c++11现在还有个问题就是windowicon无效果

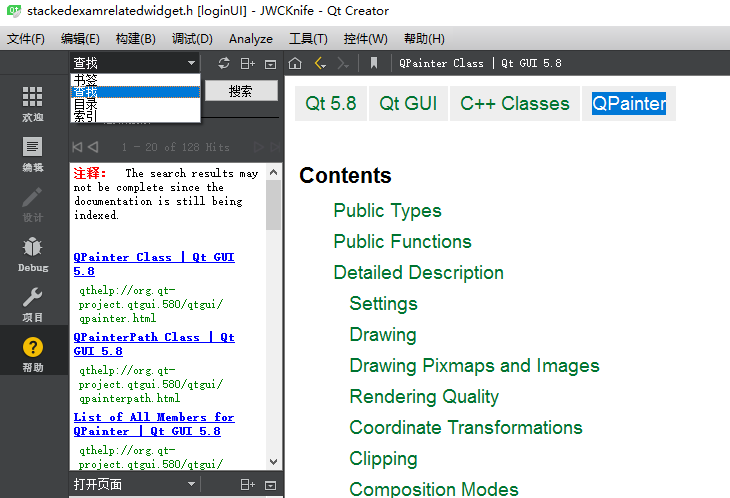
分栏显示

这个功能只要用  Qt Creator开发基本上都会用到。这个快捷键操作方法比较特别:

先按Ctrl + e后松开再按2添加上下布局的分栏

先按Ctrl + e后松开再按3添加上下布局的分栏

先按Ctrl + e后松开再按1删除所有的分栏



QSS <http://doc.qt.io/archives/qt-4.8/stylesheet-examples.html>

<https://qtdream.com/>

Qt-360 https://github.com/jun-zhang/Qt360

大项目自己对代码文件安排合理的文件夹结构进行放置，以后只需要qt中对项目右键-》增加已存在文件夹，qt就会根据文件名后缀自动进行归类。然后在pro中修改文件的路径，之后的MakeFile.debug是根据pro文件中的路径进行产生的。

控件-》显示边栏或者show Mode Selector 控制左侧工具栏的显示

C-S-F11全屏（全屏以后，最小化，退出全屏，关闭程序三个按钮全没了只能使用快捷键退出全屏）

qt界面默认都发灰，最好设置背景色。虽然我的win10上显示是白色。

qt 的login按钮，当登录信息正确会new一个窗口出来，但是在验证的过程中，如果用户多次点击，只能执行一次函数哦。

创建的新窗口应该指定位置，deepin中怎么默认显示在偏左上角而不是屏幕中间

-1.C++的GUI编程同Java不同：GUI并不是C++标准的一部分。所以，如果使用Java，那么你最好的选择就是AWT/Swing，或者也可以使SWT/JFace，但是，C++的GUI编程给了你更多的选择：wxWidget, gtk++以及Qt、Direct UI。

Qt quick和DirectUI里面的控件都是逻辑上的，最终由主窗口统一渲染出来的，和浏览器的实现几乎一致。这些都得益于于PC性能的提升，这类技术在2000年之前很少流行，都是因为太卡。这种方式的好处很多，比如自定义界面的实现，快速换肤，动态切换等，这些东西在MFC里面不是不能做，只是对程序员要求很高。Direct UI的发布包很小。

首先说wxWidget，这是一个标准的C++库，和Qt一样庞大。它的语法看上去和MFC类似，有大量的宏。据说，一个MFC程序员可以很容易的转换到 wxWidget上面来。wxWidget有一个很大的优点，就是它的界面都是原生风格的。这是其他的库所不能做到的。wxWidget的运行效率很高，据说在Windows平台上比起微软自家的MFC也不相上下。

gtk++其实是一个C库，不过由于C++和C之间的关系，这点并没有很大的关系。但是，gtk++是一个使用C语言很优雅的实现了面向对象程序设计的范例。不过，这也同样带来了一个问题——它的里面带有大量的类型转换的宏来模拟多态，并且它的函数名“又臭又长(不过这点我倒是觉得无所谓，因为它的函数名虽然很长，但是同样很清晰)”，使用下划线分割单词，看上去和Linux如出一辙。由于它是C语言实现，因此它的运行效率当然不在话下。gtk++并不是模拟的原生界面，而有它自己的风格，所以有时候就会和操作系统的界面显得格格不入。

再来看Qt，和wxWidget一样，它也是一个标准的C++库。但是它的语法很类似于Java的Swing

他们都是GUI编程的界面框架，能做的游戏估计就是贴图然后再加上一些自己的实现，要写游戏估计就需要专门的游戏引擎这些。游戏引擎就是方便进行贴图等操作。

QT的应用层主要是大型3d，vr，管理软件和器械嵌入软件。

0.Qt Creator的License，似乎策略和微软的vs类似，商业用户需要付钱。WPS以及YY就是基于Qt的。Qt 是Nokia公司。实际是在使用标准 C++ 编译器编译 Qt 源程序之前，Qt 先使用一个叫做 moc（Meta Object Compiler，元对象编译器）的工具，先对 Qt 源代码进行一次预处理，产生标准的C++代码。

qt 在java上面的名字叫做QTt Jambi（很明显，qt不适合为java弄一个图形库，因为java已经具有不少良好的图形库了），Python中叫做PyQt

1.在windows上部署qt环境，需要安装qt win框架和qt creator。不喜欢在vs下使用qt，所以就提一下qt有个vs的插件。qt win的安装需要mingw的完整安装（在安装过程中会要求指定mingw的安装目录）,之后“开始”-》Qt 4.8.5 (Build Debug Libraries)，接下来我们需要做的就是编译Library（使用安装时选中的mingw进行编译）， 看到命令行不要害怕，一般里面不会让你设置太多，只有两次交互，第一次是让你选择开源‘o’还是商业版本‘c’，键盘点击o。第二次是提示你是否同意条款。键盘点击 y。之后吧两者关联即可，在“工具”-》“选项”-》“qt”中设置好版本名称、qmake路径、MinGW（如果环境变量已经设置，那么会被自动检测）,可能还要在编译设置那里设置调试用的gdb路径。

很遗憾，这种自己组装的qt我打不起来。错误原因未知（我已经找到了匹配或者自带mingw）。最终用qt-opensource-windows-x86-mingw491\_opengl-5.4.0，这种一体式安装。<http://download.qt.io/archive/qt/>

YY语音、wireshark是用qt编写

2.示例程序分为Qt C++和Qt Quick两类。

文件说明：o（project） i（include） f（feature） l（link）

pri文件 将pro文件的一部分内容放到这里然后在pro中包含include(xxx.pri)

prf文件 当我们在pro的CONFIG中指定一个东西时，qmake就会尝试去加载相应的feature文件。使用load(xxx)在pro中被包含。

prl文件 主要和生成与使用静态库密切相关(动态库也可以有该文件，去Qt安装目录下的lib目录下看看即可) pro中使用CONFIG += create\_prl

pro文件的内容会根据文件是否还存在自动进行更新。

3.Exploer是窗口的意思，所以“在Explorer中显示”就会开启一个新窗口显示。

4.项目的“大纲视图”，会显示文件中所有类、函数和变量。要开启“与便一起同步”（这种连接同步图标都是两个环套在一起）从而定位函数。

5.帮助文档可以设置书签（工具栏上绘出现功能图标）。

6.安装Qt的时候会安装：Qt Assistant助手、Qt Desiger设计师、Qt Examples and Demos（Qt演示程序与示例）、Qt Linguist语言家

Qt Assistant是可配置且可重新发布的文档阅读器。自己定制然后和应用程序一起发布。可以做到书签和在应用程序中提供在线帮助等。

Qt Designer Qt上的GUI布局和格式构建器

Qt Linguist 提供一套加速应用程序翻译和国际化的工具。（收集UI文本提供给翻译人员）

7.文件介绍：

界面文件.ui 项目中双击进入Qt Designer。这是一个XML文件，通过编译可以生成ui头文件。叫做ui\_xxx.hh。

项目文件.pro

与用户有关的项目信息.pro.user

8.如果不是从qt里面运行界面程序，而是直接双击生成的exe，会因为缺少dll而无法运行，所需要的dll文件去Qt\bin找。要么自己移动dll到工程目录下，要么在path中设置（现在认识到了path中可以设置寻找dll）。mingwm10.dll、libgcc\_s\_dw2-1.dll、QtCore4.dll、QtGui4.dll。

补充：

如果在程序中使用了png以外格式的图片，在发布程序的时候就要将Qt安装目录下的plugins中的imageformts文件夹复制古来，保留其中自己要使用的格式对应dll。

上面是动态编译，如果是静态编译的话要是要将qt的库进行重新编译的。exe文件会很大。

9.左边一栏里面图标长得像电脑样的，叫目标选择器。

10.设置应用程序图标：

1.创建.ico图片文件。myico.rc文本文件，其中内容是:IDI\_ICON1 ICON DISCARDABLE "xxx.ico"

2.修改项目文件：在.pro最后面加上RC\_FILE +=myico.rc

11.纯代码qt：

#include <QApplication>

#include <QDialog>

#include <QLabel>

#include <QTextCodec> //文本编码转换功能

int main(int argc, char \*argv[]) {

QApplication a(argc, argv);

QTextCodec::setCodecForTr(QTextCodec::codecForLocale()); #指定转换目标为当前系统字符集，可以自己指定字符集"gb2312"

QDialog w;

QLabel label(&w); #传入地址才可以改变，这个函数还不支持引用么。

label.setText(QObject:;tr("输出这里的字符串"));

w.show();

return a.exec();//放到return这里有一个好处就是确保编写者不会有代码写在消息循环后面

}

被建议所有输出到界面上的字符串都是用tr()函数括起来. 你必须告诉tr这个窄字符串是何种编码？你不告诉它，它就用latin1。于是所谓的乱码问题就出来了:

QTextCodec::setCodecForTr(QTextCodec::codecForName(“GB2312”));

QTextCodec::setCodecForTr(QTextCodec::codecForName(“UTF-8”));

tr函数还能用来程序的国际化。

如果有ui\_xxx.h，那么直接包含进来，然后：

#icnlude "ui\_helloDialog.h"

int main(int argc, char \* argv[]) {

QApplication a(argc, argv);

QDialog w;

Ui::HelloDialog ui;

ui.setupUi(&w);//调用成员函数来显示界面

w.show();

return a.exec();

}

12.QDialog

成员函数：

resize(int ,int ) //单位是像素

13.QLabel 可以装载图片

成员函数：

move(int x,int y)

QLabel {

border-image: url(:/Resources/MainWindow/warning.png);

}

14.Qt提供的Command Prompt命令行已经配置好编译环境。 qmake -project用来生成pro文件。

直接执行qmake会产生.qmake.stash

qmake -o Makefile xxx.pro //如果没有.qmake.stash，会自动创建

补充：

cmake 似乎也是会创建一个Makefile。

15.QWidget是基本窗口类。QMainWindow和QDialog是其子类。

QMainWindow 类提供一个有菜单条、锚接窗口（例如工具条）和一个状态条的主应用程序窗口。主窗口通常用在提供一个大的中央窗口部件（例如文本编辑或者绘制画布）以及周围 菜单、工具条和一个状态条。QMainWindow常常被继承，因为这使得封装中央部件、菜单和工具条以及窗口状态条变得更容易，当用户点击菜单项或者工具条按钮时，槽会被调用。

QDialog类是对话框窗口的基类。对话框窗口是主要用于短期任务以及和用户进行简要通讯的顶级窗口（不会放到其他窗口中）。

QWidget类也可以从窗口系统接收鼠标、键盘和其他事件（不是顶级窗口）。

补充：

QFrame类是有框架的窗口部件的基类。

而且似乎控件都被称为窗口，只有顶级窗口不会被放在窗口内。

16.QMessageBox

学习网址：<http://blog.51cto.com/devbean/217694>

QMessageBox::infomation(0, "title", "text") //第一个参数可以使用this

<=>

QMessageBox::infomation(0, "title", "text", QMessageBox::Ok, 0)

直接使用QMessageBox在网络编程中，比如说对网页的返回情况，会因为批量操作导致弹出多个窗口，因为阻塞的原因，所以是点完一个继续弹出一个这样子。

QMessageBox的弹出会导致程序运行的阻塞，因为需要得到用户的反馈。

17.Qt 3d:

Qt 3D是Qt 5.5发布时最重要的一个新模块，It works on any GPU supporting OpenGL ES 2.0。

源码编译安装http://blog.csdn.net/gamesdev/article/details/43404647

在qmake.pro中增加QT += 3dcore 3drender 3dinput 3dlogic 3dextras

A Qt Quick application requires:QT += 3dcore 3drender 3dinput 3dlogic 3dextras qml quick 3dquick

头文件使用：

#include <Qt3DCore>

#include <Qt3DRender>

#include <Qt3DInput>

#include <Qt3DLogic>

#include <Qt3DExtras>

引入了QML用于简短xx.h与xx.cpp

18.qt还有个webengine，以后尝试

19.pro文件编写：

!include( ../examples.pri ) { //这样就可以在examples.pri中编写规则

error( "Couldn't find the examples.pri file!" )

}

QT += 3dcore 3drender 3dinput 3dextras #qt编程引用模块

SOURCES += \ #代码源文件加入工程里面的“资源”中

main.cpp \

orbittransformcontroller.cpp

HEADERS += \ #代码头文件加入工程

orbittransformcontroller.h

TARGET =goBang #然后就会将生成可执行文件叫做goBang，在窗口的名字上

见到过一个.pro内容是：

unix:!android {

isEmpty(target.path) {

qnx {

target.path = /tmp/$${TARGET}/bin

} else {

target.path = /opt/$${TARGET}/bin

}

export(target.path)

}

INSTALLS += target

}

export(INSTALLS)

通过QML还设置一个按钮;

Rectangle{

width: 300;height: 300

Rectangle {

id:button\_type

// property alias text: textElement.text;

width: 60; height: 30

border.color: "steelblue";border.width:2

anchors.centerIn: parent //这句话的意思估计是摆放到正中间

radius: 8

color: "mediumaquamarine"

Text {

id: textElement

anchors.centerIn: parent

font.pointSize: 10

style: Text.Raised;

text:"click me"

color: "black"

}

}

}

qml 中的 property 的作用是动态为对象添加属性，并且添加的属性可以进行跟踪（值被修改了就会发出信号）

这只是一个简单的样式，其中的按钮名字，需要在text中给出，但是如果我们需要很多按钮，就得这样重复的去写，这样很是麻烦。要是我们能做一个这样的按钮，并且后面可以直接调用这样的样式，就想一个函数一样，我们只管把按钮名字传入，就可以直接使用这个按钮，那就好多了，那么我们就可以使用QML中的alias别名。Rectangle和Text是两个属性，而按钮名字是属于Text的text

所以按照下面进行别名设置:

Rectangle {

id:button\_type

property alias text: textElement.text;

width: 60; height: 30

border.color: "steelblue";border.width:2

anchors.centerIn: parent

radius: 8

color: "mediumaquamarine"

Text { //这花括号的意义就是让自己认识的

id: textElement

anchors.centerIn: parent

font.pointSize: 10

style: Text.Raised;

// text:"click me"

color: "black"

}

}

别名使用：

Rectangle{

width: 300;height: 300

Button{ //这花括号的意义就是让自己认识的

text:"button"//设置的是Rectangle的text属性

}

}

QML Profiler (性能分析器)

Qt的SceneGraph框架。后面发现在SceneGraph下进行渲染有种种限制，SceneGraph能让我们用数据驱动的方式告诉我们要渲染什么东西，而Frame Graph能让我们用数据驱动的方式告诉我们怎样渲染。

企业版提供更多功能

Pixmap cache

Scene graph

Memory usage

Input events

QML性能优化指南

尽量避免打断批量绘制操作（剪切，透明，反锯齿等等）

正确使用Item Layers

图片格式 JPEG vs PNG

尽量使用Item作为容器，而不是Rectangle

使用企业版Qt Quick Compiler可优化页面加载速度

使用Animator替代Animation

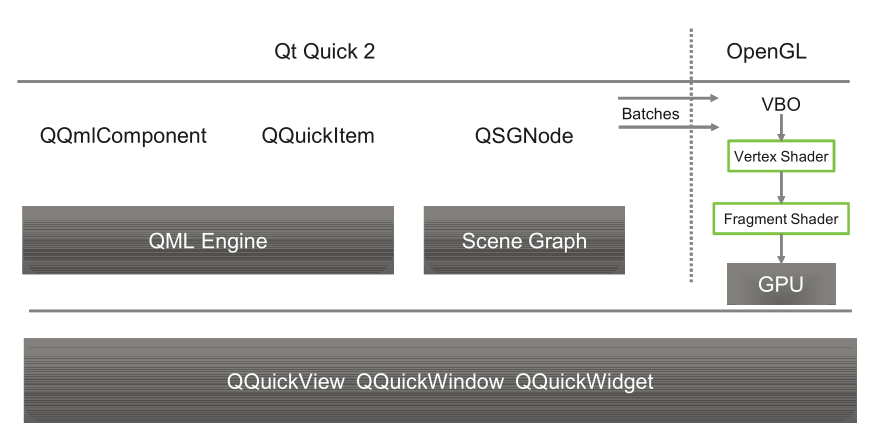
直接在Scene Graph线程执行动画，不受主线程阻塞影响

动画执行过程中不会改变原始值，不会产生多余的信号槽调用

动画不可逆

支持动画效果有限

Batches是Scene Graph对QML Items绘制的优化（实现批量绘制）。Batches越少，Scene Graph向GPU提交更少的绘制请求。打断批量绘制可能使得性能严重下降。



非常不建议使用C++来访问QML，建议在QML中使用C++

Qt 3D 示例程序学习，按F1对代码中的指定目标进行追踪查看，按一下F1会在边上出现帮助小页面，再按一下F1就会直接跳到“帮助”这个大页面。

示例Basic Shapes C++项目:

代码都很严谨啊，确保传递的类型是float，符合函数的编写

设置图形网格对象： 只设置密切关于图形的属性

m\_torus = new Qt3DExtras::QTorusMesh();

m\_torus->setRadius(1.0f);

m\_torus->setMinorRadius(0.4f);

m\_torus->setRings(100);

m\_torus->setSlices(20);

设置显示平台： 大小 位置

Qt3DCore::QTransform \*torusTransform = new Qt3DCore::QTransform();

torusTransform->setScale(2.0f);

torusTransform->setRotation(QQuaternion::fromAxisAndAngle(QVector3D(0.0f, 1.0f, 0.0f), 25.0f));

torusTransform->setTranslation(QVector3D(5.0f, 4.0f, 0.0f));

设置网格的颜色：（实现phong lighting effect）

Qt3DExtras::QPhongMaterial \*torusMaterial = new Qt3DExtras::QPhongMaterial();

torusMaterial->setDiffuse(QColor(QRgb(0xbeb32b)));

将之前的内容整合到 an entity tree：

m\_torusEntity = new Qt3DCore::QEntity(m\_rootEntity);

m\_torusEntity->addComponent(m\_torus);

m\_torusEntity->addComponent(torusMaterial);

m\_torusEntity->addComponent(torusTransform);

通过这个函数来设置entity的是否可见：

void SceneModifier::enableTorus(bool enabled)

{

m\_torusEntity->setParent(enabled ? m\_rootEntity : nullptr);//Qt3DCore::QEntity \*m\_rootEntity; private属性

}

示例Simple C++ Example：

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

{

QGuiApplication app(argc, argv);

Qt3DExtras::Qt3DWindow view;

Qt3DCore::QEntity \*scene = createScene(); //下面这个函数一直是new创建的，所以内存不会在函数退出的时候释放

// Camera

Qt3DRender::QCamera \*camera = view.camera();

camera->lens()->setPerspectiveProjection(45.0f, 16.0f/9.0f, 0.1f, 1000.0f);

camera->setPosition(QVector3D(0, 0, 40.0f));

camera->setViewCenter(QVector3D(0, 0, 0));

// For camera controls

Qt3DExtras::QOrbitCameraController \*camController = new Qt3DExtras::QOrbitCameraController(scene);

camController->setLinearSpeed( 50.0f ); //the movement speed of the camera.

camController->setLookSpeed( 180.0f ); //determines the turn rate of the camera pan and tilt

camController->setCamera(camera);

view.setRootEntity(scene); //设置根容器

view.show();

return app.exec(); //程序执行

}

Qt3DCore::QEntity \*createScene()

{

// Root entity

Qt3DCore::QEntity \*rootEntity = new Qt3DCore::QEntity;

// Material

Qt3DRender::QMaterial \*material = new Qt3DExtras::QPhongMaterial(rootEntity);

// Torus

Qt3DCore::QEntity \*torusEntity = new Qt3DCore::QEntity(rootEntity); #在根容器里面再加一个容器

Qt3DExtras::QTorusMesh \*torusMesh = new Qt3DExtras::QTorusMesh;

//....here set the attributes of torus

Qt3DCore::QTransform \*torusTransform = new Qt3DCore::QTransform;

torusTransform->setScale3D(QVector3D(1.5, 1, 0.5));

torusTransform->setRotation(QQuaternion::fromAxisAndAngle(QVector3D(1, 0, 0), 45.0f));

torusEntity->addComponent(torusMesh);

torusEntity->addComponent(torusTransform);

torusEntity->addComponent(material);

// Sphere

Qt3DCore::QEntity \*sphereEntity = new Qt3DCore::QEntity(rootEntity);

Qt3DExtras::QSphereMesh \*sphereMesh = new Qt3DExtras::QSphereMesh;

sphereMesh->setRadius(3);

Qt3DCore::QTransform \*sphereTransform = new Qt3DCore::QTransform;

OrbitTransformController \*controller = new OrbitTransformController(sphereTransform);

controller->setTarget(sphereTransform);

controller->setRadius(20.0f);

QPropertyAnimation \*sphereRotateTransformAnimation = new QPropertyAnimation(sphereTransform); //动画对象

sphereRotateTransformAnimation->setTargetObject(controller); //设置目标

sphereRotateTransformAnimation->setPropertyName("angle");

sphereRotateTransformAnimation->setStartValue(QVariant::fromValue(0));

sphereRotateTransformAnimation->setEndValue(QVariant::fromValue(360));// 0-360度设置的是度数，来表示起始位置与结束位置

//在这个项目里，0表示的是整个场景的右边，如果你是720就表示一次动画会转2圈

sphereRotateTransformAnimation->setDuration(10000); //设置动画时间，默认是250ms

sphereRotateTransformAnimation->setLoopCount(-1); //设置动画循环重复次数，如果是0就会一次都不运行，如果是-1就会一直运行

sphereRotateTransformAnimation->start();

sphereEntity->addComponent(sphereMesh);

sphereEntity->addComponent(sphereTransform); //给它设置了动画，最后加到容器中的还是这个对象

sphereEntity->addComponent(material);

}

示例Simple C++ QML：

这个项目里面在“资源”-》simple-qml.qrc-》/ -》CameraController.qml，这个文件对项目不起任何作用

还有“资源”-》simple-qml.qrc-》/ -》main.qml和QML-》main.qml文件的内容是一样的而起作用，这两个文件不管那个文件被注释了都会导致程序不能运行，后来更改了下文件发现就是映射关系

#include <Qt3DQuickExtras/qt3dquickwindow.h>

#include <QGuiApplication>

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

{

QGuiApplication app(argc, argv);

Qt3DExtras::Quick::Qt3DQuickWindow view;

view.setSource(QUrl("qrc:/main.qml")); //在C++中引用资源加不加 qrc: 前缀实际上跟引用资源的方法有关，如果在引用资源时使用了 QUrl 这个类，则必须在资源文件的路径前面加上「 qrc:」 ，否则，QUrl 只会到本地目录去搜索。

view.show();

return app.exec();

}

在“资源”-》simple-qml.qrc-》/ -》main.qml

Entity {

id: sceneRoot

Camera {

id: camera

projectionType: CameraLens.PerspectiveProjection

fieldOfView: 45

aspectRatio: 16/9

nearPlane : 0.1

farPlane : 1000.0

position: Qt.vector3d( 0.0, 0.0, -40.0 )

upVector: Qt.vector3d( 0.0, 1.0, 0.0 )

viewCenter: Qt.vector3d( 0.0, 0.0, 0.0 )

}

OrbitCameraController {

camera: camera

}

components: [ //这里使用列表来进行插入，其实由于在Entity的花括号中这本身就是已经在设计components了。这是可以去掉的

RenderSettings {

activeFrameGraph: ForwardRenderer {

clearColor: Qt.rgba(0, 0.5, 1, 1)

camera: camera

}

},

// Event Source will be set by the Qt3DQuickWindow

InputSettings { }

]

PhongMaterial {

id: material

}

//Torus draw

TorusMesh {

id: torusMesh

radius: 5

minorRadius: 1

rings: 100

slices: 20

}

Transform {

id: torusTransform

scale3D: Qt.vector3d(1.5, 1, 0.5)

rotation: fromAxisAndAngle(Qt.vector3d(1, 0, 0), 45)

}

Entity {

id: torusEntity

components: [ torusMesh, material, torusTransform ]

}

// Sphere draw and animation

SphereMesh {

id: sphereMesh

radius: 3

}

Transform {

id: sphereTransform

property real userAngle: 0.0

matrix: {

var m = Qt.matrix4x4();

m.rotate(userAngle, Qt.vector3d(0, 1, 0));

m.translate(Qt.vector3d(20, 0, 0));

return m;

}

}

QQ2.NumberAnimation {

target: sphereTransform

property: "userAngle"

duration: 10000

from: 0

to: 360

loops: QQ2.Animation.Infinite

running: true

}

Entity {

id: sphereEntity

components: [ sphereMesh, material, sphereTransform ]

}

}

再看Scene3D QML Example项目，Shadow Map QML项目挺好的

Qt 3D和其它的库不一样，它的main.qml放的是Entity，而不是我们常见的Window或者ApplicationWindow。使用的引擎不是简单的QQmlApplicationEngine类，而是Qt3D::Quick::QqmlAspectEngine类。Qt3D自己封装了一个窗口，名叫Qt3D::Window，它也不继承QQuickWindow，而是继承它的父类QWindow

camera可以理解为从什么角度进行观看，不过其实代表的是最终的渲染。计算机视觉参数fieldOfView为摄像机的纵向（并非横向）视场角（并非半视场角），aspectRatio为宽高比，nearPlaneDistance，farPlaneDistance分别为进、远剪切面。我见过是这么分别是安排的(45.0f, 16.0f/9.0f, 0.1f, 1000.0f);值。

类型知识：

phong lighting effect由ambient,diffuse and specular构成（环境即没光源时的颜色、镜面反射（光泽由浮动属性控制）、漫反射），反射的强度可以设置系数来控制。还说是为OpenGL提供了技术支持。

QMatrix4x4类 represents a 4x4 transformation matrix in 3D space. If the matrix has a special type (identity, translate, scale, etc), the programmer should follow this constructor with a call to optimize() if they wish QMatrix4x4 to optimize further calls to translate(), scale()。

Qt3DCore::QTransform对象->setMatrix

Qt3DRender::QCameraLens specifies the projection matrix that will be used to define a Camera for a 3D scene

模型的导入：

Qt3D内有关模型的导入，使用的是Assimp库，于是有些错误可能来自于Assimp库。比如说有时候会出现“Assetimporter error: Chunk is too large”这样的问题，其实这个问题不是来自Qt3D的，而是底层的Assimp库的。这里我分享一个Assimp库实现的软件，叫“assimpviewer”。它可以方便地浏览各种模型格式的文件，其下载地址在这里。它是用Assimp库和DirectX来实现的，很遗憾只有Windows版。如果assimpviewer无法打开，那么Qt3D可能也载入不了了

位置关系的设置：

MyButton

{

id: testButton

height: 50; width: 200

color: "burlywood"

border.color: "black"

border.width: 2

anchors.horizontalCenter: exitButton.horizontalCenter

anchors.bottom : exitButton.top

anchors.bottomMargin: 20

text: "test"

mouseArea.onClicked: {

Main\_GameWorld.show()

}

}

MyButton

{

id: exitButton

height: 50; width: 200

color: "burlywood"

border.color: "black"

border.width: 2

anchors.centerIn: parent //放到中间,也可以指定一个绝对位置

text: "EXIT"

mouseArea.onClicked: {

Qt.quit()

}

}

21，在Qt中，导入的项目要先配置（如果未配置那么就会在左下角的电脑上有显示“未配置”）才能运行或者调试。

22.信号与槽：

Q\_PROPERTY(Qt3DCore::QTransform\* target READ target WRITE setTarget NOTIFY targetChanged)

signals:

void targetChanged();

通过emit targetChanged();来触发信号调用setTarget函数

23.Qt里面不允许使用1f，应该是1.0f也可以1.f。反正f这个修饰前面要是浮点数形式。

24.QML的理解：JSON-like的形式，可以定义js函数，引入js文件，使用的是C++引擎来实现。

25.qt中构建目录与源文件必须为同级目录是什么原因

删掉 xxx.pro.user文件

重新打开.pro文件

自动弹出重新进行目标设置（在项目里构建设置里重新选择构建目录），重新配置项目

26.QT会在与项目目录的同目录下生成一个build文件夹，直接打开其中的exe就可以知道缺少了哪些dll然后直接放到exe的同一目录下就好，从qt安装目录中就好。qt Creator中运行的话会自动连接所需要的文件。dll名字中最后面带了个d比如说Qt53DQuickExtrasd.dll就是debug会使用的dll文件，然后就会非常的大，100KB正常dll会变成几MB的差距。

无法定位程序输入点与动态链接库xxx.dll 确认你的dll没找错，应该是从D:\Qt\Qt5.8.0\5.8\mingw53\_32\bin中找。目前Qt 3D的一些dll在这下面还没有，所以3D的就不能移植了。3D的dll是在mingw53\_32\lib下

27.从github上下载的项目如果带有.user这个项目本地配置文件，要删除，因为不然就会直接按照对方的配置（qmake、构建目录等）来尝试构建这个项目，毫无疑问结果会失败。

26.QString

没有toascii成员函数，Qt5改为toLatin1()

QString s1 = "我是中文";

QString s2("我是中文");

QString s3 = QObject::tr("我是中文")

QString s4 = QObject::tr("我是中文%1").arg（1）

QString s5 = QString("%1 哈哈哈").arg(1) 这个可以用来作为int转QString

long a = 63;

QString s = QString::number(a, 10);             // s == "63"

QString t = QString::number(a, 16).toUpper();     // t == "3F"

QString t = QString::number(a,'f',2); //保留两位小数

QString转数字：toDouble()等函数，toInt()第二个参数可以设置进制。如果是浮点数经过toInt会成为0。

QString转char \*： const char \* p = qstr.toLocal8Bit().data();  、

或者const char \*p = qstr.toStdString().data();

从前有的人喜欢使用

QTextCodec::setCodecForTr(...)

QTextCodec::setCodecForCStrings(...)

QTextCodec::setCodecForLocale(...)来做到直接编码转换，但是在qt5中已经去掉了前两个函数。

现在应该QTextCodec或这QString::fromXXX 这种东西进行转换

CONFIG += c++11 #支持C++11

windows下如果你的编译器支持C++11，那么，你可以直接使用

CONFIG += link\_pkgconfig #使用 pkg-config

PKGCONFIG += zlib dtkwidget

inst.files += libEbookDataBase.so\* #比如你要安装libEbookDataBase.so\*到目录 /usr/local/lib  
inst.path = /usr/local/lib  
INSTALLS += inst

QString s = u8"我是中文";

linux下已经直接是utf8了。

QString的每一个单位是QCharRef，所以不能进行元素与元素的相加

QT5以后QString 转ASCii的成员函数是toLatin1()。endResult.at(11).toLatin1()[2]-endResult.at(11).toLatin1()[0]; 将QString中的内容进行相减。qDebug() 输出’1’+1是50，输出’2’是2。数字转QString还是应该使用QString::number静态函数。

27.QT5创建Dialog类以后

在类中会有一个私有成员： Ui::Dialog \*ui; #必须要有这个成员否则会报错

在类的构造函数中有ui->setupUi(this);

在类的析构函数中有delete ui

因为我屏蔽了这句话ui->setupUi(this);，所以在ui编辑器中的一切制作都不会得到显示。

ui->setupUi(this);这句话最好注释掉，这会导致QLineEdit，QPushButton十分长，显示怪异。应该是窗体的初始大小不一样，然后QLineEdit，QPushButton会自动适应。

Dialog::Dialog(QWidget \*parent) :

QDialog(parent),

ui(new Ui::Dialog)

{

info =new QLabel(this);

info ->setText(tr("请输入："));

lineEdit =new QLineEdit(this);

answer =new QLabel(this);

button =new QPushButton(this);

button->setText(tr("push button"));

QGridLayout \*mainLayout =new QGridLayout();

mainLayout ->addWidget(info, 0, 0);

mainLayout ->addWidget(lineEdit, 0, 1);

mainLayout ->addWidget(answer, 1, 0);

mainLayout ->addWidget(button, 1, 1);

setLayout(mainLayout);或者在构造函数里面指定this

mainLayout->setSizeConstraint(QLayout::SetFixedSize); 这个函数可以进行布局大小的固定

//ui->setupUi(this);

connect( button, SIGNAL(clicked()), this, SLOT(showAnswer()) );

}

QLineEdit {

border: 1px solid ;

border-radius: 3px;

background: white;

selection-background-color: rgb(100, 100, 100);

font-size: 14px ;

}

设置了窗口的弧度、字体大小等一般美化操作

设置了字体选中以后的颜色

QLabel {

font: bold 14px;

}

如果对QPushuButton使用了下面的QSS样式：

QPushButton {

border: 1px solid ;

border-radius: 8px;

}



直接变成与上面介绍的QLineEdit样式一样的显示情况，这说明按钮的内部颜色并不是填充颜色而是通过border实现的。

**好看的按钮QSS样式：**





QPushButton {

border: 2px solid ;

border-radius: 8px;

border-color:rgb(100,190,245);

font: bold 14pt;

color:rgb(100,190,245);

}

QPushButton:hover {

border: 2px solid ;

border-radius: 8px;

border-color:rgb(100,190,245);

background-color:rgb(200 , 250 , 250);

font: bold 14pt;

color:rgb(100,190,245);

}

QPushButton:pressed {

border: 2px solid ;

border-radius: 8px;

border-color:rgb(100,190,245);

background-color:rgb(200 , 250 , 250);

font: bold 14pt;

color:rgb(100,190,245);

padding-left:5px;

padding-top:3px;

}

**被淡化的按钮格式: 即没有边框**

border-style: flat;

QPushButton{

color:rgb(160,160,160);

border-style: flat;

}

QPushButton:hover{

color:rgb(0,0,0);

}

QPushButton:pressed{

color:rgb(0,0,0);

}

**通过QPalette设置Button颜色：**

QPalette pal = startBtn.palette();              //startBtn是我已经定义好的QPushButton对象

pal.setColor(QPalette::ButtonText, Qt::red);    //设置按钮上的字体颜色，理论上可以，实际上就是可以

pal.setColor(QPalette::Button, Qt::green);      //设置按钮背景颜色，理论上可以，实际上不可以

如果是设置QPalette:Button需要pushButton->setAutoFillBackground(true);不过似乎还是有点问题，一旦border改掉就会失效。不改，也只是在边界处能看到点泛出来颜色。

startBtn.setPalette(pal);

**QSS设置QPushButton大小的时候： 前面需加上min-，甚至是max-**

QPushButton {

min-height: 216px;

min-width: 103px;

}有没有单位效果一样

28.QString没有包含头文件却可以直接使用，好多类型都可以直接使用，只要他颜色变了也就代表能被识别。

29.神奇的语法：

QMap<int,QVariant> map;

map.insert(3,QColor(255,0,0));

qDebug() << map[3].value<QColor>();//告诉QVariant这里是个QColor。

输出：

QColor(ARGB 1, 1, 0, 0)

qDebug() << map[3].value(QColor(0,0,0))会报错。

qDebug() << &g\_keyBuf[0]; g\_keyBuf是个字符串，于是输出一直是字符串。所以使用下面：

sprintf(buf, "%p", g\_keyBuf);

qDebug() << buf;

30.有时候会遇到ui与自己在ui编辑器中设置的ui界面脱节，qt自己的原因，这时候重新构建项目就好了。

31.Qt Creator 和 Qt Designer 是两个独立的程序。Qt Creator 包含Qt Designer的全部功能，所以，有了Qt Creator，就没必要再用Qt Designer了。

32.最新的QML相对于4.x时代的QML有了很大的改进（下面列举都是Qt Quick 2.0的特性）：

（1）基于OpenGL(ES)的场景实现，提高图形绘制的效率。

（2）QML和JavaScript主导UI的创建，后台图形绘制采用C++。高效，灵活，可扩展性强。

qt提供了一个查看qml效果的工具qmlscene ，这个工具在$QTDIR/qmlscene.exe

1 import QtQuick 2.4

2 import QtQuick.Window 2.2

3 import QtQuick.Controls 1.2

4

5 /\* 这是一个多行注释，和c语言的一样 \*/

6 // 当然这是一个单行注释

7

8 Window {

9

10 id:root; // Window元素的id，应该保证在这个qml文件中名字唯一

11 visible: true;

12 width: 460; // 直接指定窗口的宽度

13 height: 288; // 直接指定窗口的高度

14

15 Image {

16 id: bg;

17 width: parent.width; // 图片的宽度依赖父窗口的宽度

18 height: parent.height; // 图片的高度依赖父窗口的高度

19 source: "qrc:///images/sky.jpg" // 引用一张图片

20 }

21

22 Image {

23 id: rocket;

24 x: (parent.width - width) / 2; // 图片位置水平居中

25 y: 40; // 图片位置距离父窗口40

26 source: "qrc:///images/rocket.png";

27 }

28

29 Text {

30 // 没有指定id，即这是一个匿名元素

31 y:rocket.y + rocket.height + 20; // 文本顶部距离rocket图片底部20

32 anchors.horizontalCenter: parent.horizontalCenter // 设置文字水平居中

33 text: qsTr("火箭发射！"); // 设置文本显示的文字

34 color: "#ff2332"; // 设置文本颜色

35 font.family: "楷体"; // 设置字体为楷体

36 font.pixelSize: 30; // 设置文字大小

37 }

38 }

自己写的C++类，使用import person 1.0

而person.h中是

#ifndef PERSON\_H

#define PERSON\_H

#include <QObject>

class Person : public QObject

{

Q\_OBJECT

Q\_PROPERTY(QString name READ name WRITE setName)// 关键是这两行

Q\_PROPERTY(int shoeSize READ shoeSize WRITE setShoeSize) //

public:

Person(QObject \*parent = 0);

QString name() const;

void setName(const QString &);

int shoeSize() const;

void setShoeSize(int);

private:

QString m\_name;

int m\_shoeSize;

};

#endif // PERSON\_H

33.add prefix...是在项目管理中新建一个文件夹。这个文件夹不一定是真实存在的。.pro文件认为它存在即可。

34.增加音乐：

C++：

player = new QMediaPlayer;

// ...

player->setMedia(QUrl::fromLocalFile("/Users/me/Music/coolsong.mp3"));

player->setVolume(50);

player->play();

QML：

Audio

{

id: backgroundMusic

source: "Images/Incidental-music.mp3"

autoPlay: true

loops: 100

//muted: false

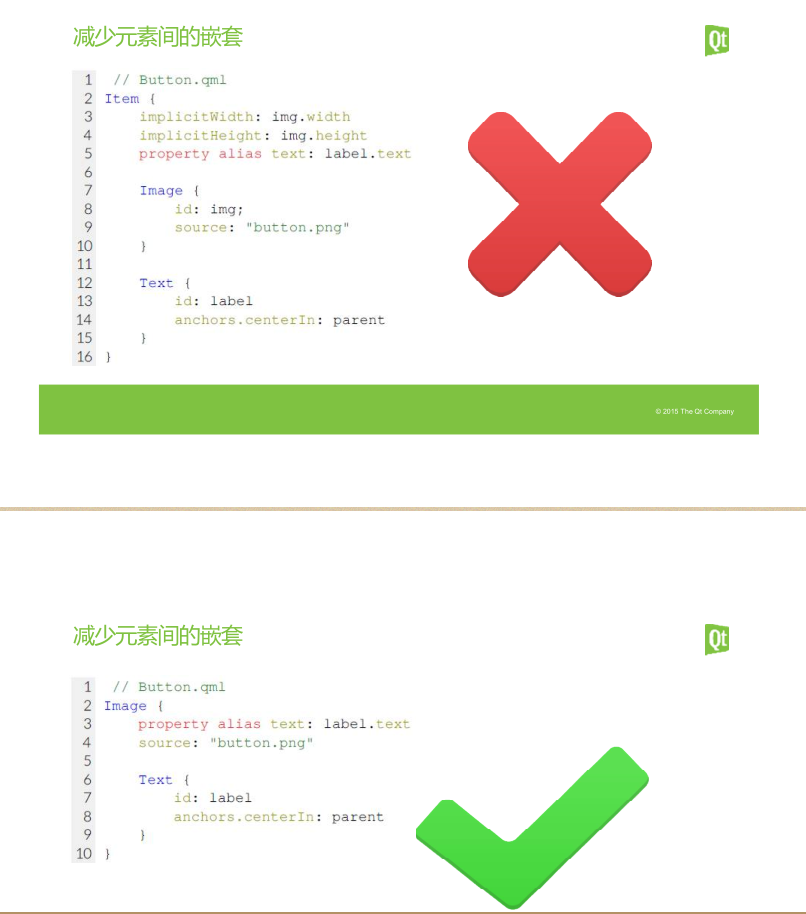
//volume: .5

}

35.QML后说：

使用readonly修饰只读属性

使用alias暴露子对象属性



Qt Quick 技术的引入，使得你能够快速构建 UI ，具有动画、各种绚丽效果的 UI 都不在话下。但它不是万能的，也有很多局限性，原来 Qt 的一些技术，比如低阶的网络编程如 QTcpSocket ，多线程，又如 XML 文档处理类库 QXmlStreamReader / QXmlStreamWriter 等等，在 QML 中要么不可用，要么用起来不方便。Qt Quick是QML的类型与函数的标准库。

其实话说回来， QML 的很多基本类型原本就是在 C++ 中实现的，比如 Item 对应 QQuickItem ， Image 对应 QQuickImage ， Text 对应 QQuickText ，……这样看来，在 QML 中访问 C++ 对象必然不成问题。然也！反过来，在 C++ 中其实也可以使用 QML 对象。QML 其实是对 JavaScript 的扩展，融合了 Qt Object 系统的宣告式编程。

id只能在本文件进行使用，算是不会垮文件的全局变量。

和Java一样，由变量名来寻找一样名字的文件

qmlscene tutorials/helloworld/tutorial1.qml #Additional module import paths can be provided using the -I flag

import QtQuick 2.0

Rectangle {

id: page

width: 320; height: 480

color: "lightgray"

border.color: "black"

border.width: 5

radius: 10

Text {

id: helloText

text: "Hello world!"

y: 30

anchors.horizontalCenter: page.horizontalCenter

font.pointSize: 24; font.bold: true

}

}

import QtQuick 2.3

Text { text: clock.time }

The text value could be filled by a dummydata/clock.qml file with a time property in the root context：（dummydata就是资源/qml.qrc/下的/）

import QtQuick 2.3

QtObject { property int time: 54321 }

在Cell.qml中

import QtQuick 2.0

Item {

id: container

property alias cellColor: rectangle.color

signal clicked(color cellColor) //希望clicked能带上颜色值

width: 40; height: 25

Rectangle {

id: rectangle

border.color: "white"

anchors.fill: parent //rectangle will have the same size as its parent

}

MouseArea {

anchors.fill: parent

onClicked: container.clicked(container.cellColor)

}

}

之后就能在main.qml中

import QtQuick 2.0

Rectangle {

id: page

width: 320; height: 480

color: "lightgray"

Text {

id: helloText

text: "Hello world!"

y: 30

anchors.horizontalCenter: page.horizontalCenter

font.pointSize: 24; font.bold: true

}

Grid {

id: colorPicker

x: 4; anchors.bottom: page.bottom; anchors.bottomMargin: 4

rows: 2; columns: 3; spacing: 3

Cell { cellColor: "red"; onClicked: helloText.color = cellColor }

Cell { cellColor: "green"; onClicked: helloText.color = cellColor }

Cell { cellColor: "blue"; onClicked: helloText.color = cellColor }

Cell { cellColor: "yellow"; onClicked: helloText.color = cellColor }

Cell { cellColor: "steelblue"; onClicked: helloText.color = cellColor }

Cell { cellColor: "black"; onClicked: helloText.color = cellColor }

}

}

设置一个动画（设置了两个状态与状态来回之间的转化）点击以后动画就会循环播放：

import QtQuick 2.0

Rectangle {

id: page

width: 320; height: 480

color: "lightgray"

Text {

id: helloText

text: "Hello world!"

y: 30

anchors.horizontalCenter: page.horizontalCenter

font.pointSize: 24; font.bold: true

MouseArea { id: mouseArea; anchors.fill: parent }

states: State {

name: "down"; when: mouseArea.pressed == true //注意这里是when

PropertyChanges { target: helloText; y: 160; rotation: 180; color: "red" }

}

transitions: Transition {

//Because we don't want the text to appear at the bottom instantly but rather move smoothly, we add a transition between our two states.“ reversible: true”就是设置了回来的动画。动画会改变的属性是y和rotation。

from: ""; to: "down"; reversible: true

ParallelAnimation {

NumberAnimation { properties: "y,rotation"; duration: 500; easing.type: Easing.InOutQuad }

ColorAnimation { duration: 500 }

}

}

}

}

Qt 提供了两种在 QML 环境中使用 C++ 对象的方式：

在 C++ 中实现一个类，注册到 QML 环境中， QML 环境中使用该类型创建对象

在 C++ 中构造一个对象，将这个对象设置为 QML 的上下文属性，在 QML 环境中直接使用改属性

不管哪种方式，对要导出的 C++ 类都有要求，不是一个类的所有方法、变量都可以被 QML 使用，因此我们先来看看怎样让一个方法或属性可以被 QML 使用：

1.只要是信号或者槽，都可以在 QML 中访问，你可以把 C++ 对象的信号连接到 QML 中定义的方法上，也可以把 QML 对象的信号连接到 C++ 对象的槽上，还可以直接调用 C++ 对象的槽或信号……所以，这是最简单好用的一种途径。

public信号也可以直接使用：（文档里面signal、slots说加上public来使用， 但是我觉得按理应该不用）

class MessageBoard : public QObject

{

Q\_OBJECT

public:

// ...

signals:

void newMessagePosted(const QString &subject);

};

MessageBoard {

onNewMessagePosted: console.log("New message received:", subject)

}

2.Q\_INVOKABLE 宏

在定义一个类的成员函数时使用 Q\_INVOKABLE 宏来修饰，就可以让该方法被元对象系统调用。这个宏必须放在返回类型前面。被建议QML中使用Q\_PROPERTY属性而不是Q\_INVOKE函数调用。

class MessageBoard : public QObject

{

Q\_OBJECT

public:

Q\_INVOKABLE bool postMessage(const QString &msg) {

qDebug() << "Called the C++ method with" << msg;

return true;

}

public slots:

void refresh() {

qDebug() << "Called the C++ slot";

}

};

// MyItem.qml

import QtQuick 2.0

Item {

width: 100; height: 100

MouseArea {

anchors.fill: parent

onClicked: {

var result = msgBoard.postMessage("Hello from QML")

console.log("Result of postMessage():", result)

msgBoard.refresh();//slots函数可以直接使用

}

}

}

//main.cpp

int main(int argc, char \*argv[]) {

QGuiApplication app(argc, argv);

MessageBoard msgBoard;

QQuickView view;

view.engine()->rootContext()->setContextProperty("msgBoard", &msgBoard);

view.setSource(QUrl::fromLocalFile("MyItem.qml"));

view.show();

return app.exec();

}

3.Q\_ENUMS

如果你要导出的类定义了想在 QML 中使用枚举类型，可以使用 Q\_ENUMS 宏将该枚举注册到元对象系统中。

class ColorMaker : public QObject

{

Q\_OBJECT

Q\_ENUMS(GenerateAlgorithm)

public:

ColorMaker(QObject \*parent = 0);

~ColorMaker();

enum GenerateAlgorithm{

RandomRGB,

RandomRed,

RandomGreen,

RandomBlue,

LinearIncrease

};

Q\_INVOKABLE GenerateAlgorithm algorithm() const;

Q\_INVOKABLE void setAlgorithm(GenerateAlgorithm algorithm);

signals:

void colorChanged(const QColor & color);

void currentTime(const QString &strTime);

public slots:

void start();

void stop();

};

一旦你使用 Q\_ENUMS 宏注册了你的枚举类型，在 QML 中就可以用 ${CLASS\_NAME}.${ENUM\_VALUE} 的形式来访问，比如 ColorMaker.LinearIncrease ，上节展示的 QML 代码片段已经使用了导出的枚举类型。

4.Q\_PROPERTY

Q\_PROPERTY 宏用来定义可通过元对象系统访问的属性，通过它定义的属性，可以在 QML 中访问、修改，也可以在属性变化时发射特定的信号。要想使用 Q\_PROPERTY 宏，你的类必须是 QObject 的子类。

例子一： any property that is writable should have an associated NOTIFY signal

//class

class Message : public QObject

{

Q\_OBJECT

Q\_PROPERTY(QString author READ author WRITE setAuthor NOTIFY authorChanged)

public:

void setAuthor(const QString &a) {

if (a != m\_author) {

m\_author = a;

emit authorChanged();

}

}

QString author() const {

return m\_author;

}

signals:

void authorChanged();

private:

QString m\_author;

};

// MyItem.qml

import QtQuick 2.0

Text {

width: 100; height: 100

text: msg.author // invokes Message::author() to get this value

Component.onCompleted: {

msg.author = "Jonah" // invokes Message::setAuthor()

}

}

//main.cpp

int main(int argc, char \*argv[]) {

QGuiApplication app(argc, argv);

QQuickView view;

Message msg;

view.engine()->rootContext()->setContextProperty("msg", &msg);

view.setSource(QUrl::fromLocalFile("MyItem.qml"));

view.show();

return app.exec();

}

例子二：

class Message : public QObject

{

Q\_OBJECT

Q\_PROPERTY(MessageBody\* body READ body WRITE setBody NOTIFY bodyChanged)

public:

MessageBody\* body() const;

void setBody(MessageBody\* body);

};

class MessageBody : public QObject

{

Q\_OBJECT

Q\_PROPERTY(QString text READ text WRITE text NOTIFY textChanged)

// ...

}

If the MessageBody type was also registered with the type system, it would be possible to assign MessageBody to the body property of a Message, all from within QML code:

Message {

body: MessageBody { //body是MessageBody指针

text: "Hello, world!"

}

}

例子三：

class MessageAuthor : public QObject

{

Q\_PROPERTY(QString name READ name WRITE setName)

Q\_PROPERTY(QString email READ email WRITE setEmail)

public:

...

};

class Message : public QObject

{

Q\_OBJECT

Q\_PROPERTY(MessageAuthor\* author READ author)

public:

Message(QObject \*parent)

: QObject(parent), m\_author(new MessageAuthor(this))

{

}

MessageAuthor \*author() const {

return m\_author;

}

private:

MessageAuthor \*m\_author;

};

Message {

author.name: "Alexandra"

author.email: "alexandra@mail.com"

}

制作可以在qml中被import的文件：

class TimeModel : public QObject

{

Q\_OBJECT

Q\_PROPERTY(int hour READ hour NOTIFY timeChanged)

Q\_PROPERTY(int minute READ minute NOTIFY timeChanged)

...

To make this type available, we create a plugin class named QExampleQmlPlugin which is a subclass of QQmlExtensionPlugin. It overrides the registerTypes() method in order to register the TimeModel type using qmlRegisterType(). It also uses the Q\_PLUGIN\_METADATA() macro in the class definition to register the plugin with the Qt meta object system using a unique identifier for the plugin.

class QExampleQmlPlugin : public QQmlExtensionPlugin

{

Q\_OBJECT

Q\_PLUGIN\_METADATA(IID QQmlExtensionInterface\_iid)

public:

void registerTypes(const char \*uri)

{

Q\_ASSERT(uri == QLatin1String("TimeExample"));

qmlRegisterType<TimeModel>(uri, 1, 0, "Time");

}

};

记得DESTDIR = imports/TimeExample。A qmldir file is required in the imports/TimeExample directory to describe the plugin and the types that it exports：

module TimeExample

Clock 1.0 Clock.qml

plugin qmlqtimeexampleplugin

使用：

import TimeExample 1.0 // import types from the plugin。版本就是1.0

Clock { // this class is defined in QML (imports/TimeExample/Clock.qml)

Time { // this class is defined in C++ (plugin.cpp)

id: time

}

hours: time.hour

minutes: time.minute

}

C++里面操作qml：（通过窗口获取与类型强转）

int main(int argc, char \*\* argv)

{

QApplication app(argc, argv);

// Register our component type with QML.

qmlRegisterType<KeyGenerator>("com.ics.demo", 1, 0, "KeyGenerator");

int rc = 0;

QQmlEngine engine;

QQmlComponent \*component = new QQmlComponent(&engine);

QObject::connect(&engine, SIGNAL(quit()), QCoreApplication::instance(), SLOT(quit()));

component->loadUrl(QUrl("main.qml"));

if (!component->isReady() ) {

qWarning("%s", qPrintable(component->errorString()));

return -1;

}

QObject \*topLevel = component->create();

QQuickWindow \*window = qobject\_cast<QQuickWindow \*>(topLevel);

QSurfaceFormat surfaceFormat = window->requestedFormat();

window->setFormat(surfaceFormat);

window->show();

rc = app.exec();

delete component;

return rc;

}

注册方式更适合qml与c++交互,缺点是c++依附于qml,不能独立运行.（C++只实现逻辑，qml根据逻辑来显示界面）

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

{

QGuiApplication app(argc, argv);

//注册组件到QML

qmlRegisterType<Login\_Gui>("Login\_Gui", 1, 0, "Login\_Gui");

QtQuick2ApplicationViewer viewer;

viewer.setMainQmlFile(QStringLiteral("qml/SH\_User/login.qml"));

viewer.showExpanded();

return app.exec();

}

而c++类独立运行,且想和qml界面交互,则可通过setContextProperty将数据提供给qml

Qt Quick用脚本QML及JS，类似于DirectUI。

而Qt Widgets用C++，即传统GUI。

36.资源下面的文件夹名字被设置为..，其中的文件还能在Qt中被正常加入，但是在windows中（也就是右键打开资源管理器）显示的是父目录。

37.widget UI

GameWidget::GameWidget(QWidget \*parent) :

QWidget(parent),

ui(new Ui::GameWidget)

{

setWindowTitle("a");

ui->setupUi(this);

}

由于UI的存在会导致构造函数中的setWindowTitle("a");无效，但是在main.cpp中构造完成以后设置setWindowTitle是有效的，说明ui文件转换为C++代码插入是在构造函数的最后面。

38.使用QWidget需要QT += widgets。加了这个以后

#include <QWidget>

#include <QGraphicsScene>

#include <QGraphicsItem>

#include <QGraphicsView>

#include <QGridLayout>都可以使用了

39.将Widget中的内容截图保存到QPixmap：render成员函数（但是不能在paintEvent中使用）

40.QPixmap主要是用于绘图，针对屏幕显示而最佳化设计，QImage主要是为图像I/O、图片访问和像素修改而设计的。

41.

labelLives是个QLabel

m\_picLives是个Qpixmap

labelLives->resize(int w, int h);并不能改变label中显示图片的大小，还是按照默认大小进行显示

最后是按照下面这种方式进行大小的限制：（但还不是自适应Label大小的方法）

QLabel \*labelMoney=new QLabel;QPixmap m\_picMoney(":/Images/dollar.png");

m\_picMoney = m\_picMoney.scaled(30, 30,Qt::KeepAspectRatio, Qt::SmoothTransformation);

labelMoney->setPixmap(m\_picMoney);

42.

QGridLayout \*bar =new QGridLayout(this);

bar->addWidget(labelMoney, 0, 0);

bar->addWidget(m\_labelMoney, 0, 1);

bar->addWidget(labelLives, 0, 10); #虽然是10，但是效果上就是2，中间并不会隔开来

bar->addWidget(m\_labelLives, 0, 11);

bar->addWidget(labelScores, 0, 20);

bar->addWidget(m\_labelScores, 0, 21);

bar->addWidget(labelWaves, 0, 30);

bar->addWidget(m\_labelWaves, 0, 31);

43.

QHBoxLayout \*gamebar =new QHBoxLayout(this);

gamebar->addStretch();

gamebar->setSpacing(50);

gamebar->addLayout(layout3);//layout3->setSpacing(3);如果不对这个进行设置，那么这里也是50的间隔，有点继承的感觉

gamebar->addLayout(layout2);

gamebar->addLayout(layout0);

gamebar->addLayout(layout);

gamebar->addLayout(layout1,Qt::AlignLeft | Qt::AlignVCenter);); 设置了对齐方式

44.自己在QWidget的继承类中再加上QWidget成员，并且配合Layout使用。还有种方法就是使用QSplit。

45.没想到资源下面的路径是：m\_picFail.load(":/Images/lose.png");。但是在qml中可以直接Images/lose.png

46.QCoreApplication is the base class, QGuiApplication extends the base class with functionality related to handling windows and GUI stuff (non-widget related, e.g. OpenGL or QtQuick), QApplication extends QGuiApplication with functionality related to handling widgets.

下面报错QWidget: Cannot create a QWidget without QApplication

#include<QApplication>

#include"MainWindow.h"

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

{

QGuiApplication app(argc, argv);

MainWindow w;

w.show();

return app.exec();

}需要将QGuiApplication改为QApplication

如果改了出现error: variable 'QApplication app' has initializer but incomplete type你需要加上

#include <QApplication>

PS：我profile中QT+=widgets

47. 配置QT5下的cocos2dx环境：

有个Port cocos2d-x to Qt5, 另外一个是当时参考的nokia的  
[https://github.com/nokia-developer/cocos2d-x-qt](https://github.com/nokia-developer/cocos2d-x-qt" \t "_blank)

直接使用Qt来编写游戏出来的样子，Qt水平烂的话真的惨不忍睹。只要是借助定时器来定时刷新。

很感谢<https://github.com/wuhaoyu1990/KingdomRush>提供代码，教学与代码中使用到的资源。

Linux下将Qt与Cocos2dx配合倒是很简单的事情，把两个都安装好以后据说在Qt新建项目的“其他项目”中间就会出现cocos2dx项目.windows下的也是因为Qt使用的是MinGW.

qDebug() 不能传入模板，所以不能传入std::endl;

**Md5：**

1.QString pwd="123456";

QString md5;

QByteArray ba,bb;

QCryptographicHash md(QCryptographicHash::Md5);

ba.append(pwd);

md.addData(ba);

bb = md.result();

md5.append(bb.toHex());

2.QString md5;

QString pwd="123456";

QByteArray bb;

bb = QCryptographicHash::hash ( pwd.toAscii(), QCryptographicHash::Md5 );

qDebug() << bb.toHex();

QCryptographicHash第一个参数与返回值都是QByteArray\

**QByteArray:**

通过toHex()转为QByteArray，这个已经可以在qDebug()中进行输出给人看了

通过toInt()转为int，但是如果转换失败，函数会返回0，比如说要转换的数太大了会溢出。

**[QByteArray](qbytearray.html" \l "QByteArray-1)**(const char \**data*, int *size* = -1)

QByteArray data;

data.append("zjh="+m\_username);

data.append("&mm="+m\_password);

**QFILE：**

QFile file("test.txt"); 这个时候会打开的文件是创建的build文件夹的根目录下的test.txt。请注意这里的路径！居然被qt重定向到build目录中。

file.open(QIODevice::ReadOnly | QIODevice::Text);

QFile htmlFile("record.html"); htmlFile.*open*(QIODevice::WriteOnly | QIODevice::Text);

QString str1 =gbk->toUnicode(str.toLatin1());

htmlFile.write(str1.toLatin1());

htmlFile.*close*(); 有个坏处，因为QByteArray与QString之间的转换可能导致乱码。所以QString要想输入文件需要通过QTextStream 实现。

QFile htmlFile("record.html"); htmlFile.*open*(QIODevice::WriteOnly | QIODevice::Text);

QTextStream in(&htmlFile);

in << str ;

htmlFile.*close*();

return; 输出以后的文件能直接被windows读取，但是要想被qt读入，需要设置为utf8格式或者改变qt读入的编码。

使用size方法得到文件的大小，但是QFile是会自动打开文件的。如果想要不打开文件就得到文件的大小可以使用QFileInfo。

**数据库操作：**

qDebug()<<QSqlDatabase::drivers(); 查看支持的数据库插件

借助SQLite自带进程内数据库，之后可以在build—xxxx文件夹下产生xxx.db数据库文件，这个数据库文件支持使用SQLite打开。除了驱动层（连接Qt与数据库管理系统），SQL接口层已经可以提供给熟悉使用SQL语法的用户，进一步封装的用户接口层中的QSqlTableModel、QSqlQueryModel、QSqlRelationTableModel等依据模型/视图来设计。

QSqlQueryModel QT说它是一种只读Model，但是其实可以通过设置Delegate来做到用户交互时的改变。

头文件：

#include <QSqlDriver>

#include <QSqlRecord> Qsqlrecord用来获取查询语句的结果

#include <QSqlError>

#include <QSqlDatabase>

#include <QSqlQuery> query用来执行语句，支持bindValue来防注入

QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");

db.setHostName() 设置主机名字

db.setDatabaseName() 设置数据库文件名字 如果不存在就会被创建 其实这里就是对路径的指定

db.setUserName() 设置连接用户名

db.setPassWord() 设置连接密码 不过坑爹之处在于原始的数据库本身不支持，预留了一个空的api对密码进行设置，官方是有收费实现的，所以加密并不能实现，除非使用第三方的dll

sqlcipher <https://github.com/sqlcipher/sqlcipher> 还使用在手机端的库，qt使用<https://github.com/devbean/QtCipherSqlitePlugin> 已经打包好，对应说明在<http://blog.csdn.net/qiangzi4646/article/details/51911607>。 编译产生两个文件.a 静态链接文件已经.dll动态链接文件。但是windows下的使用是这样的：分别在两种模式中构建出两个dll——sqlitecipher.dll、sqlitecipherd.dll，两个dll都放到D:\Qt\Qt5.8.0\5.8\mingw53\_32\plugins\sqldrivers，以后不同模式下会调用不同的dll。将一个文件夹放到D:\Qt\Qt5.8.0\5.8\Src\qtbase\src\plugins\sqldrivers中。

如果自己实现的话，需要使用跨平台的加密算法wxsqlite3等来实现空函数sqplite3\_key函数，之后还需要自己通过qt的源码来重新创建一个QtSDK。

最后需要对添加的数据库连接进行在if判断中的打开操作db.open()

QSqlDatabase db = QSqlDatabase::database("xx");获取名字叫做xx的连接，并且由于这个函数的第二个参数默认是bool open=true，所以这样子获取到的连接时打开的。

php的bind讲究格式化处理，Qt的bindValue()讲究位置：

QSqlQuerry query;

query.prepare("xxxx?????");//要被代替的地方使用问号就好  
query.bindValue(0, 1)

query.bindValue(1, "第二个参数")

query.exec() 这个函数与连接打开函数都需要进行返回值判断

prepare()了以后如果你希望进行第二次prepare就需要先执行query.clear()，而执行query.exec()不需要考虑是否已经prepare的问题。php里面没有clear这一步操作，可能因为php里面是引用，而这里是进行复制？

query.prepare("insert into user values(:acc, :pas, :sta, :nic,:ico,:sex )");  
query.bindValue(":acc", userInfo.account);

每一次prepare 都只能使用一次，第二次bind以后执行代码会出现"Parameter count mismatch"错误（出错之前的数据都成功插入了，出错时数据插入失败）。需要重新进行一次prepare，但是clear只需要调用一次。

m\_pQuery->prepare("insert into ClassSchedule values(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)");

for(int i1=0;i1<15;++i1)

m\_pQuery->bindValue(i1,"1");

SQLErrorDetectForPrepare(m\_pQuery, "插入课程表数据执行失败 in recordClassSchedule()");

for(int i1=0;i1<15;++i1)

m\_pQuery->bindValue(i1,"2");

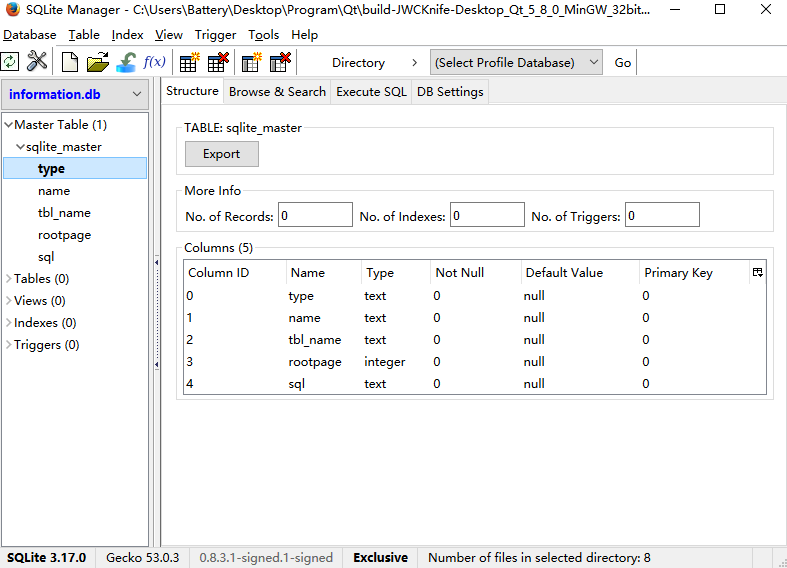
SQLErrorDetectForPrepare(m\_pQuery, "插入课程表数据执行失败 in recordClassSchedule()");

如果数据库语句编写错误也会出现这个报错。

如果函数执行不成功的话：

QSqlError lastError=query.lastError()

qDebug() << lastError.driverText() << "你的话";



执行封装函数：

bool **SQLErrorDetect** (QSqlQuery \*pQuery, const QString strCommand,const char \*info) {

if (!pQuery->exec(strCommand)){

QSqlError lastError =pQuery->lastError();

qDebug() << info << " : "<< lastError.driverText();

return false;

}

return true;

}

bool **SQLErrorDetectForPrepare** (QSqlQuery \*pQuery,const char \*info) {

if (!pQuery->exec()){

QSqlError lastError =pQuery->lastError();

qDebug() << info << " : " << lastError.driverText();

return false;

}

pQuery->clear();

return true;

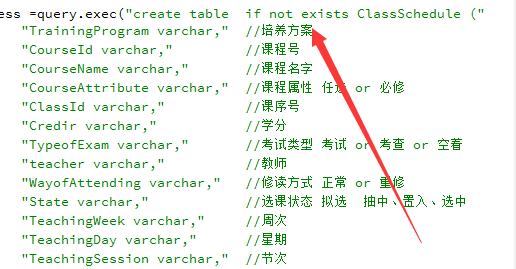
}

数据表创建失败：

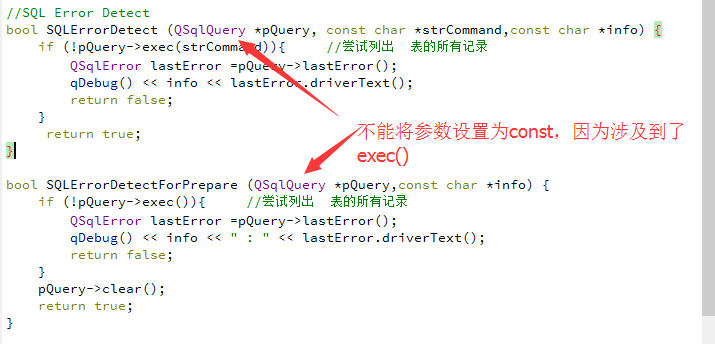
QSqlQuery query(db);

qDebug() << query.isActive();//这时候返回false

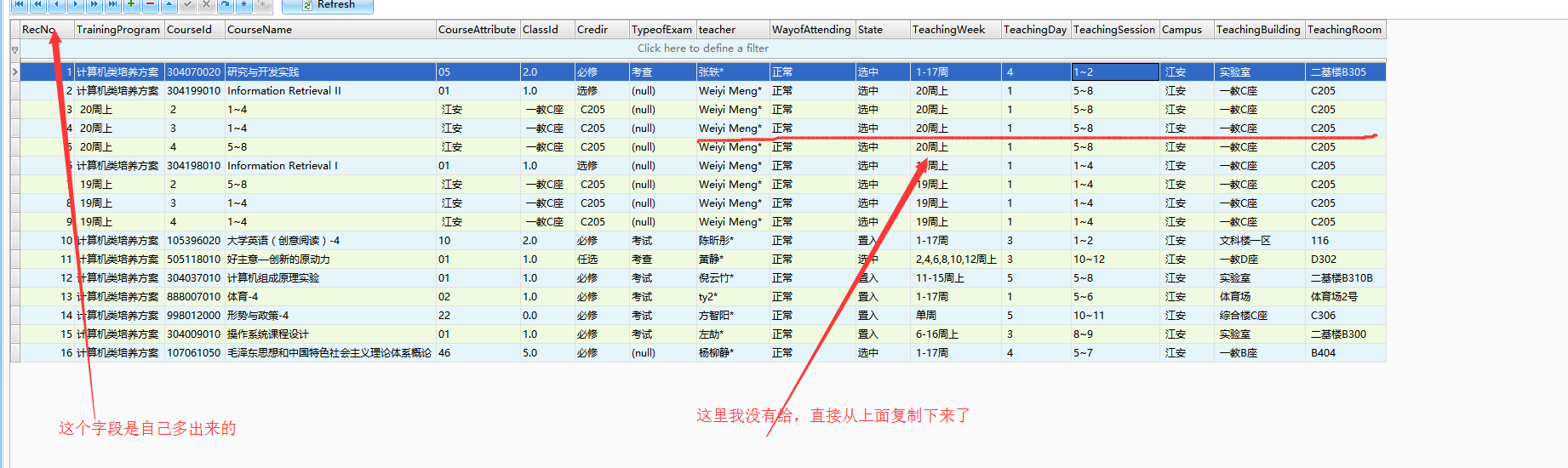
如果成功执行了一条命令并且没有被finished()或者clear()以后就会返回true，执行命令失败就会返回false。造成数据表创建失败的原因有创建语句的错误或者数据库已经建立然后你由于反复运行这个程序而倒着这套命令被反复执行就会导致出错： 解决办法是数据库语句中加上if not exists或者进行异常处理



    QSqlQuery query;  
    query.exec(“select \* from student”);  
    while(query.next())  
    {  
        qDebug() << query.value(0).toInt() << query.value(1).toString();  
    }



QSqlDatabasePrivate::addDatabase: duplicate connection name 'qt\_sql\_default\_connection', old connection removed.如果你addDatabase()建立重名字的连接就会去掉原来的连接。

多出来的字段真实并不存在，只是为了增加我们的可读性

这个自动把上面的内容复制下来有时候事件好事情，代码中一直是一条prepare

void EducationSystem::recordClassSchedule(QString &str){

pQuery->prepare("insert into ClassSchedule values(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)");

QList<QString> firstResult =findAll(str, "<tr class=(\\\")?odd(\\\")? onMouseOut=(\\\")?this.className='even';(\\\")? onMouseOver=(\\\")?this.className='evenfocus';(\\\")?>(.\*)</tr>", 7, "(&nbsp;|[\r\n\t]|<td rowspan=(\\\")?\\d(\\\")? align=(\\\")?center(\\\")?>.\*</td>)");

for(auto extract : firstResult) {

QList<QString> endResult =findAll(extract, "(<td.\*>(.\*)</td>)", 2);

int size =endResult.size();

for(int i1=0;i1<size;++i1) //对size进行一个判断

if(size==16)

pQuery->bindValue(i1,endResult.at(i1));

else

pQuery->bindValue(i1+16-size,endResult.at(i1));

SQLErrorDetectForPrepare("插入课程表数据执行失败 in recordClassSchedule()");

}

pQuery->clear();

}

datebase.exec(“xxxx”)也是可以执行SQL命令的。

addBindValue不能与bindValue混用。从维护角度， addBindValue好用很多。

To bind a NULL value, use a null [QVariant](../qtcore/qvariant.html); for example, use QVariant(QVariant::String) if you are binding a string.

如果你不希望这样子就传递空字符串去bindValue。

QSqlQuery：

SQLITE驱动不支持size()成员函数

To determine the number of rows affected by a non-SELECT statement, use [numRowsAffected](qsqlquery.html" \l "numRowsAffected)().如果是select语句，就可以使用size()成员函数来判断。

将数据库中的数据直接通过QSqlTableView进行显示。

QSqlDatabase db=QSqlDatabase::addDatabase("QSQLITE");

db.setDatabaseName("information.db");

// db.setUserName("sa"); //这两行失效了

// db.setPassword("123");

if(!db.open()){

qDebug() << "数据库连接失败";

exit(1);

}

m\_model =new QSqlTableModel(this);

m\_model->*setTable*("ClassSchedule");

qDebug() << m\_model->*select*();

m\_tableView->*setModel*(m\_model);

ui->setupUi(this);

setCentralWidget(m\_tableView);

数据库字符串里面的\n不会在View单元格中显示为换行。

注意调用select（）将恢复任何未提交的更改并删除任何插入的列。此函数的作用就是使用通过setTable（）设置的表中的数据填充模型，使用指定的过滤器和排序条件，如果成功则返回true; 否则返回false。

m\_model->setFilter("limit 0,1;")不对，以为setFilter就是使用where子句，所以应该是m\_model->setFilter("1=1 limit 0,1;") 使用了setFilter之后通过m\_model->*rowCount*() 获取记录条数

QSqlQuery或QSqlQueryModel每次最多只能缓存查询结果的256条，所以rowCount()其实最多就是256。可以通过m\_model->fetchMore();来增加缓存的数目，但是只是变成了511，依旧不是全部。所以对于大量数据要想获取整个行数的话，需要执行SQL指令，通过COUNT()函数统计。

SQLite这种文件数据库与MySql机制不一样，每条事务都有打开和关闭文件的步骤，SQLite默认将每条语句看成单独的事务。当我逐条插入数据时，就会出现大量的文件IO操作，效率自然不高。需要将多个插入操作放到一个事务中，就可以显著提升插入效率。

execBatch()是进行批处理操作

两个数据库之间进行信息复制：

bool **SQLErrorDetect** (QSqlQuery \*pQuery, const QString strCommand, const char \*info);

bool **SQLErrorDetectForPrepare** (QSqlQuery \*pQuery, const char \*info);

void **datatransfer**(void) {

QSqlDatabase db=QSqlDatabase::addDatabase("SQLITECIPHER");

db.setDatabaseName("information.db");

db.setUserName("sa");

db.setPassword("123");

db.open();

QSqlQuery \*sqlQuery =new QSqlQuery(db);

QSqlDatabase db1=QSqlDatabase::addDatabase("QSQLITE","aaa");

db1.setDatabaseName("information1.db");db1.open();

QSqlQuery \*sqlQuery1 =new QSqlQuery(db1);

SQLErrorDetect(sqlQuery,"CREATE TABLE IF NOT EXISTS PE\_exam (question varchar(150) DEFAULT NULL \

, A integer DEFAULT NULL\

, B integer DEFAULT NULL\

, C integer DEFAULT NULL\

, D integer DEFAULT NULL\

, answer varchar(35) DEFAULT NULL);", "创建");

SQLErrorDetect(sqlQuery1,"select \* from PE\_exam;", "得到数据");

while(sqlQuery1->next()) {

sqlQuery->prepare("insert into PE\_exam values(?,?,?,?,?,?)");

for(int i1=0;i1<6;++i1)

sqlQuery->addBindValue(sqlQuery1->value(i1));

SQLErrorDetectForPrepare(sqlQuery,"插入");

}

SQLErrorDetect(sqlQuery,"select count(\*) from PE\_exam;", "检验数据");

sqlQuery->next(); qDebug() << sqlQuery->value(0).toInt();

sqlQuery->clear();delete sqlQuery;

sqlQuery1->clear();delete sqlQuery1;

db.close();

db1.close();

}

**QTime：**

QTime t;

t.start();

……

qDebug() << QObject::tr("耗时%1 ms haha%2").arg(t.elapsed()).arg(1);

t.restart();

Qt不提供跨平台的sleep解决方案，不需要在主线程中使用 sleep 函数。

void **sleep**(int ms) {

QTime t;

t.start();

while(t.elapsed()<ms)

QCoreApplication::processEvents(); //保持事件循环，从而响应用户点击

}

QCoreApplication::exit();**就是退出事件循环，而这由于main.c的最后一句return就是在进行事件循环，所以会导致程序的退出。不过用这种方式退出的时候，windows弹框“程序异常退出”。说明姿势还是不对。**

**QFormLayout：**

其中的控件以两列的形式被布局在表单中

因为下面所以非常适合作文文本输入框：

addRow(const QString &labelText, QWidget \*field)来创建一个带有给定文本的QLabel及QWidget控件行

QFormLayout \*pLayout = new QFormLayout();

pLayout->addRow(QStringLiteral("用户名："), pUserLineEdit);

pLayout->addRow(QStringLiteral("密码："), pPasswordLineEdit);

pLayout->addRow(QStringLiteral("验证码："), pVerifyLineEdit);

pLayout->setSpacing(10);

pLayout->setMargin(10);

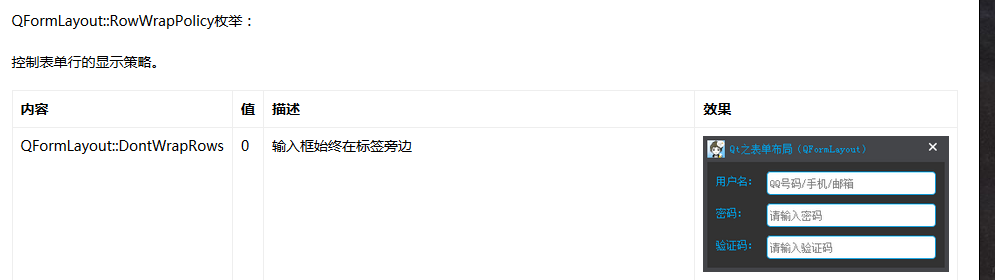
setLayout(pLayout);

和其他布局相比代码简介了很多。

* setSpacing(int spacing)
* setHorizontalSpacing(int spacing)
* setVerticalSpacing(int spacing)

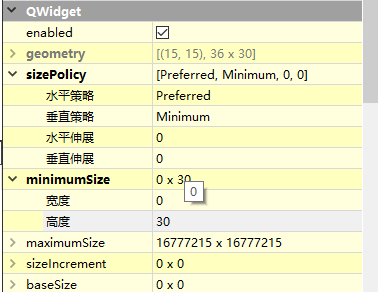
设置间距（水平间距、垂直间距）

setRowWrapPolicy(RowWrapPolicy policy)

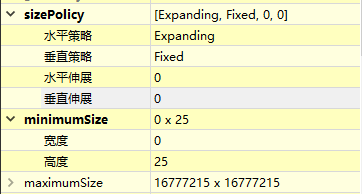
 

在Qt Designer中设置了Layout以后，对其中的元素进行大小的设置：

比较讨厌的是只能通过改政策与属性的方式，不能直接用鼠标图形化操作更改大小

这是Label的，垂直策略也可以是Fixed

如果策略更改不能成功就设置minimumSize属性中的值看看

这是TextLine的

Qt 在生成ui代码时，根据最外层的Layout自动添加一个了Widget。通过代码可以知道是QWidget \*verticalLayoutWidget;为了能让Designed创建的UI能自动适应窗口大小，mainLayout->addWidget(ui.verticalLayoutWidget);

将verticalLayoutWidget 放在目标窗体的Layout中。不过我直接给主窗口设置了Layout（右键->布局）

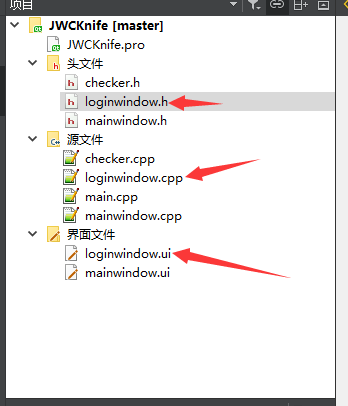
Qt Designer中改变样式还是右键-》样式表最简单，喜欢字体选择最下面的等线字体。

密码的黑点样式应该选用黑体，这样的黑点最好看。

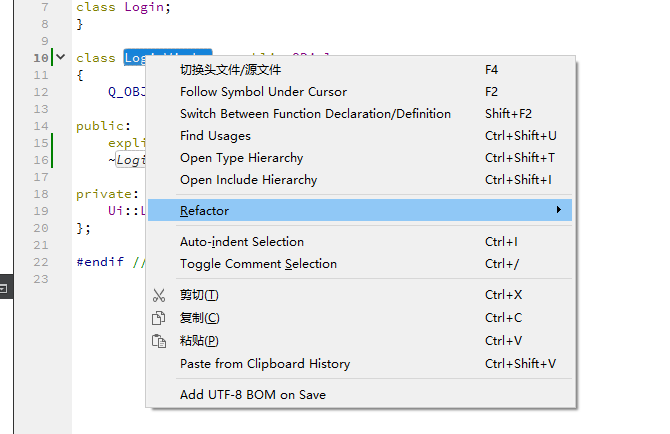
突然发现自己的界面类名字去错了不应该叫Login应该叫LoginWindow：

所以

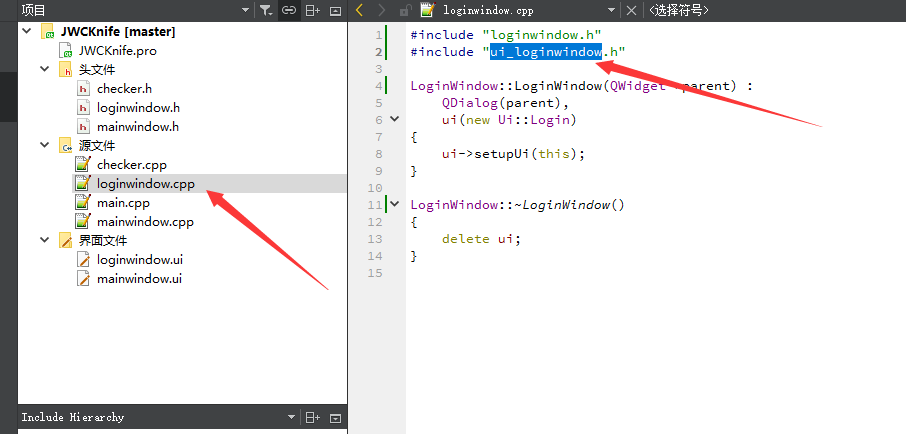
1. 改变项目中文件名字



1. 改变文件中类名：

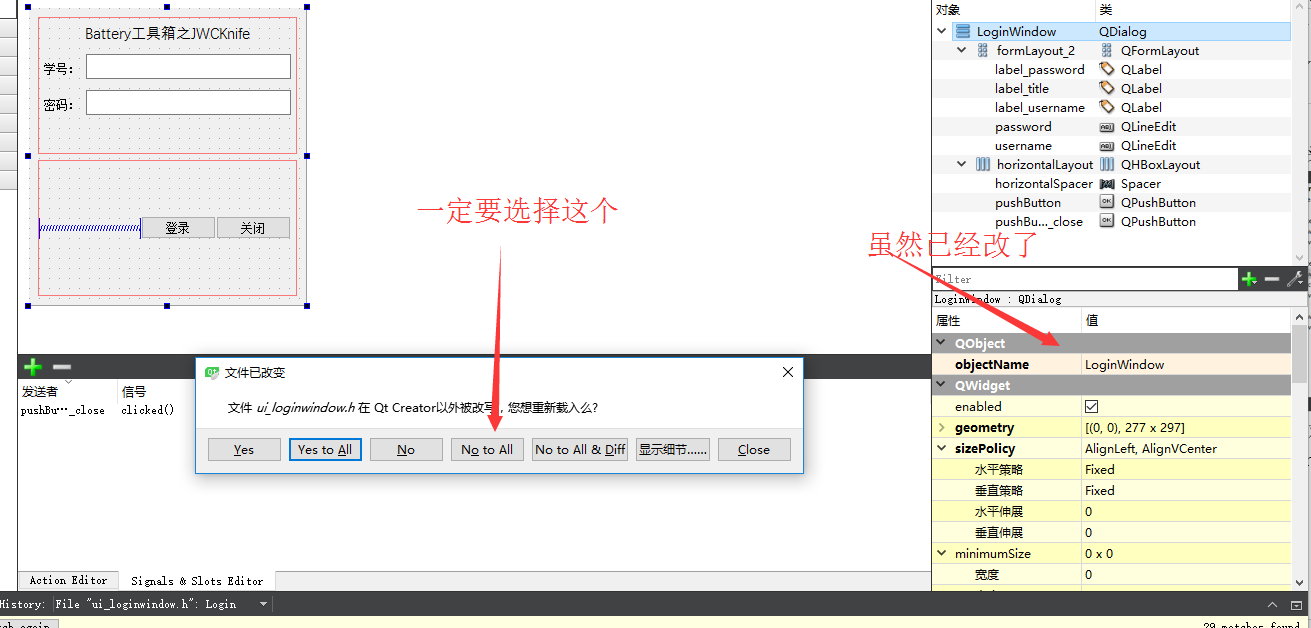


1. 改变头文件：



1. 最后对头文件右键Follow Symbol under Cursor去改变其中相关的类名代码以及loginwindow.cpp与loginwindow.h中的相关名字

遇到问题：



这时候真正的解决办法是重新构建项目

添加程序图标：

创建rc文件：IDI\_ICON1 ICON DISCARDABLE "xxx.ico"

在pro配置中引用文件：RC\_FILE += xx.rc

如果取消pro中文件的引用还需要取消设置

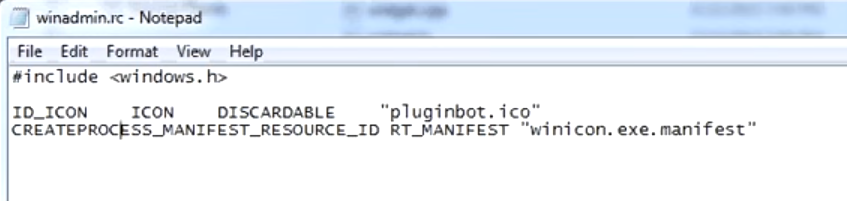
如果是设置窗口图标：

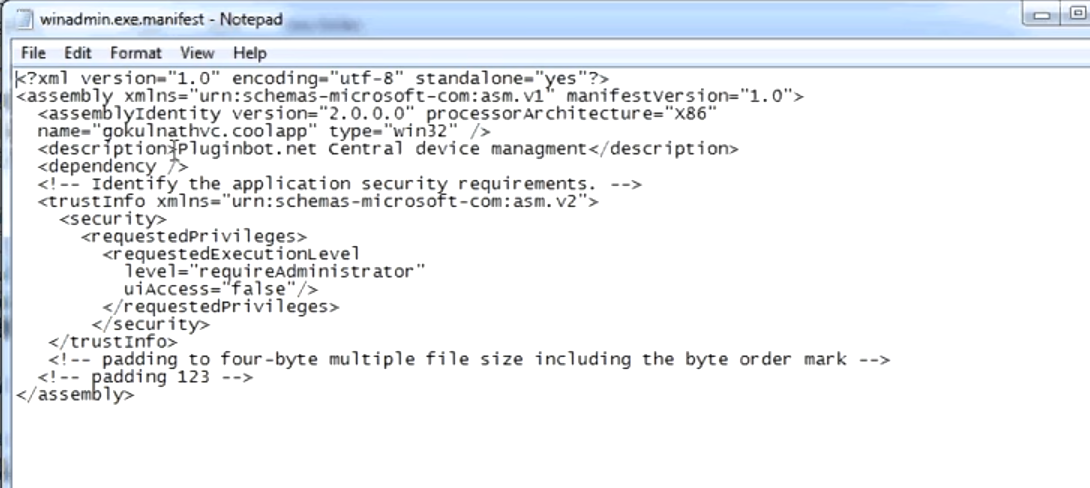
在Designer的WindowIcon属性中进行修改就好，里面可以直接选择文件。但是我第一次没有成功，后来

没想到Dialog中的“这是什么”居然是HTML



通过rc配置获取管理员权限：

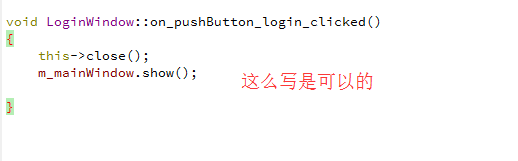




**QT 关闭主窗口时触发关闭所有打开的其他窗口：**

1.main函数里设置参数,最后一个窗口关闭时关闭程序. 默认的话不能保证立即关闭程序。

app.setQuitOnLastWindowClosed(true);



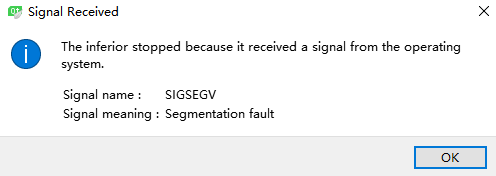
2. 在main函数中将QApplication::lastWindowClosed()信号和QApplication::quit()槽函数相关联

a.connect( &a, SIGNAL( lastWindowClosed() ), &a, SLOT( quit() ) );用close()和quit()的区别是，调用close()时，会产生一个QCloseEvent事件，而quit（）及exit()不会。这也就是我们单击X的时候产生的事件。

3.直接调用QCoreApplication::exit(int returnCode=0);

After this function has been called, the application leaves the main event loop and returns from the call to [exec](qcoreapplication.html" \l "exec)(). The [exec](qcoreapplication.html" \l "exec)() function returns *returnCode*.并不是返回给调用者。

这种方式如果你退出的时候说“程序异常结束”就说明程序的编写有问题。这个问题具体可以在调试里面看到问题的原因，比如说



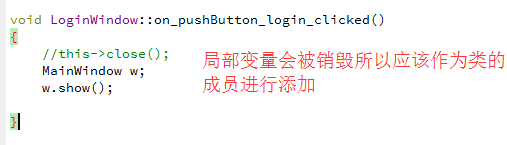
比较常见的问题就是对不该析构的地方进行析构（你进行了两次析构）。Qt建议对象树上的东西是new出来，因为对象树消除的时候会利用父对象中的链表来对所有子对象进行析构，如果是在栈上面，那就会存在被二次析构的时候。这时候就会触发程序异常结束。

**Qt窗口切换：**

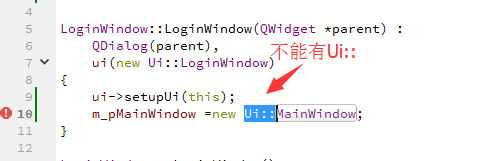
对被切换窗口的处理：要么销毁，要么隐藏，要么最小化

hide和show是最简单的方法但是不适合在还会回到原来窗口的地方，因为这就是两个窗口，窗口的大小改变与位置改变都不会进行同步。The widget is [hidden](qwidget.html" \l "hide) if it [accepts](../qtcore/qevent.html" \l "accept) the close event.

类似的切换，其实有个stackwidget（设置平移切换效果的时候直接对这个窗口对象设置就好）。

还一般是添加成指针成员

Designer基本什么都可以调整



Qt的整个网络编程和py中的urllib2库一样，设置cookiejar来记录cookie并在之后的request中都包含这个cookiejar. The [QNetworkCookieJar](qnetworkcookiejar.html) class implements a simple jar of [QNetworkCookie](qnetworkcookie.html) objects

*data* must be open for reading and must remain valid until the [finished](qnetworkaccessmanager.html" \l "finished)() signal is emitted for this reply.你必须要等信号才能读取！

QByteArray data;

data.append("pageSize=10000\

&pageNumber=0\

&actionType=1");

m\_pReply =m\_manager.post(m\_request, data);

/\*同步1.

QEventLoop loop;

connect(m\_pReply, SIGNAL(finished()), &loop, SLOT(quit()));

QTimer::singleShot(5000, &loop, SLOT(quit())); 超时退出

loop.exec();

qDebug() << m\_pReply->readAll();

同步2.

connect(m\_pReply,&QNetworkReply::readyRead,this,[=](){

QByteArray t =m\_pReply->readAll();

qDebug()<<"reply2 POST返回"<<t;

});

\*/

//异步 之所以默认是异步就是不卡顿界面。

connect(&m\_manager, SIGNAL(finished(QNetworkReply\*)), this, SLOT(replyFinished(QNetworkReply\*)));

请注意我给的connect几种回调函数的写法

注意异步发包后面调用的函数包可能更早发出去

原来可以获取到connect的第一个参数（发出信号的对象）：

connect(reply, SIGNAL(finished()), this, SLOT(onRequestCompleted())); } void MyApp::onRequestCompleted() { QNetworkReply \*reply = qobject\_cast<QNetworkReply \*>(sender()); QByteArray data = reply->readAll(); }

|  |
| --- |
| QNetworkReply\* reply = m\_pNetWorkManager->post(request, append); |

这句代码执行完之后，执行流程需要回到事件循环当中，post才会被执行。所以不能直接理解上面那条语句是发送函数。

lamdba中可以传参，但是前提就是在回调这个函数之前这个参数没有发生变化。否则只好connec传参的两种方式了（一种是总回调函数，可以通过获取QNetworkReply中的request然后再进行分开处理，一种是各自分开的回调函数）。



Qt要求槽函数的形参列表是信号函数的子集

QNetworkReply \*pTestReply =pTestManager->get(QNetworkRequest(QUrl("http://www.hellovega.cn")));

网址不能是<http://www.baidu.com>，也不能是没有协议的[www.baidu.com](http://www.baidu.com)。百度的协议已经是https了。

pTestReply->deleteLater();发送destroy信号给主事件循环

The object will be deleted when control returns to the event loop. If the event loop is not running when this function is called (e.g. deleteLater() is called on an object before [QCoreApplication::exec](qcoreapplication.html" \l "exec)()), the object will be deleted once the event loop is started. If deleteLater() is called after the main event loop has stopped, the object will not be deleted。It is safe to call this function more than once 一般是在SLOT函数中调用，一旦函数执行完以后，回到事件循环，被指定的对象就会删除。

QNetworkReply对象的删除是用户的责任

The [QNetworkReply](http://doc.qt.io/qt-5/qnetworkreply.html) class contains the data and headers for a request sent.setRequest()是个protected函数，这个函数会被自己调用，所以之后直接使用request()来获取发送出去的request就好。

while(1){

    QNetworkAccessManager \*aa = new QNetworkAccessManager();

    QNetworkReply \*bb;

    QNetworkRequest Request;

    Request.setUrl(QUrl("http://127.0.0.1"));

    bb= aa->get(Request);

    bb->abort();

    bb->close();

    bb->deleteLater();

    bb = NULL;

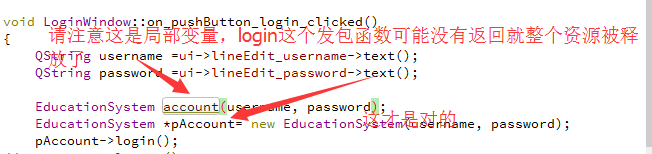
    delete aa;

    aa = NULL;

}这段代码会导致句柄泄露，如果是在main.c中，那解决方案是在while循环中的aa = NULL;后面加上一句a.processEvents()(a就是QApplication实例对象);处理所有事件即可解决泄露的问题。就是deleteLater没有被执行。

m\_pReply->readAll(); 读一遍以后缓冲区就没有内容了

所以qDebug() << m\_pReply->readAll() << m\_pReply->readAll().size();就会显示”” 503。



qt.network.ssl: QSslSocket: cannot call unresolved function xxxx

Qt的这个目录下：D:\Qt\Qt5.4.2\Tools\mingw491\_32\opt\bin。自带libeay32.dll和ssleay32.dll.

只需要把这个目录下的两个dll拷贝到D:\Qt\Qt5.4.2\5.4\mingw491\_32\bin，就不会提示qt.network.ssl: QSslSocket: cannot call unresolved function SSLv23\_client\_method。

undefined reference to `vtable for’

信号与槽之间可以进行通信的根本原因是：当在类的头文件中添加Q\_OBJECT以后QtCreator会自动的创建一个moc\_\*\*\*.cpp文件，用于实现信号与槽通信的代码。但是，有时当我们通过QtCreator创建类的时候，没有通过IDE选项选择其派生自QObject类，而是在后面添加的，则会出现QtCreator没有自动创建moc\_\*\*\*.cpp文件的情况。  解决方法：从QtCreator去除该类的头文件，然后再通过“添加现有文件”的方法，将该头文件添加进来。这样QtCreator就会自动为该类创建moc\_\*\*\*.cpp文件。 我使用了git，所以还会有个弹框问你是否git删除本地，要点击否，否则本地文件就会被删除。不知道是不是因为项目变大的原因之后每一次弹框都会导致qt Creator直接失去响应。这时候如果程序开着的话虽然界面已经关了，但是进程依然存在，需要从任务管理器那里进行关闭。

content-type missing in HTTP POST, defaulting to application/x-www-form-urlencoded. Use QNetworkRequest::setHeader() to fix this problem.

这个不是错误就是在你没有设置Content-Type头部的时候使用post就会有这个，可以不理他，不然你get的时候就会带上这个头部，虽然没有什么影响。加不加看自己咯。request.setHeader(QNetworkRequest::ContentTypeHeader, "application/x-www-form-urlencoded"); 设置为空字符串就是将这个头部取消。

QNetworkReply \*pReply =get(m\_request); 虽然可以获取地址，但是地址上内容的填充时间却是不定的，所以如果不是1.使用loop阻塞来做到同步，然后就可以在发送函数中直接使用QNetworkReply，2.使用lambda函数，3.connect使用它来指定，那就应该直接get(m\_request);

QNetworkReply \*pReply = post(m\_request, data);

connect(pReply,&QNetworkReply::finished,this,[=](){

qDebug() << gbk->toUnicode(pReply->readAll());

pReply->deleteLater();

});

当QNetworkReply到来的时候下面两个connect都会触发：

connect(pReply, SIGNAL(finished()), &loop, SLOT(quit()));

connect(this, SIGNAL(finished(QNetworkReply\*)), this, SLOT(replyComplished(QNetworkReply\*)), Qt::UniqueConnection);

qt pro里面可以使用win32{xxxx} 来进行平台代码指示

在基类A中继承了QObject并且类中有Q\_OBJECT宏, 但是在派生类B中仅仅继承了A, 这个时候, 在B中是不能使用signal/slot的. 还要进行Q\_OBJECT指定。

qt借助消息来实现异步网络活动能快很多，但是这就有了问题。QNetworkAccessManager只有一个，但是异步实现中要对结果进行判断处理，异步使得结果只能在回调函数中进行处理，所以肯定需要进行connect函数中信号与槽的传参，而这就只能使用connect(&m\_manager,SIGNAL(finished(QNetworkReply\*)), this,SLOT(replyFinished(QNetworkReply\*)));这种形式，于是如果整个发包处理的过程在执行了这个函数以后都会调用一边replyFinished函数。

异步发包爬虫还要进行可能的错误处理，不是很简单的爬虫目标，那就不能使用，因为异步意味着IO高效但乱序。根本没发把回来的包与发出去的包进行对应，不过如果网站中有什么标号比如说这一页面是第几页，这就可以使用异步进行爬取，也就能进行错误处理。但是Qt提供了有序机制：void QNetworkReply::setRequest(const [QNetworkRequest](qnetworkrequest.html) &request)和[Q](qnetworkrequest.html)NetworkRequest QNetworkReply::request() const 使得reply和相应的requests可以绑定在一起。

同步发包效率低但是是顺序的，看到返回一出错可以马上纠正。效率的提高的话只能是多线程与线程爬虫之间的范围分工。

GBK编码转换：m\_pReply->readAll()是QByteArray被隐式转换为QString

QTextCodec \*gbk = QTextCodec::codecForName("GB18030");

qDebug() << gbk->toUnicode(m\_pReply->readAll());

error: incomplete type 'QTextCodec' used in nested name specifier

使用的时候虽然有自动加亮，但是如果不包含头文件还是会出错的！

QString str =gbk->toUnicode(pReply->readAll());这样子是有效的，但是如果你先将pReply->readAll()转化为QString，那么最后得到的内容中文还是乱码。

GBK编码转URL编码：

QString **gbkURLCoding**(QString str)

{

QByteArray data =gbk->fromUnicode(str);

data =data.toPercentEncoding();

return data;

}

转换不完全版，自己安装原理写的，只能算是中文转URL编码，不能出现英文：

QString **gbkURLCoding**(QString str)

{

char hex[17]="0123456789ABCDEF";

QString result ="";

QByteArray data =gbk->fromUnicode(str);

for(auto c:data ) {

if( isprint(c) ) {

if(c==' ') result.append("+");

else result.append(c);

}else {

result.append("%");result.append(hex[(unsigned char)c/16]);result.append(hex[(unsigned char)c%16]);

}

}

return result;

}

英文字母：

字节数 : 1;编码：GB2312

字节数 : 1;编码：GBK

字节数 : 1;编码：GB18030

字节数 : 1;编码：ISO-8859-1

字节数 : 1;编码：UTF-8

字节数 : 4;编码：UTF-16

字节数 : 2;编码：UTF-16BE

字节数 : 2;编码：UTF-16LE

中文汉字：

字节数 : 2;编码：GB2312

字节数 : 2;编码：GBK

字节数 : 2;编码：GB18030

字节数 : 1;编码：ISO-8859-1

字节数 : 3;编码：UTF-8

字节数 : 4;编码：UTF-16

字节数 : 2;编码：UTF-16BE

字节数 : 2;编码：UTF-16LE

QByteArray data;data.append(gbk->fromUnicode("一个好老师。"));

qDebug() << data; "\xD2\xBB\xB8\xF6\xBA\xC3\xC0\xCF\xCA\xA6\xA1\xA3"

qDebug() << data.toHex(); "d2bbb8f6bac3c0cfcaa6a1a3"

QByteArray data;data.append("一个好老师。");

qDebug() << data;"\xE4\xB8\x80\xE4\xB8\xAA\xE5\xA5\xBD\xE8\x80\x81\xE5\xB8\x88\xE3\x80\x82" 从表示中文的字节数看出来默认是utf8，经过fromUnicode可以转换成为gbk。

EducationSystem继承自QNetworkAccessManager类

void EducationSystem::getClassSchedule(void) {

qDebug() << "enter getClassSchedule";

m\_request.setUrl(QUrl(m\_BaseUrl+"/xkAction.do?actionType=6"));

get(m\_request);

}这个函数就是发送不出去包，不过如果在前加上对其他路径的发包这个函数就会成功发送，wireshark捕捉到了发送包，却没有响应包。如果前后都加上，那么请求与响应包都有。很崩溃，无法解释这一现象….

如果加上下面的一个独立的事件循环（如果是主线程内，这样界面就可以继续响应用户的操作），那么可以正常运行：

// QEventLoop loop;

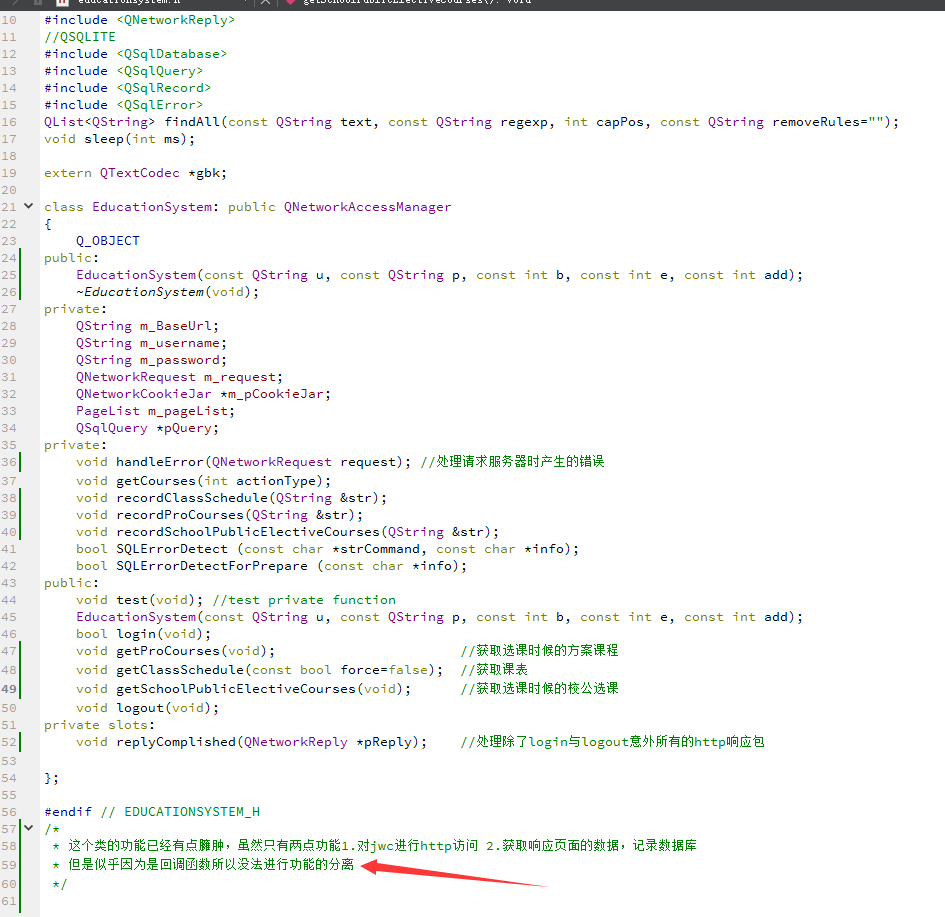
// connect(pReply, SIGNAL(finished()), &loop, SLOT(quit()));

// loop.exec();

// qDebug() << pReply->readAll();

更加无奈的是我在函数里面使用示例程序也是运行不起来的，检查了可能相关的东西，根本找不到错误。估摸着和事件循环有关系。

原来发送包里面还有个优先级的设定。



**等待提示框：**

程序退出：

exit(0); 这个函数任何时候都可以调用,直接结束进程

QApplication::setQuitOnLastWindowClosed(true); 在最后一个窗口关闭的时候退出

**QSet:**

The [QSet](qset.html) class is a template class that provides a hash-table-based set

**正则：QRegExp**

QSet<QString> EducationSystem::findAll(const QString text, const QString regexp) {

QSet<QString> resultSet;

QRegExp rx(regexp);

int pos = 0;

while ((pos = rx.indexIn(text, pos)) != -1){

pos += rx.matchedLength();

QString result = rx.cap(0);

resultSet << result; //由于是QSet所以不会存在重复内容

}

return resultSet;

}

//作用：匹配返回正则表达式中第capPos个(...)，同时第四个惨呼指定了过滤

QList<QString> findAll(const QString text, const QString regexp, int capPos, const QString removeRules) {

QList<QString> resultSet;

QRegExp rx(regexp);rx.setMinimal(true);

QRegExp rxRemove(removeRules);rxRemove.setMinimal(true);

int pos = 0;

while ((pos = rx.indexIn(text, pos)) != -1){

pos += rx.matchedLength();

QString result =(removeRules=="")? rx.cap(capPos) : rx.cap(capPos).remove(rxRemove);

// qDebug() << result; //这里是debug时进行的输出

resultSet << result;

}

return resultSet;

}

如果传入findAll的capPos为0，那么提取的时候就会是一整个。

提取被(xxx)匹配的部分：

方法一、

QString s = "a=100";

s.replace(QRegExp("(.\*)=(.\*)"), "\\1\\2=\\2");  // \1 is rx.cap(1), \2 is rx.cap(2)

方法二、

**[QRegExp](qregexp.html" \l "QRegExp) rx("(\\d+)(\\s\*)(cm|inch(es)?)");**

**int pos = rx.indexIn("Length: 36 inches");**

**[QStringList](qstringlist.html) list = rx.capturedTexts();**

**// list is now ("36 inches", "36", " ", "inches", "es") [0]是完全匹配的，其余的就是剩下(xxx)匹配的**

**rx.cap(0) 就是[0]**

**(1)就是[1]…..**

**方法三、这才是对的**

**[QRegExp](qregexp.html" \l "QRegExp) rxlen("(\\d+)(?:\\s\*)(cm|inch)");**

**int pos = rxlen.indexIn("Length: 189cm");**

**if (pos > -1) {**

**[QString](qstring.html) value = rxlen.cap(1); // "189"**

**[QString](qstring.html) unit = rxlen.cap(2); // "cm"**

**// ...**

**}**

在QRegExp中，存在'?'，为不合法的正则表达式，在QRegExp中，要使用setMinimal函数来指定贪婪算法。setMinimal函数的意思是设置最小匹配，也就是非贪婪。也就是进行最小匹配。以下情况如果指定了最小匹配会返回false，贪婪下返回true，是因为没有指定结尾，所以只会匹配一次。

QRegExp rx("(\\s)+");

qDebug() << rx.exactMatch("\r\n\t\t\t\t\t\t\t");

而下面这种情况为true

QRegExp rx("(\\s)+123");rx.setMinimal(true);

qDebug() << rx.exactMatch("\r\n\t\t\t\t\t\t\t123");

qt的正则匹配的exactMatch也是从开头开始匹配：怎么这么智障！不如使用字符串查找

QRegExp rx("500 Servlet Exception");

qDebug() << rx.exactMatch("<h1>500 Servlet Exception</h1>");

exactMatch("R") is the equivalent of [indexIn](qregexp.html" \l "indexIn)("^R$")

应该使用：

rx.indexIn("<h1>5001 Servlet Exception</h1>");

if(rx.cap(0)=="")

qDebug() << "No matches";

else

qDebug() << "have matched";

获取(.\*)代表的内容：

("<td.\*>(.\*)</td>") cap的时候pos是2

"<td.\*>(.\*)</td>" cap的时候pos是1

connect(m\_pReply, SIGNAL(finished(QNetworkReply\*)), this, SLOT(recordProCourses(QNetworkReply\*))); 前面用了SIGNAL后面就要是用SLOT。

QString text("&nbsp;\n\t\t\n\t\t\n\t\t\t计算机科学与技术培养方案</td>");

text.remove(QRegExp("[&nbsp;\n\t\r]")); 而不是[\\n](file:///\\\\n)这种，这种转义只会有一次。匹配一个数字则是[\\d](file:///\\\\d)，这里的转义也是因为C字符串。

[abc] 只是一个简单的字符类，匹配其中任意一个字符，或区间内字符  
(a|b|c) 是分组，会返回匹配到的项  
例如：(aba|abc|aca|acb) 等同 a(b[ac]|c[ab]) 符号的优先级就体现出来了

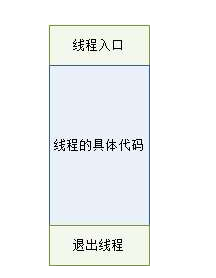
^([1-9]\d|\d)$  匹配0-99数字

|  |  |
| --- | --- |
| . | 匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。 |

如果要匹配所有字符需要[\s\S]\*？但是在qt里面.就是匹配任意一个字符。

**QT 多线程：QThread**

QThread与通常所熟知的线程（thread）有很大出入，在面向过程的语言中，我们建立一个线程的同时会传入一个函数名，这个函数名代表该线程要执行的具体代码 。但是QThread里并没有线程的具体代码，QThread只是一个接口而已，本身并不是线程，目的是为操作系统提供一个用于线程调度的“句柄”。 因为qthread类就是一个qobject，它是在主线程创建的，自然它的那些非run()函数以及各种数据成员都属于主线程。这个“句柄”即是QThread的入口，出口就是finished信号。



作为线程，QThread会毫不犹豫的为自己创建一个运行空间（QThread也应该是类的成员），一个单独的执行线索（每一个线程都有自己的事件循环），一个新的线程，但是翻阅QThread所拥有的资源，我们找不到传入函数名的地方，因此我们仿佛无法为这个新创建的线程提供具体的执行代码。

很多人因此想到了run()方法，因而继承QThread函数，好处就是看起来简单。

正确的姿势是使用moveToThread()：将需要在新线程中运行的对象moveTo到QThread中，而非继承QThread并把自身moveTo到新线程空间中（成员执行与事件循环），这个对象也就由这个线程进行释放。QThread只是系统执行线程的接口而已，并不是用于编写代码的。

serial为目标对象，可以使用线程的started信号可，也可以自建一个信号。

connect(&serialThread, SIGNAL(started()), serial, SLOT(threadStarted()), Qt::QueuedConnection); // 注意，serialThread 与 serial 并不在同一个线程中，因此使用 Qt::QueuedConnection。 而移动以后：

connect(&serialThread, SIGNAL(finished()), serial, SLOT(threadFinished()), Qt::DirectConnection); // serialThread 的 finished() 信号是在新线程中执行的，因此此处要使用 Qt::DirectConnection。threadFinished()比如说在这你需要关闭串口。

serialThread.start(QThread::HighestPriority); // 开启线程，串口接收线程的优先级较高。

serial->moveToThread(&serialThread); // 将串口接受对象移动到新线程中。一定要先开启线程与建立对象以后再移动进去。移动进去的对象需要手动释放QObject::connect(&thread, &QThread::finished, object, &QObject::deleteLater);

serial->port->moveToThread(&serialThread); // 用于接收的 port 一并移入新线程中。

UI操作只能在主线程里面做。

如果主线程结束的时候QThread还在运行就会弹出Windows Runtime Error错误框。

QThread\* workerThread = new QThread(this);

connect(camwrk, SIGNAL(destroyed()), workerThread, SLOT(quit()));

connect(this, SIGNAL(destroyed()), camwrk, SLOT(deleteLater()));

connect(workerThread, SIGNAL(finished()),

workerThread, SLOT(deleteLater()));

<http://wiki.qt.io/QThreads_general_usage>

1. moveToThread方法，是把我们需要的工作全部封装在一个类中，将每个任务定义为一个的槽函数，再建立触发这些槽的信号，然后把信号和槽连接起来，最后将这个类调用moveToThread方法交给一个QThread对象，再调用QThread的start()函数使其全权处理事件循环。于是，任何时候我们需要让线程执行某个任务，只需要发出对应的信号就可以。其优点是我们可以在一个worker类中定义很多个需要做的工作，然后发出触发的信号线程就可以执行。相比于子类化的QThread只能执行run()函数中的任务，moveToThread的方法中一个线程可以做很多不同的工作（只要发出任务的对应的信号即可）。 QObject :: moveToThread（）函数更改对象及其子对象的线程关联（如果对象具有父对象，则不能移动该对象），应该只是改动了Thread()函数返回的成员值。
2. 子类化QThread的方法，就是重写了QThread中的run()函数，在run()函数中定义了需要的工作。这样的结果是，我们自定义的子线程调用start()函数后，便开始执行run()函数。如果在自定义的线程类中定义相关槽函数，那么这些槽函数不会由子类化的QThread自身事件循环所执行，而是由该子线程的拥有者所在线程（一般都是主线程）来执行。如果你不明白的话，请看，第二个例子中，子类化的线程的槽函数中输出当前线程的ID，而这个ID居然是主线程的ID！！事实的确是如此，子类化的QThread只能执行run()函数中的任务直到run()函数退出，而它的槽函数根本不会被自己的线程执行。

使用注意事项：

1.使用moveToThread前提是QObject对象的parent必须是0，Qt认为同一颗对象树上的所有QObject对象都必须属于同一个线程。

2.线程内创建的变量要在delete 线程之前就delete掉。如果在另一个线程中删除了存在本线程的对象，如果你能够保证这个对象之后不会产生事件，到不会出现错误，但很明显这是一个不安全的行为。推荐使用QObject:deleteLater() 在稍后触发一个删除对象的信号，这个信号会进入事件循环，所以就是在存在的线程中删除了对象。

3.事件驱动的对象只能在一个线程中使用（如果被调用函数中存在new操作，那么只能通过跨线程的信号进行调用，比如发包的get、post；一般操作可以直接通过对象指针来调用成员函数），具体来说就是定时器QTimer、网络模块部分。例如，不能在另一个线程中启动定时器，或者连接套接字。QObject::thread()指向这个QOject对象存在的线程。

4.QObject及其子类的实现并不是线程安全的，甚至有时候需要自己加上互斥锁。加锁比信号与槽更加影响性能，因为是阻塞。

每一个线程都有自己的事件循环，主线程（UI线程）的事件循环是在QCoreApplication::exec()，或者对于单独的Dialog应用来说就是QDialog::exec() 开启。子线程的事件循环是在QThread::exec()开启。qApp->thread() 返回主线程指针。

QCoreApplication::sendEvent()只能将信号发送给函数调用的线程中的QObjetc，QCoreApplication::postEvent() 可以将信号发送给任意一个线程中的任意一个QObject对象。

线程提供了exit(int)方法以及quit() SLOT。

QThread::currentThreadId() 静态函数可以显示当前调用这个函数的线程的ID。

**多线程爬虫：**  
 每一个线程都分担了一定的爬取任务，比如在获取某高校某学期所有公开课信息的时候按照页数进行划分。如果是借助QNetworkAccessManager实现，那么moveToThread就需要将相关子类移到新线程中，因为HTTP响应的接收是靠一个事件循环，要想把接收函数放到新线程中处理就要将这个对象移动到该线程中。

所以对于“获取某高校某学期所有公开课信息”我就按照页数进行了划分，从0到200页。一般页数只会有150多页，所以如果在插入数据库的时候发现存在一样的记录就说明已经到了最后一页了，这时候直接删除对象，退出线程即可。

**QT进行耗时操作时需要掌握的方法：**

方法一：  
使用QThread，把逻辑功能打包到子线程。不要继承重写QThread::run，而是把逻辑功能封装为一个QObject，不要在类里写任何死循环，而是用基于信号槽的事件触发思想进行。  
这个类里，const读函数允许直接调用，而会发生修改的比如写函数和控制函数，全部做成信号槽——这样的好处是你不需要做任何线程锁，Qt的信号槽会帮你处理好数据同步操作。但代价是信号槽操作为异步操作

方法二：  
只跑一次的操作，比如异步操作一个函数，继承重写QRunnable，然后放进Qt的全局线程池里

方法三：如果这个操作是复杂的运算  
方法二比较麻烦，还要继承类。更方便的是用QtConcurrent模块（并行计算模块），传一个函数指针和对应参数进去，就可以自动完成异步执行

**Qt 串口通信：**

port = new QSerialPort();

port->setPortName("COM1");

if(!port->open(QSerialPort::ReadWrite))

{

QMessageBox WrrMsg;

WrrMsg.setInformativeText("无法打开该串口");

WrrMsg.show();

WrrMsg.exec();

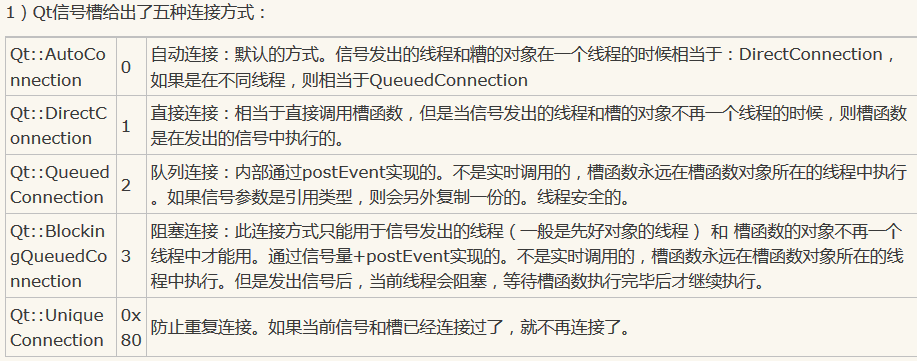
}

QNetworkAccessManager(QObject \*parent = Q\_NULLPTR) 好像parent 不同，事件循环就可能不同，如果是Q\_NULLPTR，那么似乎就是默认的事件循环。

在Qt中，每个 QObject 内部都有一个list，用来保存所有的 children，还有一个指针，保存自己的parent。当它自己析构时，它会将自己从parent的列表中删除，并且析构掉所有的children。

connect函数同样的参数是可以被多次调用的，暂时没有遇到任何问题？但其实是进行了重复连接，应该connect(this, SIGNAL(finished(QNetworkReply\*)), this, SLOT(replyComplished(QNetworkReply\*)), Qt::UniqueConnection);

解除绑定关系有  
bool QObject::disconnect ( const QObject \* sender, const char \* signal, const QObject \* receiver, const char \* method ) [static]

在对象被删除时,Qt会自动移除与这个对象相关的所有连接

connect(this, SIGNAL(finished(QNetworkReply\*)), this, SLOT(replyComplished(QNetworkReply\*)), Qt::UniqueConnection); this这个类对象指针继承自QNetworkAccessManager

之后：

QNetworkReply \*pReply = post(m\_request, data);

disconnect(pReply,SIGNAL(finished(QNetworkReply\*)), this, SLOT(replyComplished(QNetworkReply\*))); 很自然connect是一个总的，disconnect只能取消connect，但是你指定的是一个分关系。所以无效。联想iptables。

**QDebug：**

#include<QtDebug>的内容就是

#include "qdebug.h"

qDebug() << "获得到来自"+pReply->request().url().toEncoded()+"的正常的请求";的输出确实"\xE8\x8E\xB7\xE5\xBE\x97\xE5\x88\xB0\xE6\x9D\xA5\xE8\x87\xAAhttp://202.115.47.141/loginAction.do\xE7\x9A\x84\xE6\xAD\xA3\xE5\xB8\xB8\xE7\x9A\x84\xE8\xAF\xB7\xE6\xB1\x82"，这是因为全都被转成QByteArray类型、把toEncoded()变成toQString就好，不过QByteArray方便使用索引。QString可以使用const **QChar** at(int position) const，0 <= *position* < [size](qstring.html" \l "size)()。QString的索引返回的是**QCharRef**。

突然发现在调试模式下中文显示到qDebug()失败了

在ui中直接图形化设置的信号与槽的机制在我们的类代码里面不会有对应函数。

QChar：

不能把QChar类型放到switch的括号里面，会说switch quantity not an integer。使用digitValue()成员函数。char倒是可以自动转型为int，所以可以放到swtic中。

close操作过窗口以后，由于关闭所有窗口造成进程退出，进程返回值是255，好像有尽量不要使用这个返回值。没操作过窗口那返回值就是0。

QMessageBox::critical(nullptr, "错误", "数据表创建失败!请QQ联系开发人员", QMessageBox::Yes);在弹框的时候触发进程就卡在这里，知道用户对弹框进行了操作才会继续。

QT对象树： 父对象不是继承上的父类

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。

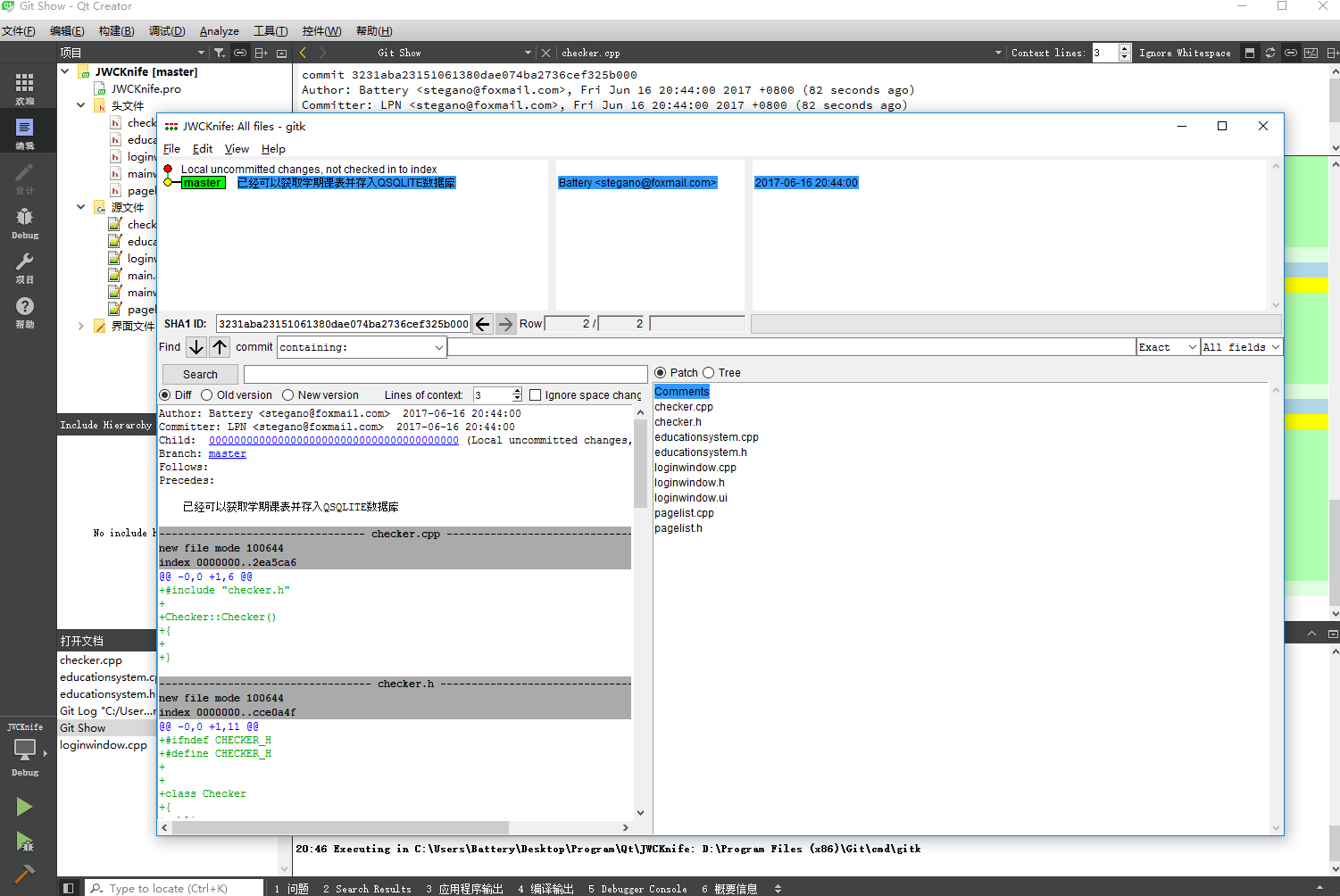
所以被建议对象树上的东西就应该new出来，而不是将内存分配到栈上。

所以窗体类型、QAction、QSystemTrayIcon都是用指针，我也就不在名字上加上p了。

QList<QString> 可以使用QStringList来代替

**Git：**

http://blog.csdn.net/hebbely/article/details/52495073



“检出分支” 的意思估计就是切换分支

Cannot find file: 进程"C:\Qt\4.8.5\bin\qmake.exe"退出，退出代码 2

这是因为中文路径的问题。

**Qt 界面美化：**

在QQ的安装目录下找到.RDB文件，使用工具进行解包即可。可以得到qq的图片图标资源。登录界面的动图是gif，一开始点开这个位置是全蓝，加载完以后才会进行显示动图。

？RDB是QQ的格式，有专门的针对QQ资源的解压工具——UniversalPackageBuilder，这个文件还会自动找到QQ的目录，直接批量解压所有工具。我没有找到登录界面的动图，但是发现解包以后那些老头像都是gif格式。 

除了使用无边框、好看的背景（渐变的纯色，可能会像qq一样有动态图）与图标、良好的QStackWidget切换方式。

Qt界面制作：

1. 这年头想要酷炫, 还是用web最方便, QT自带嵌入式webkit, 然后用d3.js [D3.js - Data-Driven Documents](http://www.d3js.org/" \t "blank)什么的, 各种酷炫的互动表随手而来.  
这里有各种各样酷炫的例子:[Gallery · mbostock/d3 Wiki · GitHub](https://github.com/mbostock/d3/wiki/Gallery" \t "blank)

2. QSS

3. QT Quick

http://www.qtcn.org/bbs/read-htm-tid-59235.html  （模仿微信）

4. C++使用图片资源，本身就已经可以做很酷很漂亮的界面：

http://www.qtcn.org/bbs/read-htm-tid-59272.html （通用界面）

http://bbs.csdn.net/topics/390508293?page=2 （360）

http://www.cnblogs.com/appsucc/archive/2012/03/14/2395657.html （另外一个360）

http://www.qtcn.org/bbs/read-htm-tid-48401.html （类似上面那个）

http://blog.csdn.net/gzshun/article/details/7592865 （研究360）

https://github.com/jun-zhang/Qt360  （Qt4模拟360 10.0界面）

http://www.cnblogs.com/appsucc/archive/2012/03/06/2382316.html （QT动画）

http://blog.sina.com.cn/s/blog\_a6fb6cc90101iol4.html （手机助手）

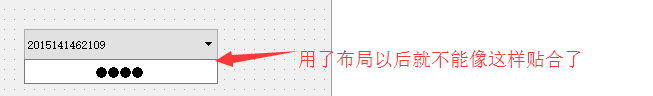
http://www.qtcn.org/bbs/read-htm-tid-58798.html （9宫格）

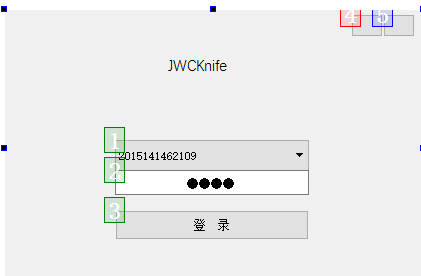
ico文件 32x32

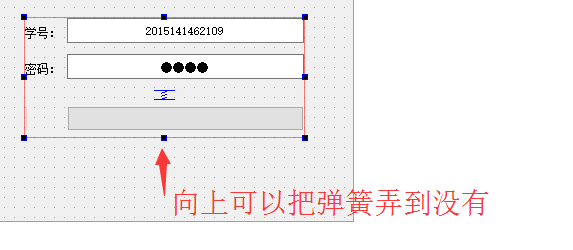
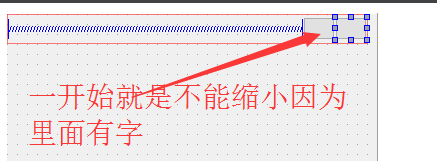
像360那样的上方工具图标 55x55



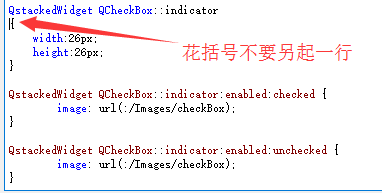
按钮按下以后，按钮上的内容向右下角偏移一点。



虽然Tab的设定是但是一进去焦点在“登录”上。



之前对窗口的显示是先显示gif或者合适的png来美化界面，不行再考虑网页显示可以考虑QWebView。还有Qt已经内置了webkit这也是一种实现。



**Qt 执行其他程序：**

//1.通过调用系统函数

system("opt/myApp"); //这是linux下的，windows下可以用WinExec等函数

//2.通过QProcess,阻塞调用

QProcess::execute("/opt/myApp"); //绝对路径和相对路径都可以，windows下要带exe后缀，系统应用如记事本notepad可直接输入打开

//3.通过QProcess,非阻塞调用

QProcess \*pro = new QProcess;

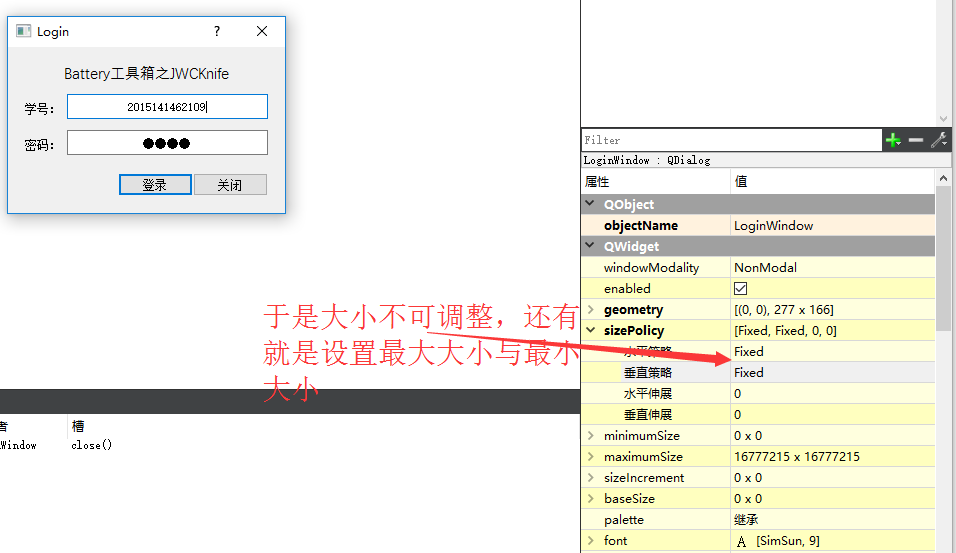
pro->start("/opt/myApp");

1. 前两种方法会阻塞进程，直到myApp程序结束，而第三种方法则不会阻塞进程，可以多任务运行。

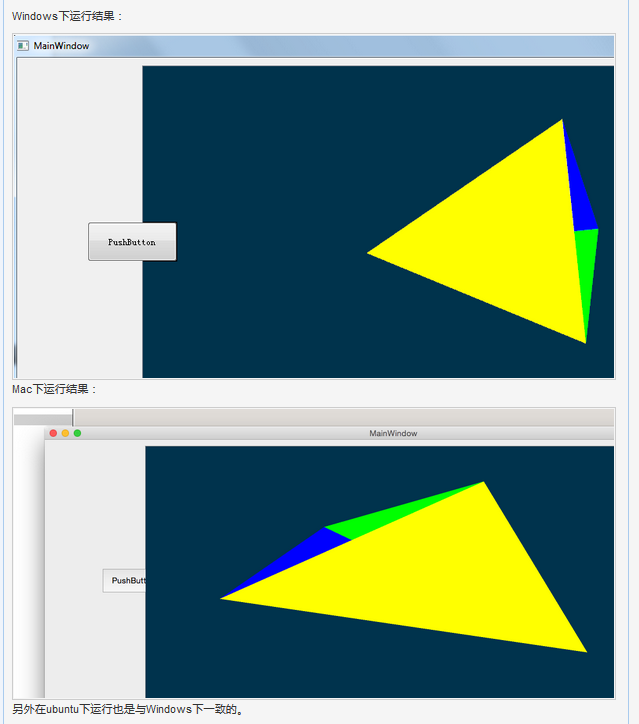
2. QT在运行的时候，要启动qws服务，如果用前面两种方法，运行的时候，要新开启一个qws，否则不能运行；而用第三种方法，则不需要再开启qws，它和主进程共用一个qws。

3. 第三种方法虽然不会阻塞，但是有可能在终端上看不到打印出来的信息。所以要在终端显示信息时，可以考虑用阻塞模式。

**设置固定窗口大小：**

1. 
2. w.setMaximumSize(); w.setMinimun

qt也有平台差异性：



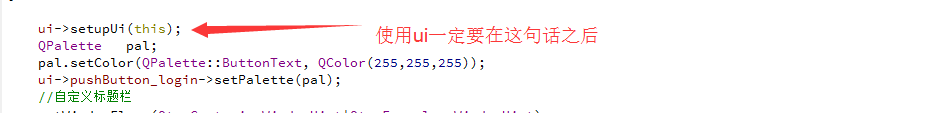
**Qt播放gif：**

使用QMovie来播放gif，如果你想要只播放一次的话，有一个简单办法，用frameCount获取到帧数，然后连接frameChanged信号，播放完一遍，调用stop即可。

**自定义标题栏：**

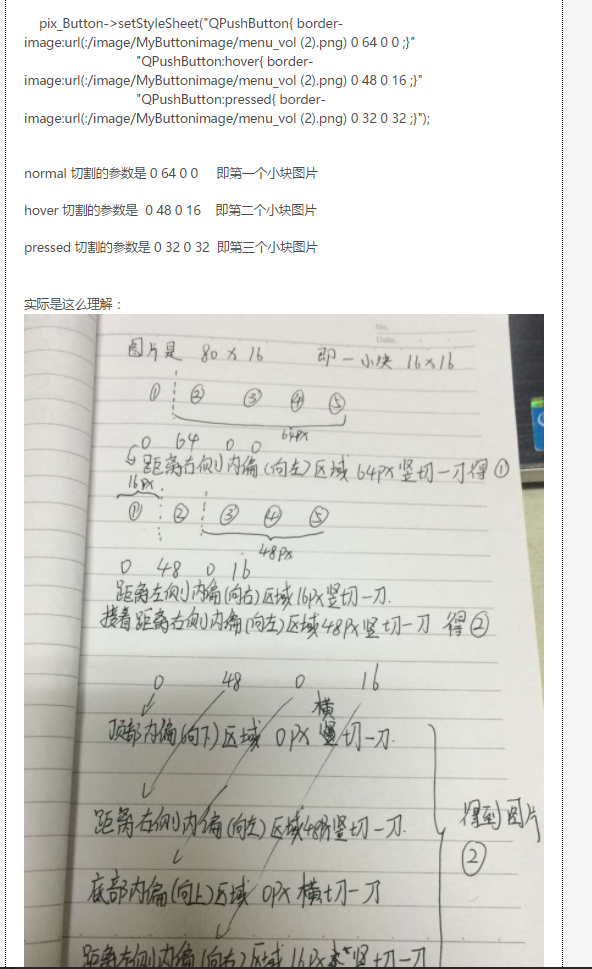
标题栏是由操作系统提供的，而操作系统没有提供任何方便修改标题栏的 API。首先想到的是，我们隐藏掉系统标题栏，然后绘制自己的进行替代。但是这么做还有一些额外的工作需要完成：提供窗口最大化、最小化、位置移动等通用操作（必要的话）

setWindowFlags(Qt::FramelessWindowHint|Qt::WindowStaysOnTopHint); 去除边框以及始终置顶



设置背景图片：

1. QSS中使用border-image: url(:/images/desktop.jpg); 属性



左边是使用border-image，右边是使用background-image。都是png文件。

1. 众所周知，使用QLabel的setPixmap（）就可以将图片显示出来，做视屏解码后显示也可以如此。但是为何我今天还要费力使用基函数drawPixmap（）来做绘图？理由有这么些吧：

一、使用QLabel很耗费CPU，播放画面过多有卡卡的感觉；

二、drawPixmap（）使用GPU处理，相对减轻了CPU的负担；

QImage image;image.load("login\_title.png");

QPainter painter(this);

painter.drawImage(0,0,image);

image.save("login\_title2.png");

或者

QPixmap pix;pix.load("login\_title.png");//路径

QPixmap pixTitleTriangle(":/Resources/MainWindow/title\_triangle.png"); //使用引入的资源

QPainter painter(this);//这样就不需要painter.begin(this); 和 painter.end();缶则就就会QPainter::begin: Painter already active

painter.drawPixmap(0,0,417,100,pix);是把图片左上角放到这个位置

注意：load是按照路径来读取文件的，所以图片应该放到build-JWCKnife-Desktop\_Qt\_5\_8\_0\_MinGW\_xxxxxx下。当然可以不使用load，直接传入文件路径给构造函数。如果路径不正确，那么不会进行报错，所以这里一定要进行检查。

注意：画图这类事情要放到paintevent函数里面去做，否则报错：

QWidget::paintEngine: Should no longer be called

QPainter::begin: Paint device returned engine == 0, type: 1然后就会画图失败，但是我遇到了一次很蛋疼的问题就是重载paintevent函数的时候写错了参数，这样子并不是重载，正确的应该是：void *paintEvent*(QPaintEvent \*);

不过画图的时候可以使用QSS：这也是我推荐的方法

#LoginWindow {

image: url(:/Resources/LoginWindow/login\_title.png);

image-position:TOP;

}

QSS中绘制了：

QWidget#title\_widget {

background-color:rgb(0,136,225);

}之后每一次paintEvent(QPaintEvent \*)都会自行进行绘制，而且绘制的顺序是在这个函数之后，所以如果使用这个函数在QSS已经绘制的区域里面进行绘制，就会被覆盖而不显示。

在Qt中，如果使用某些格式png图片，可能会报错libpng warning: iCCP: known incorrect sRGB profile这段话意思就是这个图片不是标准的色彩结构，很多ps修过的图就是这样。而解决方法其实非常简单，就是用QImage读取一下，然后直接保存回去就可以了。

不建议下面这种，还是在PS中P好图以后直接用这个图：

QPainter painter;   
    painter.begin(&image);   
    //painter.drawEllipse(10,20, 80,70); // draw an ellipse   
    painter.drawText(10,20,"南京\*\*\*\*\*\*\*有限公司 从左拍车头");   
    painter.drawText(10,40,"时间:2011-01-12 11:24:53.542");   
    painter.drawText(10,60,"速度:87Km/h 限速:60Km/h 超速比:26%");   
    painter.end();   
    image.save("kkkk.jpg","JPG");       // save picture\*/

**移动无边框窗口： 窗口拖动**

移动无边框窗体的代码网上很多，其原理都是一样的，但是是有问题的，我这里只是对其修正一下

网上的代码仅仅实现了两个事件

void LoginWindow::*mousePressEvent*(QMouseEvent \*event)

{

if (event->button() == Qt::LeftButton) {

m\_windowPosition = event->globalPos() - this->pos();

event->accept();

}

}

void LoginWindow::*mouseMoveEvent*(QMouseEvent \*event)

{

if (event->buttons() & Qt::LeftButton) {

move(event->globalPos() - m\_windowPosition);

event->accept();

}

}

但是这样就会有一个问题，就是当鼠标在一个实现了mousePressEvent的类上点击时（比如QPushButton）会被该类优先处理此事件。而不会将事件传递到窗体的mousePressEvent中。继续，当移动鼠标到这个按钮外时（假设点在了QPushButton上）会触发窗体的mouseMoveEvent从而导致计算坐标时发生错误，此时你就会看到窗体闪了一下

//下面是正确姿势，增加了一个判断状态的标志位

void LoginWindow::*mousePressEvent*(QMouseEvent \*event)

{

if (event->button() == Qt::LeftButton) {

m\_drag = true;

m\_windowPosition = event->globalPos() - this->pos();

event->accept();

}

}

void LoginWindow::*mouseMoveEvent*(QMouseEvent \*event)

{

if (m\_drag && event->buttons() & Qt::LeftButton) {

move(event->globalPos() - m\_windowPosition);

event->accept();

}

}

void LoginWindow::*mouseReleaseEvent*(QMouseEvent \*)

{

m\_drag = false;

}

如果不考虑跨平台，只针对Windows平台，那么我建议用Windows下的标准方法，模拟标题栏移动消息，既简单又高效

void MainWindow::mousePressEvent(QMouseEvent \*event)

{

    if (ReleaseCapture())

        SendMessage(HWND(this->winId()), WM\_SYSCOMMAND, SC\_MOVE + HTCAPTION, 0);

    event->ignore();

}

**QComboBox：**

QComboBox分成三个定制部分：文本框（是否可编辑），按钮（箭头标记、边框），下拉框（选项高度、子组件布局）。

按钮是QComBox的一个子组件，::down-arrow来表示

可编辑与不可编辑的QSS：QComboBox:editable {    } QComboBox:!editable {    }

要想设置为可编辑只能调用函数ui->comboBox\_username->setEditable(true);

添加内容：

|  |
| --- |
| **[addItem](qcombobox.html" \l "addItem)**(const QString &*text*, const QVariant &*userData* = QVariant()) |
| **[addItem](qcombobox.html" \l "addItem-1)**(const QIcon &*icon*, const QString &*text*, const QVariant &*userData* = QVariant()) |
| **[addItems](qcombobox.html" \l "addItems)**(const QStringList &*texts*) |

**removeItem**(int index)

QComboBox

{

font: 11pt "等线";

background:white;

padding-left:5px ;

border-top-left-radius:3px;

border-top-right-radius:3px;

border: 1px solid rgb(209 , 209 , 209);

}

QComboBox:hover

{

border: 1px solid rgb(21 , 131 , 221);

}

QComboBox QAbstractItemView::item

{

height:40px;

}

QComboBox::down-arrow

{

border-image:url(:/Resources/drop\_down\_Button.png) 0 34 0 0 ;

}34 表示从右边开始距离34，最后一个0表示从左边的0位置开始

QComboBox::down-arrow:hover

{

border-image:url(:/Resources/drop\_down\_Button.png) 0 17 0 17 ;

}

QComboBox::down-arrow:on

{

border-image:url(:/Resources/drop\_down\_Button.png) 0 0 0 34 ;

}

QComboBox::drop-down

{

width:20px;

background:transparent; 这一句让颜色与背景对的上

padding-right:5px;

}

使用QSS可以改变下拉菜单中滑动条的大小：

QScrollBar:vertical {//加宽滑动条按钮  
      border: 2px solid grey;  
      width: 30px;

}

C++中图片切割pixmap.copy这个函数，对应QSS，自己的StaticButton类继承自Widget，然后使用*enterEvent、leaveEvent、mousePressEvent、mouseReleaseEvent、changeEvent*（选中与非选中的切换）事件函数来得到状态变化，以update()函数进行重绘*。*

void StaticButton::*paintEvent*(QPaintEvent \*)

{

QPainter painter(this);

painter.drawPixmap(rect(), m\_currentPix);

}

QComboBox内部也是Model/View框架来维护下拉框内容的，因此可以直接定义一个QListView，配合布局来实现更好的下拉菜单：

ui->comboBox->setView(new QListView());

自己添加QListView操控其样式是控制下拉菜单样式的唯一方法。

ThemeRoller::ThemeRoller(QWidget \*parent)

    : QMainWindow(parent)

{

    ui.setupUi(this);

    m\_listWidget = new QListWidget(this);

        // 设置子项目代理，否则下拉框选项周围会出现虚线框

    m\_listWidget->setItemDelegate(new NoFocusFrameDelegate(this));

    ui.comboBox->setEditable(true);

    ui.comboBox->setModel(m\_listWidget->model());

    ui.comboBox->setView(m\_listWidget); //Widget也是个View

        // 在下拉框中添加5个选项

    for (int i = 0; i < 5; ++i)

    {

        ComboboxItem\* item = new ComboboxItem(this);

        item->setLabelContent(QString("Account") + QString::number(i, 10));// ComboboxItem是自定义的继承自QWidget类

        connect(item, SIGNAL(chooseAccount(const QString&)), this, SLOT(onChooseAccount(const QString&)));

        QListWidgetItem\* widgetItem = new QListWidgetItem(m\_listWidget);

        m\_listWidget->setItemWidget(widgetItem, item);

    }

}

使用成员函数来移除removeItemWidget不完整，真正的完整方法是delete指定的QListWidgetItem，或者使用takeItem()。

这种简单的直接设置text给QListWidget才可以直接被findItems找到



我是直接

QList<QListWidgetItem\*> ItemPointList=m\_listWidget->findItems(".\*",Qt::MatchRegExp);

for(int i1=0; i1<ItemPointList.size(); ++i1){…}进行遍历

QLineEdit \*QComboBox::lineEdit() const

Returns the line edit used to edit items in the combobox, or 0 if there is no line edit.

Only editable combo boxes have a line edit. 种种迹象表明了这是个组合元件。

可以实现树状结构：

int main(int argc, char \*argv[])   
{

QApplication a(argc, argv);

MainWindow w;

// 创建文件系统模型

QFileSystemModel \*model = new QFileSystemModel(&w);

// 指定要监视的目录

model->setRootPath(QDir::currentPath());

QTreeView \*tree = new QTreeView(&w);

tree->setModel(model);

tree->setRootIndex(model->index(QDir::currentPath()));

QComboBox \*b = new QComboBox(&w);

b->resize(400, 20);

b->setView(tree);

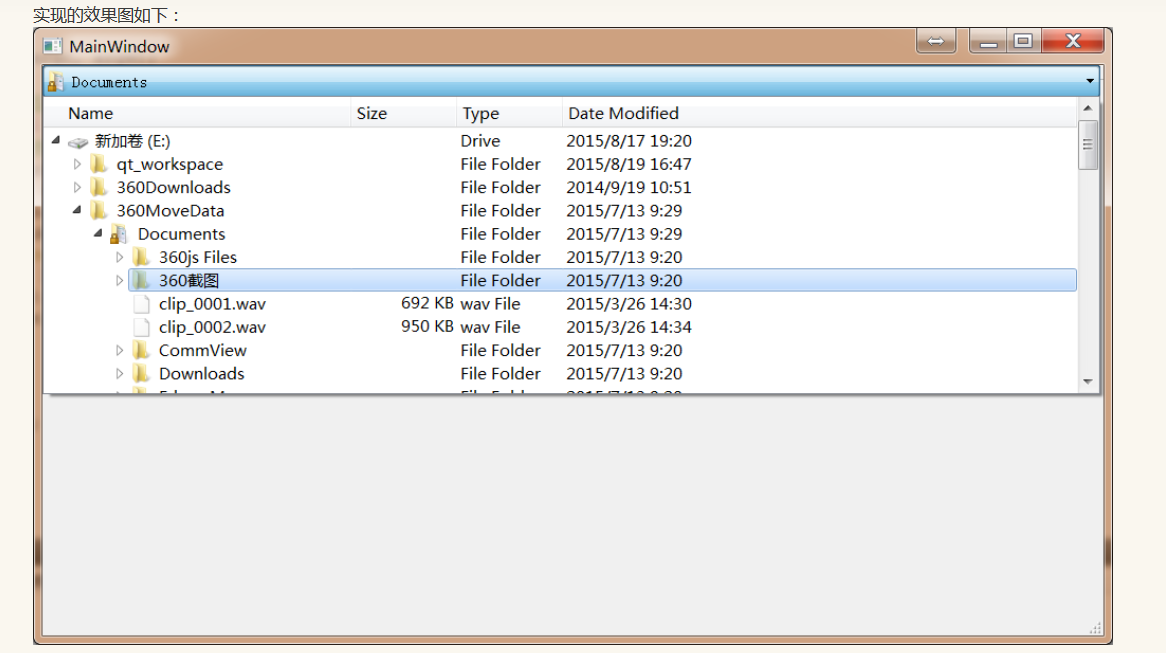
b->setModel(model);

b->show();

w.show();

return a.exec();

}



传闻事件处理需要设置窗口的整体时间处理函数然后进行过滤。

QComboBox QAbstractItemView::item{height:20px;} 据说这QSS可以改变item的高度

**操控窗口的显示：**



Qt::WindowNoState   0x00000000  The window has no state set (in normal state).

 Qt::WindowMinimized 0x00000001  The window is minimized (i.e. iconified).

 Qt::WindowMaximized 0x00000002  The window is maximized with a frame around it.

 Qt::WindowFullScreen    0x00000004  The window fills the entire screen without any frame around it.

 Qt::WindowActive    0x00000008  The window is the active window, i.e. it has keyboard focus.

void LoginWindow::on\_pushButton\_minimize\_clicked()

{

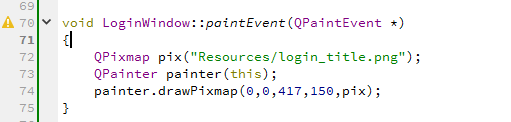
if( windowState() != Qt::WindowMinimized ){

setWindowState( Qt::WindowMinimized );

}

}

这样子就不会说我有参数没有使用了：



**Qt存储用户密码实现便捷登录：**

安全：对密码的安全保护，直接base64加密以后自己调换一下几个字母的顺序。这就必须要实现逆向才能破解。md5现在因为有库也变得不安全了。

显示：直接显示最近登录的一个账号。

1. xml记录 产生xml文件。
2. 数据库记录
3. QSettings 这个算是构造ini配置文件。

用户名控件肯定是使用QComBox，然后设置connect，使用currentText () 获取用户选择的内容，然后进行密码的显示。以下函数可以插入内容：

void insertItem ( int index, const QString & text, const QVariant & userData = QVariant() )

void insertItem ( int index, const QIcon & icon, const QString & text, const QVariant & userData = QVariant() )

void insertItems ( int index, const QStringList & list )

遗憾的事情是不能使用QSS来控制样式。只能是使用View，比如说QListView。

如果配合数据库实现，可以直接使用QSqlQueryModel。这个倒是比较直接，如果你使用默认连接的话：

              QSqlQueryModel \*model = new QSqlQueryModel;

              model->setQuery(“select \* from student”);

              model->setHeaderData(0, Qt::Horizontal, tr(“id”));

.             model->setHeaderData(1, Qt::Horizontal, tr(“name”));

QListView \*xxx =new QListViwe; xxx.setModel(model);

ui->QComBox\_xxxx->setView(xxx);

xxx->setStyleSheet(“在这里设置样式表”)

QListView继承自QAbstractItemView，所以可以使用下面这个函数：

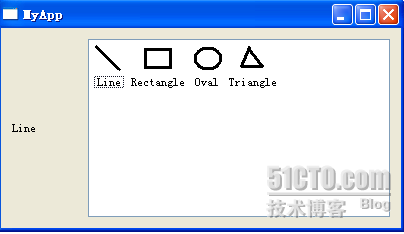
QAbstractItemView::setIndexWidget ( const QModelIndex & index, QWidget \* widget )

**QListWidget:**

list = new QListWidget;   
        list->addItem(new QListWidgetItem(QIcon(":/images/line.PNG"), tr("Line")));   
        list->addItem(new QListWidgetItem(QIcon(":/images/rect.PNG"), tr("Rectangle")));   
        list->addItem(new QListWidgetItem(QIcon(":/images/oval.PNG"), tr("Oval")));   
        list->addItem(new QListWidgetItem(QIcon(":/images/tri.PNG"), tr("Triangle")));



如果设置list->setViewMode(QListView::IconMode); 以图标为主的显示模式



QListWidget就是Model与View的组合显示。所以在设置QComboBox的时候可以setModel和setView来使用QListWidget。

**QSS选择器：**

<http://blog.csdn.net/z609932088/article/details/51011071>

.类名  作为选择器，只会作用于它自己，它的子类不受影响。

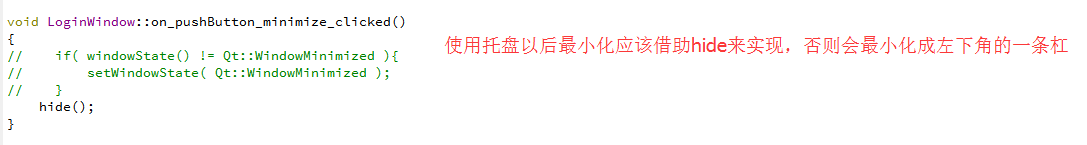
通过objetcname设置的QSS可能会作用于多个，因为可以有多个名字是一样的。

QHBoxLayout明明没有addItem成员结果Qt给我的自动补全选项里面有这个。不对我看到addItem显示是斜体，看来应该是继承来的成员函数。

信号函数的参数不能指定为const。

**右下角托盘图标**：

窗口样式设置中setWindowFlags(Qt::FramelessWindowHint|Qt::WindowStaysOnTopHint|Qt::Tool);（可以不加上Qt::Tool，不然showMinimized();最小化窗口的时候，就会最小化到整个windows界面的左下角就像是qq早期版本一样） 第一个是无边框，中间的是始终置顶，最后一个是消除任务栏图标。我只希望他有个托盘图标，和QQ一样。



显示的话还是showNormal()

void LoginWindow::createTrayIcon(void) {

m\_trayIcon->setIcon(QIcon(":/Resources/battery.ico"));

m\_trayIcon->setToolTip("JWCKnife");

QMenu \*trayIconMenu = new QMenu(this);

m\_normalAction = new QAction("打开面板", this);

connect(m\_normalAction, SIGNAL(triggered()), this, SLOT(showNormal()));

m\_quitAction = new QAction(tr("退出"), this);

connect(m\_quitAction, SIGNAL(triggered()), qApp, SLOT(quit()));

trayIconMenu->addAction(m\_normalAction);

trayIconMenu->addSeparator();

trayIconMenu->addAction(m\_quitAction);

m\_trayIcon->setContextMenu(trayIconMenu);

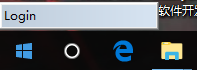
m\_trayIcon->show();

m\_trayIcon->showMessage("Battery tells","我在右下角的托盘那里哦",QSystemTrayIcon::Information,5000);

connect(m\_trayIcon, SIGNAL(activated(QSystemTrayIcon::ActivationReason)),

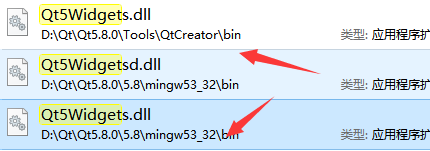
this,SLOT(trayiconActivated(QSystemTrayIcon::ActivationReason)));

}

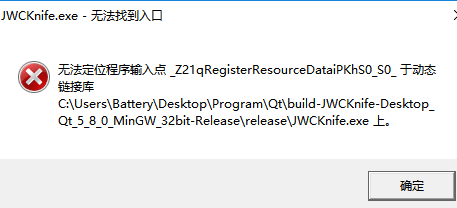
默认实现的最小化：

**发布：**





D:\Qt\Qt5.8.0\Tools\QtCreator\bin这个目录下的文件看似可以解决release版本动态链接库缺失的问题（但是用depends查看可以知道dll缺失的更多），D:\Qt\Qt5.8.0\5.8\mingw53\_32\bin才可以。可以借助depends.exe参考exe依赖的dll，尤其是我下面这种情况：

我常看以后认为已经将所有需要的dll都补全了。dll放到build-JWCKnife-Desktop\_Qt\_5\_8\_0\_MinGW\_32bit-Release或者build-JWCKnife-Desktop\_Qt\_5\_8\_0\_MinGW\_32bit-Release\release都可以，只有这两个目录可以或者环境变量中指定的目录。有库在目录下，而有不能直接运行exe文件，这样子在Qt Creator中运行的时候就会

**Starting C:\Users\Battery\Desktop\Program\Qt\build-JWCKnife-Desktop\_Qt\_5\_8\_0\_MinGW\_32bit-Release\release\JWCKnife.exe...**

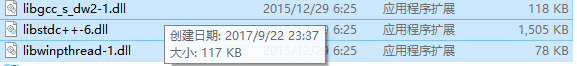
**程序异常结束。**

**C:\Users\Battery\Desktop\Program\Qt\build-JWCKnife-Desktop\_Qt\_5\_8\_0\_MinGW\_32bit-Release\release\JWCKnife.exe crashed.**

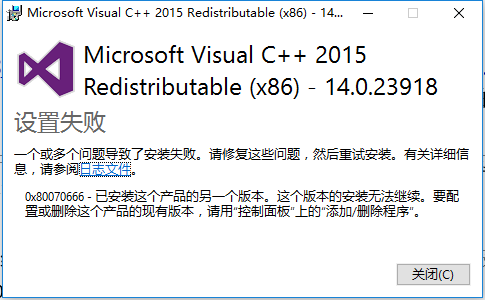
**只有部分正确的dll也不行，qt只有在你什么dll都没有放的时候才会去找dll来补上。**

**注意：windows中文件搜索搜sql是查不到Qt5Sql.dll。不过如果文件名中间使用了下划线分离的话是可以金星文件名的部分匹配的。**

**实际上是需要这三个文件中的libstdc++-6.dll：**

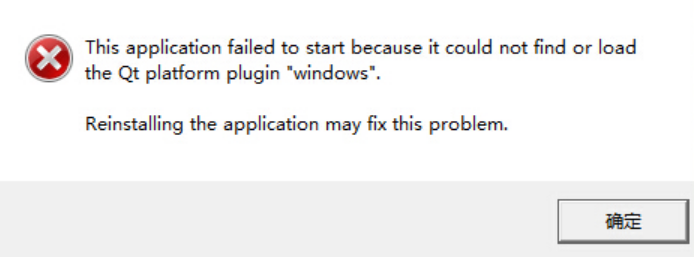


网上不少人提供的线索是使用Window update，但是因为我已经禁用服务了，所以跟新的时候会出现错误0x80070422。卸载了2008库以后却出现了缺少MSVCP140.dll（这是2015VC包的问题），于是我打算直重新安装2015VC包（从VS的安装包里面找到的VC\_redist）,but：



这个问题似乎需要先进行windows update之后再尝试安装下。真的没想到卸载2008库会导致这个问题，这个文件很重要，很多程序运行都依赖于这个文件。查看日志直接拉到最下面，发现是因为新版本已经安装的原因，后来明白：Visual C++ Redistributable for Visual Studio 2015 cannot be installed when VS2017 RC is installed.

最终我解决问题的方式是将D:\Qt\Qt5.8.0\5.8\mingw53\_32\bin全部进行复制到当前文件夹，或者在PATH中指定这个目录，作为PE文件寻找目录。我用复制的方法，成功运行文件以后打开文件，在删除所有dll，用到的dll会告诉你是哪几个….然后我发现我少了个Qt5Svg.dll，可是我应该没有用到svg才对….还需要用到libgcc\_s\_dw2-1、libstdc++-6、libwinpthread-1这三个dll。

D:\Qt\Qt5.8.0\5.8\mingw53\_32\plugins 这个目录下的文件夹也是要用到的，比如说使用了sqldriver中的qsqlite.dll，可以直接将所有的文件夹复制到虚拟机中，然后开机程序，通过文件占用的原理删除不需要的dll文件。这里面的dll必须要在对应名字的文件夹 下面，不能直接拉出来。否则会出现

或者运行了以后不能显示界面等（使用了图像解析，但是没有图像相关的dll）问题。

安装打包工具：http://blog.csdn.net/fishmai/article/details/52782332

在[软件工程](https://zh.wikipedia.org/wiki/%E8%BD%AF%E4%BB%B6%E5%B7%A5%E7%A8%8B" \o "软件工程)中，**性能分析**（**performance analysis**也称为**profiling**），是以收集程序运行时信息为手段研究程序行为的分析方法，是一种[动态程序分析](https://zh.wikipedia.org/w/index.php?title=%E5%8B%95%E6%85%8B%E7%A8%8B%E5%BC%8F%E5%88%86%E6%9E%90&action=edit&redlink=1)的方法。也就是QT中的profile运行。

**窗口大小判断：**

void MainWindow::*resizeEvent*(QResizeEvent \* e) {

qDebug() << this->size();

}而且通过这个可以知道在窗口刚启动的时候会有一次*resizeEvent*。

# QStackedWidget:

接口:

int addWidget(QWidget \* widget)

添加页面，并返回页面对应的索引

int count() const

获取页面数量

int currentIndex() const

获取当前页面的索引

QWidget \* currentWidget() const

获取当前页面

int indexOf(QWidget \* widget) const

获取QWidget页面所对应的索引

int insertWidget(int index, QWidget \* widget)

在索引index位置添加页面

void removeWidget(QWidget \* widget)

移除QWidget页面，并没有被删除，只是从布局中移动，从而被隐藏。

QWidget \* widget(int index) const

获取索引index所对应的页面

信号：

void currentChanged(int index)

当前页面发生变化时候发射，index为新的索引值

void widgetRemoved(int index)

页面被移除时候发射，index为页面对应的索引值

共有槽函数：

void setCurrentIndex(int index)

设置索引index所在的页面为当前页面

void setCurrentWidget(QWidget \* widget)

设置QWidget页面为当前页面

//add stacked widget 由于5前面的数字缺少，所以insert<=>add

StackedSelectClassWidget \*pWidget0 =new StackedSelectClassWidget("ClassSchedule");

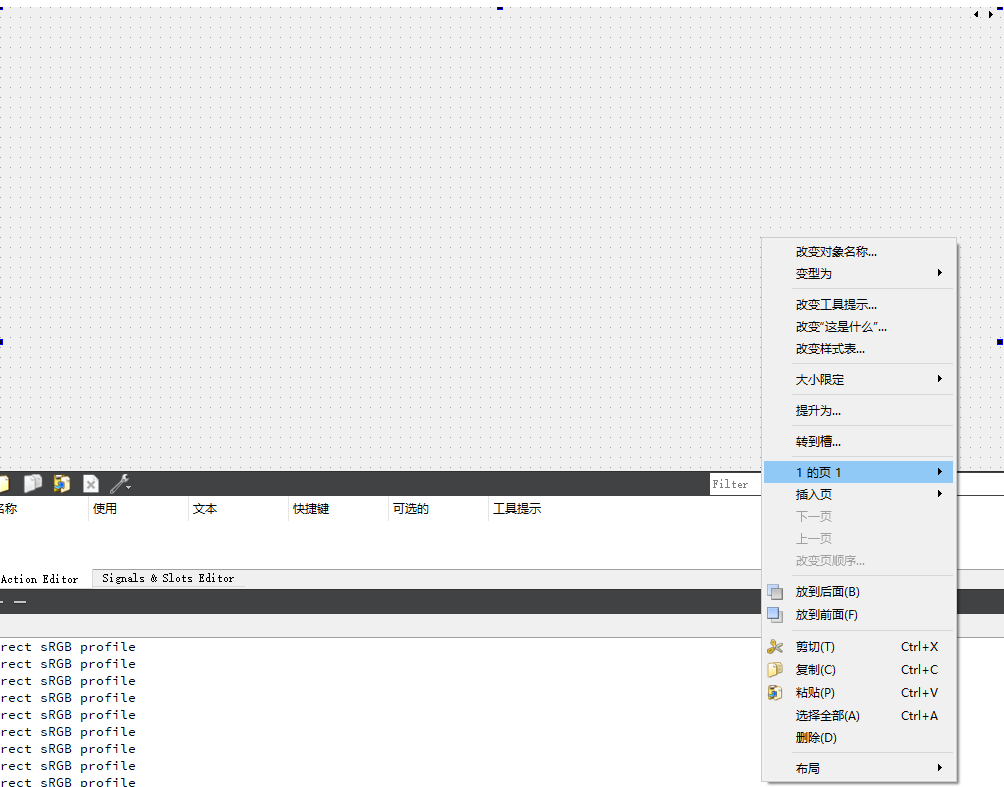
ui->stackedWidget->insertWidget(0, pWidget0);

StackedExamRelatedWidget \*pWidget1 =new StackedExamRelatedWidget(this);

ui->stackedWidget->insertWidget(5, pWidget1);

ui->stackedWidget->setCurrentIndex(0);

从QDesigner 的QStackedWidget中删除页面：

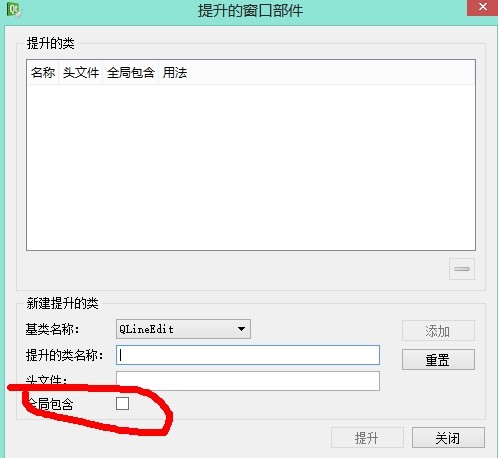


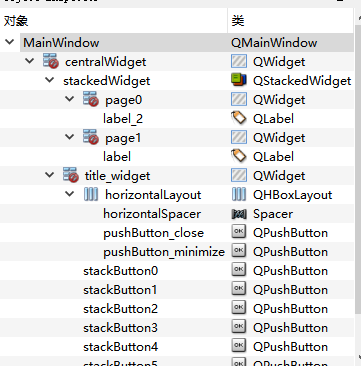
qt 清理重新构建项目可能不行，比如说我删除一个没用到的图片资源就会不停的和我说qrc\_image.cpp中要求它。只好把build-JWCKnife-Desktop\_Qt\_5\_8\_0\_MinGW\_32bit-Debug整个删除了重新建立。

ui设计里面的QstatckedWidget右上角的是用于放置窗口的时候进行窗口切换，并不会显示出来。

在ui编辑器里面QSS#是根据对象名称来定位的，但是我改了右上角的名字却没有起作用，这时候你双击要更改的元件，就会出现对象名字的编辑器。

ui提升窗口类，可以将元件提升成自己的类：

这个是include的时候使用<>还是””

虽然是这样的关系，但是也是ui->stackButton0，这时候ui是直接存有存有对象的地址的。



m\_btn[i]->setObjectName(QString::number(i)); 使用这种方法倒是可以将对象命名成数字。而在ui中是不能这样进行命名的。

最小化函数 最大化函数

窗口大小状态函数：

QWidget：：showMaximized(); 最大  
QWidget：：showMinimized();最小  
QWidget：：showNormal();还原

void MainWindow::on\_pushButton\_minimize\_clicked()

{

if(!isMinimized())

showMinimized();

}

**QSS之QCheckBox：**

QCheckBox::indicator  
{  
 width:20px;  
 height:20px;  
}  
QCheckBox::indicator:unchecked  
{  
 image:url(:/image/check\_f);  
}  
QCheckBox::indicator:checked  
{  
 image:url(:/image/check\_t);  
}

**QSS之QScrollBar：**

QScrollBar:vertical{ //垂直滑块整体

background:#FFFFFF; //背景色

padding-top:20px; //上预留位置（放置向上箭头）

padding-bottom:20px; //下预留位置（放置向下箭头）

padding-left:3px; //左预留位置（美观）

padding-right:3px; //右预留位置（美观）

border-left:1px solid #d7d7d7;//左分割线

}

QScrollBar::handle:vertical{//滑块样式

background:#dbdbdb; //滑块颜色

border-radius:6px; //边角圆润

min-height:80px; //滑块最小高度

}

QScrollBar::handle:vertical:hover{//鼠标触及滑块样式

background:#d0d0d0;

} //滑块颜色

QScrollBar::add-line:vertical{//向下箭头样式 向下与向右才是add

background:url(:/images/resource/images/checkout/down.png) center no-repeat;

} //单张图片可以这样使用，如果需要使用qss进行图片切分，那么应该使用border-image属性。而且没有//这个注释。详情可以搜索“设置背景图片”

QScrollBar::sub-line:vertical{//向上箭头样式

background:url(:/images/resource/images/checkout/up.png) center no-repeat;

}

**通过正则表达式来限制输入:**

QRegExp regx("[0-9][0-9]{0,14}");  
QValidator \*validator = new QRegExpValidator(regx,lineEdit );  
lineEdit->setValidator( validator );

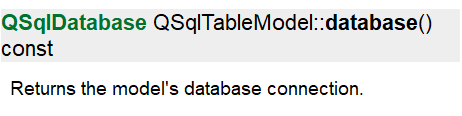
**MVC？No！： MVD**

与MVC模式不同，model/view结构没有用于与用户交互的完全独立的组件。一般来讲， view负责把数据展示给用户，也处理用户的输入。为了获得更多的灵性性（交互性），交互通过delegagte执行。它既提供输入功能又负责渲染view中的每个数据项。 控制delegates的标准接口在QAbstractItemDelegate类中定义。Delegates通过实现paint()和sizeHint()以达到渲染内容的目的，editorEvent()处理鼠标事件。然而，简单的基于widget的delegates,可以从QItemDelegate子类化，而不是QAbstractItemDelegate,这样可以使用它提供的上述函数的缺省实现。

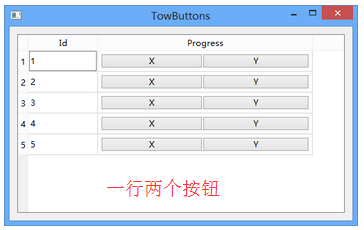
Qt提供的标准views都使用QItemDelegate的实例来提供编辑功能。它以普通的风格来为每个标准view渲染数据项。这些标准的views包括：QListView,QTableView,QTreeView。所有标准的角色都通过标准views包含的缺省delegate进行处理。一个view使用的delegate可以用itemDelegate()函数取得,而setItemDelegate() 函数可以安装一个定制delegate。需要注意的是,当一个delegate创建时，不需要安装一个widget，只有在真正需要时才创建这个用于编辑的widget，而且我们new出来以后通过Delegate的CreateEditor()返回指向它的指针，之后它的销毁不需要我们进行考虑。

当model不能满足我们的需求的时候可以考虑一下使用Proxy Model。

Model可以通过以下函数返回数据库连接：



自己重写Delegate类的paint函数可以做到：



**Model的rowCount以及columnCoun函数：**

QFileSystemModel 直接调用rowCount()获取到的指是0，因为这个模型还会有一个线程将内容加载进去。在加载之前都会是0，推荐：

connect(model,SIGNAL(directoryLoaded(QString)),this,SLOT(findItems(QString)));

还有table based model（table类型的model，比如只读的QSqlQueryModel）的rowCount()以及columnCount()都被要求是返回0的。

对于一个是显示“自动换行”（不算是自动换行，但是效果不错）的Delegate（不提供编辑功能，如果提供编辑功能需要注意不要像QStyledItemDelegate一样编辑以后 需要回车才能将数据提交，所以自己实现setEditorData方法的时候注意）： 实现了最常用的三个方法

#include "AutoTurnLineDelegate.h"

AutoTurnLineDelegate::AutoTurnLineDelegate(QObject \*parent) :QAbstractItemDelegate(parent)

{

}

void AutoTurnLineDelegate::*paint*(QPainter \*painter, const QStyleOptionViewItem &option,

const QModelIndex &index) const

{

QString text = index.model()->*data*(index, Qt::DisplayRole).toString();

painter->drawText( option.rect, Qt::TextWordWrap | Qt::AlignHCenter | Qt::AlignVCenter, text );

}

bool AutoTurnLineDelegate::*editorEvent*(QEvent \*event, QAbstractItemModel \*model, const QStyleOptionViewItem &option, const QModelIndex &index) {

if ( event->type()==QEvent::MouseButtonRelease ) {

if( true==model->*setData*(index, index.model()->*data*(index).toString()) ){

static\_cast<QSqlTableModel\*>(model)->submitAll();

emit model->dataChanged(index,index);

return true;

}else

return false;

}

}

//这个函数实现还不是自动换行，处理太简单了，需要判断一行字符个数，然后调整height，由此实现自动换行.要做上限的限制，不然表格参差不齐。

QSize AutoTurnLineDelegate::*sizeHint*(const QStyleOptionViewItem &option, const QModelIndex &index) const {

QFont font( option.font );

const QFontMetrics fm( font );

const QString text=index.model()->*data*(index).toString();

int width = fm.width(text);

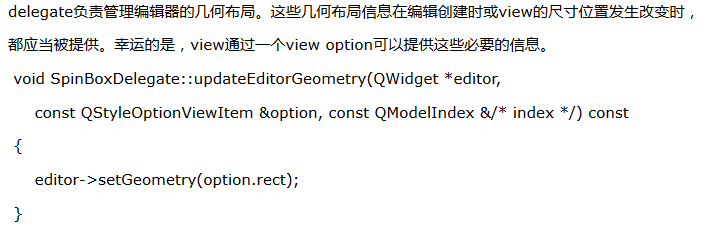
int height = fm.height();

return QSize(width ,height);

}

不过传说自动换行，还可以通过resizeRowsToContents()直接实现

需要重载四个函数：1.编辑器创建2.显示model的数据3.将编辑数据返还给model4.几何形状设置



QTableView的学习网站：<http://blog.csdn.net/u010780613/article/details/50442639> 包括合并单元格以及向单元格中插入图片。

相关函数：setSpan、rowSpan（默认返回1）、columnSpan（默认返回1）

如果setSpan(0,0,2,1);



这时候其实就是Item(0,0)将Item(1,0) 覆盖显示了，设置的话只需要设置Item(0,0)就好。

QTableWidget是QTableView的子类（自己实现了内置model，还把setmodel变为private），主要的区别是QTableView可以使用自定义的数据模型来显示内容(也就是先要通过setModel来绑定数据源)，而QTableWidget其单元格数据是QTableWidgetItem的对象来实现的(也就是不需要数据源，将逐个单元格内的信息填好即可). 一般工程应该是使用QTableView，因为这可以直接配合自己的Model. QTableWidget 存在成员函数setCellWidget，可以借此来进行复选框的少量数据表格显示设计。



默认左上角点击了以后会全选整个表格，左上角这种特殊的widget叫做CornerWidget：

void setCornerWidget(QWidget \* widget, Qt::Corner corner = Qt::TopRightCorner)//在菜单条的最左或最右添加个小部件，corner可以为Qt::TopRightCorner（最右）或Qt::TopLeftCorner(最左)

void setCornerButtonEnabled(bool choice);//是否开启对点击事件的响应，即点击之后选择整个表格

除了QTablewidget有这个CornerWidget以外，QMenuBar也有。

按照数据库中数据，动态生成列数与行数符合的TableWidget：

/\*

\* setHorizontalHeaderLabels(header); 要在本函数之后使用

\*/

void **fillTableWidget**(QTableWidget \*tableWidget, QSqlQuery \*pQuery, const QString strCommand,const char \*info) {

SQLErrorDetect(pQuery, strCommand, info);

//tableWidget->setRowCount(pQuery->size()); 这句话需要数据库支持，SQLITE不支持

//先通过加入一行数据来确定columnCount

pQuery->next();

int columnCount=0;

for(columnCount=0; pQuery->value(columnCount).isValid() ;++columnCount ) {

tableWidget->setColumnCount(1+columnCount);

tableWidget->setItem(0, columnCount, new QTableWidgetItem( pQuery->value(columnCount).toString()));

}

tableWidget->setRowCount(1);

//插入剩余数据，顺便设置行数

for(int row =1; pQuery->next() ; ++row) {

tableWidget->setRowCount(1+row);

for(int column=0; column<columnCount; ++column)

tableWidget->setItem(row, column, new QTableWidgetItem( pQuery->value(column).toString()));

}

//不能自动缩调行列数目

// for(int row=0; pQuery->next();++row ) {

// for(int column=0; pQuery->value(column).isValid() ;++column )

// tableWidget->setItem(row, column, new QTableWidgetItem( pQuery->value(column).toString()));

// }

}

QTableView（QTableWidget） 提供了setspan函数来合并指定的单元格（row与column都是从0开始计数的），当然没有必要再允许单元格的拆分。通过使用clearSpans()清除所有单元格的合并方便自己重新设置，如果希望自己实现：

<http://blog.csdn.net/ly19820701/article/details/6218366>

如果设置表格border:none，也就是没有外部边框，这时候对边界的单元格进行合并（比如说对靠左边边界的单元格进行上下合并就会导致合并以后的单元格左边边界消失）。因为涉及部分单元格的合并（跨行/跨列显示），所以不能直接setShowGrid(false)，失去了内部线条，表格的可读性大大降低。后来发际与ui->tableView->verticalHeader()->*setVisible*(false); 这句话有关系，如果讲这句话去掉以后，垂直的头依旧会不会出现，同时左边的边界线全部消失，所以消失应该才是应该的样子。后来我将垂直头设置为true显示，也没有让它显示出来，可能还有什么神奇的原因。

通过调试知道，setspan函数会调用指定了整个tableview区域的update函数，和mfc一样并不是对整个界面的刷新， 从而来减少对界面的跟新操作。

窗口的close()默认情况下也还是hide()，除非如果我们为它设置过标记位：Qt::WA\_DeleteOnClose（关闭时删除它），则出来调用hide()外，还会调用deleteLater()函数。

调用QTtableView的verticalHeader()->setDefaultSectionSize(15)可以设置tableview所有行的默认行高。

隐藏头部：#include <QHeaderView> m\_tableView->verticalHeader()->*setVisible*(false);

根据内容自动适应： 需要在setModel以后进行调用  
void QTableView::resizeColumnToContents ( int column ) [slot]  
根据内容自动调整所有列的列宽  
void QTableView::resizeColumnsToContents () [slot]  
根据内容自动调整某一行的行高  
void QTableView::resizeRowToContents ( int row ) [slot]  
根据内容自动调整所有行的行高。  
void QTableView::resizeRowsToContents () [slot]

指定列宽：m\_tableView->setColumnWidth(2,100); 但是不是固定的

ui->tableView->horizontalHeader()->setResizeMode(column, QHeaderView::Fixed); 进行固定设置

固定列宽以后内容自动进行换行：

1. 推荐使用delegate，重载paint函数

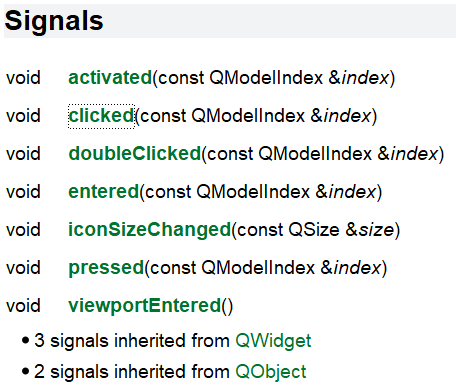
QString text = index.model()->*data*(index, Qt::DisplayRole).toString();

painter->drawText( option.rect, Qt::TextWordWrap | Qt::AlignHCenter | Qt::AlignVCenter, text );

1. 根据表格的字数长度来计算设置表格的列高度

QTableView的美化使用QSS。滚动条的自动出现是内部成员,这个也可以使用QSS来进行设置:

QAbstractItemView提供了一些信号，其中QTableView继承了

 可以知道鼠标点击的是哪个QModelIndex

如果要的是右键点击，那就是设置菜单那种写法：

ui->tableWidget->setContextMenuPolicy (Qt::CustomContextMenu);

connect(ui->tableWidget,SIGNAL(customContextMenuRequested(QPoint)),this,SLOT(showMenu(QPoint)));

void StackedScheduleWidget::showMenu(const QPoint pos)

{

//设置菜单选项

QMenu \*menu = new QMenu(ui->tableWidget);

QAction \*pnew = new QAction("第一项",ui->tableWidget);

QAction \*pnew1 = new QAction("第二项",ui->tableWidget);

// connect (pnew,SIGNAL(triggered()),this,SLOT(clickgoose()));

// connect (pnew1,SIGNAL(triggered()),this,SLOT(clickmms()));

menu->addAction(pnew);

menu->addAction(pnew1);

menu->move(cursor().pos());

menu->show();

//获得鼠标点击的x，y坐标点

int x = pos.x ();

int y = pos.y ();

QModelIndex index = ui->tableWidget->*indexAt*(QPoint(x,y));

qDebug() << index.data().toString();

}

可能还需要设置qss，因为当鼠标hover qaction的时候，字体会由黑色变成白色，和界面一个颜色，于是就看不到了。

QMenu::item:selected {

color:blue;

}

QMenu {

color:black;

}

在QAction的槽函数中要想知道被点击的QModelIndex：

1. QAction的menu()函数得到menu，在得到坐标，最后通过indexAt得到

但是经过debug发现static\_cast<QAction\*>(sender())->menu() 返回的是一个空指针。后来才知道这个是子menu，通过setmenu函数设定。后来用了action->parentWidget()->pos();但是发现返回的pos是QAction相对于QMenu的pos。所以此方法就会失败。

1. 通过使用一个类的成员来传递，做到两个槽函数之间的共享变量。

就是效率与类的可读性设计之间的抉择，反正这个功能又不会被经常点击，所以我用了1。

下面这幅图片说明QTableView的范围不仅仅是数据显示的部分

要想消除这部分多余的大小只能通过自己计算应有大小，由ui->tableView->setGeometry(rect);进行设置（既能设置大小也能设置位置）。那是我们没有必要这么做，只需要去除上面的background-color，同时设置border:none。

QStackedWidget通过C++直接显示QTableView，虽然delegate成功显示，但是点击事件没有被QTableView捕获。我猜测是因为我在使用ui的时候，同时也保留了m\_mainLayout的原因。

QStackedWidget QTableView{

color: rgb(0, 0, 0);

border: none; /\*因为边框会把滚动条也包括进去\*/

/\*border: 1px solid #C07010;边框颜色\*/

gridline-color:#C07010; /\*grid线颜色\*/

background-color: rgb(20, 20, 20);

alternate-background-color: rgb(200, 200, 200); /\*行交替颜色\*/

selection-background-color: rgb(130, 190, 100); /\*选中行背景颜色\*/

}

QStackedWidget QTableView::item:!alternate:!selected{

background-color: rgb(220, 220, 220); /\*交替行的另一颜色\*/

}

QStackedWidget QHeaderView::section{

background: qlineargradient(spread:pad, x1:0, y1:0, x2:0, y2:1, stop:0 rgba(80, 80, 80, 255), stop:1 rgba(30, 30, 30, 255));

color: rgb(240, 240, 240);

padding-left: 4px;

border: 1px solid #C07010;

min-height: 30px;

} QHeaderView中的绘制方法叫做paintSection。



QHeaderView::section:horizontal 没有指定:horizontal会导致：

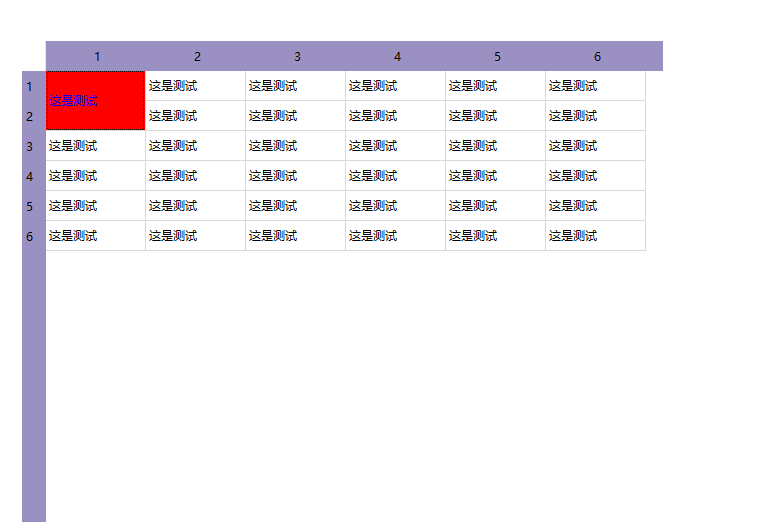


失去VerticalHeader，变成



虽然是因为意外导致垂直表头被隐藏，但是一般就是需要隐藏。可以在需要的地方通过指定QSS QHeaderView::section:vertical{min-width:40px;...} 使之出现。

如果直接QHeaderView 而没有:section就会



QTableCornerButton::section {

background-color:rgb(155,144,194);

}设置表格左上角

设置行宽、列宽通过QHeaderView实现比较方便：

QHeaderView \*verticalHeader = myTableView->verticalHeader();

verticalHeader->setSectionResizeMode(QHeaderView::Fixed);

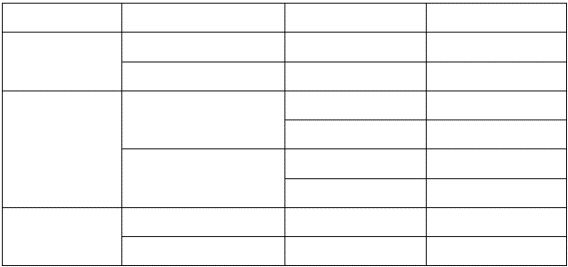
verticalHeader->setDefaultSectionSize(24);

也可以使用QTableWidget的setRowHeight()与setColumnWidth()

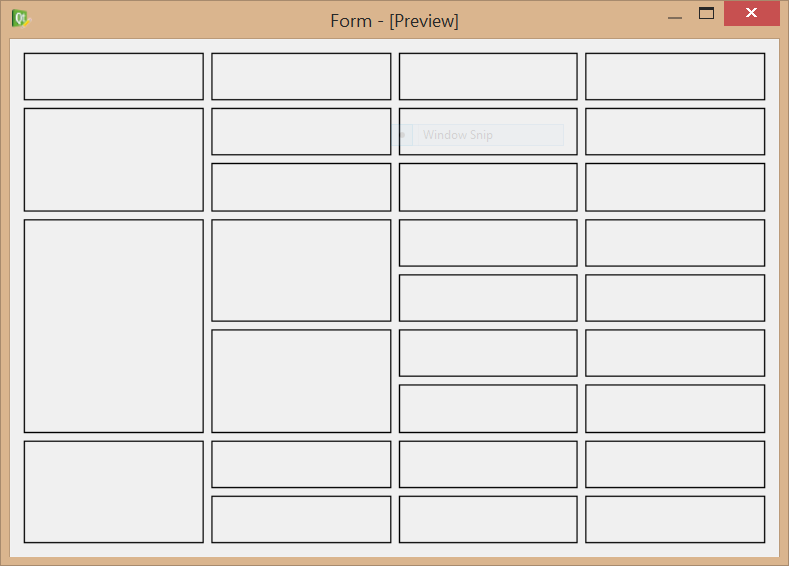
QTableWidget设置为不可修改（不可编辑）：

tableWidget->setEditTriggers(QAbstractItemView::NoEditTriggers);

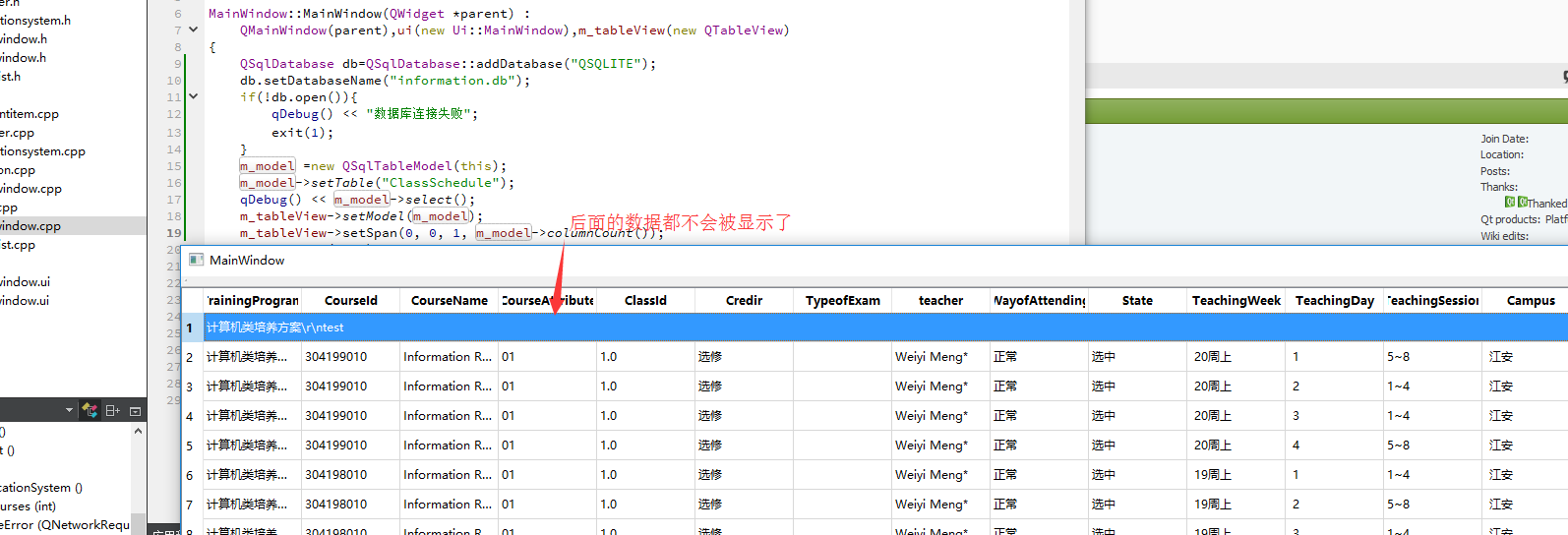
制作这种表格：



1. QTableView::setSpan()
2. using QGridLayout, add QFrames and set the rowSpan and columnSpan



QTableView的功能十分有限，甚至不能在其中进行人工的换行，不过你可以通过定义一个Delegate类，重载paint函数，在里面使用paint()的第一个参数Qpainter \*painter的成员函数painter->drawText(xxx) 进行属性的设置做到自动换行。



设置View的头部显示

1.从根源model上改变：

m\_pModel->setHeaderData(0, Qt::Horizontal, QString(""));

m\_pModel->setHeaderData(1, Qt::Horizontal, QString("时间"));

m\_pModel ->setHorizontalHeaderLabels(QStringList() << "参数" << "结果");

2.直接对QHeaderView进行设置：

m\_tableView-> setHorizontalHeader(QHeaderView \*header)

View中内容进行居中设置：

1.TableModel：：data(const QModelIndex &index, int role) const  
{  
     if(role == Qt::TextAlignmentRole){                                                   
          return int(Qt::AlignHCenter | Qt::AlignVCenter);   
     }  
}

2.自定义delegate中

QString text = index.model()->*data*(index, Qt::DisplayRole).toString();

painter->drawText( option.rect, Qt::TextWordWrap | Qt::AlignHCenter | Qt::AlignVCenter, text );

设置各行背景：  
1.使用QSS的话可以做到设置交替行的背景

2.自己写model，实现里面的data方法，注意参数role为backgroundcolor的时候，返回qcolor类型就行了

1. 不建议的方法就是自定义delegate，并且实现paint函数

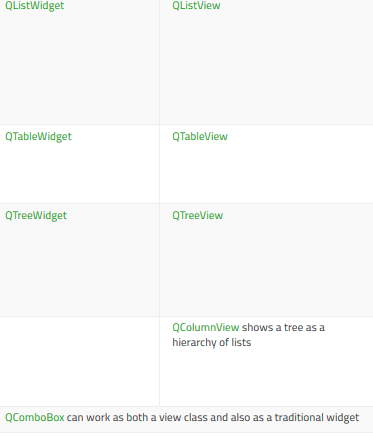
通过delegate实现view单元格的内部换行：

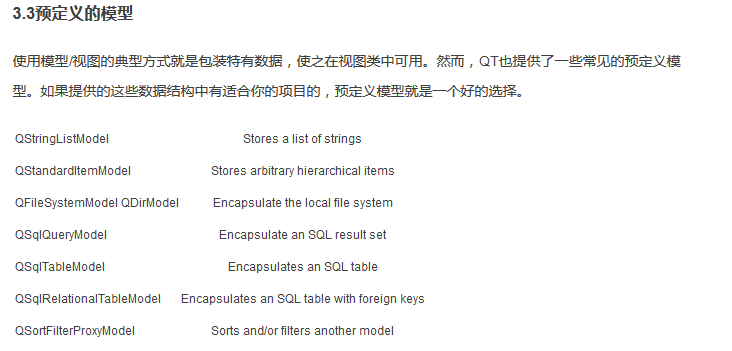
If the view and the controller objects are combined, the result is the model/view architecture.

<http://doc.qt.io/qt-5/modelview.html>

<http://www.mamicode.com/info-detail-1438357.html> （翻译稍微有点问题）

可以直接使用的显示类：





对于只能显示一条数据的元件[QLineEdit](http://doc.qt.io/qt-5/qlineedit.html), [QCheckBox](http://doc.qt.io/qt-5/qcheckbox.html)…就需要使用适配器Adapters进行数据与显示的匹配：

[QDataWidgetMapper](http://doc.qt.io/qt-5/qdatawidgetmapper.html) maps form widgets to a table row，这样会比较方便构建一个表单，表单中的每一个部分对应记录中一个内容

[QCompleter](http://doc.qt.io/qt-5/qcompleter.html) Qt内部实现了自动数据填充，这个用于搜索显示会比较方便

确定一个自定义的model：

先要确定你的数据是列表结构还是层次结构， 需要显示什么样的数据， 需不需要支持增删或编辑功能等。 根据需求来确定从哪个Model的基类派生，如一个显示字符串列表的Model可以采用QAbstractListModel， 树状层次就只能从QAbstractItemModel开始了。

QAbstractListModel类提供了非常灵活的接口，允许数据源以层次结构的形式来管理信息，也允许以某种方式对数据进行插入、删除、修改和存储。它也提供了对拖拽操作的支持。QAbstractTableModel、QAbstractTreeModel。

1.基本实现，需要实现rowCount(), data()，columnCount()，需要显示header的要去实现headerData()；如果是层次列表，则需要确定节点之间的层次关系，就需要实现index()和parent()两个函数， 一个是通过父指针和row,column座标确定一个子节点，一个是通过子节点知道它的父指针。

2.如果需要修改数据， 先要通知View我的Model数据是可以被编辑的， 就是要实现flags()这个函数， 此函数设置数据的属性，如可编辑、可被选中等； 编辑之后需要一个函数将编辑完成的数据传递给Model， 所以还要实现一个setData方法。

3.增删数据的实现：需要增删数据的Model还要告诉Model的底层：“我要增删数据了！”， “我要增删的数据是。。。”， 还有“我增删的操作已经做完了！”， 这些分别对应：调用beginInsertRows()和endInsertRows()。这两个函数需要在你增删数据前后进行调用。

对于自定义的Model类，必须要实现的虚函数：

int rowCount(const QModelIndex &parent = QModelIndex()) const Q\_DECL\_OVERRIDE ;

int columnCount(const QModelIndex &parent = QModelIndex()) const Q\_DECL\_OVERRIDE;

上面两个函数控制显示的行与列，可直接返回常数，然后就会在这个范围内调用data对数据进行设置与显示

QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const Q\_DECL\_OVERRIDE; 参数index表明了行和列信息index.row();,index.column();通过role的设置，可以对视图的显示造成影响

case Qt::FontRole:

if (row == 0 && col == 0) //change font only for cell(0,0)

{

QFont boldFont;

boldFont.setBold(true);

return boldFont;

}

break;

case Qt::BackgroundRole:

if (row == 1 && col == 2) //change background only for cell(1,2)

{

QBrush redBackground(Qt::red);

return redBackground;

}

break;



表头的设置：tableView->verticalHeader()->hide();

QVariant MyModel::headerData(int section, Qt::Orientation orientation, int role) const

{

if (role == Qt::DisplayRole)

{

if (orientation == Qt::Horizontal) {

switch (section)

{

case 0:

return QString("first");

case 1:

return QString("second");

case 2:

return QString("third");

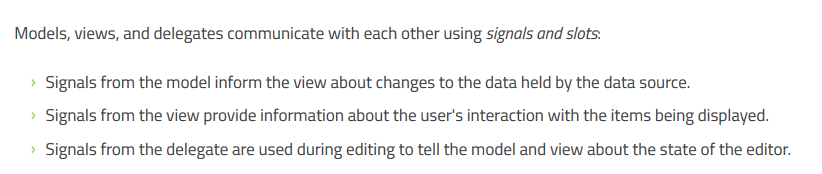
}

}

}

return QVariant();

}



Model是对数据的抽象。QStandardItemModel是一个多用途的 model，可用于表示list,table,tree views所需要的各种不同的数据结构，这model持有数据，使用的时候使用QStandardItem一条条插入数据。QDirModel维护相关的目录内容的信息,它本身不持有数据（数据就是本地的文件系统）*,*仅是对本地文件系统中的文件与目录的描述，这个model一般配合QListView或QTreeView进行使用。qtreeview还可以使用proxy Model，实现maptoSource 与 mapfromSource 函数。

model index是将数据存储与数据显示访问分开。唯一确定一个index需要三个参数：row，column和parentindex（想到TreeView，所以这第三个参数仅有时存在）。Model indexes包含一个指向创建它们的model的指针，QAbstractItemModel \*model = index.model();。model index不应该被保存，这就和C++中Vector元素的地址不能被保存一样，因为是可能会发生改变的东西。QModelIndex index = model->index(row, column, parentindex);获取index（如果指定了无效的parentindex，那么就会按照顶层作为父项进行返回），一个model的顶级项,由QModelIndex()取得，它们上式被用作父项。

**[q](qtglobal.html" \l "quintptr-typedef)uintptr** QModelIndex::**internalId**() const

这个函数返回一个qt内部定义的指向unsigned int 的指针，这个值用于连接Model的外部接口Index以及Model内部的数据结构。internalId就像是一个数组的索引。

**[QModelIndex](qmodelindex.html)** QAbstractItemModel::**createIndex**(**int** *row*, **int** *column*, **[quintptr](qtglobal.html" \l "quintptr-typedef)** *id*) const

**[QModelIndex](qmodelindex.html)** QAbstractItemModel::**createIndex**(**int** *row*, **int** *column*, **void** \**ptr* = Q\_NULLPTR) const最后一个参数是指针，用于自己实现一种内部指针的区分方式，这个指针的值会在data函数中用到，比如Tree Model就会有一样的row、column，所以不同层（不同的parent）需要一个不同的值。

不同的model中一样row,column，internalId的createIndex也是不同的。

无效的QModelIndex即直接创建的 无参QModelIndex，默认情况下row=column=-1,internalId=0。

As well as presenting data, views handle navigation between items, and some aspects of item selection. The views also implement basic user interface features, such as context menus and drag and drop. A view can provide default editing facilities for items, or it may work with a [delegate](http://doc.qt.io/qt-5/model-view-programming.html" \l "delegate-classes) to provide a custom editor.

QTableView和QTreeView,除数据项之外也可显示标题(Headers)，标题部分通过一个QHeaderView来实现，标题与view一样总是从相同的model中获取数据，通过QAbstractItemModel::headerDate()**。** view通过使用**selections**来跟踪用户选择的数据项。每个view可以维护单独使用的selections，也可以在多个views之间共享，直观的实现是secondTableView->**setSelectionModel**(firstTableView->**selectionModel**());。

默认的delegate实现在QStyledItemDelegate类中提供。Qt provides [QItemDelegate](http://doc.qt.io/qt-5/qitemdelegate.html), which inherits [QAbstractItemDelegate](http://doc.qt.io/qt-5/qabstractitemdelegate.html) and handles the most common data types (notably int and [QString](http://doc.qt.io/qt-5/qstring.html)). QStyledItemDelegate和QItemDelegate的区别就是前者使用当前的style来paint items。如果在QStyledItemDelegate子类中调用了QStyledItemDelegate::paint就会进行原来样式的绘制，所以这个函数应该在子类的paint函数中最先进行调用，否则会进行style的覆盖，是你自己的绘制失效。

The [QStylOptionViewItem](http://doc.qt.io/qt-4.8/qstyleoptionviewitem.html) class is used to describe the parameters used to draw an item in a view widget.如果你不想自己完全对这个类的对象进行设置，就使用initStyleOption将index当前的样式初始化给自己创建的对象。

视图有一个setItemDelegate()方法，替换默认的委托，然后安装用户自定义的委托。一个新的委托可以通过创建一个继承于 QStyledItemDelegate的类来实现，如果想要自己进行渲染就要重载下面的函数：

void paint(QPainter \*painter, const QStyleOptionViewItem &option,

const QModelIndex &index) const; 委托于cell的显示

QSize sizeHint(const QStyleOptionViewItem &option,

const QModelIndex &index) const; 委托于cell的大小设置

sizeHint是QWidget的成员函数，QStyledItemDelegate::sizeHint is useful only when QTableView::resizeRowsToContents, QTableView::resizeRowToContents, QTableView::resizeColumnsToContents and QTableView::resizeColumnToContents are called：

QHeaderView\* headerView = tableView->horizontalHeader();

headerView->setResizeMode(QHeaderView::ResizeToContents);

QStyledItemDelegate 类，并且重写 paint() 同时也要重写 sizeHint() 函数。paint() 函数会被每一个item独立调用，而sizeHint()函数则可以定义每一个item 的大小。在重写 paint() 函数的时候，通常需要用 if 语句找到你需要进行渲染的数据类型并进行绘制，其他的数据类型需要调用父类的实现进行绘制。

QString text = index.model()->data(index, Qt::EditRole).toString(); 每一个index可以有多种身份，所以data需要指定身份。

paint中常见的是QApplication::style()->drawPrimitive(xxx)

也可以实现自己的paint函数：

void StarRating::paint(QPainter \*painter, const QRect &rect,

const QPalette &palette, EditMode mode) const

{

painter->save();

painter->setRenderHint(QPainter::Antialiasing, true);

painter->setPen(Qt::NoPen);

if (mode == Editable) {

painter->setBrush(palette.highlight());

} else {

painter->setBrush(palette.foreground());

}

int yOffset = (rect.height() - PaintingScaleFactor) / 2;

painter->translate(rect.x(), rect.y() + yOffset);

painter->scale(PaintingScaleFactor, PaintingScaleFactor);

for (int i = 0; i < myMaxStarCount; ++i) {

if (i < myStarCount) {

painter->drawPolygon(starPolygon, Qt::WindingFill);

} else if (mode == Editable) {

painter->drawPolygon(diamondPolygon, Qt::WindingFill);

}

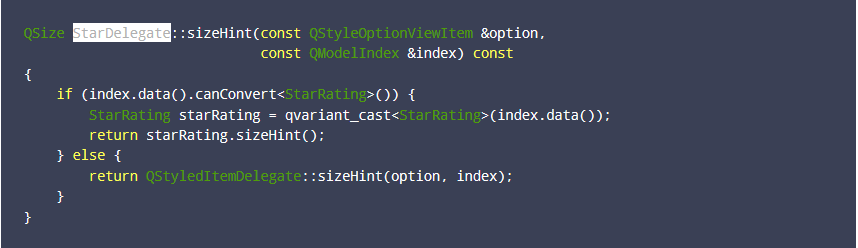
painter->translate(1.0, 0.0);

}

painter->restore();

}

StarRating 很明显就是自己设定的显示类，所以需要自己设定好sizeHint

以后对StarDelegate的sizeHint进行重载：

仅仅是设置checkbox的话，那就：



void paint(QPainter \*painter, const QStyleOptionViewItem &option,

const QModelIndex &index) const override;

QSize sizeHint(const QStyleOptionViewItem &option,

const QModelIndex &index) const override;

上面函数会直接影响QTableView的显示，最好实现sizeHint()，但是不实现对于一般的要求不影响效果，而下面的函数只有当触发编辑器的时候才会被显示。

QWidget \*createEditor(QWidget \*parent, const QStyleOptionViewItem &option,const QModelIndex &index) const override;

void setEditorData(QWidget \*editor, const QModelIndex &index) const override;

void setModelData(QWidget \*editor, QAbstractItemModel \*model,

const QModelIndex &index) const override;

如果你把函数声明最后面的const遗漏，函数名会变成斜体，但是你是重载失败的。所以函数不会被生效。



我调用了自己的居中paint之后，也调用了QItemDelegate的paint函数，验证了我上面红字的说法。

使用了paint函数以后发现，点击View表头不会再被选中标出。

如果需要自己设定编辑器，比如说弄个淘宝那样评价的星星，就需要重载createEditor、setEditorData（创建以后的值是什么）、setModelData(编辑完数据以后自动调用这个函数来更新model的值)，editingFinished信号来表示编辑已经完成。

connect(editor, &StarEditor::editingFinished,

this, &StarDelegate::commitAndCloseEditor);其实可以在setModelData函数中进行触发信号。因为这个函数已经是编辑完才会进行自动调用的函数，不过虽然这个作用可能会脱离这个函数的名字。

void StarDelegate::commitAndCloseEditor()

{

StarEditor \*editor = qobject\_cast<StarEditor \*>(sender());

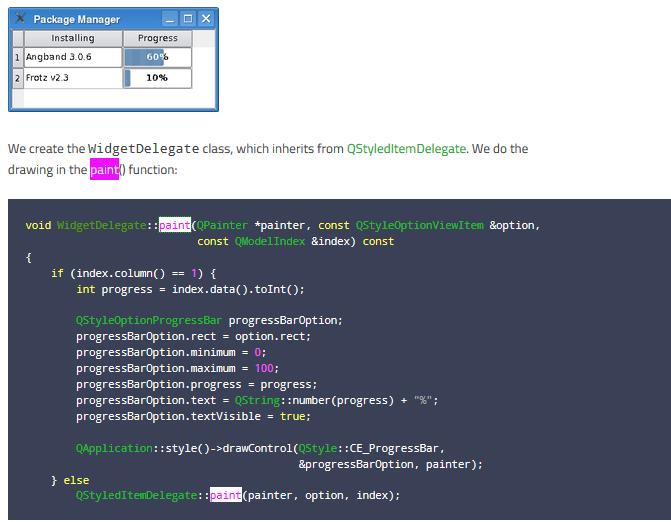
emit commitData(editor);

emit closeEditor(editor);

}

void StarDelegate::setEditorData(QWidget \*editor,

const QModelIndex &index) const



void QAbstractItemDelegate::sizeHintChanged(const [QModelIndex](http://doc.qt.io/qt-4.8/qmodelindex.html) & index)。This signal must be emitted when the [sizeHint](http://doc.qt.io/qt-4.8/qabstractitemdelegate.html" \l "sizeHint)() of index changed.

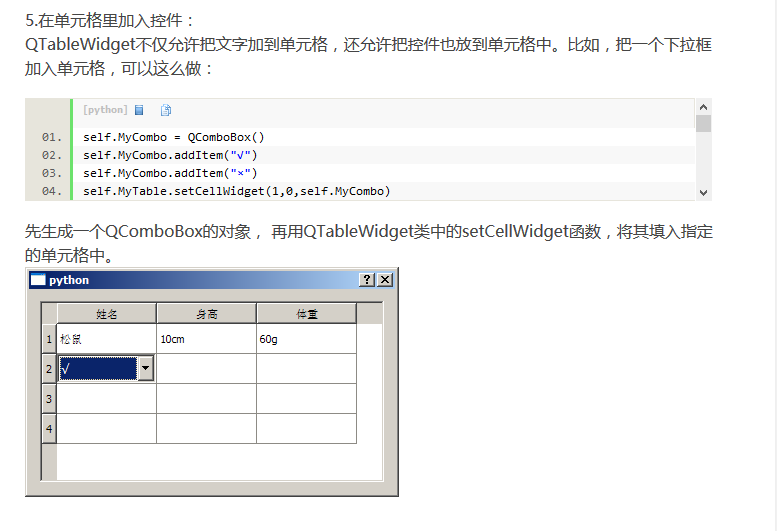
设置delegate：

m\_tableView->setItemDelegateForColumn(2, m\_tableDelegate);

m\_tableView->setItemDelegate( m\_tableDelegate);

同一个单元格不可以有多个delegate

原来QTableWidget 这种也是可以通过setItemDelegate来指定委托，毕竟是继承自QTableView。或者直接使用setCellWidget设置，如下图：



QTableWidget设置内容： QTableWidgetItem 与 setItem

newItem = QTableWidgetItem("60g")

m\_tableWidget.setItem(0, 2, newItem)

感觉从性能上来讲或许应该使用QTableView以及setmodel函数更好。

排序：

在model/view架构中，有两种方法进行排序，选择哪种方法依赖于你的底层Model。  
假如你的model是可排序的，也就是它重新实现了QAbstractItemModel::sort()函数，QTableView与QTreeView都提供了API,允许你以编程的方式对Model数据进行排序。另外，你也可以进行交互方式下的排序（例如，允许用户通过点击view表头的方式对数据进行排序），可以这样做：把QHeaderView::sectionClicked()信号与QTableView::sortByColum()槽或QTreeView::sortByColumn()槽进行联结就好了。  
另一种方法是，假如你的model没有提供需要的接口或是你想用list view表示数据，可以用一个代理model在用view表示数据之前对你的model数据结构进行转换。

使用m\_view->setSortingEnabled(true);直接使用内置的点击表头排序功能

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

 // Unindented for quoting purposes:  
 QStringList numbers;  
 numbers << "One" << "Two" << "Three" << "Four" << "Five";

 QAbstractItemModel \*model = new StringListModel(numbers);  
 //要注意的是，这里把StringListModel作为一个QAbstractItemModel来使用。这样我们就可使用model中的抽象接口，而且如果将来我们用别的model代替了当前这个model,这些代码也会照样工作。  
 QListView \*view = new QListView;  
 view->**setModel**(model);  
  view->**show**();  
  return app.exec();  
}

qcheckbox可选地提供了第三种状态——半选，通过setTristate()使之生效，通过checkState()来查询当前的状态。如果使用style()->*drawPrimitive*(QStyle::PE\_IndicatorCheckBox, &option, painter)绘制，option需要或上QStyle::State\_NoChange，剩下的两种状态在QStyle中是QStyle:State\_On、QStyle:State\_Off。对于一般的QCheckBox，三种状态分别是Qt:Checked，Qt::PartiallyChecked，Qt:Unckecked。

**在QTableView中实现QCheckBox：**

除了实现常规功能的复选框，还需要注意以下三态：不选/半选/全选

实现的方法： MVD三个方面都可以实现，不过一般是通过委托

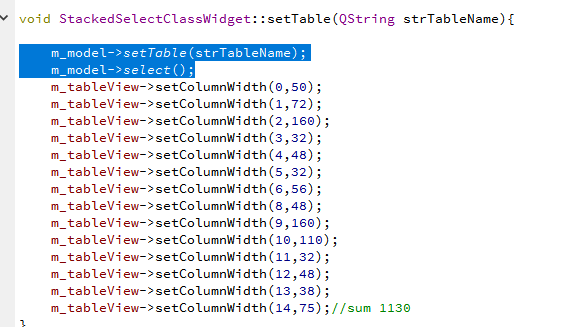
* 编辑委托。   
  方式：利用委托重载createEditor()，激活QCheckBox。   
  特点：必须双击/选中将cell变成委托的widget，才能显示CheckBox控件。一般不满足实际中的直接显示的需要。
* 使用QTableView的setIndexWidget(const QModelIndex &index, QWidget \*widget)来实现。 就是将索引列变成自己制定的方式。所以在交互性上没有得到提高。  
  此功能用来显示可视区域内对应一个数据项的静态内容。如果想显示自定义的动态内容或执行自定义编辑器部件，子类化QItemDelegate代替。也就是说，这只适合做静态数据的显示，不适合做一些插入、更新、删除操作的数据显示。
* 自定义模型QAbstractTableModel，通过flags()函数来实现。   
  方式：通过将flags()设置为Qt::ItemIsUserCheckable实现可选中，然后配合setData()与data()来实现。   
  特点：直接显示，可定义样式，默认左对齐，很难实现居中、右对齐。
* 自定义委托QAbstractItemDelegate，通过paint()函数来实现。   
  方式：通过控制editorEvent()实现鼠标的点击进行全选/半选/不选，然后由paint()实时绘制。 推荐  
  特点：这种方式比较复杂，但适合扩展，除了可以嵌入复选框，还可以绘制其它控件-按钮、图片等。

m\_model->*setTable*(strTableName);

m\_model->*select*();

对model进行限制m\_model->*setFilter*("1 limit 20");

因为select改变的时候，同时也会造成相关联的tableviwe改变，那么就要重新对各个列的宽度进行设置：



通过Model设置flag的方法设置CheckBox：

#define CHECK\_BOX\_COLUMN 0

//设置flags函数设置单元格的标志，使得 表格可选中、可复选

Qt::ItemFlags TableModel::flags(const QModelIndex &index) const

{

if (!index.isValid())

return QAbstractItemModel::flags(index);

Qt::ItemFlags flags = Qt::ItemIsEnabled | Qt::ItemIsSelectable | QAbstractTableModel::flags(index);

if (index.column() == CHECK\_BOX\_COLUMN)

flags |= Qt::ItemIsUserCheckable;

return flags;

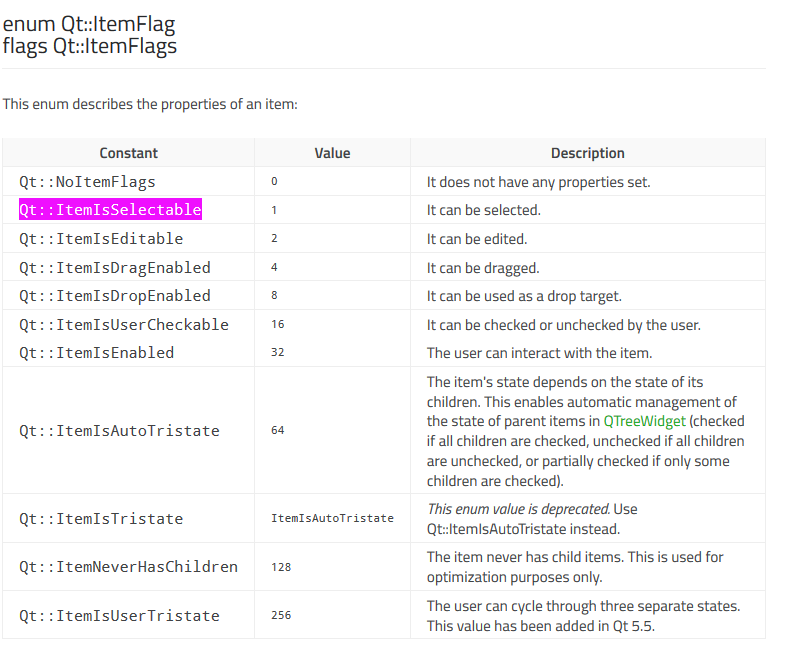
}

Qt::ItemFlags MyModel::flags(const QModelIndex &index) const

{

return Qt::ItemIsEditable | QAbstractTableModel::flags(index);

}



Model的其余函数：

下面四个函数都涉及role这个参数，进行角色的设置。

 updateData   
主要用于更新数据，刷新界面。

 data 也是返回一个值

用来显示数据，根据角色（颜色、文本、对齐方式、选中状态等）返回需要显示的数据

case Qt::CheckStateRole://如果返回是Qt::Checked就会显示一个已经选中的复选框

{

if (nColumn == CHECK\_BOX\_COLUMN)

return xxxx ? Qt::Checked : Qt::Unchecked;

}

 setData 每一次用户对内容进行编辑都会触发一次函数  
用来设置数据，根据角色（颜色、文本、对齐方式、选中状态等）触发不同的改变信号。index参数告诉我们哪一个字段被修改，value参数则提供了修改的结果

case Qt::CheckStateRole:

{

if (nColumn == CHECK\_BOX\_COLUMN)

{//CheckBox状态发生变化

先进行自己对成员中值的操作然后出发值改变信号，这个信号的作用就是让View知道数据发生了变化

emit dataChanged(index, index); //参数是左上角坐标与右下角坐标。信号告知view数据发生更改，不需要我们自己设定槽函数。这个信号完整来讲还有第三个参数， const [QVector](qvector.html)<int> &roles = QVector<int> ()，不过一般是使用默认值。

注意事项：

如果指定的范围里面item不是同一个父类，那么行为未定义。

这函数只能用在已经存在的数据上，所以如果是自己添加的数据，就需要及时使用下面这两个函数beginInsertRows(QModelIndex(), people.count(), people.count());和endInsertRows();。

return true;

}

}

如果角色有多种的话，要求model必须emit dataChanged()，否则的话还是可以仅仅使用自定义的槽函数。

 headerData   
用来显示水平/垂直表头的数据。

case Qt::DisplayRole:

{

if (orientation == Qt::Horizontal)

{

if (section == CHECK\_BOX\_COLUMN)

return QStringLiteral("状态");

}

}

role的设置：

Qt::DisplayRole 内容显示

Qt::EditRole 内容编辑

Qt:: FontRole 控制显示的字体

Qt::TextAlignmentRole，控制元素 显示文字的对齐方式的

Qt::TextColorRole，控制显示文字颜色的。

Qt::BackgroundColorRole，控制背景颜色的。

Qt::CheckStateRole 设置了以后就会显示成一个CheckBox

如果需要控制样式的话，还是直接QSS。

第四种方法：自己设置委托

需要在继承自QAbstractTableModel的自定义model类中使用Qt::UserRole(它的作用就是值的记录，但是我建议将值的记录直接交给自定义delegate，这样可以增加这个CheckboxDelegate类的重用性)，然后在自定义的Delegate（可以继承自QStyledItemDelegate），重载paint

函数进行绘制，重载editorEvent来响应鼠标事件：

**下面是通过设置绘图按钮QStyleOptionButton来模拟一个复选框：**

void DelReconQueue::paint(QPainter \*painter, const QStyleOptionViewItem &option,

                           const QModelIndex &index) const

 {

      if (index.column() == 11)

     {

         //获取值  Qt::DisplayRole 是这个函数的默认值

         bool checked = index.model()->data(index, Qt::DisplayRole).toBool();

         //按钮的风格选项

         QStyleOptionButton \*checkBoxOption = new QStyleOptionButton();

         checkBoxOption->state |= QStyle::State\_Enabled;

         //根据值判断是否选中

         if(checked)

         {

             checkBoxOption->state |= QStyle::State\_On;

         }

         else

         {

             checkBoxOption->state |= QStyle::State\_Off;

         }

         //返回QCheckBox几何形状

         checkBoxOption->rect = CheckBoxRect(option);

         //绘制QCheckBox

         **QApplication::style()->drawControl**(QStyle::CE\_CheckBox,checkBoxOption,painter);

    //注意是QStyle::CE\_系列

     }



     else

     {

         //否则调用默认委托

         QStyledItemDelegate::paint(painter, option, index);

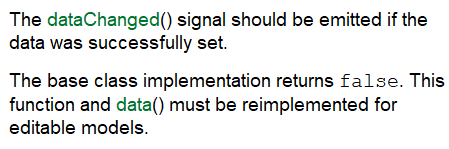
     }

       delete checkBoxOption ;

 }

我试了一下上面的三个参数，程序报错。下面的四个参数的示例倒是可以，但是示例中都是进行局部变量的分配，这可能会出现问题。最后一个参数是**[QWidget](qwidget.html)** \*widget = Q\_NULLPTR 这个参数是可选的，进行可能的辅助绘制，以防风格需要它来执行特殊效果。

QStyle::CE\_CheckBox：A QCheckBox, draws a PE\_IndicatorCheckBox, a CE\_CheckBoxLabel and a PE\_FrameFocusRect. 创建的QCheckBox并不是真正的对象，因为不能被qss 生效，只是创建一个图形，自身不具有响应机制所以要配合editorEvent重载函数使用。这是除了进行委托设置的第二种方法，也是最简单的方法。这种方法调用



QStyle::PE\_IndicatorCheckBox：On/off indicator, for example, a QCheckBox.

QStyle::CE\_CheckBoxLabel：The label (text or pixmap) of a QCheckBox.

QStyle::PE\_FrameFocusRect：Generic focus indicator.

QCheckBox是一个控件，我们使用drawControl进行绘制。绘制一个QCheckBox等同于绘制一个PE\_IndicatorCheckBox，一个CE\_CheckBoxLabel，和一个PE\_FrameFocusRect。这里算是通过QStyle进行绘制。 暂时出现了问题就是不能使用QSS对创建的CheckBox进行style调整。

void CheckBoxDelegate::paint(QPainter \*painter, const QStyleOptionViewItem &option, const QModelIndex &index) const

{

QStyleOptionViewItem viewOption(option);

initStyleOption(&viewOption, index);

if (option.state.testFlag(QStyle::State\_HasFocus))

viewOption.state = viewOption.state ^ QStyle::State\_HasFocus;

QStyledItemDelegate::paint(painter, viewOption, index);

if (index.column() == CHECK\_BOX\_COLUMN)

{

bool data = index.model()->data(index, Qt::UserRole).toBool();

如果不存在这个角色，那么index.model()->*data*(index, Qt::UserRole)是QVariant(Invalid)，toBool()以后就是false。

QStyleOptionButton checkBoxStyle;

checkBoxStyle.state = data ? QStyle::State\_On : QStyle::State\_Off;

checkBoxStyle.state |= QStyle::State\_Enabled;

checkBoxStyle.iconSize = QSize(20, 20);

//听说这里默认的大小是(-1,-1)这会导致icon这个公开成员不可见，还有一个公开成员是text（button上面的字）。

checkBoxStyle.rect = option.rect;

QCheckBox checkBox;

QApplication::style()->drawPrimitive(QStyle::PE\_IndicatorCheckBox, &checkBoxStyle, painter, &checkBox);

}

}//四个参数，QStyleOptionButton配合QCheckBox使用

|  |  |
| --- | --- |
| virtual void | **[drawPrimitive](qstyle.html" \l "drawPrimitive)**(PrimitiveElement *element*, const QStyleOption \**option*, QPainter \**painter*, const QWidget \**widget* = Q\_NULLPTR) const = 0 |

但是上面这种方法出现了问题就是不能使用QSS，估计是Qt Creator的版本问题？只能进行使用配合QStyle的QPainter通过QPixmap进行绘制：

void CheckboxDelegate::*paint*(QPainter \*painter, const QStyleOptionViewItem &option, const QModelIndex &index) const {

QPixmap pixmap; int column =index.column();

if(column==IFSEND\_COLUMN) {

if(index.model()->*data*(index).toBool()) pixmap.load(":/Resources/MainWindow/checkedHook.png");

else pixmap.load(":/Resources/MainWindow/uncheckedHook.png");

}else if(column==IFSTAR\_COLUMN) {

if(index.model()->*data*(index).toBool()) pixmap.load(":/Resources/MainWindow/checkedStar.png");

else pixmap.load(":/Resources/MainWindow/uncheckedStar.png");

}

QApplication::style()->*drawItemPixmap*(painter, option.rect, Qt::AlignHCenter|Qt::AlignVCenter, pixmap );

}

下面这对代码对格式进行设置：

QStyleOptionViewItem viewOption(option);

*initStyleOption*(&viewOption, index);

if (option.state.testFlag(QStyle::State\_HasFocus))

viewOption.state = viewOption.state ^ QStyle::State\_HasFocus;

QStyledItemDelegate::*paint*(painter, viewOption, index);

下面是我最终使用的类：

CheckboxDelegate::CheckboxDelegate(QObject \*parent) :

QAbstractItemDelegate(parent)

{

}

void CheckboxDelegate::*paint*(QPainter \*painter, const QStyleOptionViewItem &option, const QModelIndex &index) const {

QPixmap pixmap; int column =index.column();

if(column==IFSEND\_COLUMN) {

if(index.model()->*data*(index).toBool()) pixmap.load(":/Resources/MainWindow/checkedHook.png");

else pixmap.load(":/Resources/MainWindow/uncheckedHook.png");

}else if(column==IFSTAR\_COLUMN) {

if(index.model()->*data*(index).toBool()) pixmap.load(":/Resources/MainWindow/checkedStar.png");

else pixmap.load(":/Resources/MainWindow/uncheckedStar.png");

}

QApplication::style()->*drawItemPixmap*(painter, option.rect, Qt::AlignHCenter|Qt::AlignVCenter, pixmap );

}

//QModelIndex(1,3,0x0,QSqlTableModel(0x195870a0))

//QRect(50,30 49x29) option.rect输出有点神奇，这种方法下是从左上角到被点击的按钮这么大

bool CheckboxDelegate::*editorEvent*(QEvent \*event, QAbstractItemModel \*model, const QStyleOptionViewItem &option, const QModelIndex &index) {

if(index.column()!=IFSEND\_COLUMN && index.column()!=IFSTAR\_COLUMN)

return QStyledItemDelegate::*editorEvent*(event, model, option, index); //没有必要对列进行限制，因为我是通过ui->tableView->setItemDelegateForColumn 来指定设置detelagte。

if ( event->type()==QEvent::MouseButtonRelease ) {

if( true==model->*setData*(index, index.model()->*data*(index).toBool()?0:1) ){

static\_cast<QSqlTableModel\*>(model)->submitAll(); //如果没有这个函数，那么数据库的改变有延迟，需要对字段改变后来回车才能提交。static\_cast<QSqlTableModel\*>(model)->*submit*(); submit默认就会提交一行，如果没有提交，那么只能对一行记录进行改变，之后的setData函数会返回false，这是因为model中默认的submit政策导致，可以通过QSqlTableModel::setEditStrategy 设置。m\_model->*setEditStrategy*(QSqlTableModel::OnManualSubmit); 调用了以后将只能使用submitAll函数提交对model的更改，submit函数会无效。 如果使用默认的修改政策，那么即便每一次修改之后都进行submit提交以及emi dataChanged信号，可能会遇到这样的问题：有一行记录会消失在model中， 那行的显示也会变成空白。解决办法就是将model submit 的 政策改变称为submitALl函数，于是整个editEvent函数只有在最后的时候才会有调用一次submitAll以及emit信号的发送。

emit model->dataChanged(index,index); // model->*setData 应该调用这个函数*

return true;

}else

return false;

}

}在Qt的AbstractItemDelegate 文档中提到过，自定义编辑有两种方式：①复写createEditor()创建一个editor widget并显示，复写setEditorData()对editor设来自model值。②重写editorEvent()来实现 对model的修改都是使用setModelData()，使用这个方法需要关注返回值

QSize CheckboxDelegate::*sizeHint*(const QStyleOptionViewItem &option, const QModelIndex &index) const {

return QSize(option.rect.width() ,option.rect.height());

}

默认情况下option.state的值是QFlags<QStyle::StateFlag>(State\_Enabled)，点击一个cell以后通过同时输出index可以知道会先变成QFlags<QStyle::StateFlag>(State\_Enabled|State\_HasFocus|State\_Selected|State\_Active)然后变成QFlags<QStyle::StateFlag>(State\_Enabled|State\_Selected)。如果你的*editorEvent*函数的返回值是return QStyledItemDelegate::*editorEvent*(event, model, option, index); 这句话会造成不小的延迟，在消息传递上多了一步，就是你点击一个新的cell以后原来的cell也会因为这句话被paint一次。

尝试通过插入办法增加一列专门作为checkbox：

但是在model中只能进行insertRecord、insertRows、romoveRows、removeColumns不存在对column的insert操作？只是QSqlTableModle的帮助里面没有，其实QAbstractItemModel有：

bool QAbstractItemModel::insertColumn(int column, const [QModelIndex](qmodelindex.html) &parent = QModelIndex())

还有个insertColumns，不过是虚函数不能直接使用。

所以m\_tableView->setColumnHidden(0,true);也可以使用m\_model->removeColumn(0);后者在代码上与m\_model->insertColumn(0);更加统一。

MaterialDelegate::MaterialDelegate(QObject \*parent) :

QAbstractItemDelegate(parent)

{

}

**通过QApplication:style()->drawControl()绘制两个QradioButton样子按钮：**

QSize MaterialDelegate::*sizeHint*(const QStyleOptionViewItem &option, const QModelIndex &index) const {

return QSize(option.rect.width() ,option.rect.height());

}

void MaterialDelegate::*paint*(QPainter \*painter, const QStyleOptionViewItem &option, const QModelIndex &index) const {

if( index.column()==IFCHOOSE\_COLUMN ) {

QStyleOptionButton radioButtonOption;

radioButtonOption.rect =QRect(20,option.rect.y(), 30,30);

radioButtonOption.state |= QStyle::State\_Enabled;

radioButtonOption.text ="是";

if(index.model()->*data*(index).toBool()) radioButtonOption.state |= QStyle::State\_On;

else radioButtonOption.state |= QStyle::State\_Off;

QApplication::style()->*drawControl*(QStyle::CE\_RadioButton, &radioButtonOption, painter);

radioButtonOption.state =QStyle::State\_Enabled;

radioButtonOption.text ="否";

radioButtonOption.rect =QRect(70,option.rect.y(), 30,30);

if(index.model()->*data*(index).toBool()) radioButtonOption.state |= QStyle::State\_Off;

else radioButtonOption.state |= QStyle::State\_On;

QApplication::style()->*drawControl*(QStyle::CE\_RadioButton, &radioButtonOption, painter);

}

}

bool MaterialDelegate::*editorEvent*(QEvent \*event, QAbstractItemModel \*model, const QStyleOptionViewItem &option, const QModelIndex &index) {

QMouseEvent\* e =(QMouseEvent\*)event;

int returnValue =true;

if( index.column()==IFCHOOSE\_COLUMN ) {

if( QRect(20,index.row()\*30, 30,30).contains(e->x(), e->y()) ) returnValue &= model->*setData*(index, 1);

else if( QRect(70,index.row()\*30, 30,30).contains(e->x(), e->y()) ) returnValue &= model->*setData*(index, 0);

else ;

model->*submit*();

emit model->dataChanged(index,index);

}

return returnValue;

}

上面QRect(20,index.row()\*30, 30,30)、QRect(70,index.row()\*30, 30,30)导致代码存在这样一个bug，如果数据多到列表出现下滑条，当你下滑了以后机会发现按钮没有点击效果了。分别改成e->x()>=20 && e->x()<=50，e->x()>=70 && e->x()<=100

**在QTableView的Header中绘制二态QCheckbox：**

#include <QHeaderView>

#include <QPainter>

class CheckBoxHeader : public QHeaderView

{

Q\_OBJECT

public:

CheckBoxHeader(Qt::Orientation orientation, QWidget\* parent = 0);

bool isChecked() const { return m\_isChecked; }

void setIsChecked(bool val);

signals:

void checkBoxClicked(bool state);

protected:

void *paintSection*(QPainter\* painter, const QRect& rect, int index) const;

void *mousePressEvent*(QMouseEvent\* event);

private:

bool m\_isChecked;

};

#include "CheckBoxHeader.h"

CheckBoxHeader::CheckBoxHeader(Qt::Orientation orientation, QWidget\* parent /\*= 0\*/)

: QHeaderView(orientation, parent)

{

m\_isChecked = true;

}

void CheckBoxHeader::*paintSection*(QPainter\* painter, const QRect& rect, int index) const

{

painter->save();

QHeaderView::*paintSection*(painter, rect, index);

painter->restore();

if (index == 0) {

QStyleOptionButton option;

option.rect = QRect(13,7,20,20);

option.state = QStyle::State\_Enabled | QStyle::State\_Active| QStyle::State\_Off;

if (m\_isChecked)

option.state |= QStyle::State\_On;

else

option.state |= QStyle::State\_Off;

style()->*drawPrimitive*(QStyle::PE\_IndicatorCheckBox, &option, painter);

}

}

void CheckBoxHeader::*mousePressEvent*(QMouseEvent\* event)

{

setIsChecked(!isChecked());

emit checkBoxClicked(isChecked());

}

void CheckBoxHeader::setIsChecked(bool val)

{

if (m\_isChecked != val) {

m\_isChecked = val;

viewport()->update();

}

}

最后的使用：

m\_checkboxHeader =new CheckBoxHeader(Qt::Horizontal, ui->tableView\_evaluation);

ui->tableView\_evaluation->setHorizontalHeader(m\_checkboxHeader);

**在QTableView的Header中绘制一个三态状态QCheckBox：**

在二态的基础上进行更改，关键是对几个信号的利用。需要表格将半选信号传送回来。

http://blog.csdn.net/liang19890820/article/details/50772562

我自己的：

#include "StackedEvaluationWidget.h"

#include <QDebug>

StackedEvaluationWidget::StackedEvaluationWidget(QWidget \*parent) :

QWidget(parent),ui(new Ui::StackedEvaluationWidget),

m\_evaluationModel(new QSqlTableModel(this)),m\_evaluationDelegate(new EvaluationDelegate(this)), m\_tableDelegate(new AutoTurnLineDelegate(this)),

m\_recordNumber(0),m\_sqlQuery(new QSqlQuery(m\_evaluationModel->database()))

{

ui->setupUi(this);

m\_checkboxHeader =new CheckBoxHeader(Qt::Horizontal, ui->tableView\_evaluation);

ui->tableView\_evaluation->setHorizontalHeader(m\_checkboxHeader);

connect(m\_checkboxHeader, SIGNAL(checkBoxClicked(int)), this, SLOT(handleHeaderCheckBoxClicked(int)) );

connect(m\_evaluationModel, SIGNAL(dataChanged(QModelIndex,QModelIndex,QVector<int>)), this, SLOT(handleDataChanged()) );

SQLErrorDetect(m\_sqlQuery, "select count(\*) from Evaluation", "获取评教信息数量失败 in StackedEvaluationWidget::StackedEvaluationWidget(QWidget \*parent)");

sqlQuery->next();

m\_recordNumber =sqlQuery->value(0).toInt();

m\_evaluationModel->*setTable*("Evaluation");

m\_evaluationModel->*select*();

m\_evaluationModel->*setHeaderData*(IFSEND\_COLUMN, Qt::Horizontal,"");

m\_evaluationModel->*setHeaderData*(PERSON\_COLUMN, Qt::Horizontal,"被评人");

m\_evaluationModel->*setHeaderData*(COURSE\_COLUMN, Qt::Horizontal,"负责课程");

ui->tableView\_evaluation->*setModel*(m\_evaluationModel);

ui->tableView\_evaluation->setColumnWidth(IFSEND\_COLUMN, 50);

ui->tableView\_evaluation->setColumnWidth(PERSON\_COLUMN, 100);

ui->tableView\_evaluation->setColumnWidth(COURSE\_COLUMN, 200);

ui->tableView\_evaluation->setItemDelegateForColumn(IFSEND\_COLUMN, m\_evaluationDelegate);

for(int i1=PERSON\_COLUMN; i1<=COURSE\_COLUMN; ++i1)

ui->tableView\_evaluation->setItemDelegateForColumn(i1, m\_tableDelegate);

handleDataChanged();

}

StackedEvaluationWidget::~*StackedEvaluationWidget*() {

delete m\_sqlQuery;

}

//鼠标点击以后的状态state只会是0或2。未选择0以及半选1点击了以后都会变成2，全选2会变成未选择0。

void StackedEvaluationWidget::handleHeaderCheckBoxClicked(int state) {

if( state==0 ) {

SQLErrorDetect(m\_sqlQuery, "update Evaluation set IfSend=0;", "设置Evaluation中全部内容为0失败 in StackedEvaluationWidget::handleHeaderCheckBox()");

}else if( state==2 ) {

SQLErrorDetect(m\_sqlQuery, "update Evaluation set IfSend=1;", "设置Evaluation中全部内容为1失败 in StackedEvaluationWidget::handleHeaderCheckBox()");

}

m\_evaluationModel->*select*();

}

void StackedEvaluationWidget::handleDataChanged() {

SQLErrorDetect(m\_sqlQuery, "select count(\*) from Evaluation where IfSend=1;", "获取评教已选信息数量失败 in StackedEvaluationWidget::handleDataChanged()");

m\_sqlQuery->next();

int nIfSend =m\_sqlQuery->value(0).toInt();

if( nIfSend==0 ) {

m\_checkboxHeader->setCheckState(0);

}else if( nIfSend==m\_recordNumber ){

m\_checkboxHeader->setCheckState(2);

}else {

m\_checkboxHeader->setCheckState(1);

}

}

#include "CheckBoxHeader.h"

CheckBoxHeader::CheckBoxHeader(Qt::Orientation orientation, QWidget\* parent /\*= 0\*/)

: QHeaderView(orientation, parent)

{

m\_checkState = 0;

}

void CheckBoxHeader::*paintSection*(QPainter\* painter, const QRect& rect, int index) const

{

painter->save();

QHeaderView::*paintSection*(painter, rect, index);

painter->restore();

if (index == 0) {

QStyleOptionButton option;

option.rect = QRect(13,7,20,20);

option.state = QStyle::State\_Enabled | QStyle::State\_Active;

switch ( m\_checkState ) {

case 0://没有任何选择

option.state |= QStyle::State\_Off;

break;

case 1://部分选择，显示为半选状态

option.state |= QStyle::State\_NoChange;

break;

case 2://全部选择，显示为勾选状态

option.state |= QStyle::State\_On;

break;

default:

break;

}

style()->*drawPrimitive*(QStyle::PE\_IndicatorCheckBox, &option, painter);

}

}

void CheckBoxHeader::*mousePressEvent*(QMouseEvent\* event)

{

if(m\_checkState!=2) {

m\_checkState =2;

}else {

m\_checkState =0;

}

viewport()->update();

emit checkBoxClicked(m\_checkState);

}

int CheckBoxHeader::getCheckState() const {

return m\_checkState;

}

//对外（parent）提供的接口

void CheckBoxHeader::setCheckState(int state) {

m\_checkState =state;

viewport()->update();

}

通过QSqlQuery改变了数据库中的内容，要想将变化显示在QTableView上，只需要使用model->select()

我自己的代码还存在一定的缺陷，估计是因为性能问题，导致快速连续点击HeaderView中的CheckBox时，过一会就会有一次无效。

**QTableView中实现右键菜单：**

1. 默认是使用**Qt::DefaultContextMenu，**重写contextMenuEvent( QContextMenuEvent \* event )函数。
2. 我们需要进行自己的定制，使用**Qt::CustomContextMenu**，重写void customContextMenuRequested ( const QPoint & pos )。对于这个信号还需要自己进行connect设置SLOT函数。

注意：pos是该widget接收右键菜单事件的位置，一般是在该部件的坐标系中。但是对于QAbstratScrollArea及其子类例外，是对应着其视口viewport()的坐标系。如常用的QTableView、QHeaderView就是QAbstratScrollArea的子类。

table=new QTableView(this);

table->setModel(model);

table->setContextMenuPolicy(Qt::CustomContextMenu);

connect(table, SIGNAL(customContextMenuRequested(QPoint)),

SLOT(customMenuRequested(QPoint)));

void ClassScheduleView::customMenuRequested(QPoint pos) {

QModelIndex index=table->indexAt(pos);

QMenu \*menu=new QMenu(this);

menu->addAction(new QAction("增加课程", this));

menu->addAction(new QAction("Action 2", this));

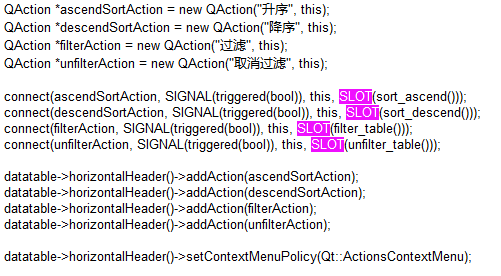
menu->addAction(new QAction("Action 3", this));

menu->popup(table->viewport()->mapToGlobal(pos));

}//之所以使用viewport()是因为可能有滚动条的出现

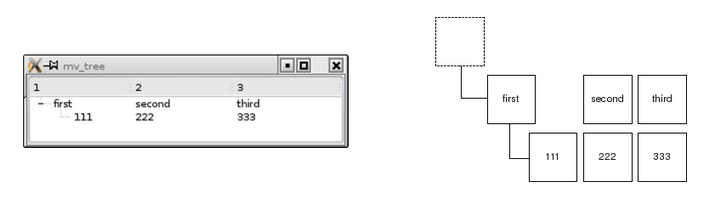
addAction是QWidget的函数

1. 还有一种是使用**Qt::ActionsContextMenu，**把部件的所有action即QWidget::actions()作为context menu显示出来。



上面的代码通过点击表头可以进行排序。

**QTreeView 与模型：**



#include <QTreeView>

#include <QStandardItemModel>

#include <QStandardItem>

#include "mainwindow.h"

const int ROWS = 2;

const int COLUMNS = 3;

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent)

{

treeView = new QTreeView(this);

setCentralWidget(treeView);

standardModel = new QStandardItemModel ;

QList<QStandardItem \*> preparedRow =prepareRow("first", "second", "third");

QStandardItem \*item = standardModel->invisibleRootItem();

// adding a row to the invisible root item produces a root element

item->appendRow(preparedRow);

QList<QStandardItem \*> secondRow =prepareRow("111", "222", "333");

// adding a row to an item starts a subtree

preparedRow.first()->appendRow(secondRow);

treeView->setModel(standardModel);

treeView->expandAll();

}

QList<QStandardItem \*> MainWindow::prepareRow(const QString &first,

const QString &second,

const QString &third)

{

QList<QStandardItem \*> rowItems;

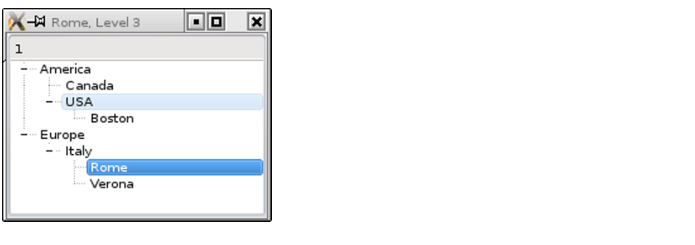
rowItems << new QStandardItem(first);

rowItems << new QStandardItem(second);

rowItems << new QStandardItem(third);

return rowItems;

}



#include <QTreeView>

#include <QStandardItemModel>

#include <QItemSelectionModel>

#include "mainwindow.h"

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent)

{

treeView = new QTreeView(this);

setCentralWidget(treeView);

standardModel = new QStandardItemModel ;

QStandardItem \*rootNode = standardModel->invisibleRootItem();

//defining a couple of items

QStandardItem \*americaItem = new QStandardItem("America");

QStandardItem \*mexicoItem = new QStandardItem("Canada");

QStandardItem \*usaItem = new QStandardItem("USA");

QStandardItem \*bostonItem = new QStandardItem("Boston");

QStandardItem \*europeItem = new QStandardItem("Europe");

QStandardItem \*italyItem = new QStandardItem("Italy");

QStandardItem \*romeItem = new QStandardItem("Rome");

QStandardItem \*veronaItem = new QStandardItem("Verona");

//building up the hierarchy

rootNode-> appendRow(americaItem);

rootNode-> appendRow(europeItem);

americaItem-> appendRow(mexicoItem);

americaItem-> appendRow(usaItem);

usaItem-> appendRow(bostonItem);

europeItem-> appendRow(italyItem);

italyItem-> appendRow(romeItem);

italyItem-> appendRow(veronaItem);

//register the model

treeView->setModel(standardModel);

treeView->expandAll();

//selection changes shall trigger a slot

QItemSelectionModel \*selectionModel= treeView->selectionModel();

connect(selectionModel, SIGNAL(selectionChanged (const QItemSelection &, const QItemSelection &)),

this, SLOT(selectionChangedSlot(const QItemSelection &, const QItemSelection &)));

}

Tree怎么显示与item的appendRow()密切相关

void MainWindow::selectionChangedSlot(const QItemSelection & /\*newSelection\*/, const QItemSelection & /\*oldSelection\*/)

{

//get the text of the selected item

const QModelIndex index = treeView->selectionModel()->currentIndex();

QString selectedText = index.data(Qt::DisplayRole).toString();

//find out the hierarchy level of the selected item

int hierarchyLevel=1;

QModelIndex seekRoot = index;

while(seekRoot.parent() != QModelIndex())

{

seekRoot = seekRoot.parent();

hierarchyLevel++;

}

QString showString = QString("%1, Level %2").arg(selectedText)

.arg(hierarchyLevel);

setWindowTitle(showString);

}

**QTreeView与数据库：**

如果将QSqlTableModel设置给QTreeView，那么最后还是按照Table的样子显示出来。Qt并不知道如何将一个Table关系转为Tree关系。所以要想连接数据库只要两种办法：1.使用proxy model，并实现映射函数 2.使用继承自QAbstractItemModel的类，比如QStandardItemModel，这种方法需要一个个item添加进去。

通过 下面的程序能够更加了解TreeView：

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <QDebug>

MainWindow::MainWindow(QWidget \*parent) :

QMainWindow(parent),

ui(new Ui::MainWindow),

m\_model(new QStandardItemModel)

{

ui->setupUi(this);

m\_model->setHorizontalHeaderLabels(QStringList()<<QStringLiteral("项目名")<<QStringLiteral("信息"));

QStandardItem\* itemProject = new QStandardItem(QStringLiteral("项目1"));

m\_model->appendRow(itemProject);

QStandardItem\* itemChild = new QStandardItem(QStringLiteral("文件夹1"));

itemProject->appendRow(itemChild);

itemProject->setChild(0/\*itemChild->index().row()\*/,1,new QStandardItem(QStringLiteral("信息说明")));

QList<QStandardItem\*> list;

list.append(itemChild);

itemChild->appendColumn(list);//但是 并不是column的形式加到后面

QStandardItem\* itemProject2 = new QStandardItem(QStringLiteral("项目2"));

m\_model->appendRow(itemProject2);

ui->treeView->*setModel*(m\_model);

}

MainWindow::~*MainWindow*()

{

delete ui;

}

void MainWindow::on\_treeView\_clicked(const QModelIndex &index)

{

QString str;

str += QStringLiteral("当前选中：%1\nrow:%2,column:%3\n").arg(index.data().toString())

.arg(index.row()).arg(index.column());

str += QStringLiteral("父级：%1\n").arg(index.parent().data().toString());

str += QStringLiteral("\tId=%1").arg(index.internalId());

qDebug() << str;

}

关键是on\_treeView\_clicked这个函数的实现

同一层（不同row、不同column）里面就是同一个internalId

初始化QStandardItem的时候能够传入图标参数

**QT代理模型： 常用基类QAbstractProxyModel**

学习网址：<http://wiki.qt.io/Proxy_model_example_code> 使用了map

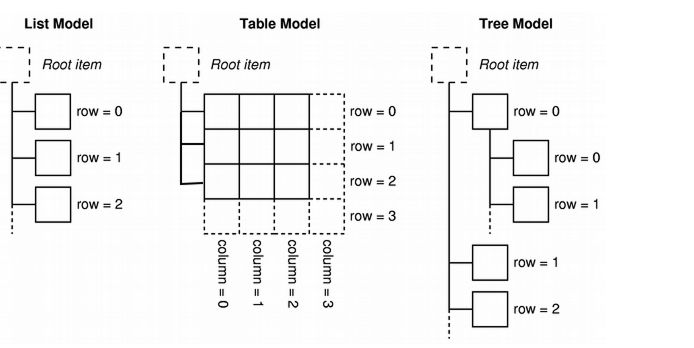
代理模型的职责是模型index的转化，由此映射产生一个新的模型。类似于数据库里面的视图，是对规则的描述。一般适用于过滤，排序功能，所以qt内部提供QSortFilterProxyModel。下面是使用示例：

proxy = new QSortFilterProxyModel(this);

proxy->setSourceModel(model);

view.setModel(proxy);

**将后台数据库中的Table格式数据显示到QTreeView：**



QModelIndex()表示最顶层的无效QModelIndex。

Tree Model的ModelIndex 也有column这个属性。

QDirModel可以同时通过QTreeView、QListView、QTableView显示。详见《QT5开发及实例》P288，似乎是将Tree Model通过TableView以及ListView显示，关键在于信号连接语句：

QObject::connect(&tree, SIGNAL(doubleClicked(QModelIndex)), &list, SLOY(setRootIndex(QModelIndex)))

QObject::connect(&tree, SIGNAL(doubleClicked(QModelIndex)), &table, SLOY(setRootIndex(QModelIndex)))

如果仅仅是使用QSqlQueryModel，那么显示出来的还是一张表格形状。所以可以知道MVD里面并不是仅仅View影响了显示，因为中间全是自动。话操作，所以model的类型也会有影响 。要想成功，（1）使用QstandardItemModel甚至根据qt提供的simple tree model来编写自己的model类 学习网址：<http://www.qtforum.org/article/33291/qsqlquerymodel-and-qtreeview.html> <https://forum.qt.io/topic/84476/consulta-sql-a-qtreeview/4> （2）通过QAbstractProxyModel对模型进行转化。将自己的Table类型的model中的ModelIndex通过代理映射称为应该的tree model的Index排列。好处在于保留了setFilter这种方便的操作。

性能上两种差不多，看起来都是将数据一点点填充进去。

class TableToTreeProxyModel : public QAbstractProxyModel

{

public:

TableToTreeProxyModel(QObject \*parent = 0);

//QAbstractProxyModel 虚函数

QModelIndex *mapToSource*(const QModelIndex &proxy\_index) const;

QModelIndex *mapFromSource*(const QModelIndex &source\_index) const;

//QAbstractItemModel 虚函数

QModelIndex *index*(int row,int column,const QModelIndex &proxy\_index) const;

QModelIndex *parent*(const QModelIndex & proxy\_child) const;

int *rowCount*(const QModelIndex &proxy\_parent) const;

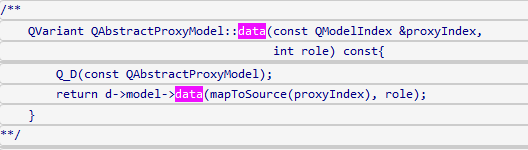
int *columnCount*(const QModelIndex &proxy\_parent) const;

public:

QVector<QString> m\_vContent;

};

data函数没重载，因为QAbstractProxyModel已经实现：



当视图对象需要访问代理模型中的某个虚拟数据项PC时，会首先调用代理模型的成员函数index()，请求代理模型为这个虚拟数据项构建一个索引ModelIndex。

模型都使用无效索引（ QModelIndex() ）作为它们的不可见根节点的索引。

毫无变动的投影ProxyModel：将TableModel 投影到 TreeView上

QModelIndex TableToTreeProxyModel::*mapToSource*(const QModelIndex &proxy\_index) const {

return sourceModel()->*index*(proxy\_index.row(), proxy\_index.column());

}

QModelIndex TableToTreeProxyModel::*mapFromSource*(const QModelIndex &source\_index) const {

return createIndex(source\_index.row(), source\_index.column(), source\_index.internalId());

}

//QAbstractItemModel 虚函数

QModelIndex TableToTreeProxyModel::*index*(int row,int column,const QModelIndex &parent ) const {

return createIndex(row, column);

}

QModelIndex TableToTreeProxyModel::*parent*(const QModelIndex &proxy\_child) const{

return *mapFromSource*(*mapToSource*(proxy\_child).parent());

}

int TableToTreeProxyModel::*rowCount*(const QModelIndex &proxy\_parent) const{

return sourceModel()->*rowCount*(*mapToSource*(proxy\_parent));

}

int TableToTreeProxyModel::*columnCount*(const QModelIndex &proxy\_parent) const{

return sourceModel()->*columnCount*(*mapToSource*(proxy\_parent));

}这时候发现原来rowCount、columnCount函数的返回值并不是0。

在填充数据的时候parent函数会在index函数之前,与用于作为index函数的第三个参数传递有关系，不过在一开始填充headerData的时候，parent这个参数是默认参数即QModelIndex()。

关于继承QAbstractItemModel的时候，必须实现如下几个函数：index(), parent(), rowCount(), columnCount(), data()， 要让Model变成可以编辑的话，必须还要实现 setData(), flags() 这两个函数，让flags()返回值有ItemIsEditable。　同时，还可以实现headerData()和 setHeaderData() 来控制View中的标题。

发现TreeView中内部节点QModelIndex只有当你点击了展开以后才会创建。

在TreeView中对于poryxmodel产生的内容，似乎不能成功执行setColumnWidth函数。

模型的被动特性给程序员带来了新的挑战。模型中的不一致性可能会导致应用程序崩溃。因为模型被大量的视图类的函数调用触发，所以很难找到哪一个函数调用导致程序崩溃，以及哪一个操作造成了这个问题。可以使用调试。

QT labs提供了叫做ModelTest的软件，当你的程序在运行时，这个软件会检查模型。每一次模型改变的时候，ModelTest都会扫描这个模型并且通过assert来报告错误。对于树形视图尤其重要，因为它们的层级特性遗留了很多不一致性的问题产生的可能性。不像视图类一样，ModelTest使用有效范围之外的索引值来测试模型。这就意味着你的应用程序在不用ModelTest时能够完美运行，但是在用ModelTest时可能会导致崩溃。因此当你用ModelTest时你需要处理所有有效范围之外的索引值。

#include "TableToTreeProxyModel.h"

#include <QDebug>

//现在的问题就是第一层的内容不会被加上第二层的内容

TableToTreeProxyModel::TableToTreeProxyModel(QObject \*parent)

: QAbstractProxyModel(parent)

{

}

TableToTreeProxyModel::~*TableToTreeProxyModel*() {

delete m\_sqlQuery;

}

bool TableToTreeProxyModel::init(void) {

m\_sqlQuery =new QSqlQuery(static\_cast<QSqlTableModel\*>(sourceModel())->database());

QString filter = static\_cast<QSqlTableModel\*>(sourceModel())->filter();

SQLErrorDetect(m\_sqlQuery, "select Attribute from PlanCourses where " +filter+" group by Attribute;", "获取方案课程属性失败 in TableToTreeProxyModel::TableToTreeProxyModel(QObject \*parent)");

while( m\_sqlQuery->next() )

m\_vData.append(m\_sqlQuery->value(0).toString());

int i1=0;

for( auto data : m\_vData ) {

SQLErrorDetect(m\_sqlQuery, "select count(\*) from PlanCourses where Attribute='"+data+"'", "获取方案课程指定属性的记录数量失败 in TableToTreeProxyModel::TableToTreeProxyModel(QObject \*parent)");

m\_sqlQuery->next();

m\_vRowCount.append(m\_sqlQuery->value(0).toInt());

m\_vIndex.append(createIndex(i1++,0));

//窗口美化,对字体美化？

}

return !m\_vData.isEmpty();

}

//QAbstractProxyModel 虚函数

//下面这个函数应该返回sourceModel中的ModelINdex

QModelIndex TableToTreeProxyModel::*mapToSource*(const QModelIndex &proxy\_index) const {

if (!proxy\_index.isValid()) return QModelIndex();

if( m\_vIndex.contains(proxy\_index) ) {

int index =proxy\_index.row(), row=0;

for(int i1=0; i1<index ; ++i1) row+=m\_vRowCount[i1];

return sourceModel()->*index*(row,0);

}else {

int index = m\_vIndex.indexOf(\*static\_cast<QModelIndex\*>(proxy\_index.internalPointer()));

int row =proxy\_index.row();

for(int i1=0; i1<index ; ++i1) row+=m\_vRowCount[i1];

return sourceModel()->*index*(row, proxy\_index.column()+1);

}

}

QModelIndex TableToTreeProxyModel::*mapFromSource*(const QModelIndex &source\_index) const {

if (!source\_index.isValid()) return QModelIndex();

int index =m\_vData.indexOf(sourceModel()->*data*(source\_index.sibling(source\_index.row(),0)).toString());

if( source\_index.column()==0 ) {

return createIndex(index,0);

}else {

int row =source\_index.row();

for(int i1=0; i1<index ; ++i1) row-=m\_vRowCount[i1];

return createIndex(row, source\_index.column()-1, const\_cast<QModelIndex\*>(&m\_vIndex[index]));

}

}

//QAbstractItemModel 虚函数

/\*

\* parent是一个临时变量，所以要根据parent找出index

\*/

QModelIndex TableToTreeProxyModel::*index*(int row,int column,const QModelIndex &parent ) const {

if(column!=0)

qDebug() << row << column << parent;

if( !parent.isValid() ) {

if(row<m\_vIndex.size()&&column==0) return m\_vIndex[row];

else return QModelIndex();

}

int index =m\_vIndex.indexOf(parent);

return createIndex(row, column, const\_cast<QModelIndex\*>(&m\_vIndex[index]));

}

QModelIndex TableToTreeProxyModel::*parent*(const QModelIndex &proxy\_child) const{

//qDebug() << "parent" << proxy\_child.row() << proxy\_child.column() << proxy\_child.internalId();Z

if( proxy\_child.internalId()==0 ) return QModelIndex();

else return \*static\_cast<QModelIndex\*>(proxy\_child.internalPointer());

}

int TableToTreeProxyModel::*rowCount*(const QModelIndex &proxy\_parent) const{

if( !proxy\_parent.isValid() ) return m\_vData.size();

else return m\_vRowCount[m\_vIndex.indexOf(proxy\_parent)];

}

/\* columnCount只能是一个数

\* 子项与父项是一个column

\* if( !proxy\_parent.isValid() ) return 1;

\* else return 4;

\* 上述代码就相当于仅仅return 1;

\*/

int TableToTreeProxyModel::*columnCount*(const QModelIndex &proxy\_parent) const{

return 4;

}

bool TableToTreeProxyModel::*hasChildren*(const QModelIndex &parent) const {

return m\_vIndex.contains(parent)||!parent.isValid();

}

Qt::ItemFlags TableToTreeProxyModel::*flags*(const QModelIndex &index) const {

if(m\_vIndex.contains(index)||!index.isValid()) return Qt::ItemIsSelectable|Qt::ItemIsEnabled;

else return Qt::ItemIsSelectable|Qt::ItemIsEnabled|Qt::ItemNeverHasChildren;

}

QVariant TableToTreeProxyModel::*data*(const QModelIndex &proxy\_index, int role) const{

if(proxy\_index.column()==0 && proxy\_index.parent().isValid() && role==Qt::DisplayRole)

return sourceModel()->*data*(*mapToSource*(proxy\_index),role).toString()=="1"?QVariant("[已修]"):"";

return sourceModel()->*data*(*mapToSource*(proxy\_index),role);

}

QVariant TableToTreeProxyModel::*headerData*(int section, Qt::Orientation orientation, int role) const {

if (orientation == Qt::Horizontal && role == Qt::DisplayRole)

switch (section) {

case 0:

return "修读情况";

case 1:

return "课程号";

case 2:

return "课程名";

case 3:

return "学分";

default:

break;

}

return QVariant();

}

void TableToTreeProxyModel::clear() {

m\_vData.clear();

m\_vIndex.clear();

m\_vRowCount.clear();

}



如果项目数量非常高，不能一次性加载它们，则只能在视图中需要时成批添加项目。 实现canFetchMore、fetchMore。

**数据存储：**

在使用Qt进行应用程序开发时，经常要自定义数据类型，而且在需要的时候还要把这些数据放到QVariant中去。因为QVariant是一个接受绝大部分类型的数据类型。为了达到这个目的，就必须使用Q\_DECLARE\_METATYPE宏：

struct struct1

{

    int a;

    double b;

};

Q\_DECLARE\_METATYPE(struct1)

如果存储的是一个·QString，qDebug()输出以后是QVariant(QString, "102031020")

比较常用的有：

* 存储数据
  + setData()
  + setItemData()
  + setUserData()
* 获取数据：
  + data()
  + itemData()
  + userData()

struct User : QObjectUserData {

int nID; // ID

QString strName; // 用户名

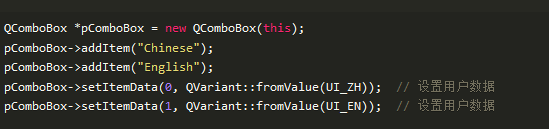
};User定义为QObjectUserData类型，为了后面setUserData()和userData()使用。如果不使用这两个接口，则不需要定义为QObjectUserData。



下面是使用正常函数的示例：



上面的用法与Android中的putExtra()一样，如果是使用了上面的宏来获取QVariant支持，那就应使用pItem->setData(Qt::UserRole,QVariant::fromValue(user))。



**Qt::UserRole：**

这个值是0x0100，这应该是所有role定义中最大的数值了

在Qt中有些类是可以给他添加角色值的，比如说QComboBox中的setItemData()与QStandardItemModel中的setData()这两个函数，默认都是在Index位置上通过Qt::UserRole添加角色值：

void QComboBox::setItemData ( int index, const QVariant & value, int role = Qt::UserRole );  
bool QStandardItemModel::setData ( const QModelIndex & index, const QVariant & value, int role = Qt::EditRole );

如果想在同一Index上添加多个角色值，我们可以利用下面的方法添加：  
QStandardItemModel \*pModel = new QStandardItemModel;  
pModel->setData(pModel->index(0, 0), 1, Qt::UserRole);  
pModel->setData(pModel->index(0, 0), 2, Qt::UserRole + 1);  
这种方法多于表格处理，或者是QComboBox中需要获取其他信息。  
在默认情况下，我们在利用这样的函数时我们并不加添加角色的，如果想利用这个功能我们就可以使用上面的方法来做。这里给大家提一个特例，QComboBox中的addItem()函数，他默认情况下是直接给Qt::UserRole这个角色添加值，而函数的原型中却没有给出这个角色，原型如下：  
 void QComboBox::addItem ( const QString & text, const QVariant & userData = QVariant() );  
Adds an item to the combobox with the given text, and containing the specified userData (stored in the Qt::UserRole). The item is appended to the list of existing items.

text是显示的字符串，userData是在后台使用的值。如果不进行自己制定的话就是传入控制那么值就是从0开始进行控制（我猜从0开始）。取值的时候使用itemData()函数进行获取。

一个QComboBox的clear（）函数，都会导致dataChanged（）信号的发射，这时候index的序号是-1.

一个QComboBox的addItem（）函数，也会导致dataChanged（）信号的发射。

**Spacer：QSpacerItem**

QBoxLayout已经设置了默认的spacing和margin

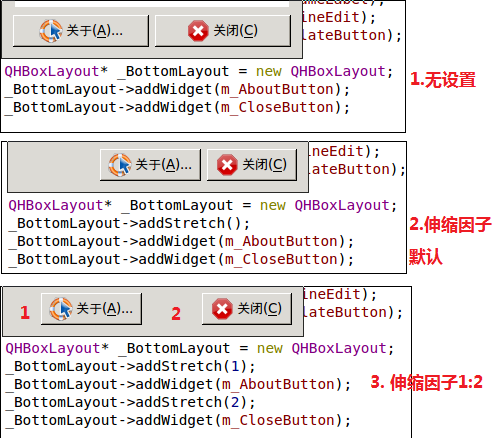
通常情况下，不需要直接使用这个类，Qt内建布局管理器在操控空白区时提供以下功能：

| **类** | **函数** |
| --- | --- |
| QHBoxLayout | addSpacing()、addStretch()、insertSpacing()、insertStretch() |
| QGridLayout | setRowMinimumHeight()、setRowStretch()、setColumnMinimumWidth()、setColumnStretch() |

1. QVBoxLayout \*buttonLayout = new QVBoxLayout;
2. buttonLayout->addStretch(1);
3. buttonLayout->addWidget(Button1);
4. buttonLayout->addStretch(1);
5. buttonLayout->addWidget(Button2);
6. buttonLayout->addStretch(1);
7. buttonLayout->addWidget(Button3);
8. buttonLayout->addStretch(6);

上面这段代码将空白部分切分为9份，然后按照比例进行分配。

如果没有加上stretch用于伸缩，那么默认情况下会把中间占据了



本质上来讲spacing和stretch都是QSpaceItem控件，占用空间用的，只是一个固定的，一个可伸缩的：

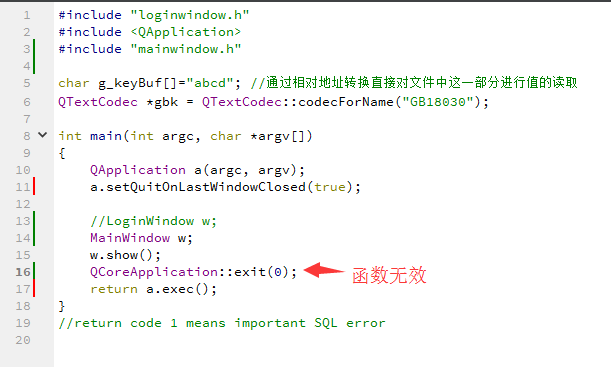
addSpacing(int size) 添加一个大小为size的不可伸缩space。

addStretch(int stretch=0) 添加一个可收缩的space

void QBoxLayout::addSpacerItem([QSpacerItem](http://doc.qt.io/qt-4.8/qspaceritem.html) \* spacerItem) 添加一个QSpacerItem

pHLayout->addSpacerItem(new QSpacerItem(20, 20, QSizePolicy::Expanding));

**退出程序：**



这个函数其实就是告诉exec()进行函数返回给出返回值。如果event loop还没有开始运行那么这个函数无效。

**国际化语言：**

QString text1 = QObject::tr("hello"); QString text2 = QString("hello");

tr是用来实现国际化，如果你为这个程序提供了中文翻译包(其中hello被翻译成中文"你好")，那么text1的内容将是中文"你好"；如果你为程序提供且使用日文翻译包，那么text1的内容将是日文。

tr是经过多级函数调用才实现了翻译操作，是有代价的，所以不该用的时候最好不要用。

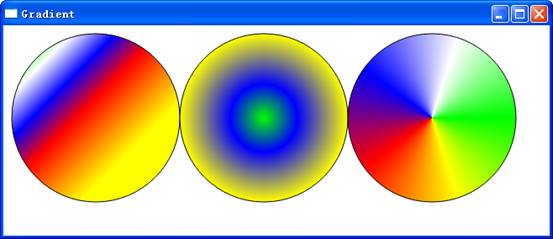
**QTextEdit：**

相对于QLineEdit这个是多行文本编辑。

获取text的函数名字有点特殊toPlainText()

**渐变色：**

线性渐变(QLinearGradient)、辐射渐变(QRadialGradient)、角度渐变(QConicalGradient)



看得出来stop是设置坐标比例？第二段代码中y2为1，所以stop设置的是y。

我比较喜欢在QSS中使用：

background:qlineargradient(spread:pad, x1:0, y1:0, x2:1, y2:0, stop:0 rgba(0, 0, 0, 255), stop:1 rgba(255, 255, 255, 255));

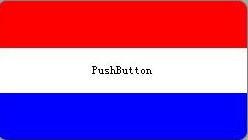


qlineargradient(spread:pad, x1:0, y1:0, x2:0,

y2:1, stop:0 rgba(255, 0, 0, 255), stop:0.339795 rgba(255, 0, 0, 255),

stop:0.339799 rgba(255, 255, 255, 255), stop:0.662444 rgba(255, 255, 255, 255),

stop:0.662469 rgba(0, 0, 255, 255), stop:1 rgba(0, 0, 255, 255))

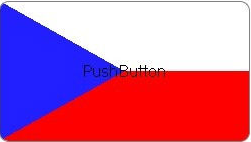


background:qconicalgradient(cx:0.5, cy:0.5, angle:0, stop:0

rgba(255, 255, 255, 255), stop:0.373979 rgba(255, 255, 255, 255), stop:0.373991

rgba(33, 30, 255, 255), stop:0.624018 rgba(33, 30, 255, 255), stop:0.624043

rgba(255, 0, 0, 255), stop:1 rgba(255, 0, 0, 255))



**QSS：**

setStyleSheet() 这个函数可以对一个变量进行设置

QStyle::drawPrimitive 不支持QSS

设置单位为px的宽度的时候，前面的数字如果是小数，就会被四舍五入为整数然后显示。

**QStyleOption Class**

The [QStyleOption](qstyleoption.html) class stores the parameters used by [QStyle](qstyle.html) functions. [More...](#details)

|  |  |
| --- | --- |
| Header: | #include <QStyleOption> |
| qmake: | QT += widgets |
| Inherited By: | [QStyleOptionButton](qstyleoptionbutton.html), [QStyleOptionComplex](qstyleoptioncomplex.html), [QStyleOptionDockWidget](qstyleoptiondockwidget.html), [QStyleOptionFocusRect](qstyleoptionfocusrect.html), [QStyleOptionFrame](qstyleoptionframe.html), [QStyleOptionGraphicsItem](qstyleoptiongraphicsitem.html), [QStyleOptionHeader](qstyleoptionheader.html), [QStyleOptionMenuItem](qstyleoptionmenuitem.html), [QStyleOptionProgressBar](qstyleoptionprogressbar.html), [QStyleOptionRubberBand](qstyleoptionrubberband.html), [QStyleOptionTab](qstyleoptiontab.html), [QStyleOptionTabBarBase](qstyleoptiontabbarbase.html), [QStyleOptionTabWidgetFrame](qstyleoptiontabwidgetframe.html), [QStyleOptionToolBar](qstyleoptiontoolbar.html), [QStyleOptionToolBox](qstyleoptiontoolbox.html), and [QStyleOptionViewItem](qstyleoptionviewitem.html) |

这些类就是提供给QStyle函数进行使用，对象里面包含了所有绘制的配置信息。下面就是一般使用的举例：

void MyPushButton::paintEvent([QPaintEvent](http://doc.qt.io/qt-4.8/qpaintevent.html) \*)

{

[QStyleOptionButton](http://doc.qt.io/qt-4.8/qstyleoptionbutton.html) option;

option.initFrom(this);

option.state = isDown() ? [QStyle](http://doc.qt.io/qt-4.8/qstyle.html)::State\_Sunken : [QStyle](http://doc.qt.io/qt-4.8/qstyle.html)::State\_Raised;

if (isDefault())

option.features |= [QStyleOptionButton](http://doc.qt.io/qt-4.8/qstyleoptionbutton.html)::DefaultButton;

option.text = text();

option.icon = icon();

[QPainter](http://doc.qt.io/qt-4.8/qpainter.html) painter(this);

style()->drawControl([QStyle](http://doc.qt.io/qt-4.8/qstyle.html)::CE\_PushButton, &option, &painter, this);

} drawControl是QStyle类的绘制函数。一般见到的是QApplication::style()->drawPrimitive。如果进行自己对这个函数的实现，由于函数参数是基类：

void MyStyle::drawPrimitive(PrimitiveElement element,

const [QStyleOption](http://doc.qt.io/qt-4.8/qstyleoption.html" \l "QStyleOption) \*option,

[QPainter](http://doc.qt.io/qt-4.8/qpainter.html) \*painter,

const [QWidget](http://doc.qt.io/qt-4.8/qwidget.html) \*widget)

{

if (element == PE\_FrameFocusRect) {

const [QStyleOptionFocusRect](http://doc.qt.io/qt-4.8/qstyleoptionfocusrect.html) \*focusRectOption =

qstyleoption\_cast<const [QStyleOptionFocusRect](http://doc.qt.io/qt-4.8/qstyleoptionfocusrect.html) \*>(option);

if (focusRectOption) {

// ...

}

}

// ...

}

所以就需要自己的在函数过程中对类型进行强制转换。使用的是

The [qstyleoption\_cast](http://doc.qt.io/qt-4.8/qstyleoption.html" \l "qstyleoption_cast)() function will return 0 if the object to which option points is not of the correct type.

许多Qt 开发人员经常会发现Qt 提供的风格不能很好的满足客户的个性化的需要，Qt 提供了三种方法来满足定制风格的需求：

（1 ）继承并实现QStyle 的子类；

（2 ）利用Qt 样式表；

（3 ）利用QGraphicsView 、QGrphicsScene 、QGraphicsItem 框架来绘制。

**QStyle：**

这个在自定义的delegate的paint函数中经常会出现，比如说增加复选框的时候就是通过

QApplication::style()->*drawPrimitive*(QStyle::PE\_IndicatorCheckBox, &checkBoxStyle, painter, &checkBox);实现。

Qstyle 类将UI 的组成元素分为三类：

（1 ）Primitive Elements : 基本元素，如边框，倒角，箭头等一些基本的，被多个控件所复用的元素。基本元素本身不能单独存在。

（2 ）Control Elements : 控件元素，如按键，复选框等，不单是这些单独的控件，也包括复杂控件中的组成元素，如滚动条中的滑块。它和基本元素不同的地方是，它能够单独处理用户交互操作。

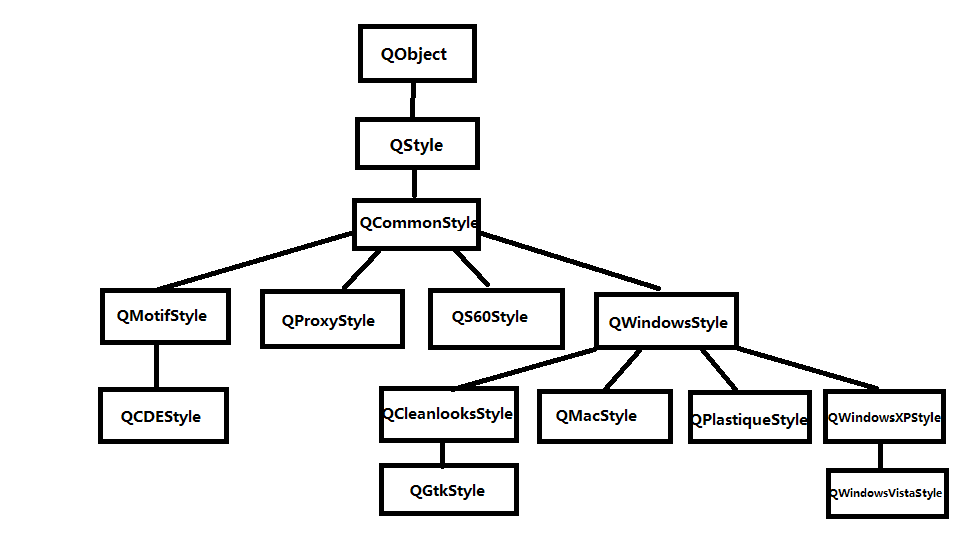
（3 ）Complex Control Elements : 复杂控件元素，复杂控件元素包含有子控件元素。例如combo boxes （复合选择框）等。它的用户交互响应，取决于用户操作的对象是哪个子控件。

QStyle 是一个封装了实现不同平台上 GUI 观感的基类，它控制了所有控件的界面风格（直接是对整个窗口的设定），定义了大量的枚举类型和十几个函数。 不仅仅可以用于widget的绘制，可以用于任何可绘制设备，但是需要对QPainter进行正确的绘制。QStyle 中只实现了 drawItem(), itemRect(), visualRect() 三个函数：

（ 1 ） drawItem() ，负责绘制文本和象素图；

（ 2 ） itemRect() ，返回文本或图像所占的区域；

（ 3 ） visualRect() ，返回逻辑坐标。



创建 style 的第一步是选择一个 Qt 提供的 style 作为父类，一般会选择 QWindowsStyle 、 QMacStyle 、 QMotifStyle 来作为父类，当然也可以选择 QCommonStyle 、 QStyle 作为父类，但是由于 QCommonStyle 、 QStyle 中许多 GUI Style 函数细节并没有实现，因此选择后者的工作量会比前者大的多。之后就是用子类进行绘制：

子类化QStyle一般是通过重新实现几个虚函数来实现的。

1）polish(QPalette& palette)：通常在此函数内指定配色方案，也即配置调色板

2）polish(QWidget\*)&unpolish(QWidget\*)：当样式应用到窗口部件时，polish(QWidget\*)就会调用，从而允许我们进行最后的定制，当动态改变样式的时候，unpolish就会调用，来撤销polish的影响。polish(QWidget\*)一般用做窗口部件的事件过滤器。

polish函数中widget->setAttribute(Qt::WA\_Hover, true);

3）styleHint返回关于样式外观的提示

功能：对指定的widget返回由option指定的样式的hint，返回值为整型。当查询的widget需要返回更多的信息，而不仅仅是由styleHint()返回的整型值时，我们就可以使用returnData来返回这些额外的信息。大致格式如下：

int styleHint(StyleHint hint, const QStyleOption \*option, const QWidget \*widget,

QStyleHintReturn \*returnData) const

{

switch (hint) {

case SH\_DitherDisabledText:

return int(false);

case SH\_EtchDisabledText:

return int(true);

default:

return QMotifStyle::styleHint(hint, option, widget, returnData);

}

}

4）pixelMetric返回像素值，来影响窗口部件的绘制

 pixelMetric(PixelMetric metric,   const QStyleOption \*option,

                        const QWidget \*widget) const

第一个参数是一个枚举类型，主要是描述了像素公制可取的一些值，一个像素公制值是单个像素在在样式中表现的尺寸。函数看起来就是对会用到的枚举值进行设定。

 {

     switch (metric) {

     case PM\_ComboBoxFrameWidth:

         return 8;

     case PM\_ScrollBarExtent:

         return QMotifStyle::pixelMetric(metric, option, widget) + 4;

     default:

         return QMotifStyle::pixelMetric(metric, option, widget);

     }

 }

5）drawPrimitive(),drawControl(),drawComplexControl()具体的执行绘制。

QApplication::style()

1. void BronzeStyle::polish(QPalette &palette)
2. {
3. QPixmap backgroundImage(":/images/background.png");
4. QColor bronze(207,155,95);
5. QColor veryLightBlue(239,239,247);
6. QColor lightBlue(223,223,239);
7. QColor darkBlue(95,95,191);
9. palette=QPalette(bronze);
10. palette.setBrush(QPalette::Window,backgroundImage);
11. palette.setBrush(QPalette::BrightText, Qt::white);
12. palette.setBrush(QPalette::Base, veryLightBlue);
13. palette.setBrush(QPalette::AlternateBase, lightBlue);
14. palette.setBrush(QPalette::Highlight, darkBlue);
15. palette.setBrush(QPalette::Disabled, QPalette::Highlight,Qt::darkGray);
17. }

**QLayout:**

对于addItem()这个函数，一般是不会使用的：This function is not usually called in application code. To add a widget to a layout, use the [addWidget](qlayout.html" \l "addWidget)() function; to add a child layout, use the addLayout() function provided by the relevant [QLayout](qlayout.html) subclass.

没有使用 m\_ok->setMinimumWidth(200);或m\_ok->setMaximumWidth(200); 来进行大小设定，那么 qDebug() << m\_ok->size(); 输出是大小和显示的对不上。 同时意识到qDebug支持很多类型的输出。

QPalette：qt 调色板

每个窗口部件都可以拥有自己的调色板，也可以更加宏观的为应用程序指定一个调色板，调色板拥有三个颜色组，为窗口部件的 不同状态定义不同的颜色，三个工作组是：

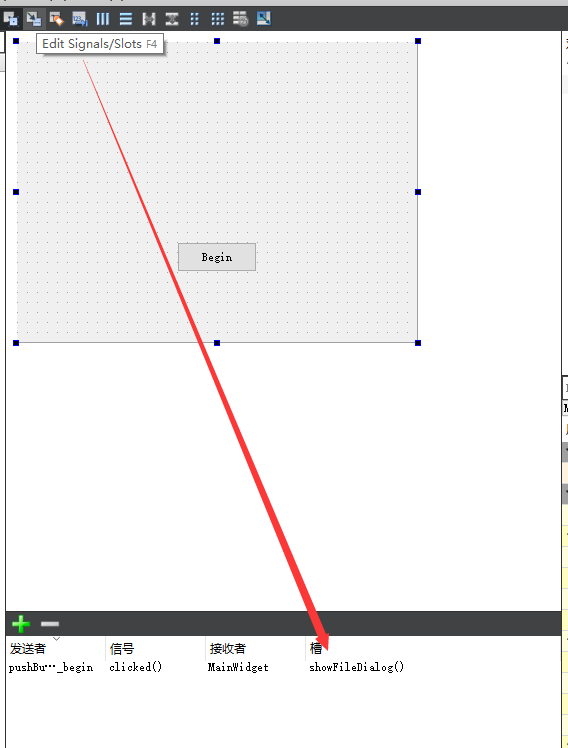
1）Active组，用于·当前活跃的窗口，即具有键盘或鼠标焦点的窗口  
2）Inactive组，其他窗口  
3）Disabled组



使用的时候就对各种性质进行设置。

想通过this->setWindowFlags(Qt::WindowMinimizeButtonHint|Qt::WindowCloseButtonHint);来实现只有最小化与关闭按钮，但是不行，还是会显示不可点击的最大化按钮。

this->setWindowFlags(Qt::Dialog); 会显示一个问号与一个关闭按钮。



UI编辑器中只能走这里才能在下面设置自定义的槽函数。

**QFileDialog：**

void MainWidget::showFileDialog(void) {

m\_fileName =QFileDialog::getOpenFileName(this);

if( !m\_fileName.isEmpty() ) {

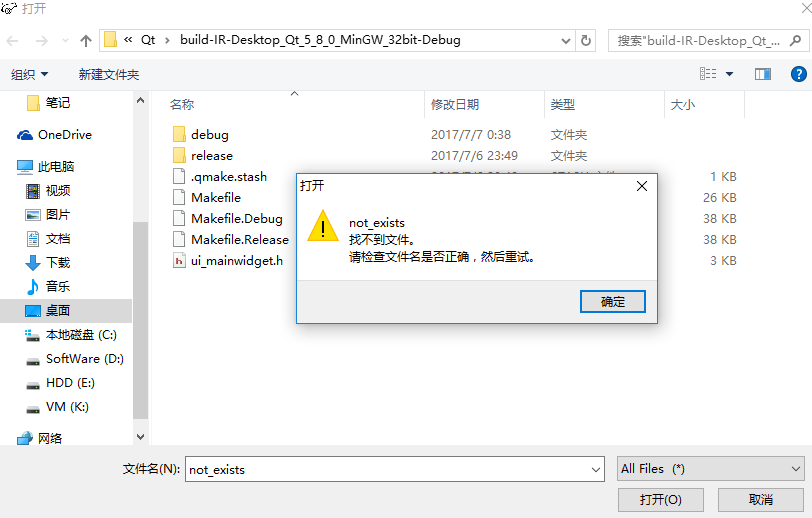
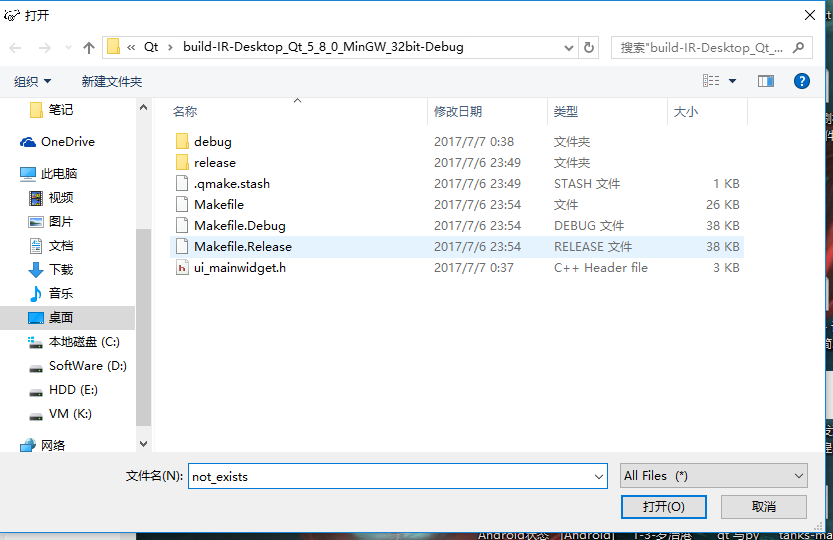
}else {

QMessageBox::critical(this, "Error", "This file is not exists",QMessageBox::NoButton);

}

}

虽然存在windows的过滤机制，但是因为qt文件对话框实现的问题，倒是windows对于不存在的文件过滤机制无效。



然后点击取消，发现m\_fileName输出为空。

win10中使用QFileDialog有在输出窗口显示：

shell\comdlg32\fileopensave.cpp(14274)\comdlg32.dll!760C2833: (caller: 760F211C) ReturnHr(1) tid(154c) 80004005 未指定的错误

CallContext:[\PickerModalLoop] 这是因为windows10系统的原因

**艺术字：**

配合字体包来实现艺术字，估计没有PS出来的艺术字图片小，但是就是扩展性、重用性高。适合分装成类。

**前景颜色设置：**

1. QSS color:white;

2.c++

QPalette   pal;  
pal.setColor(QPalette::ButtonText, QColor(255,0,0));

pushButtonCancel->setPalette(pal);

QPalete::Window,通常指窗口部件的背景色;

QPalette:WindowText,通常指窗口不见的前景色;

QPalette::Base,指文本输入窗口部件(比如QtextEdit,QLinedit等)的背景色.

QPalette::Text,与QPalette::Base一块使用,指文本输入窗口部件的前景色;

QPalette::Button,指按钮窗口部件的背景色;

QPalette::ButtonText,指按钮窗口部件的前景色

ui->pushButton\_login->setPalette(pal);

**字体设置：**

1.font.family: "楷体"; // 设置字体为楷体

2.QFont font;

font.setFamily("Microsoft YaHei");

painter.setFont(font);

上面的方法限定使用系统字体，windows就提供微软雅黑，如果要加载自己的字体：

int fontId =QFontDatabase::addApplicationFont(".ttc或者.ttf路径");

QStringList fontFamilies = QFontDatabase::applicationFontFamilies(fontId);

// 创建字体

QFont font;

font.setFamily(fontFamilies.at(0)); //FontAwesome

void MainWindow::paintEvent(QPaintEvent \*event)

{

Q\_UNUSED(event);

QPainter painter(this);

QTransform transform;

transform.rotate(45);

// 设置画笔颜色

painter.setPen(QColor(0, 160, 230));

QFont font;

font.setFamily("Microsoft YaHei");

// 大小

font.setPointSize(16);

// 斜体

font.setItalic(true);

// 设置下划线

font.setUnderline(true);

// 设置上划线

font.setOverline(true);

// 设置字母大小写

font.setCapitalization(QFont::SmallCaps);

// 设置字符间距

font.setLetterSpacing(QFont::AbsoluteSpacing, 20);

// 使用字体

painter.setFont(font);

// 获取字体信息

QFontInfo info = painter.fontInfo();

info.family();

info.italic();

// 转换

painter.setTransform(transform);

painter.drawText(rect(), QStringLiteral("一去丶二三里"));

}

直接设置整个程序的字体：

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

{

QApplication a(argc, argv);

//a.setQuitOnLastWindowClosed(true);

//LoginWindow w;

QFont font;

int fontId =QFontDatabase::addApplicationFont(":/Resources/titlewidget.ttf");

QStringList fontFamilies = QFontDatabase::applicationFontFamilies(fontId);

font.setFamily(fontFamilies.at(0));

font.setPointSize(10);

1. setFont(font);

...

}

Qt中一般数组使用QVector代替，但是QString的应该使用QStringList，因为有些函数的返回就是这个。

**QPushButton:**

选中checkable后，Button变成切换按钮(toggle button)，可以有两种状态：按下checked/弹起默认。一般状况下checkable是不选中的，Button默认为触发按钮(trigger button)，按下去马上弹起。setCheckable常常配合setChecked使用。

m\_vButton是一个QVector<QPushButton>

m\_vButton[i1]->setFont(font);

m\_vButton[i1]->setText(slistComposition.at(i1));

m\_vButton[i1]->setFlat(true); //设置为平面也就是去除边框

m\_vButton[i1]->setGeometry(10,10,200,200); 设置位置或大小只有这个函数了void QWidget::setGeometry(int x, int y, int w, int h) 如果你的w或者h比其中的内容还小，那么这个参数的设置就会无效

QSS：border-style: flat; 会导致没有焦点框：

QPushButton{

color:rgb(160,160,160);

border-style: flat;

}

QPushButton:hover:pressed{ 这里是错误的写法，应该要分开写

color:rgb(0,0,0)

}

基本配合stackedwidget进行按钮变化的，使用qss设置样式+setCheckable。因为如果使用C++来控制样式的话，需要大量的代码，QFont控制字体，QPalette控制颜色。其实性能上来讲估计差不多，QSS最终也是会转变成代码。

之前已经设置每一个按钮是checkable

void TitleWidget::handleStackButtonClick(void)

{

m\_selectedStackButtonIndex =m\_vButton.indexOf(static\_cast<QPushButton\*>(sender())););

for(int i1=0; i1<m\_vButton.size() ;++i1) {

m\_vButton[i1]->setChecked(false); }

m\_vButton[m\_selectedStackButtonIndex]->setChecked(true);

} 但是这种变化效果存在一定的延迟，主要是因为 这个框的变化是渐变的引发。

这个框的名字叫做焦点框，要想去除：

方法1：可以通过代码ui->pushButton->setFocusPolicy(Qt::NoFocus)或在Qt Creator的属性列表中设置。

方法2：如果在嵌入式设备中需要通过按键切换控件，最简单的方法就是通过控件的focus来实现，就不能使用方法1了。此时可以通过qss样式表来去掉虚线框，代码如下所示。

ui->pushButton->setStyleSheet("outline: none");

方法3：也是通过qss样式表来实现，代码如下所示。

ui->pushButton->setStyleSheet("padding: -1");

方法4: drawFocus函数

方法5：QSS 中设置border: 1px solid white; 我真的没有想到5种方法只有这种起了效果，没想到设置border就能去掉整个框。

void TitleWidget::handleStackButtonClick(void)

{

m\_selectedStackButtonIndex =m\_vButton.indexOf(static\_cast<QPushButton\*>(sender()));

QPalette pal;

pal.setColor(QPalette::ButtonText, QColor(160,160,160));

for(int i1=0; i1<m\_vButton.size() ;++i1) {

if(i1==0) m\_font.setBold(true);

else m\_font.setBold(false);

m\_vButton[i1]->setPalette(pal);

}

pal.setColor(QPalette::ButtonText, QColor(0,0,0));

m\_vButton[m\_selectedStackButtonIndex]->setPalette(pal);

//stacked widget switch

}

qt项目中显示的文件名字完全是因为pro文件中的设置，你把pro文件中的大小写改了就能将项目中的文件名也改了，同时会尝试对系统中文件名的更改，由于windows的逗逼机制，windows下文件名需要自己更改，我现在文件名认为应该设定与类名一样，大小写一样。更多pro文件中的属性：<http://blog.chinaunix.net/uid-26126915-id-4449101.html>

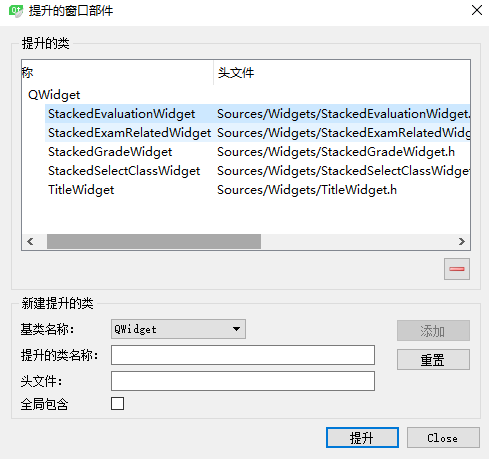
我被一个问题纠结了许久：实现一个StackedxxxWidget自定义的类，但是在犹豫到底使用C++去完成界面布局呢，还是使用Designer先定好界面布局。使用Designer很方便，而且代码也好看，本来想用Designer中的“提升为”这个功能，注意这个功能对构造函数有要求，其实就是它帮你实现了代码，但是它只会使用无参构造函数。而我必须要有参数传入给这个自定义类，如果使用Designer便需要设置一个专门的Init，我在需要传参的地方调用这个类。 提升为某一个Widget子类的时候，发现窗口的大小居然不是ui文件中被提升对象的大小。

不同点：不过StackedWidget的自定义类总共有6个，每一个都会带来一定的延时操作（即便我已经将网络操作后台化了），我一旦在一个地方进行传参并执行，那么这时候程序会卡住。如果是自己实现C++类，通过构造函数传参，只有当StackedxxxWidget 的关联button被点击的时候才会被构造。不过仔细思考了下每一个的延时应该不高才对。不过问题是即便使用了“提升为”使用了自定义控件，默认其中没有ui参数。

需要注意构造函数的执行时间也不是一样的。所以如果涉及到数据库的打开，可能需要放到

designer得到ui\_\*.h（把整个类、QSS、函数的具体实现代码都放在这里）之最先执行的，在程序窗口显示的时候

有时候使用designer的时候，也在考虑要不要加上一层对布局的控制。但是和Android一样，想想就知道每加一层布局对性能肯定不好



通过提升以后没想到最后的界面是两个ui文件的合并，最终显示的时候还是有pushButton\_3这个QPushButton。



创建了ui指针，在析构函数的时候需要delete。

:-1: error: No rule to make target '../JWCKnife/Resources/MainWindow/checkBox\_checked.png', needed by 'debug/qrc\_images.cpp'. Stop. 这个错误表示designer ui 的qss中有一个图片引用路径错误。

static\_cast<QMouseEvent\*>(event)->pos()) 这条**语句一直报**错，结果发现被颜色变化骗了，没有加上QMouseEvent的头文件。

qDebug\qWarning\ qCritical\qFatal ： 后面几个会阻止程序的继续执行

如果字符串过大，qDebug()并不会将之输出。

在qDebug中输出QString，有些情况\r\n会被转义输出，有些情况不会。

不要忘记了qt中最适合显示文字的是label控件。

**随机数：**

#include <QTime>

qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));

int result =qrand()%300;

ui->label\_result->setText(QString::number(result));

**窗口透明化：**

这几个参数也可以放到MainWindow的构造函数中

MainWindow w;

//设置整个显示窗口的不透明度，默认就是1.0

w.setWindowOpacity(1.0);

//取消窗口的标题

w.setWindowFlags(Qt::FramelessWindowHint);

//设置主widget为透明。 类似的可以使用mask实现

w.setAttribute(Qt::WA\_TranslucentBackground);

w.show();

**弹出参数设置窗口：**

如果涉及对路径的显示，使用QLineEdit：

    （1）调用lineEdit->setEnabled(false)。确实不可编辑了。不过路径太长时就只能看到后部分了。没关系，再想别的办法就是了。

    （2）setFocusPolicy(Qt::NoFocus); 设置它不可获得焦点。OK.不可编辑，又能查看完整的显示文本。



**超时限制（定时器 信号）：QTimer**

如果是timeout事件可以用stop，singleShot用stop都无效

1. 定时器信号

QTimer \*timer = new QTimer( myObject );

[connect](http://www.kuqin.com/qtdocument/qobject.html" \l "connect)( timer, SIGNAL([timeout](http://www.kuqin.com/qtdocument/qtimer.html" \l "timeout)()), myObject, SLOT(timerDone()) );

timer->[start](http://www.kuqin.com/qtdocument/qtimer.html" \l "start)( 2000, TRUE ); // 2秒单触发定时器

2. 单触发定时器（只能一个connect的关系）

QTimer::singleShot(100, &loop, SLOT(quit()));

时间到了直接触发一个信号。这种方式的缺点是不能被stop。

对于QEventLoop loop; 的结束方式（是因为超时结束还是因为成功发包有返回结束），这时候就有两种选择，其一：定时器信号以及信号处理函数，通过自己指定多个connect关系，从而设置一个内部变量来反馈信息，缺点是需要专门设置一个成员变量以及专门设置对其进行更改的函数，而且信号的传递时比较慢的。其二：在执行loop前记录时间，这时候就能设置一个单触发定时器，循环结束的时候判断时间差。 这就需同时使用QTime以及QTimer库。

Qt调试的时候进入源码查看：

需要先进行设置，选择：工具 -> 选项 -> 调试器 -> 概要在“源码路径映射”处，点进“添加 Qt 源码”按钮，选择 Qt 源码安装目录（例如：D:\Qt\Qt5.7.1\5.7\Src） 。如果不进行设置，那么只能看到堆栈的变化，但是不会跳转到源码进行可读性的人为查看。

之后就能通过调试进入Qt的源码查看内部实现机制

信号的一个比较有用的功能就是讲一个类的信号像异常一样传递出来，但是异常是按照堆栈的方式一个个传递出来，信号测试类的方式。

自定义信号的时候，并没有设置这个信号的函数。直接

signals:

void my();

signals: 而这种写法是因为都没有返回值，所以直接省略了

haveFlushClassSchedule();

之后就能用了。我自定义具体的函数体了反而说重复定义。信号不同于一般的函数，这个函数是qt内部实现，即便是有参数，那也只是在emit的时候加上就好了。

使用信号与槽的功能的类必须要包含Q\_OBJECT宏。同时好像要继承自Qt内部的某一个类、

有参数的信号

**周期操作：**

timer = new QTimer(this);// singleShot属性置为true时只触发一次

timer->setInterval(1000);

connect(timer, SIGNAL(timeout()) , this, SLOT(timerHit()));

timer->start();

我使用QTimer进行循环周期操作，发现一个用于网络操作的子线程在执行QTimer指定的函数中间（函数还没有执行完），如果用户点击了主界面触发网路子线程另外一个网络操作函数，就会直接去执行那个函数，接着的响应回调等等都有可能是在这个被定时器不断循环执行的函数中间执行，估计是开启了一个新线程，但是对象却还是同一个对象，于是就要注意多线程中最常见的问题。我遇到的是一个变量运行着运行着就自己变了（QSqlQuery的查询结果被改变）。

**带复选框的属性结构显示：**

<http://blog.csdn.net/rl529014/article/details/51355968>

**DLL 封装：**

qt 中比较常见的是将界面类做成一个dll文件，然后通过调用这个dll文件来使用界面。甚至这个dll能在VC下使用。如果将功能封装成dll，就是为了实现程序的升级以及exe文件体积的缩小。

学习网址：<http://www.cnblogs.com/findumars/p/5129050.html>

<http://www.cnblogs.com/newstart/archive/2013/01/23/2873611.html>

扫描plugin目录下的所有文件，寻找出dll文件并尝试加载它。如果他为合法的插件，提供了我们需要的接口（主界面上的显示、插件的启用与关闭即dll文件的加载与停止，其余功能的使用），就把它加入插件信息数组。

**程序打包： 安装包**

好处应该就是自己找到相关的dll文件，方便了很多。需要自己创建两个xml文件：config.xml设置安装包中的内容，packages.xml设置关于自己要打包的程序的配置。

<https://download.qt.io/official_releases/qt-installer-framework/> 下载页面

<http://wiki.qt.io/Qt-Installer-Framework> 还是弄的不错的

<https://doc.qt.io/qtinstallerframework/index.html> （帮助文档）

<https://blog.csdn.net/baidu_33570760/article/details/73467400> （中文使用示例）

<http://blog.csdn.net/sunshinwong/article/details/37764719> <https://doc.qt.io/qtinstallerframework/ifw-online-installers.html> （在线安装/升级需要创建仓库）

..\..\bin\binarycreator.exe -c config\config.xml -p packages YourInstaller.exe config.xml是自己设置的配置文件

如果是linux下，可以使用Qt Installer Framework(可以制作在线安装)，或者使用其余打包软件。相比较Windows，linux可以直接使用ldd命令查看相关的依赖。但是因为linux存在过多有差异的发行版，linux真打包起来需要很多很多so文件，于是最终打包文件一般都会很大。

无依赖打包形式：Flatpak、Snap、AppImage(下面有例子)、sfpkg

Flatpak：<http://docs.flatpak.org/en/latest/getting-started.html>

AppImage打包： 使用了网络与数据库，小程序最后有150MB，打包了很多不必要的so文件

appimagetool打包应用，appimaged注册和取消注册应用于系统daemon服务...

不过qt使用的是https://github.com/probonopd/linuxdeployqt，创建一个目录，二进制文件放到./bin下，qml放到/qml下，linuxdeployqt name -appimage就开始打包，缺少的目录与文件会自动默认创建。

so文件缺少要安装对应版本库：虽然qt中没有用到这个库，缺少不影响运行，但是linuxdeployqt借助ldd递归检查依赖的时候，要求一个也不能少：

ERROR: ldd outputLine: "libpq.so.5 => not found"

ERROR: for binary: "/opt/Qt5.10.1/5.10.1/gcc\_64/plugins/sqldrivers/libqsqlpsql.so"

ERROR: Please ensure that all libraries can be found by ldd. Aborting.

上面的错误意为打包目标依赖的libqsqlpsql.so缺少依赖的libpq.so.5

libmysqlclient.so.18=>libmysqlclient18

libpq.so.5=>libpq5

安装了对应库以后，so文件就会出现在/usr/lib/x86\_64-linux-gnu下

**版本更新：**

设置一个xml文件来管理各个零件的版本匹配，若存在需版本控制文件的增加以及删除的话，我想替换掉xml是最简单的办法。如果访问访问通过比较判断出来不是最新，以及是否强制跟新等设定，之后再进行进一步操作。网络操作可以选择开启一个新的线程。桌面端的游戏应用的更新就是很好的示例。

更新的程序部分可以是外在的update小程序，也可以是内嵌应用。一般选用更新的小程序是为了能够杀死正在执行的程序（用户提示以后，启动更新小程序并正常结束自己更好），释放对文件的使用以后方便执行安装包来成功替换文件。

最新版本的程序可以通过很多种方式来传递，比较常见的是http，ftp，scp。手机平台中下载文件应该是原生的tcp socket。

版本号可以根据前一个版本号自动生成下一个版本号，也可以人为的按版本号规则命名。不过基于上一次来生成这一次的版本号也是种很好的想法，使用md5等固定长度的运算就能够现。每一次更新都应该给出更新摘要以及更新的人为可读版本号。

**主程序关闭：**

如果QProcess::startDetached方法返回true，则关闭主程序，调用exit(0)，或者quit()方法；

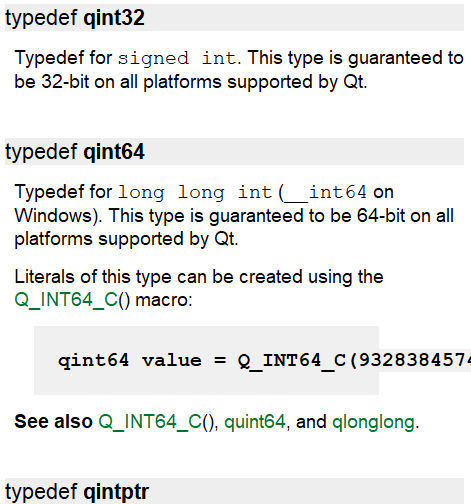
qApp->exit(0); //qApp 是自动创建的全局变量

qApp->arg(0); //获取传入main函数的index=0的参数



通过设置不同widget的显示样式，配合用户的点击操作就能够实现手机上类似的效果。

**跨平台类型：**



**日历：**

日历功能的实现需要使用viewpager或者使用gridview.setAdapter 实现翻页

每一页的具体内容，也可以使用gridview实现，或者自定义view

GridView：<http://blog.csdn.net/howard_liu1314/article/details/39993271>（同时包含农历、时间的一些计算函数）

http://blog.csdn.net/u010057266/article/details/45312475

自定义view：<http://blog.csdn.net/wwj_748/article/details/42244865>

推荐设置指定日期

**动画：**

动画类：QPropertyAnimation

常用方法：

setStartValue 设置起始情况

setEndValue 设置结束情况

setDuration

setLoopCount -1的时候就会无限循环

setPropertyName 设置操纵的属性。可以在构造函数中，通过第二个参数指定。

start

QPropertyAnimation \*sphereRotateTransformAnimation = new QPropertyAnimation(sphereTransform); //动画对象

sphereRotateTransformAnimation->setTargetObject(controller); //设置目标

sphereRotateTransformAnimation->setPropertyName("angle");

sphereRotateTransformAnimation->setStartValue(QVariant::fromValue(0));

sphereRotateTransformAnimation->setEndValue(QVariant::fromValue(360));// 0-360度设置的是度数，来表示起始位置与结束位置

//在这个项目里，0表示的是整个场景的右边，如果你是720就表示一次动画会转2圈

sphereRotateTransformAnimation->setDuration(10000); //设置动画时间，默认是250ms

sphereRotateTransformAnimation->setLoopCount(-1); //设置动画循环重复次数，如果是0就会一次都不运行，如果是-1就会一直运行

sphereRotateTransformAnimation->start();

error: undefined reference to `LoginWindow::on\_pushButton\_close\_clicked()' 通过designeer 设置了槽函数以后，直接将代码中的槽函数删除或者注释掉以后，就会出现找不到这个问题。双击问题可以知道 生成的代码文件中还是用到了这个函数。解决办法是cpp文件删除了以后还要去头文件里面删除对应申明，不然就会自动加上，因为“转到槽”功能设置的函数名字太特殊了。设置了这个名字的函数在头文件，之后生成目录/debug/moc\_xxx.cpp 中就会自动调用这个函数。

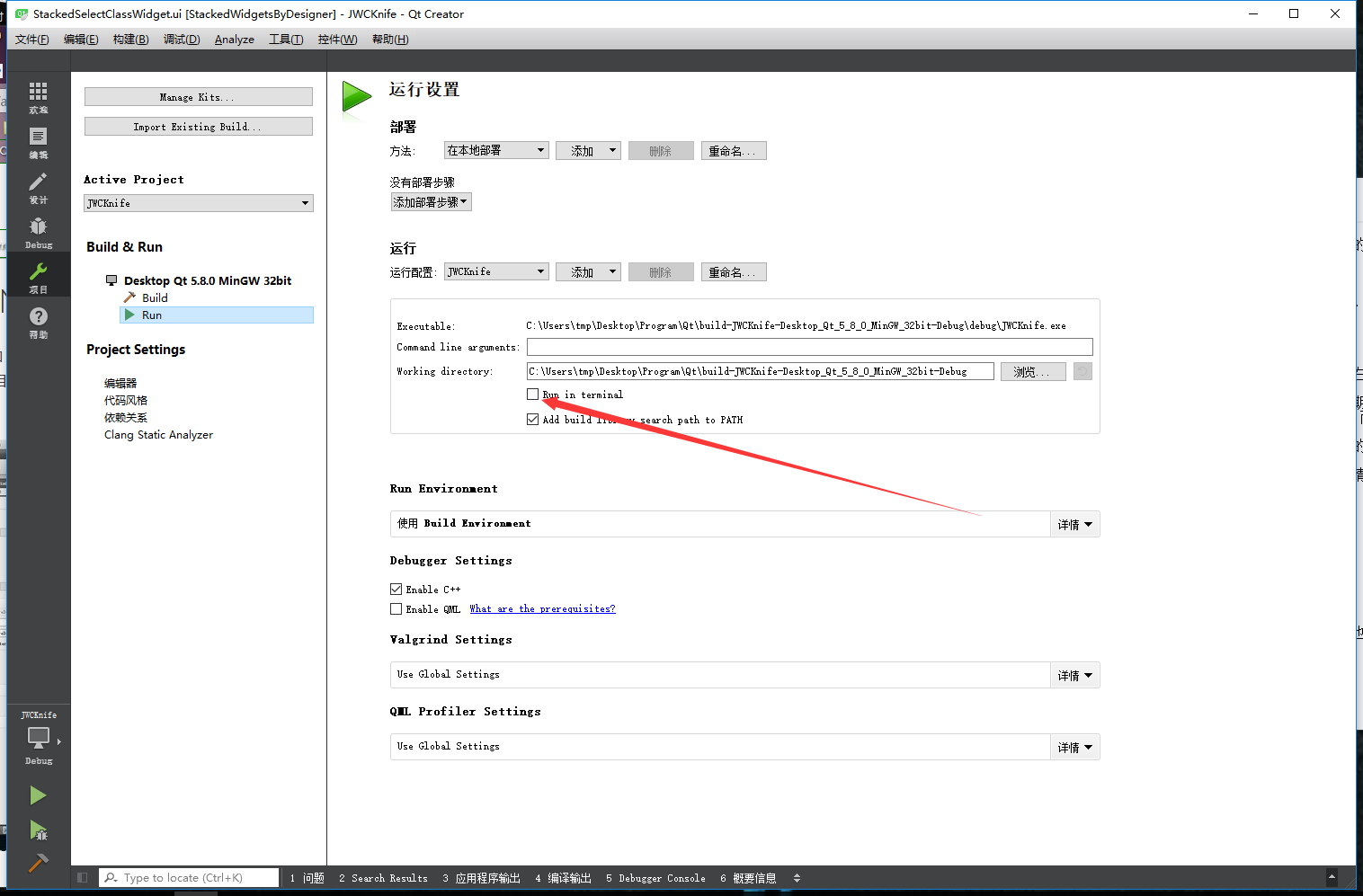
connect(m\_quitAction, SIGNAL(triggered()), qApp, SLOT(quit()));

qApp 这个是一个项目全局变量，是一个指向应用程序的指针

**控制台操作：**

学习网址：<http://blog.csdn.net/junbujianwpl/article/details/51446430>

1. 在pro文件中添加 CONFIG+= console   
   2、在最左侧选择项目(ctrl + 5)->构建运行 选择运行 将 Run in terminal选中



之后就能够使用cout 以及 cin 等对标准输入输出操作。

**绘图：**

QCustomPlot 是一个超强超小巧的qt绘图类，非常漂亮，非常易用

**图片与文字共存的按钮： QToolButton**

ui->toolButton\_0->setToolButtonStyle(Qt::ToolButtonTextBesideIcon);

ui->toolButton\_0->setIcon(QIcon(":Resources/MainWindow/checkedStar.png"));

ui->toolButton\_0->setText("test");

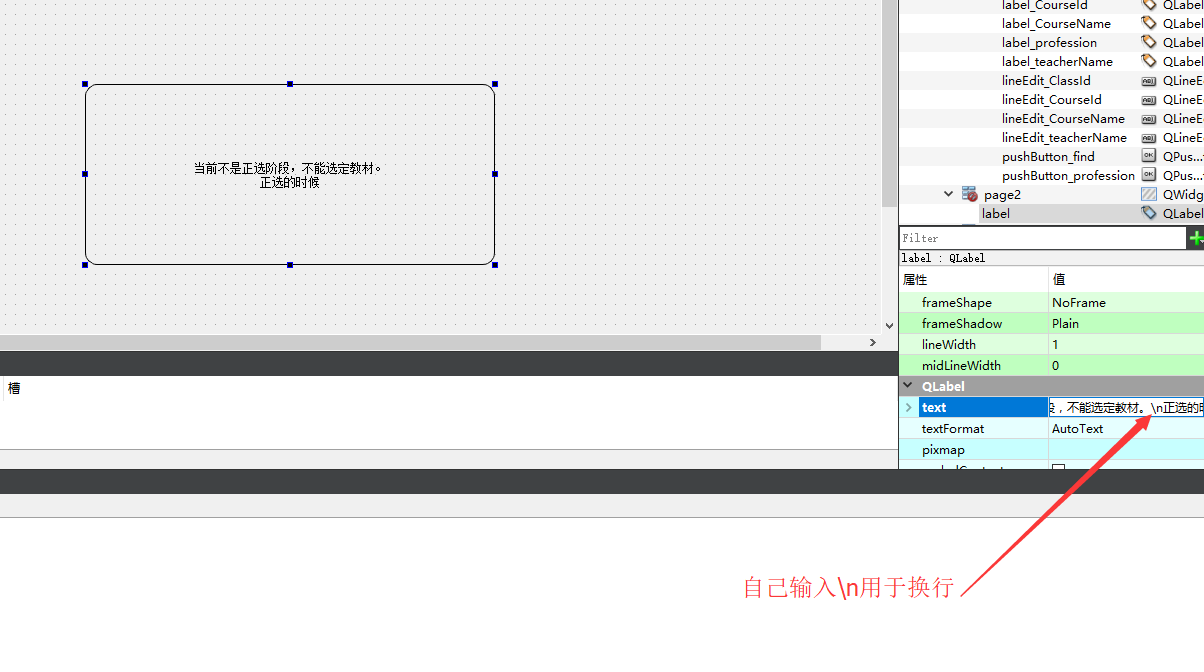
**QRadioButton：**

一个QRadioButton产生的效果

“同一个父窗体”或“一个button group”，这就是分组。如果没有进行分组，则默认拥有相同父窗体的radio buttons都将具有相互排他性。可以使用GroupBox将一组组按钮分开来。传闻还有一个QButtonGroup，在Designer中通过选中多个按钮以后，右键“添加到按钮组”。

这个Group会自动对加入的button进行id赋值，自动赋值是从-2开始的。

**QT Designer中设置文本内容换行：**



**QT 游戏：**

Youtube教学视频：<https://www.youtube.com/watch?v=8ntEQpg7gck>

undefined reference to 'vtable for \*\*\*(类)'

继承自QAbstractItemDelegate并声明了paint、sizeHint、editorEvent函数，但是没有定义。

**QT内嵌浏览器：**

通过COM组件可以调用系统自带的浏览器，当然是IE啦。Qt提供QAxObject和QAxWidget对象帮助访问COM。2.自己引入QtWebEngine，但是这种方式很消耗空间。所以大部分时候推荐使用方式一。

**QLineEdit中有按钮：**



创建一个继承自QLineEdit的类，创建好按钮，示例中的按钮是带有menu的QToolButton。

QToolButton \*optionsButton = new QToolButton;

#ifndef QT\_NO\_CURSOR

optionsButton->setCursor(Qt::ArrowCursor);

#endif

optionsButton->setFocusPolicy(Qt::NoFocus);

optionsButton->setStyleSheet("\* { border: none; }");

optionsButton->setIcon(icon);

optionsButton->setMenu(menu);

optionsButton->setPopupMode(QToolButton::InstantPopup);

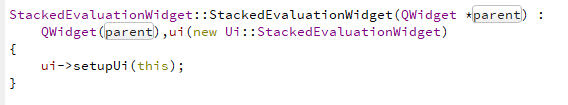
QWidgetAction \*optionsAction = new QWidgetAction(this);

optionsAction->setDefaultWidget(optionsButton);

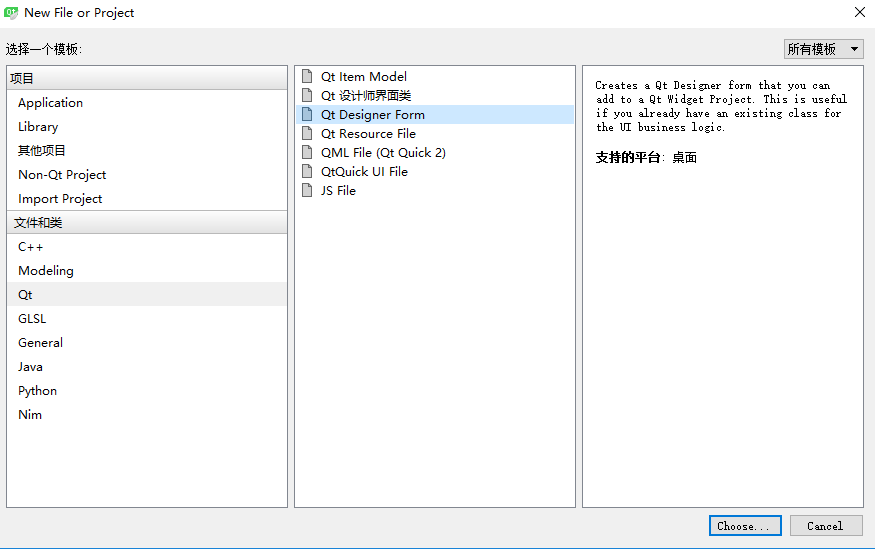
addAction(optionsAction, QLineEdit::LeadingPosition);

**自己创建ui文件与C++ Class .h .cpp文件，进行连接:**

QT会将ui1文件自动变成ui\_xxxxx.h，到时候在cpp文件中



以下就是对UI文件的创建：



这种连接有一种缺陷，不能在Designer中对按钮通过右键“转到槽”来自动设置槽函数。自己手动设置就好：void on\_pushButton\_reset\_clicked();

按钮根据QTableView的记录显示情况来自动设置位置，不需要调整designer中的qtableview的大小。因为move函数，是覆盖上去。

QSqlQuery \*sqlQuery =new QSqlQuery(QSqlDatabase::database());

QPoint pos = ui->pushButton\_materials\_send->pos();

SQLErrorDetect(sqlQuery, "select count(\*) from Materials", "获取所有开课数量失败 in StackedSelectClassWidget::setTable()");

sqlQuery->next();

int recordNumber =sqlQuery->value(0).toInt();

pos.setY( (recordNumber+1)\*30+ui->tableView\_materials->y()+10 );

ui->pushButton\_materials\_send->move(pos);

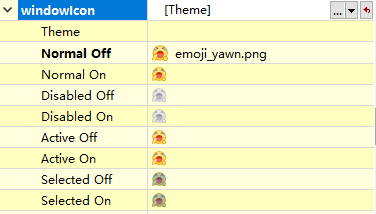
delete sqlQuery;

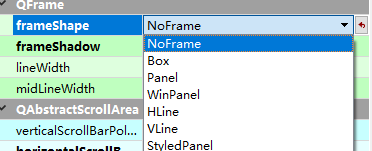
QVBoxLayout、QHBoxLayout 都不能作为构件构造函数的参数，因为不是QWidget类型。他们是QObejct, QLayoutItem->QLayout

**QT窗口关闭事件： QCloseEvent**

<http://blog.csdn.net/u010780613/article/details/50504867>

一张图片在不同状态会有不同的合适的显示：





**QFrame这里可以设置边框有无以及样式**

m\_scrollArea->setFrameShape(QFrame::NoFrame); 就可以去除边框

但是有的控件是lineEdit->setFrame(false);

**滚动条QScrollArea**（一个viewport）：

QScrollArea \*s = new QScrollArea(page0);

s->setWidget(page0);

上面这两行代码会导致程序不能正常启动，因为setWidget是将其设置为一个child，而初始化的时候却又将其设置为一个parent。

tableWidget->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

这句命令会导致滚动条不显示，但是鼠标滚轮依旧有效。

示例：

QScrollArea \* scrollArea =new QScrollArea(this);scrollArea->resize(400,300);

QWidget \*page =new QWidget(this); page->resize(1000,1000);

scrollArea->setWidget(page);

QWidget 构造函数中的参数不一定要是scrollArea，setWidget执行以后，就会将QWidget的父亲作为scrollArea的viewport。setWidget不仅仅是将目标设置为自己的“孩子”。类比，layout的addWdiget估计也差不多。

隐藏滚动条：（滚动条没有显示，但是还可以通过滚轮交互）

QTableWidget \*table = new QTableWidget(100,100,this);

....

table->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

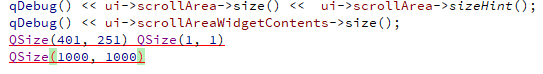
关闭滚动条：（关了交互部分的功能，还是有滚动条显示，但是不能使用）

QTableWidget \*table = new QTableWidget(100,100,this);

....

table->verticalScrollBar()->setDisabled(true);

*sizeHint*：有滚动条的窗体通过*sizeHint*()函数返回scroll area的size。这个函数本意是返回一个qt推荐的窗口大小。



**QScrollArea与QStackedWidget：**

如果使用QScrollArea包含QStackedWidget并且将QStackedWidget的一个page，使用了setWidget。这时候的效果就是QScrollArea的滚动条能够上下拉，但是不支持鼠标滚轮，同时QStackedWidget失效，不能在page之间互相切换。我估摸着是QScrollArea将QStackedWidget盖住了。真正的做法或许是在QStackedWidget中加入QScrollArea，并在QScrollArea下面加上一个page。

setMinimumHeight 不能阻止支持滚动的窗口，因为挤而“缩小”。

m\_stackedWidget =new QStackedWidget(m\_scrollArea);

qDebug() << parent->width() << parent->height();

m\_stackedWidget->setGeometry(0,0,parent->width(),parent->height());

上述代码中parent->width(),parent->height()不能获取到数值程序不能正常启动。界面不会正常显示。最后的办法是无奈地使用常数。

**重新调整窗口大小有两种办法：**

1. setGeometry(0,0,1000,670); 前两个参数为左上角坐标
2. resize(1000,670);

qt的窗口函数width()这么警告:

**Note:** Do not use this function to find the width of a screen on a [multiple screen desktop](qdesktopwidget.html). Read [this note](qdesktopwidget.html" \l "screen-geometry) for details.

将QVBoxLayout中的内容水平居中·：

1.page0Layout->addWidget(lineEdit);page0Layout->setAlignment(lineEdit, Qt::AlignHCenter);

而page0Layout->addWidget(lineEdit, Qt::AlignHCenter);无效

同时要求 如果QWidget是在代码中创建的，需要给他指定一个大小，因为不然，大小就是QSize(0, 0)，就没有居中效果

1. QVBoxLayout外面再包一层QHBoxLayout

如果想要专门用于显示的一行式文本框

1. 使用QLineEdit并且lineEdit->setReadOnly(true); 或者自己继承类暗中这么实现。
2. QLabel

**搜索候选项：**

使用**[textChanged](qlineedit.html" \l "textChanged)**信号

**Qt设置单元格背景颜色（QT 课程表）：**

1、常规的方法：在子类化Moel时重载data/setData函数，在函数中判定当role==Qt.BackgroundRole时，data返回保存的本单元格指定颜色的QColor对象；而setData函数就是把指定的颜色保存在本单元格数据中。  
这种方法通常应该自定义单元格数据的类，类成员除了文本数据还应该包括颜色数据。  
2、可用但不太常用的方法：重载QtableView的painEvent函数，或者重载Delegate的paint函数。这种方法要在函数中判定某个单元格的颜色，并对QBrush等画图工具作出不同设置，从而达到为某个单元格“画”不同背景的效果。  
通常可以在Model中保存一个列表，列表记录了背景不同的单元格，而QtableView或Delegate则通过判定单元格是否在列表中来设置不同的画图工具。  
3、 Delegate::paint函数

ui->tableWidget->setRowCount(6);

ui->tableWidget->setColumnCount(6);

for(int row=0; row<6; ++row)

for(int column=0; column<6; ++column)

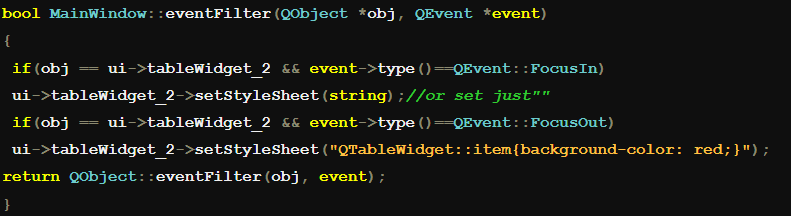
ui->tableWidget->setItem(row, column, new QTableWidgetItem("这是测试"));

ui->tableWidget->setSpan(0,0, 2, 1);

ui->tableWidget->item(0, 0)->setForeground(Qt::blue); 字体颜色

ui->tableWidget->item(0, 0)->setBackground(Qt::red); 单元格背景颜色

**QT事件过滤器：**



“xxx” does not name a type xxx是一个自己创建的类，有一次出现这个问题是因为xxx这个名字，将它换成一个短一点的名字就好了。

因为显示器的原因，自己电脑上看起来白很多，但是在截图的时候，因为加了层阴影，到是看出来qt默认的背景色不是纯白，有点偏灰。我将程序放到关总电脑上执行就显示成明显的灰色。

**时间：**

QDateTime dt;

QTime time;

QDate date;

dt.setTime(time.currentTime());

dt.setDate(date.currentDate());

QString currentDate = dt.toString("yyyy:MM:dd:hh:mm:ss");

qDebug()<<currentDate<<"-=-=-=-=-=-=-=-";

QDateTime current\_date\_time = QDateTime::currentDateTime();

QString current\_date = current\_date\_time.toString("yyyy-MM-dd hh:mm:ss ddd");

qDebug()<<current\_date<<"-=-=-=-=-=-=-=-";

**下拉框：**

m\_usernameListWidget->setGeometry(0,0,0,0);

void LoginWindow::fillUsernameComboBox() {

ui->comboBox\_username->setEditable(true);

ui->comboBox\_username->lineEdit()->setPlaceholderText("请输入本科系统账号");//需要comboBox\_username->setEditable(true)

SQLErrorDetect(m\_pQuery, "select \* from Users order by frequence desc", "获取账号列表失败");

ui->comboBox\_username->setModel(m\_usernameListWidget->model());

ui->comboBox\_username->setView(m\_usernameListWidget);

if( m\_pQuery->next() ) {

QString tmpUsername =m\_pQuery->value(0).toString(); //不能直接进行ui->comboBox\_username->setCurrentText，while会清空

ui->lineEdit\_password->setText(m\_pQuery->value(1).toString());

do {

AccountItem \*pAccountItem =new AccountItem();

pAccountItem->setText(m\_pQuery->value(0).toString());

QListWidgetItem\* widgetItem = new QListWidgetItem(m\_usernameListWidget);

m\_usernameListWidget->setItemWidget(widgetItem, pAccountItem);

connect( pAccountItem, SIGNAL(deleteAccount(QString)),this, SLOT(deleteAccount(QString)) );

connect( pAccountItem, SIGNAL(showAccount(QString)),this, SLOT(showAccount(QString)) );

}while(m\_pQuery->next());

ui->comboBox\_username->setCurrentText(tmpUsername);

}

}

其中Item为自定义类：

#include "AccountItem.h"

#include <QHBoxLayout>

AccountItem::AccountItem(QWidget \*parent) : QWidget(parent),m\_username(new QLabel),m\_delete(new QPushButton)

{

QHBoxLayout \*main\_layout =new QHBoxLayout();

main\_layout->addWidget(m\_username);

main\_layout->addStretch();

main\_layout->addWidget(m\_delete);

this->setLayout(main\_layout);

connect( m\_delete, SIGNAL(clicked()), this, SLOT(on\_pushButton\_delete\_clicked()) );

}

void AccountItem::setText(const QString text) {

m\_username->setText(text);

}

QString AccountItem::text(void) {

return m\_username->text();

}

void AccountItem::on\_pushButton\_delete\_clicked(void) {

emit deleteAccount(m\_username->text());

}

void AccountItem::*mouseReleaseEvent*(QMouseEvent \*) {

emit showAccount(m\_username->text());

}

配合一下QSS样式：

QComboBox

{

font: 11pt "等线";

background:white;

padding-left:5px ;

border-top-left-radius:3px;

border-top-right-radius:3px;

border: 1px solid rgb(209 , 209 , 209);

}

QComboBox:hover

{

border: 1px solid rgb(21 , 131 , 221);

}

QComboBox::down-arrow

{

border-image:url(:/Resources/drop\_down\_Button.png) 0 36 0 0 ;

}

QComboBox::down-arrow:hover

{

border-image:url(:/Resources/drop\_down\_Button.png) 0 18 0 18 ;

}

QComboBox::down-arrow:on

{

border-image:url(:/Resources/drop\_down\_Button.png) 0 0 0 36 ;

}

QComboBox::drop-down

{

width:20px;

background:transparent;

padding-right:5px;

}

QListView {

margin:3px;

}

QListWidget QPushButton{

background:transparent;

border-image:url(:/Resources/delete\_username.png) 0 30 0 0 ;

width:15px;height:15px;

}

QListWidget QPushButton:hover{

border-image:url(:/Resources/delete\_username.png) 0 15 0 15 ;

}

QListWidget QPushButton:pressed{

border-image:url(:/Resources/delete\_username.png) 0 0 0 30;

}

QComboBox QAbstractItemView::item

{

background:transparent;

height:30px;

}

QComboBox QAbstractItemView::item:hover

{

background:rgb(21 , 131 , 221);

}

qt中子项优先于父项构建，这也是C++的规则。

**Debug 与Realease：**

Debug模式下进行debug的时候，可以随时设置断点。

Debug模式下可以正常在Qt Creator中启动的软件，放到Realease模式下，不一定能够正常运行，一般症状是界面不显示并且过一会程序就会异常结束。一般这就是程序出错，如果将功能都封装好，通过注释可以很容易地判断出来是哪里的问题：



Qt QString的cotains函数从html的返回文件中判断中文一直返回false。判英文是可以的，同时输出到qDebug()是正常的中文显示。这时候不仅仅使用中文内容进行contains判断，而是加上了标签名字：<font color=\"#990000\">没有完成评估,不能查看课表信息!</font>。

qDebug有时候会因为输出的内容过多，而输出不了。这种情况只在输出一个网站的时候遇到过。

QList<QString> 能直接给QStringList赋值

­­LoginWindow这个类的界面拥有EducationSystem网络访问类，与MainWindow数据显示主界面，在刚刚点下login按钮的时候，我根据网络的返回情况发送信号，并已经在构造函数中将这些信号链接给MainWindow中的几个子窗口，但是发现子窗口中的slot函数根本不执行，大概是因为子窗口没有被创建好。等窗口显示了以后，想办法触发，信号倒是发送成功。

LoginWindow在显示MainWindow的时候，但是设置了网络返回，如果失败就会弹窗告知原因，而这时候，我发现弹窗的时候主程序的运行是暂停下来的，弹出的窗口也不是置顶，于是被loginwindow挡住，整个程序就因此卡主。

#LoginWindow {

image: url(:/Resources/LoginWindow/login\_title.png);

image-position:TOP;

}

这种写法只有在QDialog中有效，窗口如果是QWidget在designer有效但是执行时无效。

**XML：** 远远不如数据库  
<http://blog.csdn.net/taiyang1987912/article/details/46695245>

使用extern QTextCodec \*gbk;并不需要包含头文件

**文件上传：**

二进制 POST传输数据，使用的Content-Type: application/octet-stream:

<http://m.blog.csdn.net/timothy93bp/article/details/77580365>

表单上传文件，使用的Content-Type: multipart/form-data;

POST /G2S/eWebEditor/aspx/upload.aspx?action=save&type=file&style=mini&cusdir=&skey= HTTP/1.1

Accept: image/gif, image/jpeg, image/pjpeg, application/x-ms-application, application/xaml+xml, application/x-ms-xbap, \*/\*

Referer: http://cc.scu.edu.cn/G2S/eWebEditor/dialog/file.htm

Accept-Language: zh-Hans-CN,zh-Hans;q=0.5

User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 10.0; WOW64; Trident/7.0; .NET4.0C; .NET4.0E; .NET CLR 2.0.50727; .NET CLR 3.0.30729; .NET CLR 3.5.30729; InfoPath.3)

Content-Type: multipart/form-data; boundary=---------------------------7e12873b507e0

Accept-Encoding: gzip, deflate

Host: cc.scu.edu.cn

Content-Length: 709

Connection: Keep-Alive

Cache-Control: no-cache

Cookie: \_\_guid=80263045.2957263564511600600.1508405234987.6143; ASP.NET\_SessionId=gh15x2epsjwblkug15rav0ig; user\_name=MjAxNTE0MTQ2MjEwOQ%3d%3d; login\_code=MQ%3d%3d

-----------------------------7e12873b507e0

Content-Disposition: form-data; name="MAX\_FILE\_SIZE"

51200000

-----------------------------7e12873b507e0

Content-Disposition: form-data; name="uploadfile"; filename="C:\Users\tmp\Desktop\123.zip"

Content-Type: application/x-zip-compressed

PK...........K+OtCR....................txt}.K

. ...}.Q.Q.g...A.....A.j.x...Z;.0[M.&.b..0(.'.@`r/.<}P..A.S.W... ...+.;.8.....PK..?..........K+OtCR.........$....... ................txt

. ...........h/.t..

..".t..

..".t..PK..........^...|.....

-----------------------------7e12873b507e0

Content-Disposition: form-data; name="originalfile"

C:\Users\tmp\Desktop\123.zip

-----------------------------7e12873b507e0—

奇怪的是我没有在页面中找到这几个name

“Content-Type: application/x-zip-compressed”这个东西由浏览器决定，服务端代码在限制检查的时候也可以加上这个的判断，因为如果是“Content-Type: application/octet-stream”就说明可能是二进制文件，即便你将后缀名改了也不影响浏览器对文件的识别，缺点是支持的浏览器少了。cc.scu.edu.cn对后缀名的检测是通过js进行的。

zip文件

firefox : application/octet-stream

chrome : application/octet-stream  
oprea : application/zip  
IE : application/x-zip-compressed

rar文件

firefox : application/x-rar-compressed  
chrome : application/octet-stream  
oprea : application/x-rar-compressed

通过浏览器得到的页面经过了浏览器的调整，还是用bp的Repeat来得到原始网页最好。

**URL编码：**

QByteArray ba = strInput.toUtf8();

QByteArray byteArrayPercentEncoded = ba.toPercentEncoding();

QString(byteArrayPercentEncoded.data())

一般来说，在QApplication创建之前不支持创建QObjects，根据平台的不同，可能会导致出现奇怪的崩溃。 这意味着QObject的静态实例也不被支持。 一个正确构造的单线程或多线程应用程序应该使QApplication成为第一个被创建的，最后被销毁的QObject。