# async线程退出：

async配合future::get\_wait()，当get\_wait超时返回时，进程依然等待子线程运行结束，才会退出

**class** Test{

**public**:**void** open(){

**for** (**int** i = 0; i < 5; i++){

            std::this\_thread::sleep\_for(std::chrono::seconds(1));

            cout << i << "..." << endl;

        }

    }

};

**int** main()

{

    Test t;

    std::future<**void**> fut = std::async(std::launch::async,&Test::open,&t);

    std::future\_status status = fut.wait\_for(std::chrono::seconds(2));

**if** (status == std::future\_status::deferred)

        cout << "deferred" << endl;

**else** **if** (status == std::future\_status::timeout)

        cout << "timeout" << endl;

**else** **if** (status == std::future\_status::ready)

        cout << "ready" << endl;

**return** 0;

}

/\*

运行结果：

0...

timeout

1...

2...

3...

4...

\*/

# pthread的join和deatch：

pthread有两种状态joinable状态和unjoinable状态

pthread\_create创建的线程默认是joinable，如pthread\_create(&tid, NULL, ThreadFunc, NULL)

unjoinable属性可以在pthread\_create时指定，或在线程创建后在线程中pthread\_detach自己

int pthread\_detach(pthread\_t tid);

int pthread\_join(pthread\_t thread, void \*\*retval);

QT开发GUI程序既可以采用QT设计器——QT Designer，也可以采用编写代码的方式

在按钮（Push Button）上右键->转到槽，则可添加事件处理函数

QApplication管理GUI程序的控制流和主要设置

F1 查看帮助

F2 跳转到函数定义

# QT开发环境

QTCreator和QT Visual Studio Tools

IDE可采用QT Creator，也可以采用Visual Studio

QT Creator中：

F1 查看帮助

F2 跳转到函数定义（和Ctrl+鼠标左键一样的效果）

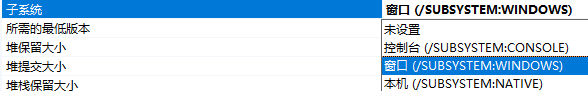
\*.ui文件转为ui\_\*.h：

在QT安装目录bin文件夹下找到uic.exe

uic.exe \*.u –o ui\_\*.h

VS2015之前集成Qt使用 Qt Add-In，但在VS 2015中Qt Add-In不可用了，但是提供了一个扩展QT Visual Studio Tools

VS下要查看qDebug内容，则在修改项目属性->链接器->系统，将默认的窗口改为控制台，这样运行时会出现控制台



在环境变量Path中同时添加

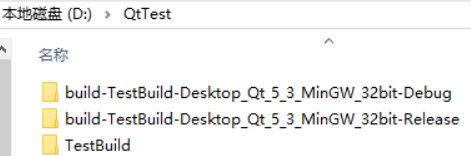
D:\Qt\5.12.0\mingw73\_64\bin和D:\Qt\5.12.0\msvc2017\_64\bin会有冲突

如在VS上开发，ctrl+F5可以运行，但点击生成的exe则不可以

因为点击exe时查找dll，先找到D:\Qt\5.12.0\mingw73\_64\bin，而实际需要的是D:\Qt\5.12.0\mingw73\_64\bin下的dll，所以出错，将D:\Qt\5.12.0\msvc2017\_64\bin置于D:\Qt\5.12.0\mingw73\_64\bin前边则正常

修改QTCreator项目构建目录

QTCreator默认项目构建目录（相当于VS中的Debug与Release目录）与工程目录同级



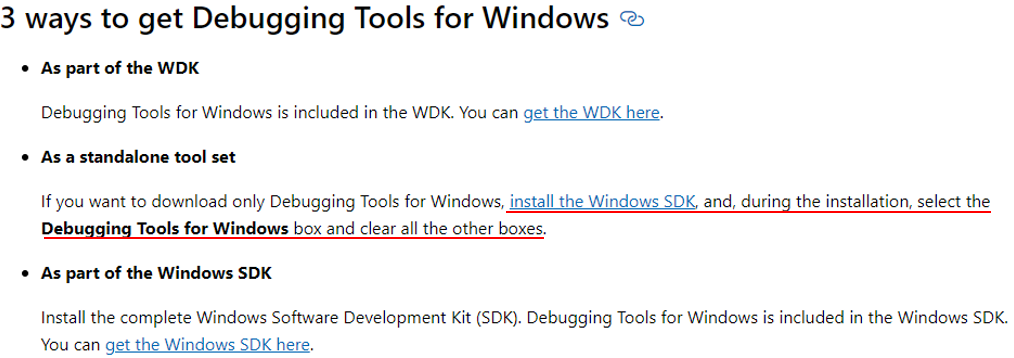
项目将工具–>选项–>构建和运行中Default build directory修改为./%{CurrentBuild:Name}（改为你想要的目标目录都行）

QT Creator工程的.pro文件相当于VS的.sln文件

QT Creator采用MSVC编译器调试

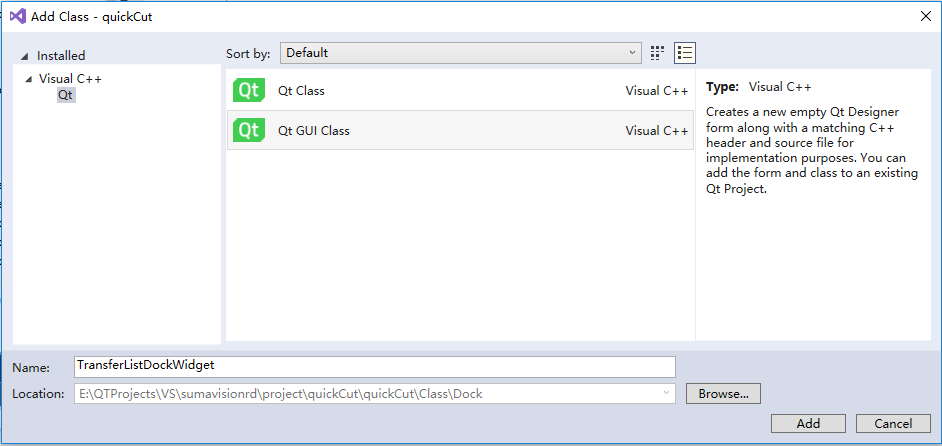
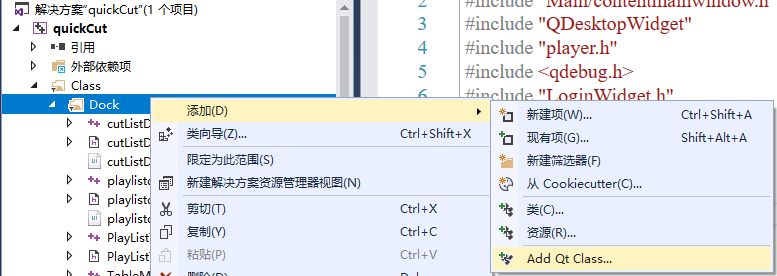
QT采用MSVC编译器，调试时提示缺少“Unable to create a debugger engine”。解决方法：

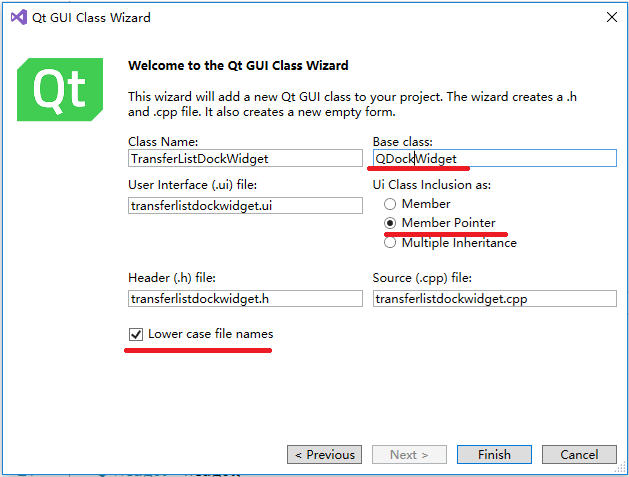
1. 微软官网下载Debugging Tools for Windows，选用第二种方法，安装



1. 工具->选项->构建套件，系统会自行系统自行检测到Debugger，如果没有，则手动添加：将Path设定为”Windows Kits所在目录\Windows Kits\安装版本\Debuggers\编译器版本\cdb.exe”，例如：C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\cdb.exe
2. Qt MSVC 使用Qt-Creator Debug 产生了”The cdb process terminated”的信息，但是调试根本没有开始。原因在于MSVC的版本与Debugger(cdb.exe)的版本不一致。若为64bit请选择x64下的cdb.exe，否则选择x86文件夹下的cdb.exe。

QT Visual Studio Tools添加带.ui文件的QT类，即Qt GUI Class



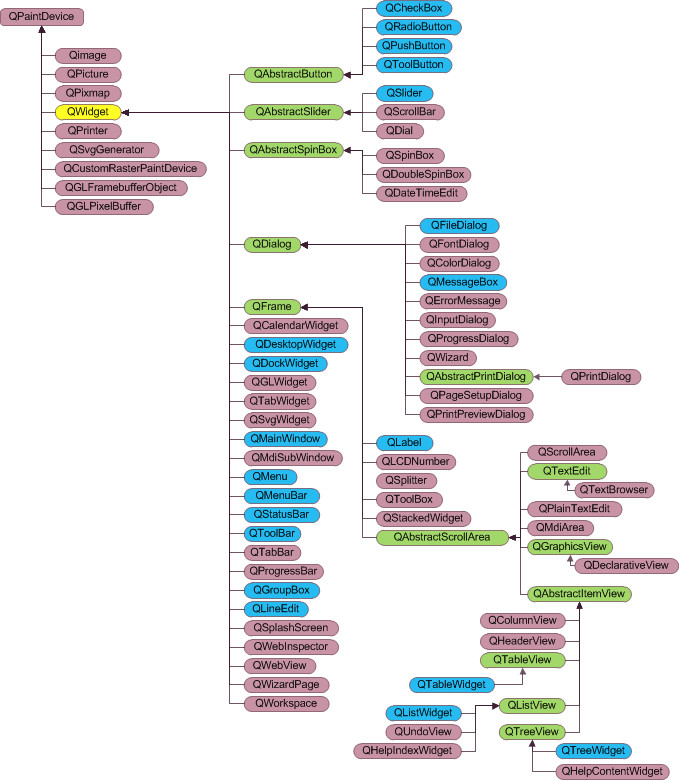


QT Visual Studio Tools不带.ui文件的QT类，即Qt class

既可以在解决方案管理器右键->添加->Add Qt Class->Qt class，这样会自动生成包含一些预定义内容的.c和.cpp文件

也可以像普通C++程序一样右键依次添加.cpp和.h文件，手动添加所有内容

# QT继承体系：



class QWidget : public QObject, public QPaintDevice

# QApplication：

QApplication管理GUI程序的控制流和主要设置

**应用程序运行路径：**

QString curPath = QCoreApplication::applicationDirPath();//"E:/QTProject/Folder/x64/Debug"

QCoreApplication::applicationFilePath(); // "E:/QTProject/Folder/x64/Debug/Folder.exe"

QCoreApplication::applicationName(); // "Folder"

**宏qApp：**

QApplication 继承 QGuiApplication，而 QGuiApplication 继承 QCoreApplication

QCoreapplication中：#define qApp QCoreApplication::instance()

QGuiApplication中：#define qApp (static\_cast<QGuiApplication \*>(QCoreApplication::instance()))

QApplication中：#define qApp (static\_cast<QApplication \*>(QCoreApplication::instance()))

即QCoreapplication是个单例类，qApp即相当于QCoreApplication::instance()

**QCoreApplication与QSetting：**

要使用QSetting无参构造函数，则下面代码片段中a.set\*两个语句不可省略

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

{

QApplication a(argc, argv);

a.setApplicationName("QuickEditor"); //或者QCoreApplication::setApplicationName("QuickEditor");

a.setOrganizationName("sumavision");//或者QCoreApplication::setOrganizationName("sumavision"); RotateWidget r;

r.show();

return a.exec();

}

QSetting的有如下两个构造函数：

QSettings ( const QString & organization, const QString & application = QString(), QObject \* parent = 0 )

QSettings ( QObject \* parent = 0 )

当不传入organization和application参数，则使用的QCoreApplication::setApplicationName和QCoreApplication::setOrganizationName设置的值

**获取桌面分辨率：**

QDesktopWidget\* p= QApplication::desktop();

* const [QRect](http://epic-alfa.kavli.tudelft.nl/share/doc/qt4/html/qrect.html) QDesktopWidget::screenGeometry ( int screen = -1 ) const

Returns the geometry of the screen with index screen. The default screen is used if screen is -1.

* const [QRect](http://epic-alfa.kavli.tudelft.nl/share/doc/qt4/html/qrect.html) QDesktopWidget::availableGeometry ( int screen = -1 ) const

Returns the available geometry of the screen with index *screen*. What is available will be subrect of [screenGeometry](http://epic-alfa.kavli.tudelft.nl/share/doc/qt4/html/qdesktopwidget.html#screenGeometry)() based on what the platform decides is available (for example excludes the dock and menu bar on Mac OS X, or the task bar on Windows). The default screen is used if *screen* is -1.

**覆盖QApplication::notify()监控全局事件：**

keyPressEvent、mouseReleaseEvent，这些函数加在某一控件里，那么只有焦点在那一控件上的时候，才会响应这些函数来实现相关功能

如果需要监控全局下的键盘快捷键，无论焦点在哪个界面上，则继承QApplication并覆盖 QApplication::notify()函数是唯一方法

Qt的事件会汇聚到 QCoreApplication::notify，然后再分发出去，因此整个程序的消息可以在此过滤

class CustomApp : public QApplication

{

protected: bool notify(QObject\*, QEvent \*) override{

if (e->type() == QEvent::KeyPress || e->type() == QEvent::MouseButtonPress){

dosomething();//如发射信号

}

return QApplication::notify(obj, e);

}

};

# \_\_super：

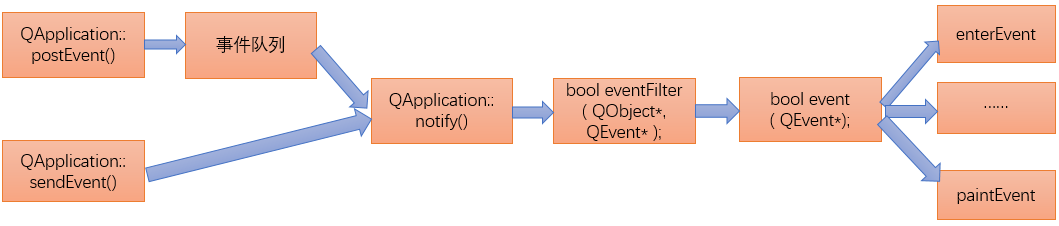
C＋＋本身没有\_\_super，这是visual c＋＋的扩展

\_\_super::member\_function();

即多层继承，会调用与当前类辈分最近的重载基类的虚函数。

# QT的事件传递、过滤、接受、忽略：

事件派发过程图：



* 安装事件过滤器有两个步骤: (假设要用**A**来监视过滤**B**的事件)  
       首先，调用 B 的installEventFilter( const QOject \*A )，以A的指针作为参数。 这样所有发往B的事件都将先由A的eventFilter()处理。  
       然后，重载 A 的QObject::eventFilter()函数，在eventFilter() 中书写对事件进行处理的代码。

返回true，则事件被过滤掉了不再传给B的event()函数，返回false，则相反

* 重新实现QObject::event()。一般用在Qt没有提供该事件的处理函数时。也就是，我们增加新的事件时。返回true，则事件不再送给形如mousePressEvent()的特定事件处理函数，返回false，则相反
* 事件转发给父组件

对于某些类别的事件（**并非所有事件，包括 鼠标，滚轮，按键等**），如果在整个事件的派发过程结束后还没有被处理，那么这个事件将会向上转发给它的父widget，直到最顶层窗口。如，事件最先发送给QCheckBox，如果QCheckBox没有处理，那么由QGroupBox接着处理，如果QGroupBox没有处理，再送到QDialog，因为QDialog已经是最顶层widget，所以如果QDialog不处理，QEvent将停止转发。

* 特定的事件处理函数例如mousePressEvent(), keyPressEvent(), paintEvent()等，它们都是在event()函数中通过switch...case调用的，返回值都是void，处在事件循环队列的最末端，一般由这些函数调用accept()函数和ignore()函数来决定此事件会不会转发给父对象
* **在QApplication 上安装事件过滤器**

QApplication a(argc, argv);

a->installEventFilter()

这之所以被单独列出来是因为： QApplication 上的事件过滤器将捕获应用程序的所有事件，而且第一个获得该事件。也就是说事件在发送给其它任何一个event filter之前发送给QApplication的event filter。

举例：

TestQT继承QMainWindow，CustomButton继承QPushButton，

TestQT的centralWidget中增加CustomButton按钮

void CustomButton::mousePressEvent(QMouseEvent \*event)

{

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

qDebug() << "CustomButton left clicked";

qDebug() << "event->globalPos() is:" << event->globalPos();

qDebug() << "event->pos() is:" << event->pos();

\_\_super::mousePressEvent(event);

event->ignore();

}

else

{

return \_\_super::mousePressEvent(event);

}

}

void TestQT::mousePressEvent(QMouseEvent \*event)

{

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

qDebug() << "TestQT::mousePressEvent left clicked";

qDebug() << "event->globalPos() is:" << event->globalPos();

qDebug() << "event->pos() is:" << event->pos();

}

return \_\_super::mousePressEvent(event);

}

bool TestQT::eventFilter(QObject \*watched, QEvent \*event)

{

if (event->type() == QEvent::MouseButtonPress)

{

QMouseEvent\* mouseev = (QMouseEvent\*)(event);

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

qDebug() << "TestQT::eventFilter left clicked, watched=" << watched;

qDebug() << (watched == ui.success\_btn);

qDebug() << "event->globalPos() is:" << mouseev->globalPos();

qDebug() << "event->pos() is:" << mouseev->pos();

return false;

}

}

return \_\_super::eventFilter(watched,event);

}

//运行结果：

事件由按钮接收，分发到CustomButton::mousePressEvent，由于在该函数中设置了ignore，因此事件继续传播给了父组件

在TestQT的构造函数中设置了this->installEventFilter(this)，因此事件先被TestQT自己的eventFilter函数处理

CustomButton left clicked

event->globalPos() is: QPoint(875, 628) //全局坐标，即屏幕坐标

event->pos() is : QPoint(38, 10) //鼠标点击点相对于按钮的位置

TestQT::eventFilter left clicked, watched = TestQT(0x538cff7d0, name = "TestQT") **//事件传播给父组件，watched已经是父组件而不是按钮了**

false

event->globalPos() is : QPoint(875, 628)

event->pos() is : QPoint(109, 331) //鼠标点击点相对于父组件的位置

TestQT::mousePressEvent left clicked

event->globalPos() is : QPoint(875, 628)

event->pos() is : QPoint(109, 331)

Qt 做了特殊的设计：事件对象默认是 accept 的，而作为所有组件的父类QWidget的默认实现则是调用ignore()。这么一来，如果用户自己实现事件处理函数，不调用QWidget的默认实现，就等于是接受了事件；如果用户调用QWidget的默认实现，则忽略了事件，因此如果一个类继承了QWidget，并且在mousePressEvent中调用\_\_super:: mousePressEvent()，则该事件会传递给父组件



# 加载图片

QImage logo;

if (logo.load(logo\_path,"png"))

{

QImage resizedimg = logo.scaled(ui.LogoIcon->size());

ui.LogoIcon->setPixmap(QPixmap::fromImage(resizedimg));

}

bool QImage::load ( const QString & fileName, const char \* format = 0 )

第二个参数指定文件格式，如果指定错误，则加载失败

如果第二个参数为nullptr，则probes the file for a header to guess the file format.

QImage::load("a.png")，如果实际是jpg格式的文件，扩展名被改为png，则加载失败；

QImage::load("a.jpg")，如果实际png文件扩展名被改为jpg，则加载失败

不需要指定格式的加载方法：

QImageReader reader(logo\_path);

reader.setDecideFormatFromContent(true);//设定通过文件实际内容格式打开文件

if (reader.canRead())

{

QImage logo\_img;

if (reader.read(&logo\_img)) {}

}

QTimer：

单次触发：QTimer::singleShot(1, this, []() {}); 1ms后触发，如果第一个参数为0，则立即触发

bool QTimer::isActive()，检测定时器是否在正在运行，调用start后运行，主动调用stop后停止或者设置setSingleShot(true)且触发了一次后

QTimer正在运行时可再次设置interval、setSingleShot，然后再次调用start()重启定时器

QTimer::singleShot(0, this, []() {});

QTimer\* timer = new QTimer(this);

timer->setSingleShot(false);

timer->setInterval(3000);

connect(timer, &QTimer::timeout, this, [timer]()

{

static int i = 0;

qDebug() << "trigger i=" << i << ", " << QDateTime::currentDateTime();

i++;

if (i == 3)

{

timer->setInterval(5000);

timer->setSingleShot(true);

timer->start();

}

});

timer->start();

//trigger i = 0, QDateTime(2019 - 11 - 23 19:49 : 05.125 中国标准时间 Qt::LocalTime)

//trigger i = 1, QDateTime(2019 - 11 - 23 19:49 : 08.098 中国标准时间 Qt::LocalTime)

//trigger i = 2, QDateTime(2019 - 11 - 23 19:49 : 11.099 中国标准时间 Qt::LocalTime)

//trigger i = 3, QDateTime(2019 - 11 - 23 19:49 : 16.098 中国标准时间 Qt::LocalTime)

# QWebEngineView捕获事件

Unfortunately the widget QWebEngineView does not catch almost any events (except mouse enter and exit, recently added keyboard events)

Almost all events (like mouse move or paint) are handled by QWebEngineView child delegate of private type RenderWidgetHostViewQtDelegateWidget that is derived from QOpenGLWidget

解决方法：

**class** JumpWebEngineView : **public** QWebEngineView

{

    Q\_OBJECT

**public**:

**explicit** JumpWebEngineView(QWidget \*parent = nullptr);

signals:

**void** signalMouseButtonPress();

**protected**:

**bool** eventFilter(QObject \*obj, QEvent \*ev) override;

**bool** event(QEvent\* ev) override;

**protected**:

    QObject\*  m\_child;

};

JumpWebEngineView::JumpWebEngineView(QWidget \*parent): QWebEngineView(parent), m\_child(nullptr){}

**bool** JumpWebEngineView::eventFilter(QObject \*obj, QEvent \*ev)

{

**if** (obj == m\_child && (ev->type() == QEvent::MouseButtonPress))

    {

        emit signalMouseButtonPress();

**return** **false**;

    }

**return** \_\_super::eventFilter(obj, ev);

}

**bool** JumpWebEngineView::event(QEvent\* ev)

{

    //鼠标按下事件不会被传递给QWebEngineView，而是被QWebEngineView child delegate处理

**if** (QEvent::ChildAdded == ev->type())

    {

        QChildEvent \*child\_ev = **static\_cast**<QChildEvent\*>(ev);

        m\_child = child\_ev->child();

**if** (m\_child)

        {

            m\_child->installEventFilter(**this**); //不能省略，否则JumpWebEngineView::eventFilter仍捕获不到鼠标按下等事件

        }

    }

**else** **if** (QEvent::ChildRemoved == ev->type())

    {

        QChildEvent \*child\_ev = **static\_cast**<QChildEvent\*>(ev);

        QObject \*obj = child\_ev->child();

**if** (m\_child == obj)

        {

            m\_child = nullptr;

        }

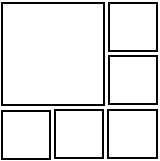
    }

**return** \_\_super::event(ev);

}

# QGridLayout：

QGridLayout中一个控件可以占多个栅格，如实现6分屏



m\_gridlayout->addWidget(m\_players[0], 0, 0, 2, 2); //0号位置

m\_gridlayout->addWidget(m\_players[1], 0, 2, 1, 1); //1号位置

m\_gridlayout->addWidget(m\_players[2], 1, 2, 1, 1); //2号位置

m\_gridlayout->addWidget(m\_players[3], 2, 0, 1, 1); //3号位置

m\_gridlayout->addWidget(m\_players[4], 2, 1, 1, 1); //4号位置

m\_gridlayout->addWidget(m\_players[5], 2, 2, 1, 1); //5号位置

其中，m\_players为Player\*数组，Player是自定义类，派生于QWidget

移除组件：m\_gridlayout->removeWidget(m\_players[i]);

获取QGridLayout中当前的组件数：m\_gridlayout->count()

但QGridLayout:: rowCount()返回的行数、QGridLayout:: columnCount()返回的列数并不等于实际的行列数

//QGridLayout在管理Grid里面的对象的时候将槽和对象分开的，槽就是行列数量

//addWidget的时候只会增加，removeWidget的时候不会减少

由行列获取控件指针：

QLayoutItem\* item = m\_gridlayout->itemAtPosition(row,col);

Player\* player = (Player\*)(item->widget());

遍历QGridLayout中的所有组件：

for (int i = 0; i < m\_gridlayout->count(); i++)

{

QLayoutItem\* item = m\_gridlayout->itemAt(i);

Player\* player = (Player\*)(item->widget());

}

# connect搭配lambda的一个陷阱：

可以是

connect(io, SIGNAL(readyRead()),this, SLOT(slotReadyRead())

或者connect(io, &QIODevice::readyRead, [this,io](){})

不能是connect(io, SIGNAL(readyRead()), [](){})这种形式

TestQt是主窗口，继承自QMainWindows，其中有按钮名为play\_btn

在TestQt的构造函数中有如下代码：

connect(ui.play\_btn, &QPushButton::clicked, this, [this](bool)

{

int\* loop = new int;

qDebug() << "out of connect lambda loop is" << (long long)loop;

connect(this, &TestQT::startupPlayBackFinished, this, [loop]()

{

qDebug() << "in connect lambda loop is" << (long long)loop;

});

emit this->startupPlayBackFinished();

auto calculate = [loop]()

{

qDebug() << "int general lambda loop is" << (long long)loop;

};

calculate();

});

运行结果如下：

第一次点击按钮

out of connect lambda loop is 2286508206144

in connect lambda loop is 2286508206144

int general lambda loop is 2286508206144

第二次点击按钮

out of connect lambda loop is 2286508205824

in connect lambda loop is 2286508206144

in connect lambda loop is 2286508205824

int general lambda loop is 2286508205824

可以发现，第二次点击按钮时，竟然触发了两次TestQT::startupPlayBackFinished的槽函数，原因如下：

第一次按钮按下，执行了connect(this, &TestQT::startupPlayBackFinished, this, [loop](){}}

第二次按钮按下，又执行了connect(this, &TestQT::startupPlayBackFinished, this, [loop](){}}，至此已经有两个槽函数绑定到this->startupPlayBackFinished()了

修改为

connect(this, &TestQT::startupPlayBackFinished, this, [this,loop]()

{

qDebug() << "in connect lambda loop is" << (long long)loop;

disconnect(this, SIGNAL(startupPlayBackFinished()), 0, 0);

});

则运行结果为：

第一次点击按钮

out of connect lambda loop is 1828696057008

in connect lambda loop is 1828696057008

int general lambda loop is 1828696057008

第二次点击按钮

out of connect lambda loop is 1828696057776

in connect lambda loop is 1828696057776

int general lambda loop is 1828696057776

# QDialog、QMainWindow、QWidget：

QWidget::winId()函数能够返回 当前Widget的HWND句柄

窗口部件（也称组件）分为顶级窗口部件和子窗口部件。

子窗口部件：被嵌入到一个父窗口部件的部件

顶级窗口部件：无父窗口部件。通常情况下，顶级窗口部件是默认有框架和标题栏的窗口。QMainWindow、QDialog都是顶级窗口部件。

QDialog虽然有parent()，但QDialog总是顶级窗口部件。如果它有一个父对象parent()，则它的默认位置在父对象的正中间。

MyMainWindow::MyMainWindow(QWidget \*parent): QMainWindow(parent)

{

AudioCapture\* audiocap = new AudioCapture(this);

audiocap->setModal(true);

audiocap->show();

}

以上代码中，AudioCapture为自定义QT组件

如果AudioCapture继承自QWidget，则是一个子窗口部件。默认位于主窗口MyMainWindow的左上角，且可能被主窗口已有的其它子部件遮挡，导致显示不全

如果AudioCapture继承自QDialog，则默认位于主窗口MyMainWindow的中央。是一个独立地浮于整个MyMainWindow的上方的对话框，可以自由移动、缩放，而不是被“嵌入”主窗口固定死。

如果是

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

{

QApplication a(argc, argv);

QWidget w;

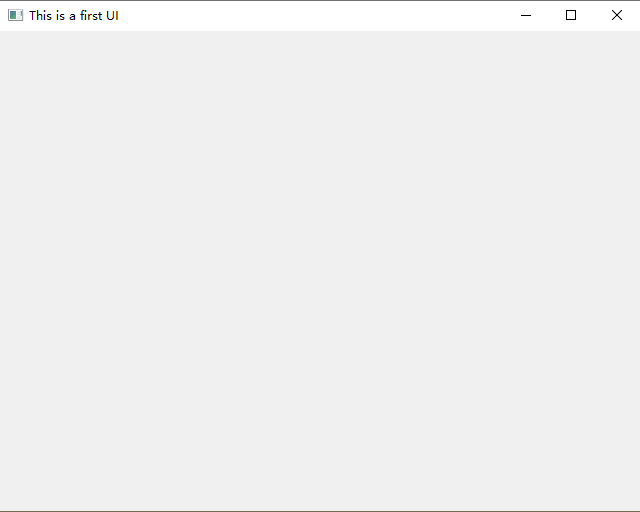
w.setWindowTitle("This is a first UI");

w.show();

return a.exec();

}

则效果如下图。因为代码中的QWidget w对象没有父组件，那w就是顶级窗口部件，拥有窗口的最大、最小、关闭等功能按钮，并且可以通过SetWindowTitle()成员函数设置窗口的标题等等



如果是

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

{

QApplication a(argc, argv);

TestQTSyntax w; //TestQTSyntax是自定义的QMainWindow的子类

w.show();

QDialog wi;

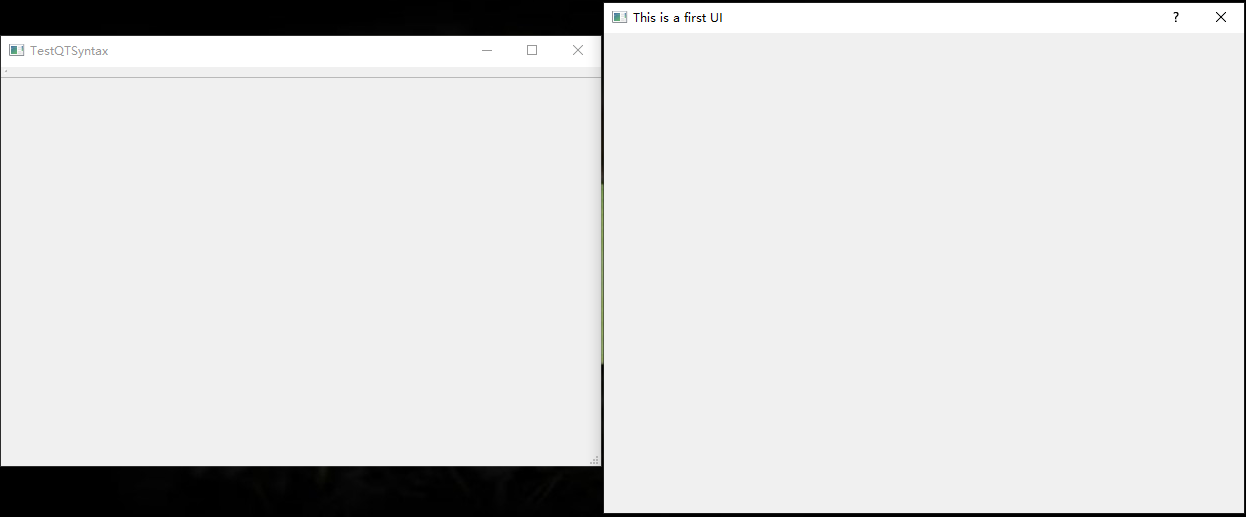
wi.setWindowTitle("This is a first UI");

wi.show();

return a.exec();

}

则This is a first UI对话框和TestQTSyntax窗口完全独立，关闭其中一个不影响另一个



如果是

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

{

QApplication a(argc, argv);

TestQTSyntax w; //TestQTSyntax是自定义的QMainWindow的子类

w.show();

QDialog wi(&w);

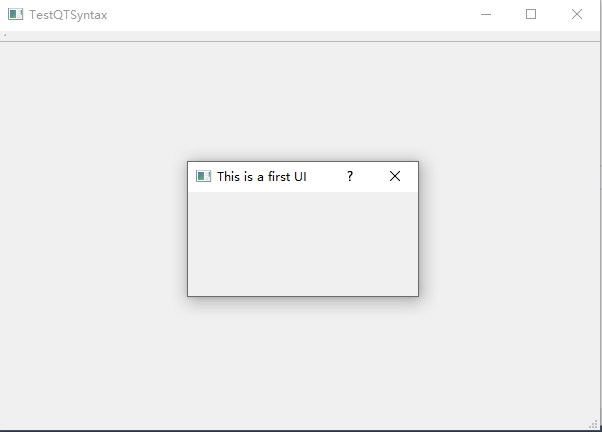
wi.setWindowTitle("This is a first UI");

wi.show();

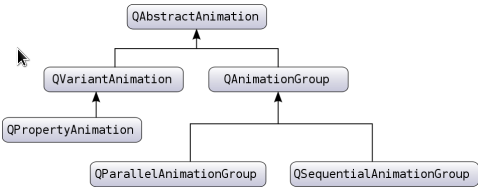
return a.exec();

}

此时效果如下图，This is a first UI对话框位于TestQTSyntax窗口正中央，二者都可独立自由移动，不受互相影响，但关闭TestQTSyntax窗口时This is a first UI对话框也随之自动关闭



QT动画：



* 动画循环次数

QT动画设置循环次数：setLoopCount(int n)，默认一次，0则动画不运行，-1则动画无限循环

QPropertyAnimation\* m\_anim = createDanmuAnim();

m\_anim->start(QAbstractAnimation::DeleteWhenStopped);

QPropertyAnimation\* RollingNotifier::createDanmuAnim()

{

QPropertyAnimation\* anim = new QPropertyAnimation(this, "pos");

anim->setStartValue(QPoint(0,0));

anim->setEndValue(QPoint(1000, 0));

if (m\_msg.style.rolling\_mode == QStringLiteral("从右到左"))

{anim->setDirection(QAbstractAnimation::Forward);}

else if (m\_msg.style.rolling\_mode == QStringLiteral("从左到右"))

{anim->setDirection(QAbstractAnimation::Backward);}

anim->setLoopCount(-1);//setLoopCount (-1)则一直循环直到被stop

anim->setDuration(3000);

anim->setEasingCurve(QEasingCurve::Linear);

return anim;

}

动画开始后、未结束前，再次设置起始结束点、持续时间，则下一个循环会生效，如下

m\_anim ->setStartValue(QPoint(0, 400));

m\_anim ->setEndValue(QPoint(1000, 400));

m\_anim->setDuration(5000);

* 动画终止

动画终止有以下几种情况：

1. 动画运行完毕正常结束，会发送stateChanged(QAbstractAnimation::Running, QAbstractAnimation::Stopped)信号和finished信号
2. 手动调用stop()提前结束动画，发送stateChanged信号，不发送finished信号
3. 新的动画触发，则正运行的动画自动结束，相当于对正运行的动画调用了stop()

第三种情况举例：

（1）QPropertyAnimation\* anim1= new QPropertyAnimation(this, "pos");

……// anim1设置属性

QPropertyAnimation\* anim2= new QPropertyAnimation(this, "pos");

……// anim2设置属性

anim1->start();

anim2->start()

第一个动画刚启动，就被终止，实际上仅第二个动画被执行

即使使用并行动画组也不能使这两个动画并行：

QParallelAnimationGroup group;

group.addAnimation(anim1);

group.addAnimation(anim2);

group.start(); //结果是两个动画都没有被执行

并行动画组仅对两个不同属性的动画有效，如透明度和位置，如果两个动画都设置pos属性，则依旧会冲突

（2）按下按钮触发动画，动画持续时间10s，第一次按下按钮后，过了3s，此时第一个动画还未结束，再次按下按钮，则新动画触发，旧动画自动终止且不会发送finished信号

connect(this, &CustomizedPushButton::clicked, this, [this]()

{

int x = this->geometry().x();

QPropertyAnimation\* up = new QPropertyAnimation(this, "pos");

up->setDuration(10000);

up->setLoopCount(1);

up->setStartValue(QPoint(x + 10, this->y()));

up->setEndValue(QPoint(x, this->y()));

up->setEasingCurve(QEasingCurve::OutBounce);

connect(up, &QPropertyAnimation::finished, [x, up, this]()

{

qDebug() << up << " " << up->currentTime() << ", slot finished";

});

connect(up, &QPropertyAnimation::stateChanged,

[up, this](QAbstractAnimation::State new\_state, QAbstractAnimation::State old\_state)

{

if (old\_state == QAbstractAnimation::Stopped && new\_state == QAbstractAnimation::Running)

{

qDebug() << up << " " << up->currentTime() << ", start";

}

if (old\_state == QAbstractAnimation::Running && new\_state == QAbstractAnimation::Stopped)

{

qDebug() << up << " " << up->currentTime() << ", go to stop state";

}

});

up->start(QAbstractAnimation::DeleteWhenStopped);

});

运行结果：

//QPropertyAnimation(0x2258a5065b0) 0, start

//QPropertyAnimation(0x2258a5065b0) 3392, go to stop state

//QPropertyAnimation(0x2258a505ca0) 0, start

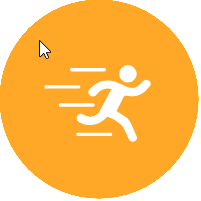
//QPropertyAnimation(0x2258a505ca0) 10000, go to stop state

//QPropertyAnimation(0x2258a505ca0) 10000, slot finished

# QT定制按钮

按钮有图标，圆形

按下后，会上下弹跳，连续点击（上次点击后的动画还没结束就再次点击）按钮位置不会漂移



CustomizedPushButton::CustomizedPushButton(QWidget \*parent)

: QPushButton(parent)

{

this->setFixedSize(100,100);

this->setIcon(QIcon(QStringLiteral("奔跑.png"))); //图片大小为200\*200

this->setIconSize(QSize(100,100)); //回自动缩放图片而不是裁剪

this->setStyleSheet("CustomizedPushButton{border:0px}");

//设置为无边框，否则能看出按钮是个矩形

//设置无边框后，由于Icon是个圆形，圆外全透明，因此按钮看起来也是个圆形

connect(this, &CustomizedPushButton::clicked, this, [this]()

{

static int y;

static bool first\_time = true;

if (first\_time)

{

y = this->geometry().y();

first\_time = false;

}

QPropertyAnimation\* up = new QPropertyAnimation(this, "pos");

up->setDuration(150);

up->setLoopCount(1);

up->setStartValue(QPoint(this->x(), y));

up->setEndValue(QPoint(this->x(), y-100));

up->setEasingCurve(QEasingCurve::OutQuad); //模拟小球从地板弹起

QPropertyAnimation\* down = new QPropertyAnimation(this, "pos");

down->setDuration(150);

down->setLoopCount(1);

down->setStartValue(QPoint(this->x(), y-100));

down->setEndValue(QPoint(this->x(), y));

down->setEasingCurve(QEasingCurve::InQuad); //模拟小球自由落体

QSequentialAnimationGroup\* group = new QSequentialAnimationGroup(this);

group->setLoopCount(5);

group->addAnimation(up);

group->addAnimation(down);

group->start(QAbstractAnimation::DeleteWhenStopped); //结果是两个动画都没有被执行

});

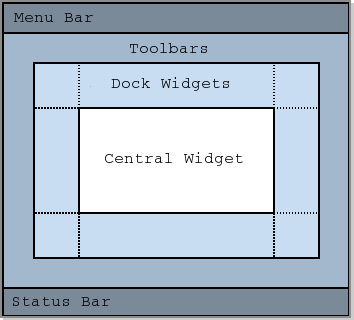
}

# QMainWindow：

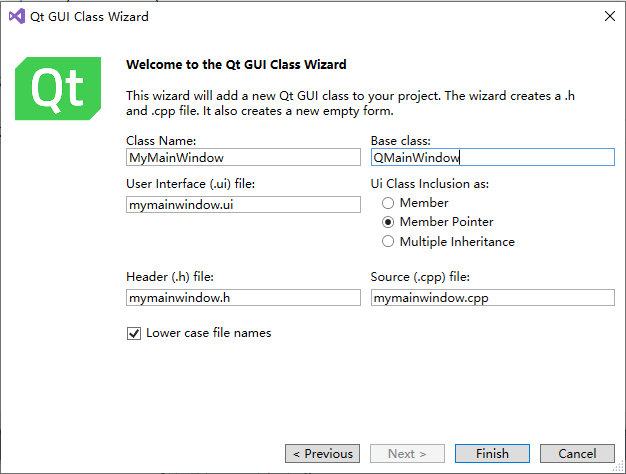
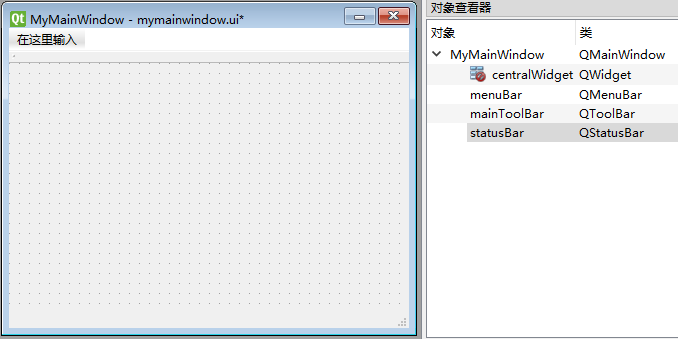
QWidget的子类有：QMainWindow、QDialog等

通过子类化QMainWindow可以创建一个应用程序的窗口

MainWindow的结构分为五个部分：菜单栏（Menu Bar）、工具栏（Toolbars）、停靠窗口（Dock Widgets）、状态栏（Status Bar）和中央窗口（Central Widget）



VS创建Qt GUI Class，选择QMainWindow为基类，则默认包含中央窗口、菜单栏、工具栏、状态栏这几部分，其中除中央窗口外的其它部分可手动删除。如下：

QWidget \* QmainWindow:: centralWidget()返回主窗口的central widget

QMenuBar\* QmainWindow::**menuBar**()返回主窗口的菜单栏

#include "mainwindow.h"

#include <QApplication>

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

{

    QApplication a(argc, argv);

    MainWindow w;   //MainWindow继承自QmainWindow

    w.setWindowTitle(QString::fromUtf8("simplest calculator")); //设置窗口标题

    w.show(); //窗口默认不可见的，调用 show()函数使变为可见。

    QWidget\* centwid=w.centralWidget();

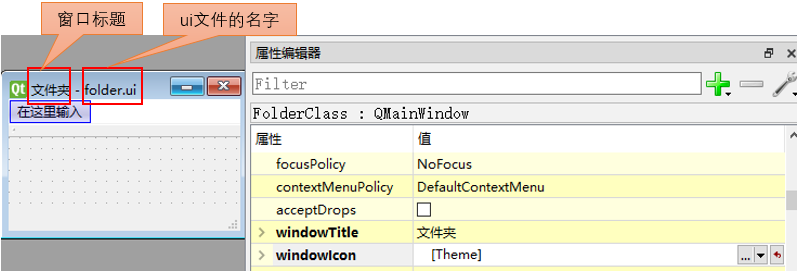
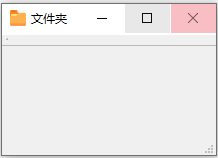
    centwid->setStyleSheet("background-color: gray;"); //将中央窗口设置为灰色

**return** a.exec(); //程序进入消息循环，等待可能的输入进行响应

}

# QMainWindow的窗口标题、图标

新建一个QMianWindows，在QT Designer打开自动生成的.ui文件

在QT属性编辑器中：

设置windowTitle（窗口标题）选项相当于在代码中调用setWindowTitle(QString::fromUtf8("文件夹"));

设置windowIcon（窗口图标）选项相当于setWindowIcon(QIcon("Resources/folder\_512x512.png"));

注意，QT Designer中不能显示出窗口图标，即使在QT属性编辑器进行了设置。图中，窗口标题前的蓝色QT图标，是QT Designer打开.ui文件时自动显示的，并不是窗口真正的图标

由图片路径构造QIcon作为窗口图标，图标大小根据窗口大小自动调整，并不是图片的大小

要捕获QMainWindow标题栏上的鼠标双击等事件，继承之并覆盖bool event(QEvent\*)函数

# 向QMianWindow增加组件：

* 增加QSpinBox至工具栏：

Folder::Folder(QWidget \*parent) : QMainWindow(parent)

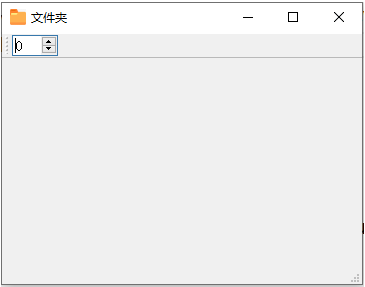
{

ui.setupUi(this);

lineWidthSpinBox = new QSpinBox;

ui.mainToolBar->addWidget(lineWidthSpinBox);

}



* 增加QSpinBox至centralWidget：

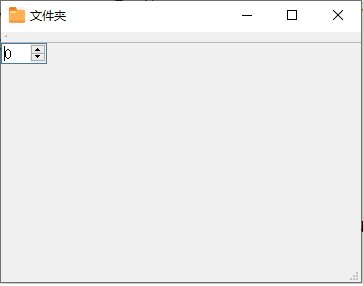
Folder::Folder(QWidget \*parent) : QMainWindow(parent)

{

ui.setupUi(this);

lineWidthSpinBox = new QSpinBox(ui.centralWidget);

}



* 添加至QMainWindow的特定位置：

Folder::Folder(QWidget \*parent) :

QMainWindow(parent)

{

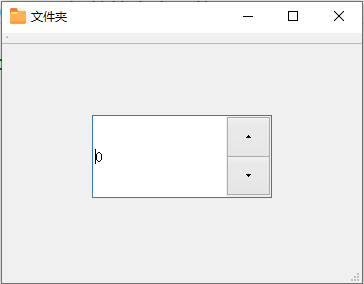
ui.setupUi(this);

lineWidthSpinBox = new QSpinBox(this);

lineWidthSpinBox->setGeometry(this->x() + this->width() / 4, this->y() + this->height() / 3, this->width() / 2, this->height() / 3); //设置组件的大小、位置，如果不设置，则有默认位置大小

//如果上文new QSpinBox时没有传入parent参数，则QSpinBox显示不出来

}



# Qt5 error LNK2019 无法解析的外部符号:

右键已有QT Creator工程，添加QT设计师界面类，则同时出现namedialog.h、namedialog.cpp、namedialog.ui文件，但编译时报错Qt5 error LNK2019 无法解析的外部符号

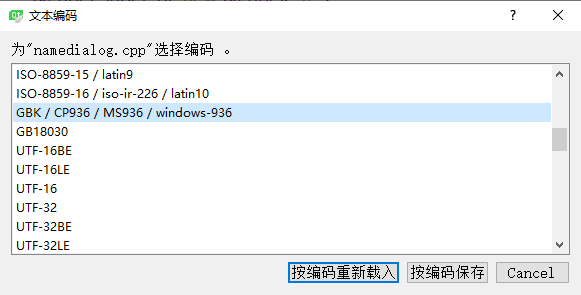
解决办法：将当前工程的构建目录，即形如build-(工程名)-Desktop\_Qt\_5\_0\_2\_MSVC2010\_32bit-Debug 的文件夹，删除，再构建项目即可运行

# Qt Creator源代码中出现中文

Qt Creator源代码中出现中文时编译报错: error: C2001: 常量中有换行符

分析：QT Creator源代码的编码格式与编译器不匹配

解决方法：如果一定要用MSVC编译器，则将源代码的编码格式设置为GBK，即菜单栏-->编辑-->Select Encoding-->GBK-->按编码保存



简体中文Windows系统下：



# QT控件显示中文：

采用MSVC编译器，源代码采用GBK编码时

QLabel\* label\_qi=new QLabel("齐");

控件显示乱码，



分析：

QString内部采用的是 16位的Unicode，即每个字符在内存中是2字节的Unicode

对QString qs="齐";语句：汉字“齐”在内存中为0xC6EB（MSVC编译器对字符串GBK编码置于内存），要将char\* 转成Unicode的QString字符串的，需要告诉QString这串char\* 中究竟是什么编码，即采用以下函数：

QString QString::fromAscii ( const char \* str, int size = -1 )

QString QString::fromLatin1 ( const char \* str, int size = -1 )

QString QString::fromLocal8Bit ( const char \* str, int size = -1 )

QString QString::fromUtf8 ( const char \* str, int size = -1 )

在简体中文Windows下，local8Bit是GBK

因此，采用MSVC编译器，源代码采用GBK编码，要使控件显示中文，应该

QLabel\* label\_qi=new QLabel(QString::fromLocal8Bit("齐"));

QLabel\* label\_chu=new QLabel("sad");

QLabel\* label\_qin=new QLabel("bad");

QLabel\* label\_yan=new QLabel("fuck");



或者QLabel\* label\_qi=new QLabel(QStringLiteral("齐"));

QStringLiteral是一个宏，在编译期把代码里的字符串常量直接构造为QString对象

# QBoxLayout::addStretch

void QBoxLayout::addStretch ( int stretch = 0 )按比例分配空白

QLabel\* label\_qi=new QLabel(QString::fromLocal8Bit("齐"));

QLabel\* label\_chu=new QLabel(QString::fromLocal8Bit("楚"));

QLabel\* label\_qin=new QLabel(QString::fromLocal8Bit("秦"));

QLabel\* label\_yan=new QLabel(QString::fromLocal8Bit("燕"));

QHBoxLayout \*layout = new QHBoxLayout;

layout->addWidget(label\_qi);

layout->addStretch(3);

layout->addWidget(label\_chu);

layout->addStretch(1);

layout->addWidget(label\_qin);

layout->addStretch(1);

layout->addWidget(label\_yan);

水平布局，除Qlabel控件自身占据的空间外，其余为空白部分，将这些空白部分按比例分为3：1：1



# QPushButon：

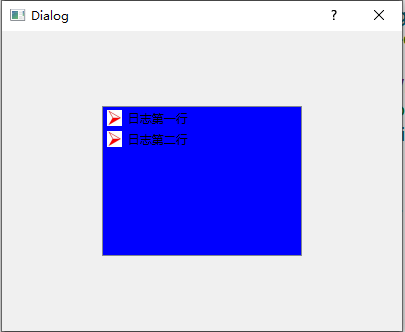
QPushButton::setEnabled(false)，则按钮无效，不可点击，如下图中结束、离开两个按钮即



向QMenu添加QAction，要使QAction无效，act->setDisabled(true);

# QListWidget：

QListWidget以列表形式显示数据，形如。既可添加文字也可添加图标



由图片路径构造Qicon，显示时Qicon大小自动调整为合适

LogDialog::LogDialog(QWidget \*parent) :QDialog(parent),ui(**new** Ui::LogDialog)

{

    ui->setupUi(**this**);

    QListWidget\* listwgt=**new** QListWidget(**this**);

    listwgt->setGeometry(**this**->x()+**this**->width()/4,**this**->y()+**this**->height()/4,

**this**->width()/2,**this**->height()/2); //设置组件的大小、位置

    listwgt->setStyleSheet("background-color: blue;"); //设置颜色

    QListWidgetItem\* item=**new** QListWidgetItem();

    item->setIcon(QIcon("../resources/log\_arrow.JPG")); //以QT Creator构建目录为基准的相对路径

    item->setText(QString::fromLocal8Bit("日志第一行"));// MSVC编译器编译器

listwgt->addItem(item);

    item=**new** QListWidgetItem(QIcon("../resources/log\_arrow.JPG"),QString::fromLocal8Bit("日志第二行"));

    listwgt->addItem(item);

}

# QSignalMapper：

QSignalMapper类可以看成是信号的翻译和转发器，它可以把一个无参的信号翻译成带int参数或QString参数或QObject\*参数或QWidget\*参数的信号，并将之转发

setMapping函数的参数只有四种，并且要严格按照格式写入，第一种const QString&，第二种int，第三种QObject\*，第四种QWidget\*，对于后两种，需要的是他们的子类，则在信号处理的函数里进行类型转化

MyWidget::MyWidget(QWidget \*parent) :  QWidget(parent)

{

    QString buttonText = "btn1,btn2,btn3,btn4,btn5,btn6,btn7,btn8,btn9,btn10";

    QStringList textList = buttonText.split(",");

    signalMapper = **new** QSignalMapper(**this**);

    QGridLayout \*gridLayout = **new** QGridLayout;

**for**(**int** i=0; i<textList.size(); ++i)

    {

        //动态创建按钮

        QPushButton \*button = **new** QPushButton(textList[i]);

        //原始信号传递给signalMapper

        connect(button,SIGNAL(clicked()),signalMapper,SLOT(map()));

        //设置signalMapper的转发规则，转发为参数为QString类型的信号，并把textList[i]的内容作为实参传递

        signalMapper->setMapping(button, textList[i]);

        gridLayout->addWidget(button, i/3, i%3);

    }

    //将转发的信号连接到最终的槽函数上

    connect(signalMapper,SIGNAL(mapped(QString)),**this**,SLOT(doClicked(QString)));

    setLayout(gridLayout);

}

信号->槽函数

信号->QSignalMapper::map()槽函数-> QSignalMapper信号->槽函数

map()作为QSignalMapper的一个槽函数，根据setMapping规则转发mapped()信号

# QComboBox组合框：



QComboBox的常用方法和属性：

* 在QComboBox的最后添加一项。

lineStyleComboBox = new QComboBox;

lineStyleComboBox->addItem("实线", static\_cast<int>(Qt::SolidLine));

lineStyleComboBox->addItem("虚线", static\_cast<int>(Qt::DashLine));

addItem的第二个参数，可以在之后被读出来，如

ui->combox\_sample\_rate->addItem("48 KHz", QVariant(48000));

ui->combox\_fmt->addItem("Signed 16", qVariantFromValue(std::pair<QAudioFormat::SampleType, int>(QAudioFormat::SignedInt, 16)));

ui->combox\_sample\_rate->itemData(ui->combox\_sample\_rate->currentIndex()).toInt()

const QVariant var = ui->combox\_fmt->itemData(ui->combox\_fmt->currentIndex());

std::pair<QAudioFormat::SampleType, int> type\_bits = var.value<std::pair<QAudioFormat::SampleType, int>>();

* int count () const返回列表项总数。
* int currentIndex () const当前显示的列表项序号。
* QString currentText () const返回当前显示的文本。
* 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 )

插入一项或多项至序号index处。

* void insertSeparator ( int index )在序号为index的项前插入分隔线
* void setItemText ( int index, const QString & text )改变序号为index项的文本。



信号与槽：connect(lineStyleComboBox, SIGNAL(activated(int)), this, SLOT(showLineStyle()));

访问数据：lineStyleComboBox->currentData().toInt()

在MainWindow.ui中添加组合框后，设置包含的选项

MainWindow::**MainWindow**(QWidget \*parent) :QMainWindow(parent),ui(new Ui::MainWindow)

{

ui->setupUi(this);

ui->mood\_combox->addItem(QString::fromLocal8Bit("实线"));

ui->mood\_combox->addItem(QString::fromLocal8Bit("虚线"));

ui->mood\_combox->addItem(QString::fromLocal8Bit("点线"));

ui->mood\_combox->insertSeparator(2);//在序号为index的项前插入分隔线

}

右键添加槽函数，

void MainWindow::**on\_mood\_combox\_currentIndexChanged**(const QString &arg1)

{

QMessageBox::information(this, QString::fromLocal8Bit("提示"),

QString::fromLocal8Bit("你选择了")+arg1);

}

这样，每次程序刚启动，主窗口尚未显示，就会弹出“你选择了实线”消息框，然后点击确定后才是MainWindow界面，如何避免？

将n\_mood\_combox\_currentIndexChanged(函数)改为

void MainWindow::**on\_mood\_combox\_currentIndexChanged**(const QString &arg1){

static bool first\_call=true;

if(first\_call==true){first\_call=false;return;}

else{QMessageBox::information(this, QString::fromLocal8Bit("提示"),

QString::fromLocal8Bit("你选择了")+arg1);}

}

# QSpinBox和QDoubleSpinBox步进框

QSpinBox和QDoubleSpinBox都继承自QAbstractSpinBox类，大多数属性相同。

QSpinBox可用于显示和输入整数，并可以在显示框中添加前缀或后缀。

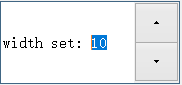
QDoubleSpinBox可用于显示和输入小数，并可以在显示框中添加前缀和后缀。

lineWidthSpinBox = new QSpinBox(this);

lineWidthSpinBox->setPrefix("width set: ");

lineWidthSpinBox->setMinimum(10); //设置可输入的最小值，不设置则采用默认值

lineWidthSpinBox->setMaximum(1000); //设置可输入的最大值，不设置则采用默认值



用户可以输入数据，也可以点击带有上下箭头的步进按钮来增减

信号与槽：connect(lineWidthSpinBox, SIGNAL(valueChanged(int)), imgDrawWidget, SLOT(setLineWidth(int)));

# QMessageBox消息弹窗：

派生体系：QWidget->QDialog->QMessageBox

QMessageBox::information(this, QString::fromLocal8Bit("警告"),QString::fromLocal8Bit("有种你再点击一次"));

# QT的父子对象机制：

QT的父子对象机制是在QWidget和QOject中实现的。使用父对象来创建一个对象的时候，父对象会把这个对象添加到自己的子对象列表中。当这个父对象被删除的时候，它会遍历它的子对象类表并且删除每一个子对象，然后子对象们自己再删除它们自己的子对象，这样递归调用直到所有对象都被删除。

需要显式手动删除的对象是那些使用new创建的并且没有父对象的对象。如果在删除一个对象的父对象之前手动删除它，QT会自动地从它的父对象的子对象列表中移除它。

父子对象机制是main函数里面mainwidget/dialog/mainWindow要分配在栈上的原因。其他new出来的东西都以这个widget作为父亲。当程序最后结束了，mainwidget弹栈。父类被销毁。子类跟着被销毁。所以是手动new出来的。没有父类，不删除就会造成内存泄漏

#include "mainwindow.h"

#include <QApplication>

#include <QTextCodec>

#include <QLabel>

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

{

QApplication a(argc, argv);

MainWindow w;

QLabel \*label =new QLabel("hello",&w);

//这里使用new之后不需要执行delete，因为label的父类是w，而w是在栈中创建，在程序关闭的时候会自动释放，所以作为w的子类内存也被释放。

QLabel \*label1 =new QLabel("world");

//这里必须需要执行delete label1，否则会造成内存泄漏，因为label没有父类，所以不会自动释放内存

w.show();

a.exec();

delete label1;

return 0;

}

new某个组件的时候，没有设置父对象，但之后调用了addWidget，也会形成父子关系。因此也不需要手动delete

lineStyleLabel = new QLabel("线型: ");

qDebug()<< ui->mainToolBar; // QToolBar(0x29aae7a0, name="mainToolBar")

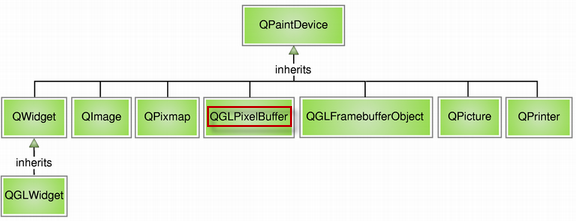
qDebug()<<lineStyleLabel->parent(); // QObject(0x0)

ui->mainToolBar->addWidget(lineStyleLabel);

qDebug()<<lineStyleLabel->parent(); // QToolBar(0x29aae7a0, name = "mainToolBar")

# QPixmap、QPainter的关系

绘制设备：



* QT5中，QGLPixelBuffer已被废弃
* QGLWidget和QGLFramebufferObject，是关于OpenGL的相关类
* QPixmap专门为图像在屏幕上的显示做了优化；QBitmap是QPixmap的一个子类，它的色深限定为1（二值图），可以使用QPixmap的isQBitmap()函数来确定这个QPixmap是不是一个QBitmap
* QImage专门为图像的像素级访问做了优化。
* QPicture则可以记录和重现QPainter的各条命令

void ImgDrawWidget::resizeEvent (QResizeEvent \*event)

{

//创建一个新的QPixmap对象，size()表示当前客户区的大小

QPixmap \*newPix = new QPixmap(size());

newPix->fill (Qt::transparent);

//以QPixmap为参数构建QPainter，之后绘制操作都是绘制在QPixmap上

QPainter p(newPix);

//在newPix上绘制，绘制内容为m\_pixmap，左上角坐标为QPoint(0,0)

p.drawPixmap (QPoint(0,0), \*m\_pixmap);

m\_pixmap = newPix; //将newPix赋值给m\_pixmap作为新的工作画板

QWidget::resizeEvent (event);

}

# QLineEdit::setValidator

QIntValidator类提供了一个验证器，确保字符串描述的整数在有效范围内

ui->hostPort->setValidator(new QIntValidator(0, 65535, ui->hostPort));

hostPort是QLineEdit\*类型，用于输入端口

此外还可借助正则表达式限定输入到QLineEdit的字符串的格式

QRegExp rx("^((2[0-4]\\d|25[0-5]|[01]?\\d\\d?)\\.){3}(2[0-4]\\d|25[0-5]|[01]?\\d\\d?)$");

ui->hostIp->setValidator(new QRegExpValidator(rx, this));

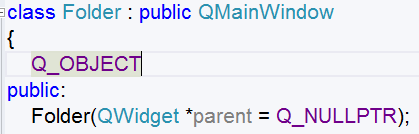
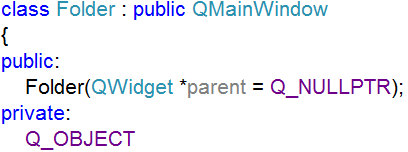
hostPort是QLineEdit\*类型，用于输入ip

# QT的信号和槽函数：

信号和槽是QT自行定义的一种通信机制，仅依靠C/C++ 语言标准并不能实现，因此要正确的处理信号和槽，必须借助一个称为 moc.exe（Meta Object Compiler）的 QT 工具，该工具是一个 C++预处理程序，它为高层次的事件处理自动生成所需要的附加代码。

当一个信号被发射时，与其相关联的槽将被立刻执行，就像一个正常的函数调用一样（但比直接调用非虚函数慢10倍）。信号-槽机制完全独立于任何GUI事件循环。只有当所有的槽返回以后发射函数（emit）才返回

一个类若要支持信号和槽，就必须继承QObject或QObject 的子类，且将Q\_OBJECT宏置于类的private区，形如

或

connect()函数原型：

**bool** QObject::connect(**const** QObject \* sender, **const** **char** \* signal,

**const** QObject \* receiver, **const** **char** \* member)

如果接收者为this，则可以省略，即

QObject::connect(**const** QObject \*asender, **const** **char** \*asignal, **const** **char** \*amember, Qt::ConnectionType atype)

{**return** connect(asender, asignal, **this**, amember, atype);}

connect关联的信号和槽的签名要么等同，即参数类型和参数个数相同，要么槽的参数个数少于信号的参数个数的，但缺少的参数必须是信号参数的最后一个或几个参数。否则，编译报错

connect()函数支持：

1. 一个信号a连接到另一个信号b，则a信号的发射触发b信号的发射
2. 一个信号连接到多个槽
3. 多个信号连接到同一个槽

connect (Objectl,SIGNAL(signal2),Object2,SIGNAL(slot2));

connec (Object3,SIGNAL(signal2) ,Object2,SIGNAL(slot2));

SIGNAL() 和SLOT() 是宏，它们返回其参数的C语言风格的字符串 const char\* 。

#define SIGNAL(arg) #arg

#define SLOT(arg) #arg

但不意味着以下两个语句等同，因为SIGNAL宏由moc.exe解析

connect(button, SIGNAL(clicked()), this , SLOT(showArea()));//正确连接了信号与槽  
connect(button, "clicked()",this, "showArea()"); //未能连接信号与槽

当信号与槽没有必要继续保持关联时，使用 disconnect 函数来断开连接

bool QObject::disconnect(const QObject \* sender, const char \* signal, const Object \* receiver, const char \* member)

在disconnect函数中0可以用作一个通配符，分别表示任何信号、任何接收对象、接收对象的任何槽函数：

* 断开与某个对象相关联的任何对象。当在某个对象中定义了一个或者多个信号，这些信号与另外若干个对象中的槽相关联，如果要切断这些关联的话，可以利用这个方法

disconnect( myObject, 0, 0, 0 ) 或者

myObject->disconnect()

* 断开与某个特定信号的任何关联。

isconnect( myObject, SIGNAL(mySignal()), 0, 0 ) 或者

myObject->disconnect( SIGNAL(mySignal()) )

* 断开两个对象之间的关联

disconnect( myObject, 0, myReceiver, 0 )

或者myObject->disconnect(myReceiver )

槽函数是类的成员函数。定义槽时，必须借助slots宏，且slots前面必须有private或者public修饰符，否则编译报错。形如



槽函数可以像普通的对象成员函数一样被直接调用

但connect连接到信号的必须是槽函数，不能是普通成员函数

slots宏由moc.exe解析

信号可以重载，信号没有函数体定义，并且信号的返回类型都是void。

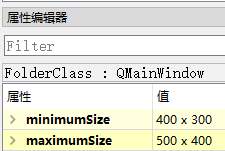
signals关键字指出进入了信号声明区，随后即可声明自己的信号。

signals:

void overflow();

# 设置窗口大小：

QT Designer中设置：



等同于：

QWidget::setMaximumSize(500,400);

QWidget::setMinimumSize(400, 300);

如果最大size和最小size一致，则窗口不能缩放

设置的最大size如果较小，则最大化按钮自动失效

但无论怎么设置最小size、最大size，都不影响最小化按钮

# 设置字体：

QLabel\* label = new QLabel(this);

label->setText(QString::fromLocal8Bit ("秦"));

label->setFont(QFont("Times",36,QFont::Weight::Bold,true)); //字体类型、大小、加粗、是否斜体

QDateTime QDateTime::currentDateTime()获取本地系统时间和日期

QDate QDateTime::data() //日期部分

QTime QDateTime::time() //时间部分

# QTimer定时器

设置定时时间，start(int time)和setSingleShot(true) 。

start(int time)表示每隔 "time" 毫秒就会重启定时器，可以重复触发定时，可利用stop()将定时器掉； setSingleShot(true) 则是仅启动定时器一次

label\_dt = new QLabel;

label\_dt->setFont(QFont("Times",20,QFont::Weight::Bold,true));

label\_dt->setText(QDateTime::currentDateTime().toString());

label\_d = new QLabel;

label\_d->setFont(QFont("Times", 20, QFont::Weight::Bold, true));

label\_d->setText(QDateTime::currentDateTime().date().toString());

label\_t = new QLabel;

label\_t->setFont(QFont("Times", 20, QFont::Weight::Bold, true));

label\_t->setText(QDateTime::currentDateTime().time().toString());

QVBoxLayout\* vlayout = new QVBoxLayout(this);

vlayout->addWidget(label\_dt);

vlayout->addWidget(label\_d);

vlayout->addWidget(label\_t);

vlayout->addStretch();

ui.centralWidget->setLayout(vlayout);

QTimer\* timer = new QTimer(this);

timer->start(1000);

connect(timer,SIGNAL(timeout()),this,SLOT(slot\_timeout()));

void Folder::slot\_timeout()

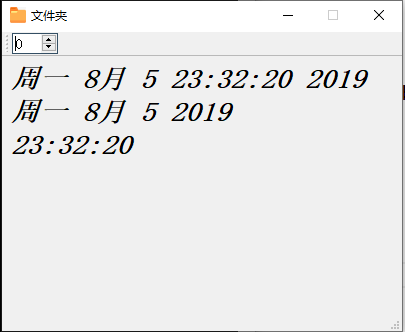
{

label\_dt->setText(QDateTime::currentDateTime().toString());

label\_d->setText(QDateTime::currentDateTime().date().toString());

label\_t->setText(QDateTime::currentDateTime().time().toString());

}

每隔1s更新一次显示的数据

# 对话框关闭、内存释放：

QMainWindow上有Join按钮，点击则弹出自定义的JoinDialog对话框，模态

最佳实现（1）：

void MainWindow::onClickJoin()

{

JoinDialog\* joinDialog = new JoinDialog(this);

joinDialog->setAttribute(Qt::WA\_DeleteOnClose);

joinDialog->setModal(true); //或者dlg.setAttribute(Qt::WA\_ShowModal, true);

joinDialog->show();

}

是一个模态对话框

每次点击Join按钮、叉掉，new一个JoinDialog，叉掉时释放堆中内存

设置QWidget::setAttribute(Qt::WA\_DeleteOnClose)，则当关闭对话框（叉掉或调用QDialog::close()）时，会释放内存。

叉掉对话框，等价于调用QDialog::close()

最佳实现（2）：

connect(ui.dlg\_btn, &QPushButton::clicked, this, [this](bool)

{

QDialog dlg;

dlg->setModal(true); //或者dlg.setAttribute(Qt::WA\_ShowModal, true);

dlg.show();

QEventLoop loop;

connect(&dlg, SIGNAL(finished(int)), &loop, SLOT(quit()));

loop.exec();

});

源码：

int QDialog::exec()

{

Q\_D(QDialog);

...

setAttribute(Qt::WA\_ShowModal, true);

...

show();

...

QEventLoop eventLoop;

(void) eventLoop.exec(QEventLoop::DialogExec);

...

}

如果是：

void MainWindow::onClickJoin(){

JoinDialog joinDialog(this);

joinDialog.setModal(true);

joinDialog.show();

}

则点击Join按钮后对话框一闪而过，没法在其上进行操作。因为函数onClickJoin结束，栈上对象joinDialog被析构

如果是：

void MainWindow::onClickJoin(){

JoinDialog\* joinDialog = new JoinDialog(this);

joinDialog->setModal(true);

joinDialog->show();

}

则每次点击Join按钮，new一个JoinDialog，叉掉对话框时，内存未被释放。下次点击Join按钮，又new一个新的JoinDialog，叉掉，内存未释放……直到父组件QMianDialog要销毁时，堆中多个JoinDialog内存才被释放。如果是将第一句代码替换为JoinDialog\* joinDialog = new JoinDialog，则内存将泄漏，即使父组件销毁，堆中多个JoinDialog内存也不会被释放。

# 模态、非模态：

模态对话框设置：

QDialog::setModal(true); //或者dlg.setAttribute(Qt::WA\_ShowModal, true);由QWidget提供

QDialog::show();

不等价于QDialog::exec();

两者都将对话框设置为模态，但

前者不会阻塞线程，即用户未操作对话框，代码都会继续执行QDialog::show()后边的语句

后者会阻塞线程，只有当对话框关闭后，语句QDialog::exec()才返回，代码继续向后执行

模态对话框指在子对话框弹出时，焦点被强行集中于该子对话框，子对话框不关闭，用户将无法操作其他的窗口。非模态对话框则相反，用户仍然可以操作其他的窗口，包括该子对话框的父对话框

除了直接调用setAttribute外，QWidget 提供了一个易用的函数，来设置窗体的模态。其源码如下：

void QWidget::setWindowModality(Qt::WindowModality windowModality)

{

data->window\_modality = windowModality;

// setModal\_sys() will be called by setAttribute()

setAttribute(Qt::WA\_ShowModal, (data->window\_modality != Qt::NonModal));

setAttribute(Qt::WA\_SetWindowModality, true);

}

注意：该函数的参数取值：NonModal、WindowModal、ApplicationModal 分别对应默认情况下的：

QDialog::show()

QDialog::open()

QDialog::exec()

# QString：

QString::fromStdString(duration) std::string转QString

QString::toStdString()QString转std::string

int x = QString("-").toInt(); //0，转换失败返回0

# 使用播放器库提供的接口：

第三方库提供的播放器接口：

WINDOWS\_CONTROL\_API IPlayerControl\* createWindowsControl(**const** **char**\* pluginPath, **const** **char**\*libPath, **const** **char**\* cfgPath, **void**\* hWnd);

**class** IPlayerControl{

**public**:

**virtual** **bool** play(**const** **char**\* src) = 0;

**virtual** **bool** stop() = 0;

**virtual** **void** registerPlayCallback(PlayState\_Callback\* p\_callback) = 0;

}

**class** PlayState\_Callback{

**public**:

**virtual** **void** updateDurationState(**const** **long** **long** cur\_duration) = 0;

**virtual** **void** updateSpeedState(**const** **int** cur\_speed) = 0;

};

如何使用？

设计子类继承PlayState\_Callback，覆盖父类的纯虚函数，形如

**class** Im\_PlayState\_Callback

{

**public**:

**virtual** **void** updateDurationState(**const** **long** **long** cur\_duration) {cur\_duration\_=cur\_duration;}

**virtual** **void** updateSpeedState(**const** **int** cur\_speed) {cur\_speed\_=cur\_speed;}

**private**:

**long** **long** cur\_duration\_;

**int** cur\_speed\_;

};

设计新类，对IPlayerControl进行包装，形如

**class** PlayControl{

**public**:

**bool** play(**const** **char**\* src) {ipc->play(src);}

**bool** stop() {ipc->stop();}

**long** **long** getDuration() {**return** callback.cur\_duration\_;}

**int** getSpeed(){**return** callback.cur\_speed\_;}

**private**:

    IPlayerControl\* ipc; //构造函数中createWindowsControl()赋值给ipc、ipc->registerPlayCallback(&callback)

    Im\_PlayState\_Callback callback;

};

如果第三方库对当前播放时间以回调的方式提供

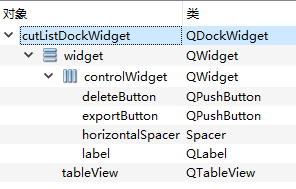
则在回调函数里发送信号，在类Player中设计槽函数——移动界面上时间条的滑块位置，完美！

# QDockWidgt停靠窗口：

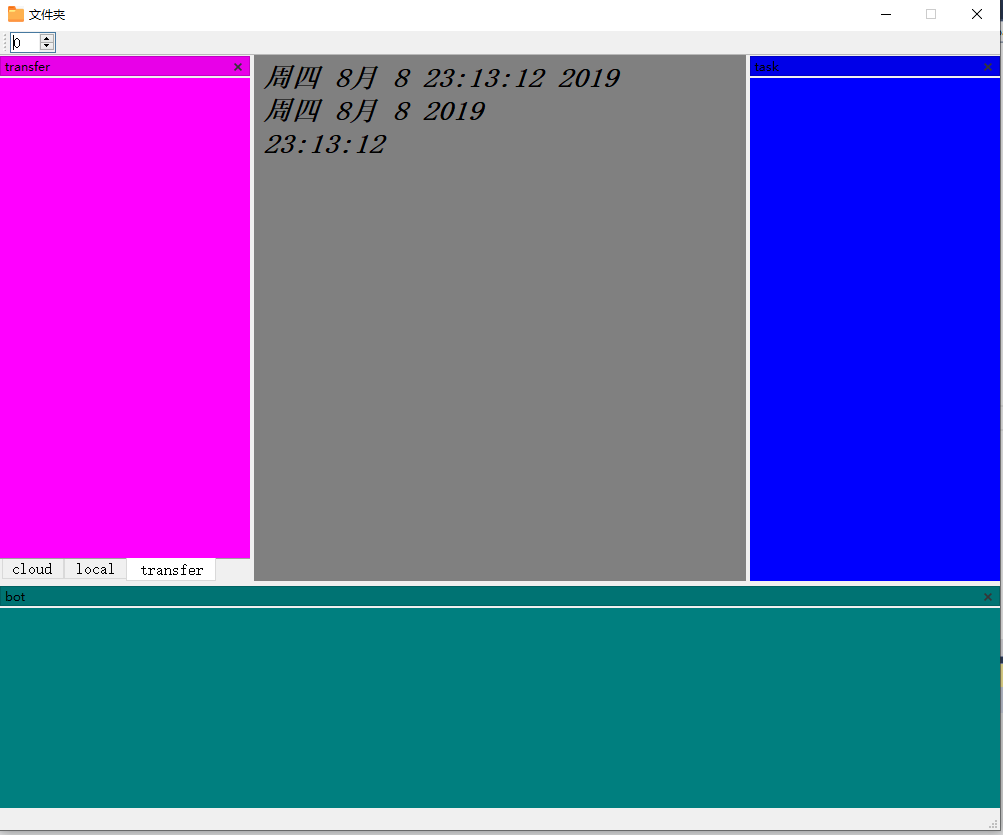
QDockWidget可设置最大size、最小size（setMinimumSize）、最小宽高、最大宽高

QDockWidget::resize()不能用于设置初始窗口大小

QDockWidget仅仅作为其子控件（QDockWidget::setWidget()设定的控件）的一个包装，对QDockWidget或其子类使用resize()或者setGeometry()均不管用



这里cutListDockWidget为QDockWidget的子类，widget即cutListDockWidget类对象的一个子控件



中间是一个时钟，每1秒更新一次时间

四周为5个停靠窗口

最底下为QMainWindow的状态栏，可以通过在主窗体的构造函数中delete ui.statusBar去掉

#include<QMessageBox>

#include<QLabel>

#include<QDebug>

#include<QTextCodec>

#include<string>

#include<iostream>

#include<algorithm>

#include<QTime>

#include<QTimer>

#include<QLayout>

#include<QDockWidget>

#include "folder.h"

**class** MyWidget :**public** QWidget

{

**public**:

    MyWidget(QWidget \*parent, **int** init\_w, **int** init\_h) :

        QWidget(parent),init\_width(init\_w), init\_height(init\_h) { }

    QSize sizeHint() **const**

    {

**return** QSize(init\_width, init\_height);

    }

**private**:

**int** init\_width;

**int** init\_height;

};

Folder::Folder(QWidget \*parent) : QMainWindow(parent)

{

    ui.setupUi(**this**);

    setMinimumSize(1000, 800);

    setMaximumSize(1500, 1200);

    //工具栏添加QSpinBox控件

    lineWidthSpinBox = **new** QSpinBox;

    ui.mainToolBar->addWidget(lineWidthSpinBox);

    connect(lineWidthSpinBox, SIGNAL(valueChanged(**int**)), SLOT(slot\_set\_linewidth(**int**)));

    //--------centralWidget设置--------------------

    label\_dt = **new** QLabel;

    label\_dt->setFont(QFont("Times",20,QFont::Weight::Bold,**true**));

    label\_dt->setText(QDateTime::currentDateTime().toString());

    label\_d = **new** QLabel;

    label\_d->setFont(QFont("Times", 20, QFont::Weight::Bold, **true**));

    label\_d->setText(QDateTime::currentDateTime().date().toString());

    label\_t = **new** QLabel;

    label\_t->setFont(QFont("Times", 20, QFont::Weight::Bold, **true**));

    label\_t->setText(QDateTime::currentDateTime().time().toString());

    QVBoxLayout\* vlayout = **new** QVBoxLayout(ui.centralWidget);

    vlayout->addWidget(label\_dt);

    vlayout->addWidget(label\_d);

    vlayout->addWidget(label\_t);

    vlayout->addStretch();

    QWidget\* center\_widget = **new** QWidget(**this**);

    center\_widget->setStyleSheet("background-color:gray");

    center\_widget->setLayout(vlayout);

    setCentralWidget(center\_widget);

    //也可以直接

    //ui.centralWidget->setStyleSheet("background-color:gray");

    //ui.centralWidget->setLayout(vlayout);

    //---------centralWidget设置----------------

    //以下为5个停靠窗口设置

    QDockWidget\* cloud\_list\_dock = **new** QDockWidget(**this**);

    cloud\_list\_dock->setWindowTitle("cloud");

    cloud\_list\_dock->setStyleSheet("background-color:#FF0000");

    QDockWidget\* local\_list\_dock = **new** QDockWidget(**this**);

    local\_list\_dock->setWindowTitle("local");

    local\_list\_dock->setStyleSheet("background-color:#00FF00");

    QDockWidget\* transfer\_list\_dock = **new** QDockWidget(**this**);

    transfer\_list\_dock->setWindowTitle("transfer");

    transfer\_list\_dock->setStyleSheet("background-color:#FF00FF");

    QDockWidget\* task\_list\_dock = **new** QDockWidget(**this**);

    task\_list\_dock->setWindowTitle("task");

    task\_list\_dock->setStyleSheet("background-color:#0000FF");

    QDockWidget\* bot\_list\_dock = **new** QDockWidget(**this**);

    bot\_list\_dock->setWindowTitle("bot");

    bot\_list\_dock->setStyleSheet("background-color:#007F7F");

    //默认可关闭、可移动、可悬浮

    //这里设置为可关闭、可移动、不可悬浮

    cloud\_list\_dock->setFeatures(QDockWidget::DockWidgetClosable| QDockWidget::DockWidgetMovable);

    local\_list\_dock->setFeatures(QDockWidget::DockWidgetClosable | QDockWidget::DockWidgetMovable);

    transfer\_list\_dock->setFeatures(QDockWidget::DockWidgetClosable | QDockWidget::DockWidgetMovable);

    task\_list\_dock->setFeatures(QDockWidget::DockWidgetClosable | QDockWidget::DockWidgetMovable);

    bot\_list\_dock->setFeatures(QDockWidget::DockWidgetClosable | QDockWidget::DockWidgetMovable);

    //--------设置QDockWidget的初始size-------

    QWidget\* cloud\_widget = **new** MyWidget(**this**, **this**->width() / 4, **this**->height() \* 3 / 4);

    QWidget\* local\_widget = **new** MyWidget(**this**, **this**->width() / 4, **this**->height() \* 3 / 4);

    QWidget\* transfer\_widget = **new** MyWidget(**this**, **this**->width() / 4, **this**->height() \* 3 / 4);

    QWidget\* task\_widget = **new** MyWidget(**this**, **this**->width() / 4, **this**->height() \* 3 / 4);

    QWidget\* bot\_widget = **new** MyWidget(**this**, **this**->width(), **this**->height() \* 1 / 4);

    cloud\_list\_dock->setWidget(cloud\_widget); //QDockWidget::setWidget()设置的值即QDockWidget::widget()返回的值

    local\_list\_dock->setWidget(local\_widget);

    transfer\_list\_dock->setWidget(transfer\_widget);

    task\_list\_dock->setWidget(task\_widget);

    bot\_list\_dock->setWidget(bot\_widget);

    //------设置QDockWidget的初始size---------

    addDockWidget(Qt::LeftDockWidgetArea, cloud\_list\_dock);

    addDockWidget(Qt::LeftDockWidgetArea, local\_list\_dock);

    addDockWidget(Qt::LeftDockWidgetArea, transfer\_list\_dock);

    addDockWidget(Qt::RightDockWidgetArea, task\_list\_dock);

    addDockWidget(Qt::BottomDockWidgetArea, bot\_list\_dock);

    ////Sets the given dock widget area to occupy the specified corner

    setCorner(Qt::TopLeftCorner, Qt::LeftDockWidgetArea);

    setCorner(Qt::TopRightCorner, Qt::RightDockWidgetArea);

    setCorner(Qt::BottomLeftCorner, Qt::BottomDockWidgetArea);

    setCorner(Qt::BottomRightCorner, Qt::BottomDockWidgetArea);

    tabifyDockWidget(cloud\_list\_dock, local\_list\_dock);  //把多个dock变成一个tab形式的窗体

    tabifyDockWidget(cloud\_list\_dock, transfer\_list\_dock);

    setDockNestingEnabled(**true**);

    QTimer\* timer = **new** QTimer(**this**);

    timer->start(1000);

    connect(timer,SIGNAL(timeout()),**this**,SLOT(slot\_timeout()));

}

**void** Folder::slot\_set\_linewidth(**int** width)

{

    line\_width = width;

    QMessageBox::information(**this**, "Line Width","line width get is "+QString::number(width));

}

**void** Folder::slot\_timeout()

{

    label\_dt->setText(QDateTime::currentDateTime().toString());

    label\_d->setText(QDateTime::currentDateTime().date().toString());

    label\_t->setText(QDateTime::currentDateTime().time().toString());

}

# 控件位置：

QPoint mainwind\_point = ((QMainWindow\*)(**this**->parent()))->pos(); //{x=95,y=34}，主窗口相对于屏幕的位置

QRect mainwind\_rect = ((QMainWindow\*)(**this**->parent()))->geometry();//{x=95,y=34,width=1728,height=910}

QPoint dock\_point = **this**->pos();//{x=1228,y=68}，节目单停靠窗口相对于主窗口的位置

QRect dock\_rect = **this**->geometry();//{x=1228,y=68,width=500,height=822}

QPoint treeview\_point = **this**->ui->treeView->pos(); //{x=0,y=45}，QTreeView相对于停靠窗口的位置

QRect treeview\_rect = **this**->ui->treeView->geometry();//{x=0,y=45,width=500,height=755}

QPoint headerview\_point = **this**->ui->treeView->header()->pos(); //{x=0,y=0}，QHeaderView相对于QTreeView的位置

QRect headerview\_rect = **this**->ui->treeView->header()->geometry();//{x=0,y=0,width=500,height=25}

QRect cell\_rect=ui->treeView->visualRect(index);  //{x=20,y=0,width=80,height=25}//首个单元格相对于QTreeView表格体的位置

# 设置控件背景色：

在QT Designer中设置styleSheet

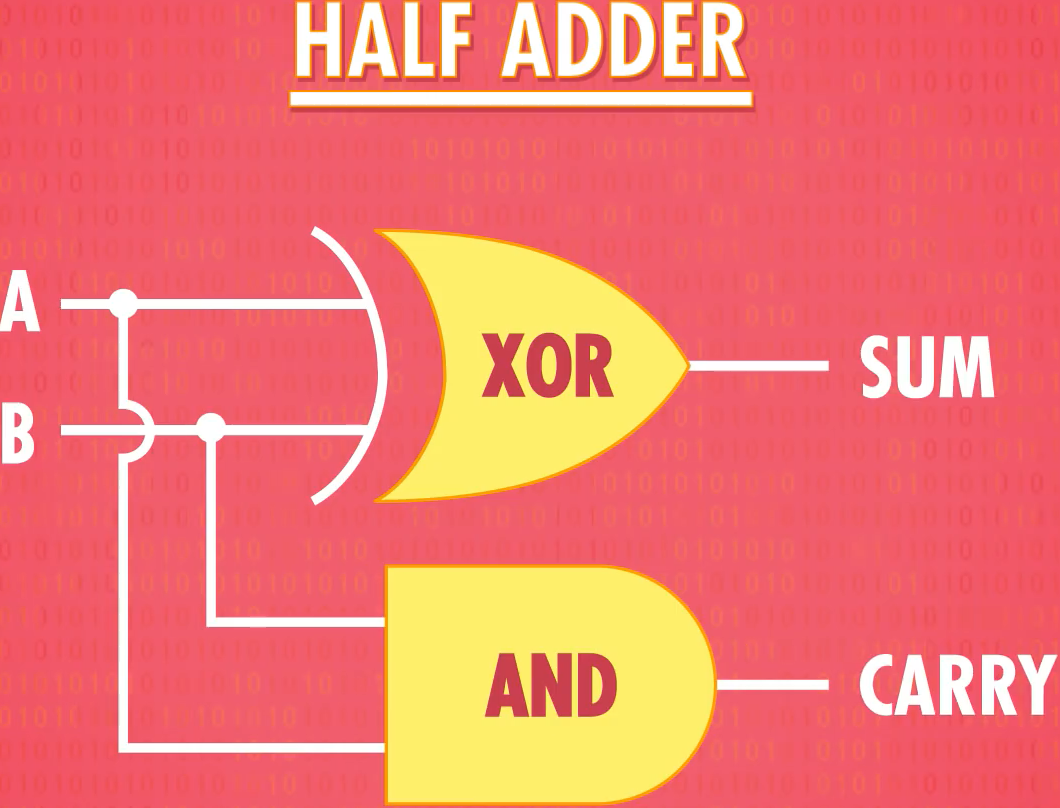
QPushButton{color:#FFFFFF;} //按钮上文字的颜色

QLabel{color:#FFFFFF;} //文字颜色

QLineEdit{color:#FFFFFF;} //文字颜色

QTreeView{color:#FFFFFF;} //QTreeView的整体背景色

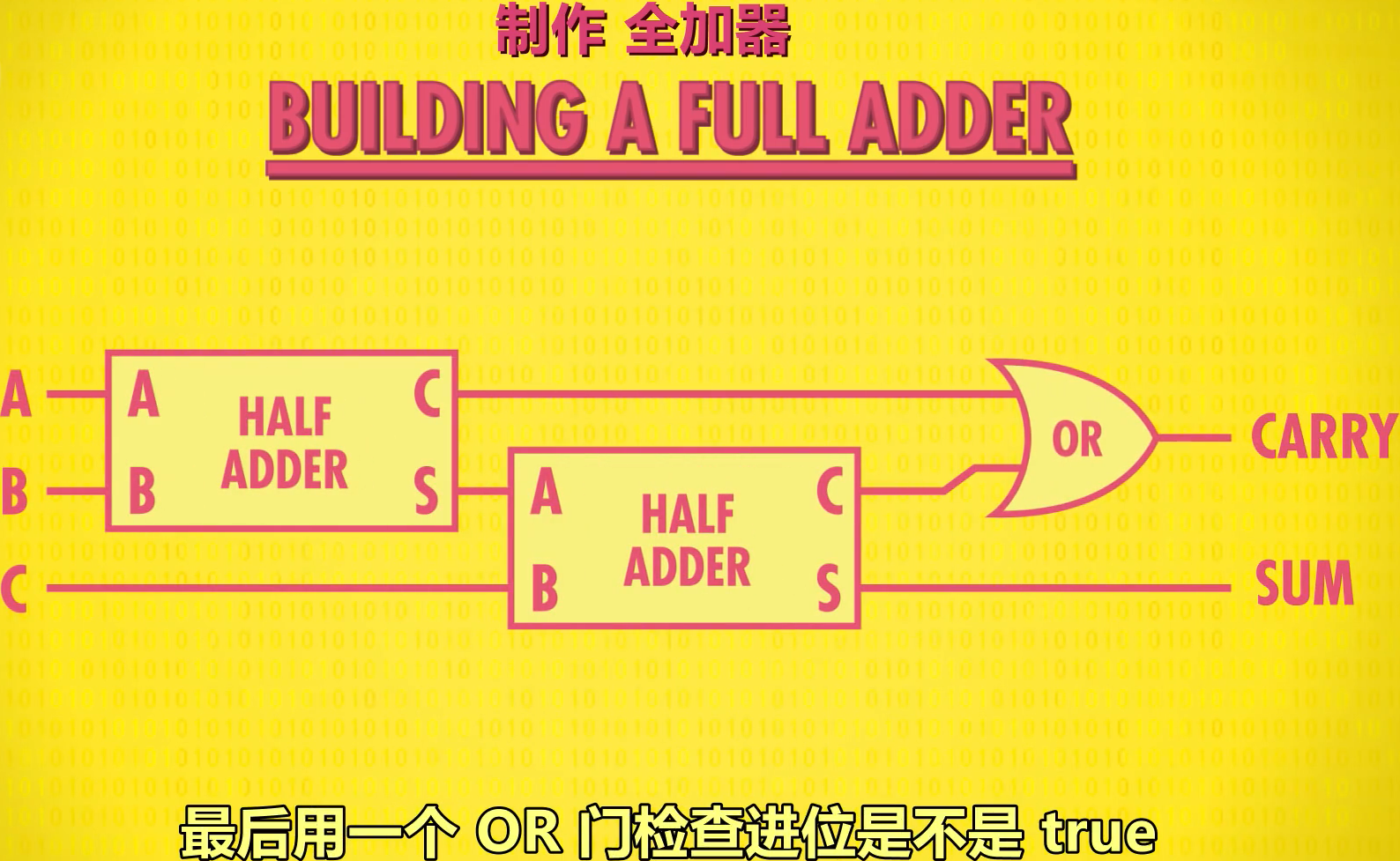
# 数电之半加器：



1+1=0、0+1=1、1+0=1、0+0=0

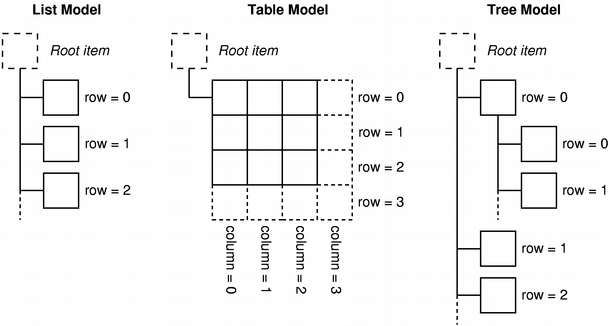
异或门（计算和sum）、与门（计算进位carry）

全加器：



# model/view 架构

只要是QAbstractItemModel的子类，都提供一种表格形式的层次结构



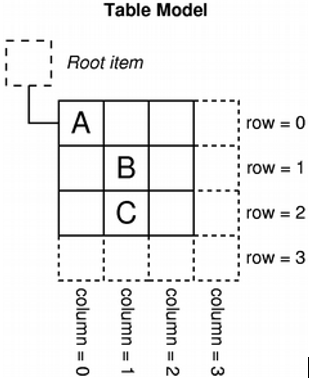
模型索引（model index）：用于访问模型的特定元素的特定部分

视图和委托使用索引来请求所需要的数据

索引保存有创建的它的那个模型的指针

QAbstractItemModel \*model = index.model();

模型索引包括临时索引和持久索引。由于模型随时可能重新组织其内部的结构，因此临时索引很可能变成不可用的。如果需要长期有效的数据片段，必须创建持久索引。持久索引保证其引用的数据及时更新。临时索引（也就是通常使用的索引）由QModelIndex类提供，持久索引则是QPersistentModelIndex类



获取模型索引：

QModelIndex QAbstractItemModel::index(int row,int column, const QModelIndex &parent=QModelIndex()) const

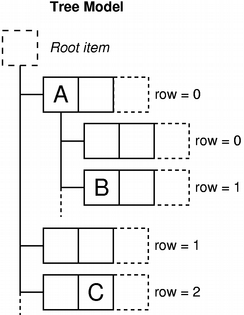
QModelIndex indexA = model->index(0, 0, QModelIndex()); //A

QModelIndex indexB = model->index(1, 1, QModelIndex()); //B

QModelIndex indexC = model->index(2, 1, QModelIndex()); .//C

表格模型中，行号、列号即可确定一个数据项，index的第三个参数采用默认值即可

对于树型结构，**每一个**节点都有可能是另外一个表格，因此每一个数据项需要指明其父节点，即index()函数的 parent 参数：



QModelIndex indexA = model->index(0, 0, QModelIndex());

QModelIndex indexC = model->index(2, 1, QModelIndex());

QModelIndex indexB = model->index(1, 0, indexA);

一个索引（QModelIndex）对应一个数据项，数据项包含一系列不同的数据角色，这些角色定义在Qt::ItemDataRole枚举中

写：void QStandardItem::setData ( const QVariant & value, int role = Qt::UserRole + 1 ) [virtual]

读：

* QVariant QStandardItem::data ( int role = Qt::UserRole + 1 ) const [virtual]
* QVariant value = model->data(index, role);
* QVariant QModelIndex::data ( int role = Qt::DisplayRole ) const

而QStandardItem\* item = new QStandardItem(“hello”)

等价于

QStandardItem\* item = new QStandardItem;

item->setData(“hello”, Qt::DisplayRole);

QList<QStandardItem\*> item\_list;

QStandardItem\* name\_item = new QStandardItem(programme.name);

name\_item->setData(programme.programme\_id, Qt::UserRole + 1);

name\_item->setData((int)programme.author, Qt::UserRole + 2);

item\_list.append(name\_item);

m\_treeview\_model->appendRow(item\_list); //修改model、view随之改变

后文要访问一个数据项的不同数据角色：

m\_treeview\_model->index(i, 0).data(Qt::UserRole + 1).toString()

m\_treeview\_model->index(i, 0).data(Qt::UserRole + 2).toInt()

获取一个QModelIndex的父索引，子索引，同级索引

QModelIndex parent () const

QModelIndex child ( int row, int column ) const

QModelIndex QModelIndex::sibling(int row, int column) const;//遍历索引级别相同的项

QModelIndex siblingAtColumn(int col)//同一行col列的数据项的QModelIndex

如：

QModelIndex curIndex = ui->treeView->indexAt(pos); //当前点击的元素的index

QModelIndex index = curIndex.sibling(curIndex.row(), 0); //该行的第1列元素的index

QString url = ui->treeView->currentIndex().siblingAtColumn(3).data().toString(); //该行第三列元素

QMimeData类是Qt内部的一个容器，它记录着大量拥有Mime Type（元类型）的数据。可以把它当成是一个map，每一个元素都是”Mime Type, Mime Data”这样的键值对，但是使用的时候不要像map一样使用，只是用于理解

QMimeData \* data = new QMimeData;

data->setData("drag", "鼠标拖拽");

当Qt检测到有Mime Data被释放到以该模型为model的组件上时，相应model的这个函数被调用。设计者需要重新实现这个函数来处理被释放的数据，通常是将数据存储到（插入或追加到）model的数据结构中

bool QAbstractItemModel::dropMimeData(const QMimeData \*data, Qt::DropAction action, int row, int column, const QModelIndex &parent)

QMimeData \*QAbstractItemModel::mimeData(const QModelIndexList &indexes) const

从indexes取数据

QAction \* QDockWidget::toggleViewAction () const,这个函数直接返回了一个与QDockWindow对象一对一关联的QAction，不用我们另外再建立QAction对象与其建立connect连接。

QStandardItemModel会获取Item的所有权，因此在销毁时同时释放各个Item的内存， 然而View并没有获取QStandardItemModel的所有权(因为同一个Model可以被多个View使用)，因此需要程序员手工释放QStandardItemModel对象内存；参考资料[8]则指出，当调用QStandardItemModel::clear()之后，所有item的内存都会被释放

# QModelIndex与QStandardItem

QModelIndex 转换成QStandardItem：

QStandardItem \* QStandardItemModel::​itemFromIndex(const QModelIndex & index) const

QStandardItem 转换成 QModelIndex：

QModelIndex QStandardItem::​index() const

# QT解析Json串：

QJsonParseError error;

QJsonDocument document = QJsonDocument::fromJson(jsonstr.toUtf8(), &error);

if (error.error != QJsonParseError::NoError)

{

return QString();

}

if (document.isNull() || document.isEmpty())

{

return QString();

}

* json串中是字符串数组时：

QVariantList varlist= document.toVariant().toList();

* json串中是数组，每个数组元素又是键值对：

QJsonArray arr = document.array();

for (int i = 0; i < arr.count();i++)

{

QJsonValue value = arr.at(i);

QJsonObject obj = value.toObject();

int serial = obj["serial"].toInt();

QString bundle\_id = obj["bundleId"].toString();

}

* json串中是键值对

if (json.isObject())

{

QJsonObject obj = json.object();

int serial = obj["serial"].toInt();

QString bundle\_id = obj["bundleId"].toString();

}

# QT拼Json串：

* json串中是键值对

QJsonObject search\_ret;

search\_ret.insert("id", "mousedown");

search\_ret.insert("scope", scope);

QJsonDocument doc;

doc.setObject(search\_ret);

QByteArray byte = doc.toJson(QJsonDocument::Compact);

QString ret(byte);

HTML：

<script> 标签用于定义客户端脚本，比如 JavaScript。

script元素既可以包含脚本语句，也可以通过src属性指向外部脚本文件。

必需的type属性规定脚本的 MIME 类型

包含脚本语句：

<script type="text/javascript">

document.write("Hello World!")

</script>

src属性指向外部.js文件：

<script type="text/javascript" src="./qwebchannel.js"></script>

CSS的margin属性：

margin: style /\*单值语法 所有边缘 \*/ 举例： margin: 1em;

margin: vertical horizontal /\*二值语法 纵向 横向 \*/ 举例： margin: 5% auto;

margin: top horizontal bottom /\*三值语法 上 横向 下\*/ 举例： margin: 1em auto 2em;

margin: top right bottom left /\*四值语法 上 右 下 左\*/ 举例： margin: 2px 1em 0 auto;

margin: inherit

Q\_PROPERTY

Q\_PROPERTY(QString text MEMBER m\_send\_text NOTIFY signal\_send\_text)

//Q\_PROPERTY前两个参数为类型、属性名

//如果没有MEMBER，则必须要有READ

//如果一个属性不需要 READ ，但又想使用属性系统，可以使用MEMBER来注册，MEMBER后面是成员变量的名字

//NOTIFY 后面跟该类中已经定义的一个信号函数，只要该属性的值发生更改，就会发出该信号。这个信号函数必须采用零个或一个参数，该参数必须与属性的类型相同

本地桌面与Web混合开发：

基于[Chromium](http://www.chromium.org/)的[Qt WebEngine](http://qt-project.org/wiki/QtWebEngine)是一个非常成熟的web浏览引擎，旧版QT使用的是WebKit

目前QT官方推荐用QtWebChannel来桥接C++应用和HTML/JavaScript应用。QtWebChannel可以很方便的实现C++和HTML/JavaScript的双向通信，同时实现C++和HTML/JS的解耦，方便开发人员的分工及系统集成

[QWebEngineView](http://doc.qt.io/qt-5/qwebengineview.html)\*view = new[QWebEngineView](http://doc.qt.io/qt-5/qwebengineview.html)(parent);

view->load([QUrl](http://doc.qt.io/qt-5/qurl.html)("http://qt-project.org/"));

view->show();

ui.webview->page()->setWebChannel(channel);

QWebEnginePage \*QWebEngineView::page() const ：返回指针指向底层web page

void QWebEnginePage::setWebChannel([QWebChannel](https://doc.qt.io/qt-5/qwebchannel.html) \*channel)：sets the web channel instance to be used by this page to *channel* and installs it in the main JavaScript world.

With this method the web channel can be accessed by web page content. If the content is not under your control and might be hostile, this could be a security issue and you should consider installing it in a private JavaScript world.

添加图标时，必须/最好先添加到.qrc文件里

QFileSystemModel::rmdir() 只能删除空目录

QFileSystemModel::remove() 既能删除文件又能删除目录

当QNetworkReply发送SIGNAL(finished())信号，槽函数中不能再调用

QNetworkReply::abort(); //关闭网络连接

QNetworkReply::deleteLater(); //销毁对象

否则程序崩溃

QStandardItem\* url\_item = new QStandardItem(“hello”); //显示的文本

url\_item->setData(“world”);

QString str = url\_item->data().toString; //返回的是setDate设置的字符串，”world”，而不是传入构造函数的文本

ui->combox\_sample\_rate->addItem("48 KHz", QVariant(48000));

ui->combox\_fmt->addItem("Signed 16", qVariantFromValue(std::pair<QAudioFormat::SampleType, int>(QAudioFormat::SignedInt, 16)));

ui->combox\_sample\_rate->itemData(ui->combox\_sample\_rate->currentIndex()).toInt()

const QVariant var = ui->combox\_fmt->itemData(ui->combox\_fmt->currentIndex());

std::pair<QAudioFormat::SampleType, int> type\_bits = var.value<std::pair<QAudioFormat::SampleType, int>>();

QMenu：

将QAction连接到不带参数的函数：

menu.addAction(QStringLiteral("刷新"), this, SLOT(on\_refreshTreeView()));

将QAction连接到带参数的函数：

QAction\* act = new QAction(QString::fromLocal8Bit("下载"));

connect(act, &QAction::triggered,[this,&url]() {on\_downLoadTreeView(url); });

menu.addAction(act);

**void** slot\_rightclicked(**const** QPoint pos)

{

    QMenu menu;

    QAction\* act = **new** QAction(QStringLiteral("添加到节目单"));

    act->setDisabled(**true**);

    connect(act, &QAction::triggered, [**this**]() {QMessageBox::information(**this**, "提示", "不支持"); });

    menu.addAction(act);

    menu.exec(QPoint(QCursor::pos().x() + 1, QCursor::pos().y() + 1));  //显示菜单

}

**void** slot\_rightclicked(**const** QPoint pos)

{

QMenu\* m\_menu = **new** QMenu(**this**);

    QMenu \*menu4 = m\_menu->addMenu(QStringLiteral("切换到4分屏"));

    menu4->addAction(QStringLiteral("通道1~4"), **this**, [**this**](){});

menu4->addAction(QStringLiteral("通道5~8"), **this**, [**this**](){});

    QMenu \*menu6 = m\_menu->addMenu(QStringLiteral("切换到6分屏"));

menu6->addAction(QStringLiteral("通道1~6"), **this**, [**this**]() {});

    menu.exec(QPoint(QCursor::pos().x() + 1, QCursor::pos().y() + 1));  //显示菜单

}

# 焦点事件

在应用程序中，都会有一个当前窗口，即当前获得焦点事件的窗口，这个窗口可以接受键盘的输入

ui->treeView->setFocusPolicy(Qt::NoFocus); //设置如何接受焦点事件，不接受焦点事件

QStackedWidget

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

final关键字：

C++11的关键字final有两个用途。**第一，它阻止了从类继承**；**第二，阻止一个虚函数被子类覆盖**

class TaskManager {/\*..\*/} final;

class PrioritizedTaskManager: public TaskManager {

}; //compilation error: base class TaskManager is final

如果一个派生类试图覆盖final函数，编译器就会报错：

struct A{

virtual void func() const;

};

struct B: A{

void func() const override final; //OK

};

struct C: B{

void func()const; //error, B::func is final

};

QSlider

void setTracking(bool enable)如果启用跟踪(默认)，滑块在拖动滑块时发出valueChanged()信号。如果禁用跟踪，则滑块仅在用户释放滑块时才发出valueChanged()信号。

设置鼠标点击时移动的步进值:void setPageStep(int)之所以有鼠标点击时的步进值设置，是因为在默认情况下，点击QSlider，QSlider的滑动条并不会移动到所点击的位置，而是向前或向后移动PageStep大小。

QSlider的setValue()方法会激发信号valueChanged()

QSlider::sliderPressed仅在滑块被按下时触发，并非鼠标点击进度条任意位置都会触发

QSlider::sliderReleased滑块被释放时触发，非鼠标释放触发

# 启动进程：

QString program = "C:/Windows/System32/cmd.exe";

QStringList arguments;

arguments << "/c" << "ffplay" << "C:/Users/myth/Videos/doctor.mp4"; //第一个字符串"/c"不能省略

QProcess \*cmd\_process = new QProcess(&w);

cmd\_process->start(program, arguments);

QString program = "D:/Git/git-bash.exe";

QStringList arguments;

arguments <<"ffplay" << "C:/Users/myth/Videos/doctor.mp4"; //如果加上这句则git-bash的窗口一闪而过，且不播放视频

QProcess \*cmd\_process = new QProcess;

cmd\_process->startDetached(program, arguments);

cmd\_process->waitForFinished(-1);

# Qt事件循环：

QPushButton的mousePressEvent()函数中发射clicked()信号

继承QPushButton自定义按钮，如果

void CustomButton::mousePressEvent(QMouseEvent \*event)

{

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

{

qDebug() << "left";

}

}

则点击按钮后，不会在发送clicked()信号

“事件循环”（Qt常用）, 也可以叫做”消息循环”（windows编程常用）

QEventLoop eventLoop;

connect(netWorker, &NetWorker::finished,&eventLoop, &QEventLoop::quit);

QNetworkReply \*reply = netWorker->get(url);

eventLoop.exec();

其中exec是启动事件循环，调用exec以后，调用exec的函数就会被“阻塞”，直到EventLoop里面的while循环结束。

Qt作为一个跨平台的UI框架，其事件循环实现原理, 就是把不同平台的事件循环进行了封装

windows下的事件循环

while (GetMessage(&msg, NULL, 0, 0)) //从消息队列读消息，或根据线程的某些标识返回MESSAGE\_QUIT、WM\_PAINT等消息，没有消息则一直wait（放弃cpu时间片）

{

TranslateMessage(&msg); //翻译消息

DispatchMessage(&msg); //派发消息

}

这样的while()循环体现了“事件驱动”

QApplication::exec()对主事件循环调用exec()，因此调用QApplication::exec()前，程序不能响应任何用户交互

但模态组件如QMessageBox可以在QApplication::exec()前响应用户事件，因为其调用exec开启了一个局部QEventLoop

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

{

QApplication a(argc, argv);

QMessageBox box;

box.exec(); //可以正常与用户交互，或者QMessageBox::information()也可以

return 0;

}

Application还提供了sendEvent和poseEvent两个函数

sendEvent发出的事件会立即被处理，也就是“同步”执行。

postEvent发送的事件会被加入事件队列，在下一轮事件循环时才处理，也就是“异步”执行。

bool QCoreApplication::sendEvent ( [QObject](http://epic-alfa.kavli.tudelft.nl/share/doc/qt4/html/qobject.html) \* *receiver*, [QEvent](http://epic-alfa.kavli.tudelft.nl/share/doc/qt4/html/qevent.html) \* *event* )   [static]

Sends event *event* directly to receiver *receiver*, using the [notify](http://epic-alfa.kavli.tudelft.nl/share/doc/qt4/html/qcoreapplication.html#notify)() function. Returns the value that was returned from the event handler.

The event is *not* deleted when the event has been sent. The normal approach is to create the event on the stack, for example:

QMouseEvent event(QEvent::MouseButtonPress, pos, 0, 0, 0);

QApplication::sendEvent(mainWindow, &event);

QCoreApplication::sendEvent()相当于win API中的SendMessage()发送事件到同一线程

QCoreApplication::sendEvent()直接调用QCoreApplication::notify

SendMessage()到同线程时，直接调用窗口处理函数，一般名为WndProc

void QCoreApplication::postEvent ( [QObject](http://epic-alfa.kavli.tudelft.nl/share/doc/qt4/html/qobject.html) \* *receiver*, [QEvent](http://epic-alfa.kavli.tudelft.nl/share/doc/qt4/html/qevent.html) \* *event* )   [static]

将事件送入事件队列后立即返回

必须在堆上分配，因为一旦从事件队列中取出该事件，则该事件将被删除，

The event must be allocated on the heap since the post event queue will take ownership of the event and delete it once it has been posted. It is *not safe* to modify or delete the event after it has been posted.

When control returns to the main event loop, all events that are stored in the queue will be sent using the [notify](http://epic-alfa.kavli.tudelft.nl/share/doc/qt4/html/qcoreapplication.html#notify)() function.

void QCoreApplication::postEvent 相当于win API中的PostMessage()

qDebug() << QStringLiteral("播放") << QDateTime::currentDateTime();

Sleep(5000); //win api，等价于Qt的QThread::msleep(5000)，等价于std的std::this\_thread::sleep\_for(std::chrono::milliseconds(5000));

qDebug() << QStringLiteral("播放失败") << QDateTime::currentDateTime();

会导致界面卡死，程序会阻塞在Sleep(5000)语句处，且主事件循环被卡住

**子层事件循环具有父层事件循环的所有功能，所以当在主线程中启动各种exec()（比如QEventLoop::exec()）时，虽然会打断main函数中的QApplication::exec()，但是Gui界面还是可以正常响应，不会出现卡住的现象**

QEventLoop loop;

qDebug() << QStringLiteral("播放") << QDateTime::currentDateTime();

QTimer::singleShot(5000, [&loop]()

{

qDebug() << QStringLiteral("播放失败") << QDateTime::currentDateTime();

loop.quit();

});

loop.exec();

qDebug() << QStringLiteral("下一语句") << QDateTime::currentDateTime();

不会导致界面卡死，虽然程序会阻塞在loop.exec()语句处，但主事件循环不受影响

运行结果：

"播放" QDateTime(2019-11-26 10:57:52.972 中国标准时间 Qt::LocalTime)

"播放失败" QDateTime(2019-11-26 10:57:57.969 中国标准时间 Qt::LocalTime)

"下一语句" QDateTime(2019-11-26 10:57:57.970 中国标准时间 Qt::LocalTime)

对耗时操作封装，避免主界面卡死

**int** res;

QEventLoop\* loop = **new** QEventLoop;

connect(**this**, &Player::startupPlayBackFinished, **this**, [**this**, &loop]()

{

stopLoadingAnin();

**if** (!loop->isRunning())

    {

**delete** loop;

        loop = nullptr;

    }

**else**

    {

        loop->quit();

    }

    disconnect(**this**,SIGNAL(startupPlayBackFinished()),0,0);

});

std::**thread** t([playctrl, xml, &res, **this**]()

{

    res = playctrl->play(xml);

    emit startupPlayBackFinished();

});

**if** (loop)

{

    loop->exec();

}

t.join();

**if** (loop)

{

**delete** loop;

}

**return** res;

Qt 程序时事件驱动的，每个程序，我们需要调用 QApplication::exec() 来启用事件循环。

int QCoreApplication::exec()

{

...

QEventLoop eventLoop;

int returnCode = eventLoop.exec();

...

return returnCode;

}

而

int QEventLoop::exec(ProcessEventsFlags flags)

{

Q\_D(QEventLoop);

...

while (!d->exit)

processEvents(flags | WaitForMoreEvents | EventLoopExec);

...

return d->returnCode;

}

而

bool QEventLoop::processEvents(ProcessEventsFlags flags)

{

Q\_D(QEventLoop);

if (!d->threadData->eventDispatcher)

return false;

if (flags & DeferredDeletion)

QCoreApplication::sendPostedEvents(0, QEvent::DeferredDelete);

return d->threadData->eventDispatcher->processEvents(flags);

}

进一步：这将调用平台相关的函数，比如在windows下

bool QGuiEventDispatcherWin32::processEvents(QEventLoop::ProcessEventsFlags flags)

{

if (!QEventDispatcherWin32::processEvents(flags))

return false;

if (configRequests) // any pending configs?

qWinProcessConfigRequests();

return true;

}

事件循环和线程没有必然的联系，事件循环可以用在QThread中，而且从Qt4.4开始，QThread的run函数默认就调用了自己的事件循环。

对QDialog来说，当它自己的QEventLoop启用时，主程序的 QEventLoop 当然是处于暂停状态了。说到底，就是两个死循环，一个在内，一个在外，只有里面的退出后，外边的循环才会执行。不过由于两个循环执行的命令是基本一样的，都是调用并处理程序收到的各种事件，因此界面没有卡死

所有带有Q\_OBJECT的类都应该放在头文件中，因为 moc 只会处理头文件中的Q\_OBJECT

对以下程序，点击一次按钮后，快速再点击一次，输出为

in QEventLoop

in QEventLoop

out QEventLoop

out QEventLoop

connect(ui.testQELBtn, &QPushButton::clicked, [this](bool)

{

QEventLoop loop;

qDebug() << "in QEventLoop";

QTimer::singleShot(5000,&loop,&QEventLoop::quit);

loop.exec();

qDebug() << "out QEventLoop";

});

# 可重入与线程安全：

linux系统下的可重入与线程线程安全

异步可重入是Unix/Linux这种支持signal的系统特有的，叫 async-signal-safe。

很多线程安全的函数都是不可重入的，例如malloc、pthread\_mutex\_lock

进程(线程)正在执行函数fun()，在函数fun()还未执行完的时候，突然进程接收到一个信号sig, 此时，需要暂停执行fun()，要转而执行sig信号的处理函数sig\_handler()，而sig\_handler()也恰好调用了函数fun()。Linux中信号的处理是以软终端的形式进行的，那么，当sig\_handler()执行完返回之后，CPU会继续从fun()被打断的地方往下执行。如果func()函数在不被上述中断影响下的执行结果与出现上述中断情况下的最终执行结果一致，则认为是异步可重入

在Unix/Linux系统中，signal是以软中断的方式分发的，signal handler可能在任何时候打断一个进程的任意一个线程而执行（如果该线程没有屏蔽该signal的话）

Windows的signal是通过特定的线程分发的，所以它不会打断应用的线程。所有线程安全的函数在Windows的signal handler中都可以使用

Qt之可重入与线程安全

对于一个类，如果不同的实例可以被不同线程同时使用而不受影响，就说这个类是可重入的；如果这个类的所有成员函数都可以被不同线程同时调用而不受影响，即使这些调用针对同一个对象，那么我们就说这个类是线程安全的。由此可以看出，线程安全的语义要强于可重入

# QThread：

class FileDecryptor : public QObject

{

Q\_OBJECT

public slots:

void decrypt()

{

//dosomething();

emit decryptionFinished();

}

signals:

void decryptionFinished();

};

QThread m\_decryptionthread;

FileDecryptor m\_decryptor;

m\_decryptor.moveToThread(&m\_decryptionthread);

connect(&m\_decryptor, SIGNAL(decryptionFinished()), this, SLOT(decryptionFinished()));

connect(this, SIGNAL(signalStartDecryption()), &m\_decryptor, SLOT(decrypt()));

m\_decryptionthread.start();

emit signalStartDecryption();

从 Qt 4.4 开始，QThread就已经不是抽象类了。QThread::run()不再是纯虚函数，而是有了一个默认的实现。这个默认实现其实是简单地调用了QThread::exec()函数，开始了一个事件循环

QThread::run()

{

QThread::exec();

}

**在主线程中创建并启动的局部事件循环具有主事件循环的所有功能**

**在线程中创建并启动的事件循环（调用QThread::exec()启动）只负责该线程中对象的事件分发**

线程的事件循环用于为线程中的所有QObjects对象分发事件；默认情况下，这些对象包括线程中创建的所有对象，或者是在别处创建完成后被移动到该线程的对象

一个QObject所属的线程（thread affinity，依附）是指它所在的那个线程，也即是创建它的那个线程（没有被移动的话）。调用QObject::thread()可以查询一个QObject所属线程

我们可以使用线程安全的QCoreApplication::postEvent()函数向一个对象发送事件。它将把事件加入到对象所在的线程的事件队列中，因此，如果这个线程没有运行事件循环，这个事件也不会被派发。

QObject类可重入，但不是线程安全

QWidget及其所有的子类，都不可重入的，因此也不是线程安全

多线程同时访问某个QWidget时，需要对数据加锁

从另外的线程访问一个对象时，该对象可能正在处理所属线程的事件循环派发的事件！因此，不能在另外的线程直接delete一个QObject对象，相反，需要调用QObject::deleteLater()函数，这个函数会给对象所在线程发送一个删除的事件

QObject所属线程可以改变，方法是调用QObject::moveToThread()函数。该函数会改变一个对象及其所有子对象的线程依附性。由于QObject不是线程安全的，所以只能在该对象所在线程上调用这个函数。此外，Qt 要求QObject的所有子对象都必须和其父对象在同一线程。这意味着：

1. 不能对有父对象（parent 属性）的对象使用QObject::moveToThread()函数
2. 不能在QThread中以这个QThread本身作为父对象创建对象，例如：

class Thread : public QThread {

void run() {

//这个函数里创建的所有都属于新线程

QObject \*obj = new QObject(this); // 错误！

}

};因为QThread对象所依附的线程是创建它的那个线程，而不是它所代表的线程

Qt 还要求，在代表一个线程的QThread对象销毁之前，所有在这个线程中的对象都必须先delete。要达到这一点并不困难：我们只需在QThread::run()的栈上创建对象即可。

当我们将信号与槽连接起来时，QObject::connect()的最后一个参数将指定连接类型：

* Qt::DirectConnection：直接连接意味着槽函数将在信号发出的线程直接调用
* Qt::QueuedConnection：队列连接意味着向接受者所在线程发送一个事件，该线程的事件循环将获得这个事件，然后之后的某个时刻调用槽函数
* Qt::BlockingQueuedConnection：阻塞的队列连接就像队列连接，但是发送者线程将会阻塞，直到接受者所在线程的事件循环获得这个事件，槽函数被调用之后，函数才会返回
* Qt::AutoConnection：自动连接（默认）意味着如果接受者所在线程就是当前线程，则使用直接连接；否则将使用队列连接

上面每种情况中，发送者所在线程都是无关紧要的！在自动连接情况下，Qt 需要查看信号发出的线程是不是与接受者所在线程一致，来决定连接类型。注意，Qt 检查的是信号发出的线程与接受者所在线程，而不是信号发出的对象所在的线程与接受者所在线程

class Thread : public QThread

{

Q\_OBJECT

signals:

void aSignal();

protected:

void run() {

emit aSignal();

}

};

/\* ... \*/

Thread thread;

Object obj;

QObject::connect(&thread, SIGNAL(aSignal()), &obj, SLOT(aSlot()));

thread.start();

这里将会使用队列连接，发送信号的对象在主线程，但发送信号这一动作是在Thread所代表的线程

class Thread : public QThread

{

Q\_OBJECT

slots:

void aSlot() {

/\* ... \*/

}

protected:

void run() {

/\* ... \*/

}

};

/\* ... \*/

Thread thread;

Object obj;

QObject::connect(&obj, SIGNAL(aSignal()), &thread, SLOT(aSlot()));

thread.start();

obj.emitSignal(); //会执行emit aSignal();

这里的obj发出aSignal()信号时，使用哪种连接方式？答案是：直接连接。因为Thread对象所在线程发出了信号，也就是信号发出的线程与接受者是同一个。在aSlot()槽函数中，我们可以直接访问Thread的某些成员变量，但是注意，在我们访问这些成员变量时，Thread::run()函数可能也在访问！这意味着二者并发进行：这是一个完美的导致崩溃的隐藏bug。

# QManinWindow、QWidget、QFrame等setLayout

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

{

QApplication a(argc, argv);

QFrame frame;

frame.resize(600, 400);

frame.setObjectName("myframe");

frame.resize(600, 400);

frame.setStyleSheet("QFrame#myframe{border-image:url(cmdctrl.png)}"); //不能是jpg格式图片

QVBoxLayout\* lay = new QVBoxLayout;

frame.setLayout(lay);

lay->addStretch();

QHBoxLayout\* sublayup = new QHBoxLayout;

lay->addLayout(sublayup);

sublayup->addStretch();

sublayup->addWidget(new QPushButton("go"));

sublayup->addStretch();

lay->addStretch();

QHBoxLayout\* sublaymid = new QHBoxLayout;

lay->addLayout(sublaymid);

sublaymid->addStretch();

sublaymid->addWidget(new QPushButton("go"));

sublaymid->addStretch();

lay->addStretch();

QHBoxLayout\* sublaybot = new QHBoxLayout;

lay->addLayout(sublaybot);

sublaybot->addStretch();

sublaybot->addWidget(new QPushButton("go"));

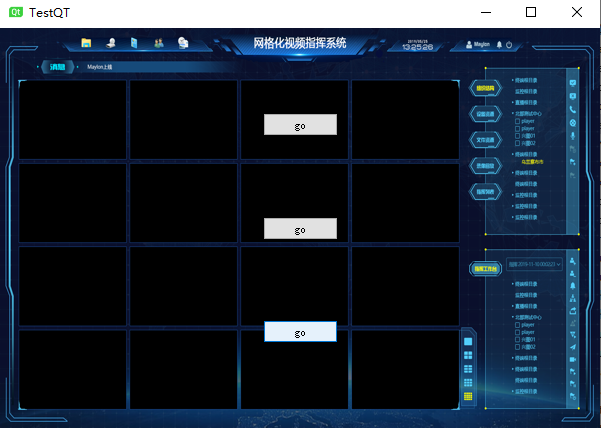
sublaybot->addStretch();

lay->addStretch();

frame.show();

return QApplication::exec();

}



背景图片随着窗体大小缩放

QMainWindow要实现上述效果，将代码修改如下

QApplication a(argc, argv);

QMainWindow mainwind;

mainwind.resize(600, 400);

mainwind.setObjectName("myframe");

mainwind.resize(600, 400);

mainwind.setStyleSheet("QMainWindow#myframe{border-image:url(cmdctrl.png)}");

QVBoxLayout\* lay = new QVBoxLayout(&mainwind);

QWidget\* center=new QWidget;

center->setLayout(lay);

mainwind.setCentralWidget(center);

对QFrame、QWIdget等可以直接setLayout，再向参数所代表的布局添加组件、子布局

但对QMainWindow，只能

QWidget\* center=new QWidget;

center->setLayout(lay);

mainwind.setCentralWidget(center);

//再向lay中添加组件、子布局

# QFileInfo:: fileName()

QFileInfo info("local://C:/Users/myth/Videos/a.txt");

QString name=info.fileName(); //"a.txt，无论文件是否存在"

QFileInfo info("https://www.bilibili.com/video/movie.mp4");

QString name = info.fileName(); //"movie.mp4，无论文件是否存在"

QFileInfo info("/a .txt");

qDebug() << info.absoluteFilePath(); //"E:/a.txt"

# QDateTime与QDateTimeEdit：

QDateTime now = QDateTime();

QDateTime after = now.addMSecs(30 \* 60 \* 1000);

QString strnow = now.toString("yyyy/MM/dd HH:mm:ss"); //空

bool bnow = now.isValid(); //false

bool bafter = after.isValid(); //false

ui.dateTimeEdit->setDateTime(QDateTime::currentDateTime()); //2020/1/7 10:12

ui.dateTimeEdit->setDateTime(QDateTime());

QString str = ui.dateTimeEdit->dateTime().toString(); //仍为2020/1/7 10:12

ui.dateTimeEdit->setDateTime(QDateTime());

QDateTime dt = ui.dateTimeEdit->dateTime();

qDebug() << dt.isValid(); //true

qDebug() << dt.toString("yyyy/MM/dd HH:mm:ss"); //2000/01/01 00:00:00"

# 窗口资源的销毁、对象析构

~QtResManage()

mainwind emit destroyed signal

~CustomizedPushButton()

void QObject::destroyed ( QObject \* obj = 0 ) [signal]

This signal is emitted immediately before the object obj is destroyed, and can not be blocked.

All the objects's children are destroyed immediately after this signal is emitted.

See also deleteLater() and QPointer.

void QWidget::destroy ( bool destroyWindow = true, bool destroySubWindows = true ) [protected]

Frees up window system resources. Destroys the widget window if destroyWindow is true.

destroy() calls itself recursively for all the child widgets, passing destroySubWindows for the destroyWindow parameter. To have more control over destruction of subwidgets, destroy subwidgets selectively first.

This function is usually called from the QWidget destructor.

# QAudioDeviceInfo::availableDevices：

const QAudioDeviceInfo default\_dev = QAudioDeviceInfo::defaultInputDevice();

//警告：QAudioDeviceInfo::availableDevices(QAudio::AudioInput)返回的列表中找不到设备名为default\_dev的QAudioDeviceInfo

QAudioDeviceInfo::availableDevices返回的设备中有重复的，应该去重复如下：

QList<QAudioDeviceInfo> dups = QAudioDeviceInfo::availableDevices(QAudio::AudioInput);

struct myLessFunc

{

bool operator() (const QAudioDeviceInfo& lhs, const QAudioDeviceInfo& rhs)

{

return lhs.deviceName() < rhs.deviceName(); //从大到小排序

}

};

std::set<QAudioDeviceInfo, myLessFunc> undups;

for (const auto& i : dups)

{

undups.insert(i);

}

QList<QAudioDeviceInfo> devinfos;

for (auto iter = undups.begin(); iter != undups.end(); iter++)

{

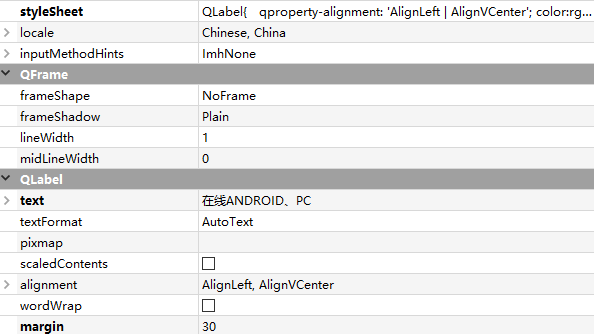
devinfos.push\_back(\*iter);

}

# 样式表:



QLabel{ qproperty-alignment: 'AlignRight | AlignVCenter'; color:rgb(55,161,49); background-color:transparent;border:none;background-image:url(:Resources/others/u1948.png); background-position: left;background-repeat: no-repeat}



继承QWidget 的组件，设置样式表，则需要重写 paintEvent()，因为而样式表要通过 paint 被绘制到窗口中，而QWidget的paintEvent()是空的

# 检测信号槽连接的函数：

virtual void connectNotify(const QMetaMethod &signal)该对象的信号被连接时自动调用的函数

virtual void disconnectNotify(const QMetaMethod &signal) 有信号断开连接时自动调用的函数

示例：

void TestQT::connectNotify(const QMetaMethod &signal)

{

    QString type\_name = signal.typeName(); //信号函数的返回类型

    QString name = signal.name(); //信号函数的函数名

    if (signal == QMetaMethod::fromSignal(&TestQT::startupPlayBackFinished))

    {

        qDebug() << QString("The startupPlayBackFinished signal has been connected");

    }

    else if (signal == QMetaMethod::fromSignal(&TestQT::signalChangeColor))

    {

        qDebug() << QString("The signalChangeColor signal has been connected");

    }

}

TestQT主界面上有个ui.play\_btn按钮，槽函数如下

    connect(ui.play\_btn, &QPushButton::clicked, this, [this](bool)

    {

        connect(this, &TestQT::startupPlayBackFinished, this, []()

        {

            qDebug() << "hello";

        });

        emit this->startupPlayBackFinished();

    });

};

则每点击一下play\_btn按钮，都会打印出"The startupPlayBackFinished signal has been connected"

且第一次点击

hello

第二次点击

hello

hello

调试利器，有多少人连我啊？ - receivers() 函数

int receivers(const char \*signal) const

// 示例

if (receivers(SIGNAL(valueChanged(QByteArray))) > 0) {

....

}

**对于一个类，如果不同的实例可以被不同线程同时使用而不受影响，就说这个类是可重入的；如果同一个对象可以被不同线程同时使用而不受影响，那么我们就说这个类是线程安全的**

**虽然QObject是可重入的，但是 GUI 类，特别是QWidget及其所有的子类，都是不是可重入的。它们只能在主线程使用。即不能多线程使用QWidget或其子类的对象**

**在你从另外的线程访问一个对象时，它可能正在处理所在线程的事件循环派发的事件！如播放器时间回调函数里不能直接ui->processSlider->setValue(value)更新进度条；**

**基于同样的原因，你也不能在另外的线程直接delete一个QObject对象，相反，你需要调用QObject::deleteLater()函数，这个函数会给对象所在线程发送一个删除的事件。**

**跨线程的信号槽也是类似的。当我们将信号与槽连接起来时，QObject::connect()的最后一个参数将指定连接类型：**

* **Qt::DirectConnection：直接连接意味着槽函数将在信号发出的线程直接调用**
* **Qt::QueuedConnection：队列连接意味着向接受者所在线程发送一个事件，该线程的事件循环将获得这个事件，然后之后的某个时刻调用槽函数**
* **Qt::BlockingQueuedConnection：阻塞的队列连接就像队列连接，但是发送者线程将会阻塞，直到接受者所在线程的事件循环获得这个事件，槽函数被调用之后，函数才会返回。接收者和发送者绝对不能在一个线程，否则程序会死锁**
* **Qt::AutoConnection：自动连接（默认）意味着如果接受者所在线程就是发出信号的线程，则使用直接连接；否则将使用队列连接**

**不能做的是：**

* **在线程还在运行时退出程序。使用QThread::wait()函数等待线程结束**
* **在QThread对象所管理的线程仍在运行时就销毁该对象。如果你需要某种“自行销毁”的操作，你可以把finished()信号同deleteLater()槽连接起来**

**播rtp-es：**

**<media type='stream' url='rtp-es://+127.0.0.1:25002@127.0.0.1' mode='lowdelay' rtp-reorder-cnt='5'><prog id='1'><audio id='0' pid='0' info-present='0' codec='g711-mulaw' channel='1' sample-rate='8000' sample-width='16' sel-channel='0'/></prog></media>**

**source为RtpEsSource**

**svn的Revert回到上次自己update时服务器的版本，而不是得到服务器最新版本，revert后再update则与服务器最新版本保持一致**

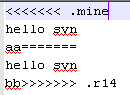
**commit时冲突，再update后产生test.txt、test.txt.mine、test.txt.r13、test.txt.r14，其中**

**test.txt.mine是用户当前状态**

**test.txt.r13是服务器13版本，用户在13版本上修改有了test.txt.mine**

**test.txt.r14是服务器14版本，也是服务器的最新版本**

**test.txt是svn将最新版14与用户的修改合并后的文件，如**



**手动解决冲突后，Resolve消除掉test.txt.mine、test1.txt.r13、test1.text.r14这三个文件，重新commit**

**QSplitter::setOpaqueResize(true)** **设定在拖拽分割条时，是否实时更新。若为true，则实时更新；否则在拖拽时显示一条虚线，释放鼠标后，更新size**

**对QMainWindow在构造函数中设置了最小size，设置最大化显示**

const QRect desktop = qApp->desktop()->availableGeometry(); //1920\*1040

const int width = desktop.width() \* 8 / 10; //1536

const int height = desktop.height() \* 18 / 20; //936

setMinimumSize(width, height);

setWindowState(windowState()| Qt::WindowMaximized); //初始时最大化

**先进入resizeEvent()函数，此时this->width()、this->height()返回的即是设置的最小size，即1536\*936**

**再进入showEvent函数，此时this->width()、this->height()返回的即是设置的最小size，即1536\*936**

**再进入resizeEvent()函数，此时this->width()、this->height()返回的是最大化size，即1920\*1017**

**如果注释掉setWindowState(windowState()| Qt::WindowMaximized)，则不会第二次进入resizeEvent()函数**

**minimumSizeHint()是一个虚函数**

**除非设置了minimumSize()或者其尺寸控制模式被设置为QSizePolicy::Ignore，否则Qt的布局管理器不会把部件的大小调整为比最小大小提示还要小**

/\* 下拉后，整个下拉窗体每项的样式 \*/

QComboBox QAbstractItemView::item {

height: 50px; /\* 项的高度（设置pComboBox->setView(new QListView());后，该项才起作用） \*/

}

**Qt设置QWidget或其派生类对象透明**

this->setWindowFlags(Qt::Window | Qt::FramelessWindowHint);

this->setAttribute(Qt::WA\_NoSystemBackground);

this->setAttribute(Qt::WA\_TranslucentBackground);

this->setAttribute(Qt::WA\_TransparentForMouseEvents);

**Qt设置QWidget或其派生类对象全屏**

this->setWindowState(this->windowState() ^ Qt::WindowFullScreen);

this->show();

Qt检测鼠标悬停事件前必须：

widget->setAttribute(Qt::WA\_Hover);

# 保存文件：

//使用QSaveFile来确保所有数据都安全写入，并且在写入失败时不会损坏现有文件。

//我们使用QFile::Text标志来确保在Windows上，“\n”被转换为“\r\n”以符合Windows约定。

bool MainWindow::saveFile(const QString &fileName)

{

QString errorMessage;

//在当前状态下，改为 Qt::WaitCursor 光标

QGuiApplication::setOverrideCursor(Qt::WaitCursor);

QSaveFile file(fileName);

if (file.open(QFile::WriteOnly | QFile::Text)) {

QTextStream out(&file);

out << textEdit->toPlainText();

if (!file.commit()) {

errorMessage = tr("Cannot write file %1:\n%2.")

//返回路径名，其中“/”分隔符转换为适合基础操作系统的分隔符。

.arg(QDir::toNativeSeparators(fileName), file.errorString());

}

}

else {

errorMessage = tr("Cannot open file %1 for writing:\n%2.")

.arg(QDir::toNativeSeparators(fileName), file.errorString());

}

//设置回之前的光标

QGuiApplication::restoreOverrideCursor();

if (!errorMessage.isEmpty()) {

QMessageBox::warning(this, tr("Application"), errorMessage);

return false;

}

setCurrentFile(fileName);

//(&message, timeout = 0)

//timeout的单位为milli-seconds。

//显示File Saved，持续时间2秒

statusBar()->showMessage(tr("File saved"), 2000);

return true;

}

setWindowModified(textEdit->document()->isModified());