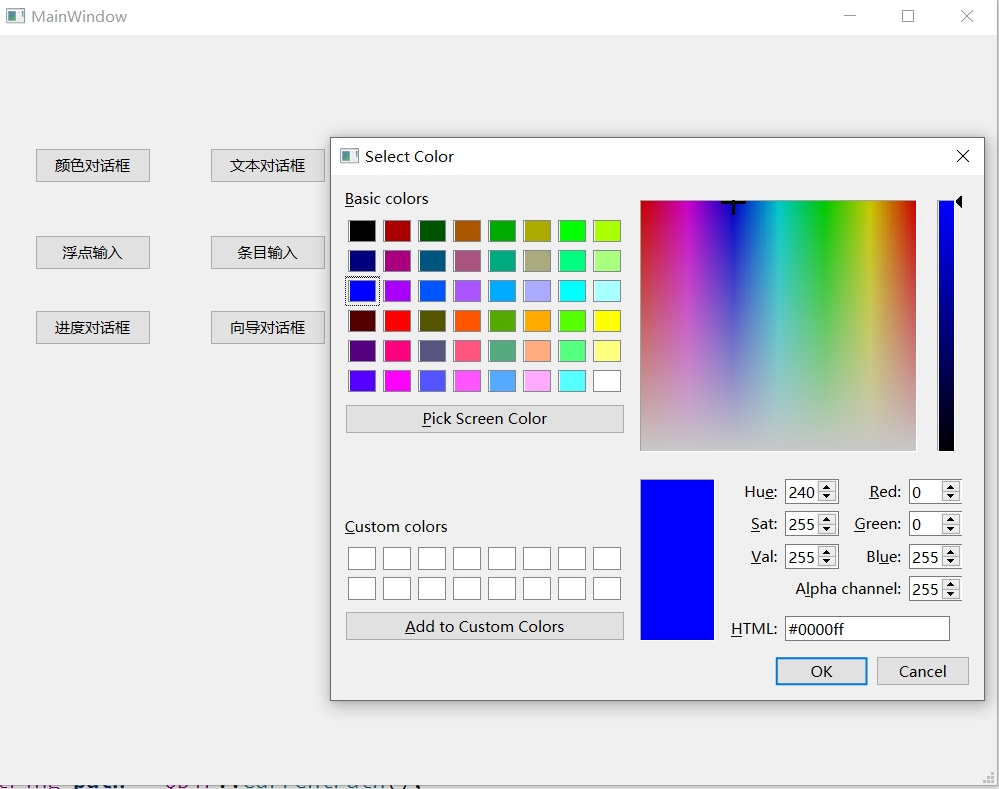
这个是QT学习的开始。

首先，我们将去了解几个QT中常用的对话框。

1~颜色对话框



这个对话框是我们在软件中常见到的选择颜色界面



类名： QColorDialog

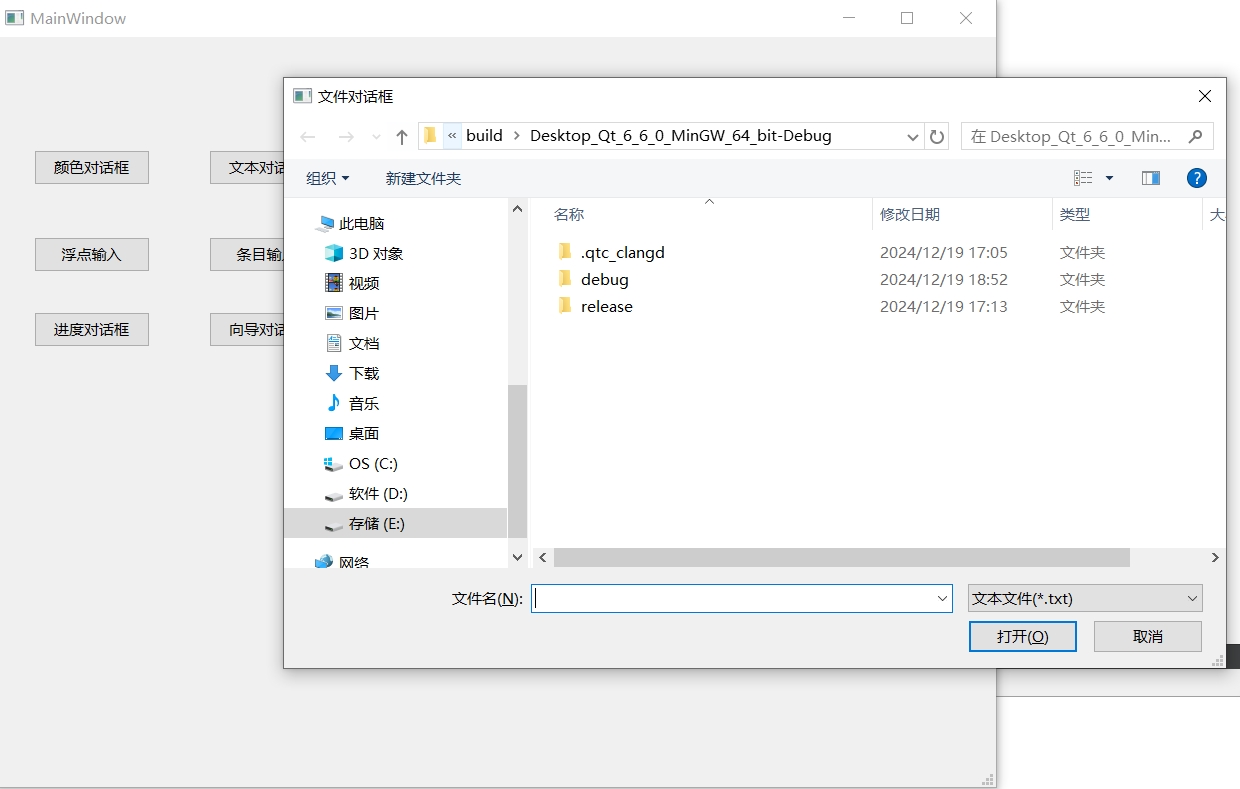
构造函数参数: 第一个参数用于指定一个默认选择的颜色，即进去过后可以看到的颜色，第二个参数指定这个对话框的父窗口，挂到对应的对象树上。

对于这个对话框，右下角会有俩个默认的[OK][Cancel]按钮。对于这俩个按钮，当我们选择完或者自定义完我们的颜色并按下按钮后，这个颜色对话框将会被关闭，在默认情况下，只有你站在最后关闭窗口是使用[OK]按钮，最后窗口的颜色属性都会被保存。否则，还是只会保存上一次设定的颜色值，没有就是默认的颜色属性。

我们可以使用QColorDialog类的currentColor方法进行获取这个颜色的属性，当然可能看不懂它的输出。

接下来看到下一个对话框，**文件对话框**

这个对话框的样式其实跟我们平时的文件系统的样式很相像





类名：QFileDialog

这个对话框比较特殊，我们不是通过创建一个类对象来创建的，而是调用一个静态方法。我们来看到这个静态方法QFileDialog::getOpenFileName()的参数列表。

这个静态方法的调用比较复杂，需要四个参数。

1~这个文本对话框的父窗口对象

2~这个对话框的标题，就是整个窗口左上角的那段文字

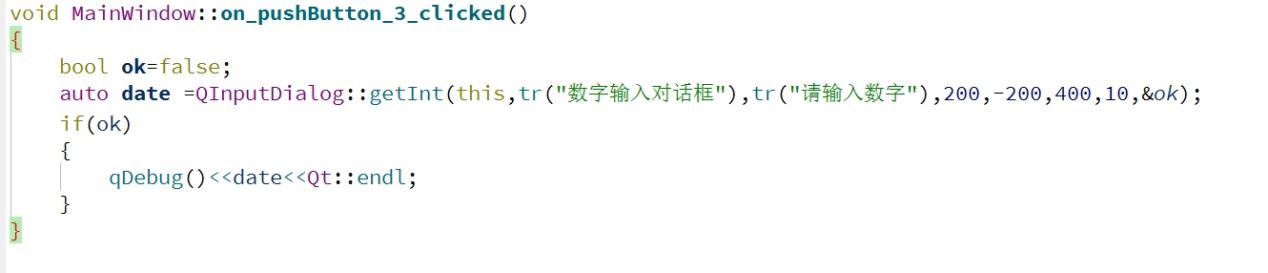
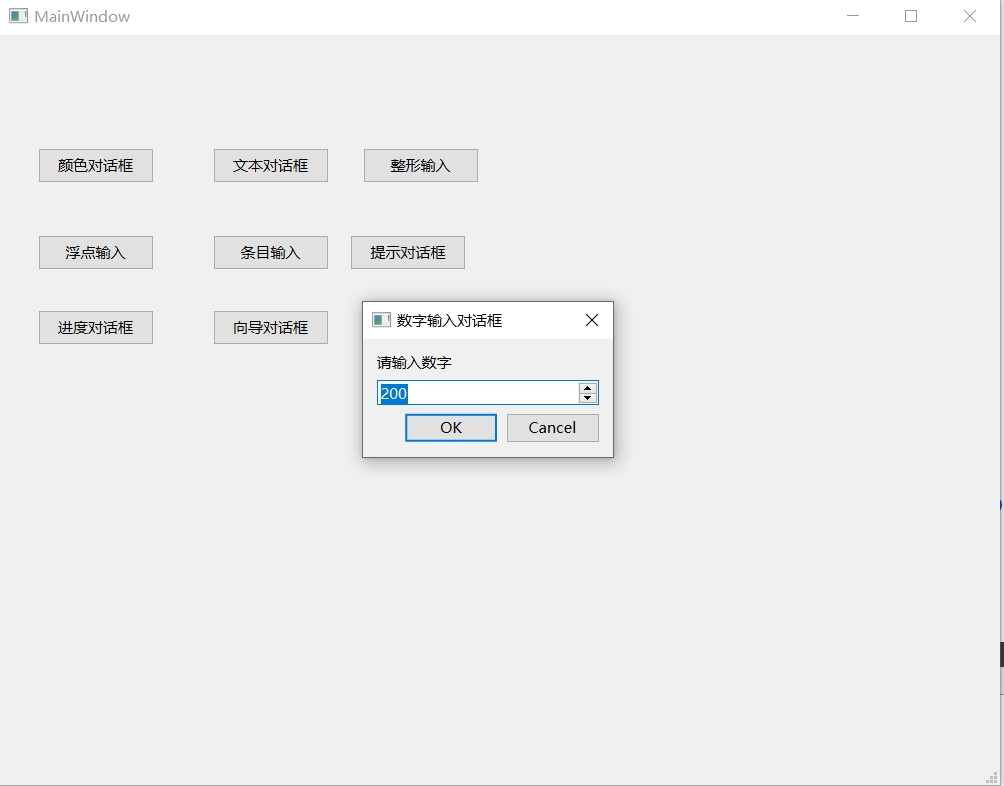
3~指定一个默认路径，当我们去打开这个文本对话框后，这个对话框将会显示这个默认路径下的文件

4~过滤器，这个过滤器有几个命名规则，自己看吧。该参数得是一个QString对象，其中应该是类似于【文本文件(\*.txt)】,前面的文字可以理解为注释，重点在于后面这个括号中的信息，使用**\*+后缀名**来指定我们将要过滤的文件，使用这个过滤器之后，我们整个对话框将会筛选全部文件直到剩下对应的后缀文件。特殊的，**\*.\***表示不过滤任何文件。

介绍完参数，我们来看一下这个静态方法的返回值，这个方法在调用后将会返回一个QString类型的字符串，这个字符串是我们在这个文本对话框中打开的文件的地址。通过保存这个地址，我们可以使用其在项目之后进行一些特殊的操作。

QString filter = tr("文本文件(\*.txt);;图片文件(\*.jpg \*.gif \*.png);;所有文件(\*.\*)");

接下来看到输入对话框(QInputDialog)，该对话框还存在分类包括文本输入对话框，整数输入对话框，浮点数输入对话框，条目输入对话框。先来看到整形输入对话框。



整形文本输入对话框

类名:QInputDialog(这个是系列对话框的类)

这个对话框更上面的那个文件对话框一样，不需要我们去创建一个具体的对象，只需要去调用对应的静态方法即可，对应的整形输入文本对话框的静态方法为QInputDialog::getInt()

接下来看到参数，这个函数需要的参数也比较多，一个个来看

1~父窗口对象指针，没什么好说的，就是挂到对应的父窗口上，类型是QWidget\*,即只要是QT中自带的窗口类都能够挂，自己建的窗口类得注意这一点。

2~一个QString对象常量引用，用于指定当前窗口的名称，即左上角的那个窗口名

3~一个QString对象常量引用，用于指定这个整形输入窗口的提示文本，需要注意这里的常量引用参数

4~一个int型数据，用于指定打开窗口时上面的默认值

5~一个int型数据，用于指定窗口输入数据的最小值

6~一个int型数据，用于指定窗口输入数据的最大值

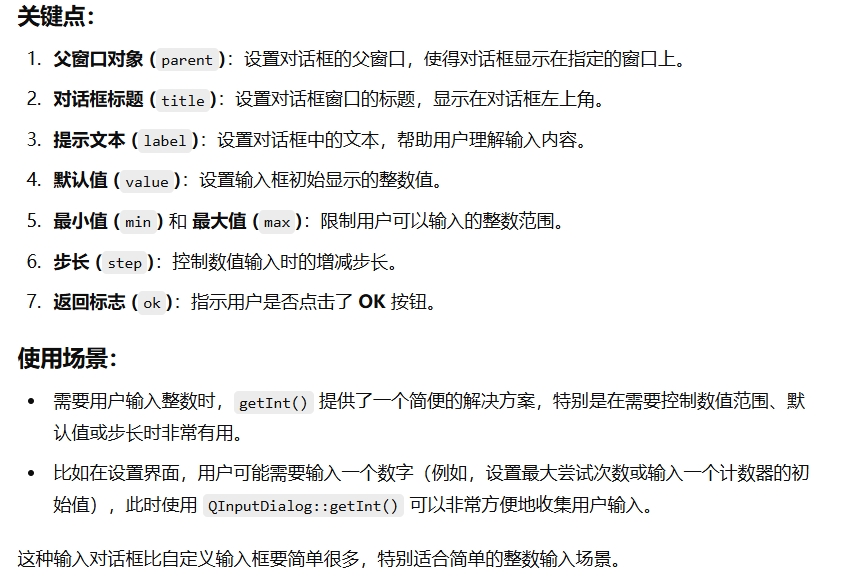
7~一个int型数据，用于指定窗口输入数据的增长条的步长

8~一个bool指针，这个是用于这个函数去返回的标志，需要在外部设置一个变量进行储存，通过这个可以检测窗口是否正确运行。检测这个指针指向的变量，若为true，窗口正确运行，反之则出错。

接下来还得看到这个静态方法的返回值，该方法的返回值是我们关闭窗口是这个窗口显示的整数值，可以通过int型数据进行储存。

static int getInt(QWidget \*parent, const QString &title, const QString &label, int value = 0,

int min = INT\_MIN, int max = INT\_MAX, int step = 1, bool \*ok = nullptr);



接下来看到第二种，浮点数输入对话框。这个其实跟上面的整形输入对话框照猫画虎，大致相同。

方法:QInputDialog::getInt();

参数列表：

1~父窗口对象

2~QString常量引用对象，指定父窗口标题

3~QString常量引用对象，指定窗口提示文本

4~double对象，用于指定默认值

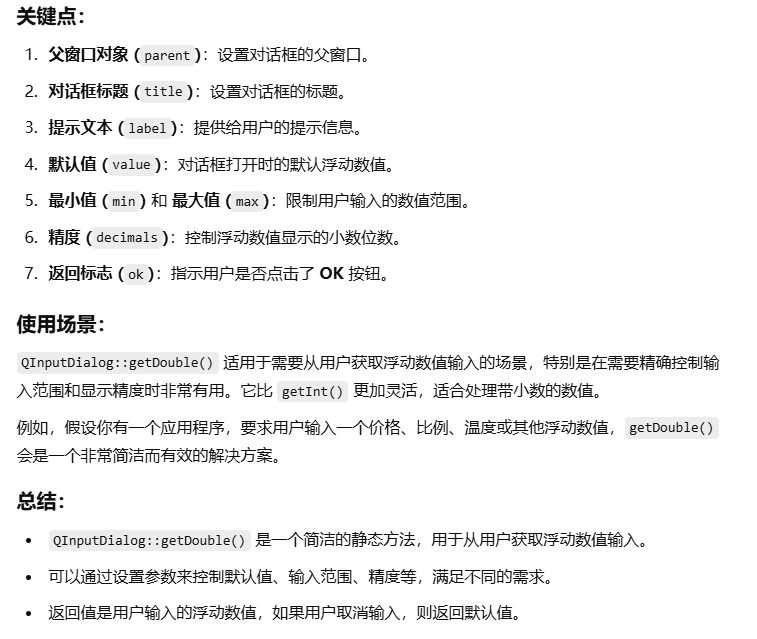
5~double对象，用于指定最小值

5~double对象，用于指定最大值

6~int对象，用于指定精度，即保留小数点后几位，需要注意，这个对话框的默认步长为0.1

7~bool指针，返回标志，用于标识用户是否按下OK按钮。

这个方法的返回值是一个double型数据，返回当前对话窗的最后显示的数据。可用对应容器接收并使用。



接下来看到下一个对话框，**条目对话框**。

这些个对话框其实都是一个抽象类出来的，所以我们加快点速度

方法：QInputDialog::getItem(this,tr("条目输入对话框"),tr("输入或选择条目"),items,0,true, &ok);

参数列表:

1~父窗口指针

2~QString常量引用对象，指定父窗口标题

3~QString常量引用对象，指定窗口提示文本

4~一个QStringList对象，这个对象是将要压入这个对话框的条目，需要在外部构造并输入对应的数据。比如

QStringList items;

items << tr("条目1") << tr("条目2");

这个是重载了对应的<<符，所以直接用吧

5~一个整形型，用于指定默认的条目，需要注意的就是，前面的这个QStringList其实就是一个类似于vector之类的容器，你压入的顺序可以以下标的顺序进行访问，所以0就指定了最开始压入的那个条目显示。

6~一个bool值，用于标识我们可不可以修改条目，如果不可以的话视觉效果就是选择一个下拉窗的条目，可以的话我们就可以自己去修改窗口的条目了。

7~bool指针，返回标志，用于标识用户是否按下OK按钮。

接下来看到对应的返回值，其实它的返回值是一个QString对象，内容是最后一次显示并带有保存的条目值，这个自己试一下就知道怎么回事了。

void MainWindow::on\_pushButton\_6\_clicked()

{

QStringList items;

items << tr("条目1") << tr("条目2");

bool ok = false;

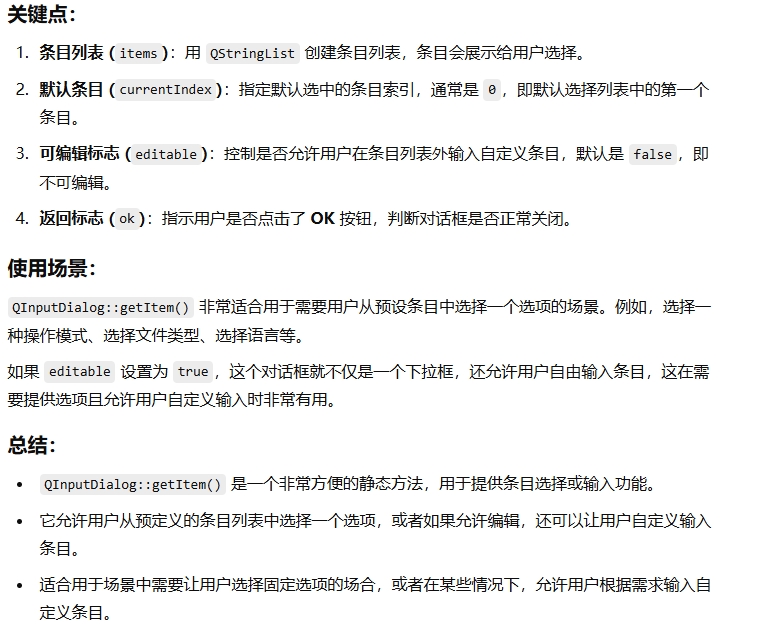
auto itemData = QInputDialog::getItem(this,tr("条目输入对话框"),tr("输入或选择条目"),items,0,true, &ok);

if(ok){

qDebug() << "item is " << itemData << endl;

}

}



接下来看到提示对话框了

这个提示对话框中，存在四个子类对话框，不过答题都是相同的，所以我们这里只分析一个就行了，其他自行分析

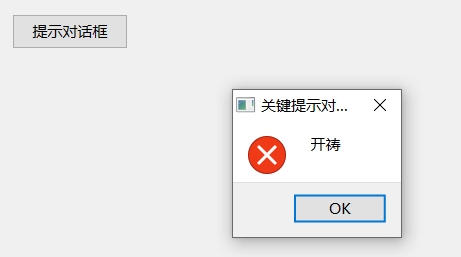
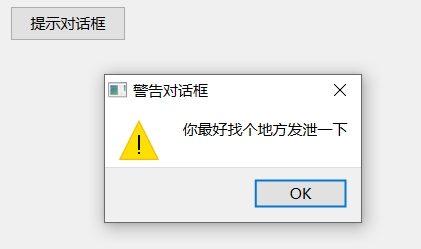
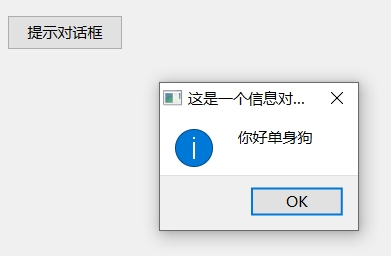
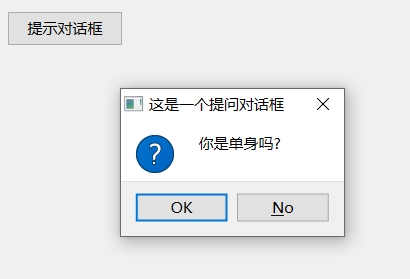
提示对话框均为静态方法

类名QMessageBox

四个不同类型的字对话框方法

question方法，information方法，waring方法，critical方法

这些方法的主要区别就是对应的对话窗的图标有所不同，其中question方法对应的图标是一个蓝色圆圈带中心白色问号，information是蓝色圆圈带白色感叹号，waring是黄色三角带白色感叹号，critical是红色圆圈带白色叉。



来分析下方法的参数，以下面这个举例

QMessageBox::question(this,tr("提问对话框"),tr("你是单身吗"),QMessageBox::Yes, QMessageBox::No);

在这一系列参数中

1~父窗口指针

2~QString常量引用对象，指定父窗口标题

3~QString常量引用对象，指定窗口提示文本

4~对应的QMessageBox宏，通过添加这些宏可以添加我们的对话窗按钮，可以添加多个按钮吗，在宏之间使用|进行间隔即可

5~QMessageBox宏，用于指定默认的按钮，指定之后进入这个对话窗后对应的按钮会有一个蓝色边框。需要注意，如果这个默认按钮没有出现在前面的设置中，将会直接创建在这个对话框中。

void MainWindow::on\_pushButton\_7\_clicked()

{

auto ret = QMessageBox::question(this,tr("提问对话框"),tr("你是单身吗"),QMessageBox::Yes, QMessageBox::No);

if(ret == QMessageBox::Yes || ret == QMessageBox::No){

qDebug() << "ret is " << ret << endl;

}

auto ret2 = QMessageBox::information(this,tr("通知对话框"),tr("你好单身狗"),QMessageBox::Ok);

if(ret2 == QMessageBox::Ok){

qDebug() << "ret2 is " << ret2 << endl;

}

auto ret3 = QMessageBox::warning(this,tr("警告对话框"),tr("你最好找个地方发泄一下"),QMessageBox::Ok);

if(ret3 == QMessageBox::Ok){

qDebug() << "ret3 is " << ret3 << endl;

}

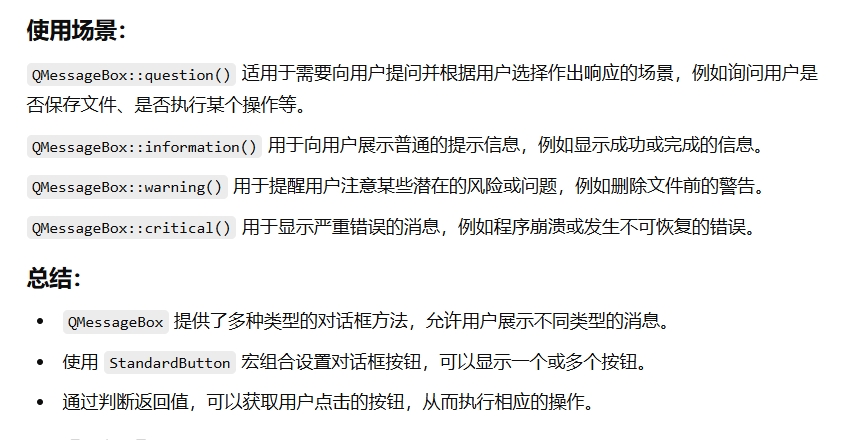
auto ret4 = QMessageBox::critical(this,tr("关键提示对话框"), tr("我梦寐以求是真爱和自由"),QMessageBox::Ok);

if(ret4 == QMessageBox::Ok){

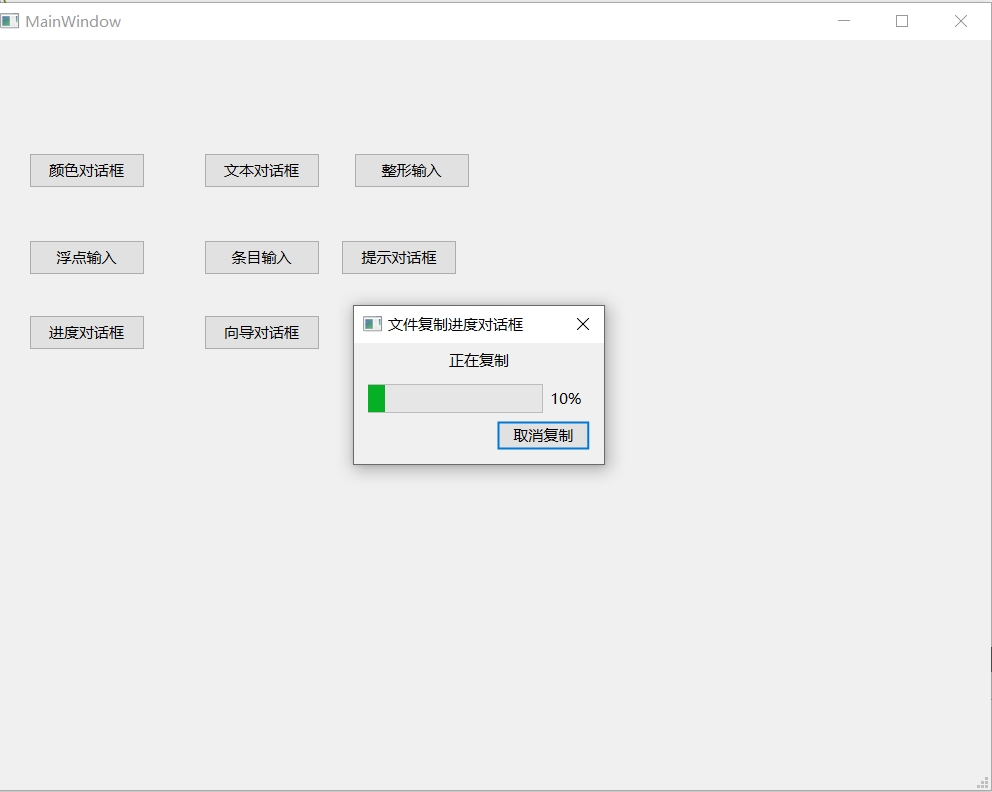
qDebug() << "ret4 is " << ret4 << endl;

}

}



接下来看到倒数第二个常用对话框，进度条对话框



进度条对话框就不像前面几个对话框了，这个对话框需要通过一个普通的方法进行创建，就比如

this->myProDlg=new QProgressDialog(tr("正在复制"),tr("取消复制"),0,5000,this);

我们来分析下这里的参数列表

1~QString常量引用对象，用于标识这个进度条对话框正在做的事情

2~QString常量引用对象，用于标识这个进度条对话框右下角按钮的内容

3~int型数据，进度条的最小值

4~int型数据，进度条的最大值

5~QWidget\*指针，用于挂载对象

这里需要来分析下我们的进度条计算原理，我们假设我们的最大值为5000，最小值为0，我们使用\_counter来进行计算，这个变量会根据计时器来进行自动的自增，步长为1，需要注意的是，即使我们的这个变量改变了，我们也需要使用setValue方法进行更新视图。这里的进度条的百分比数计算

(现有的\_counter-原有的\_counter)/(最大值-最小值)\*100%

当然，这里是一种很粗糙的说法，具体的等到之后我需要用到的时候再去进行一次深入的了解吧。

进度条的增长速度与 \_counter 的增长速度直接相关

当我们复制或进行文件传输时，会显示一个进度条的对话框，我们可以使用QProcessDialog来操作。

void MainWindow::on\_pushButton\_8\_clicked()

{

// 进度提示框一闪而逝，不是逻辑问题，是5000量级太小了

QProgressDialog progressDialog(tr("正在复制"),tr("取消复制"),0,5000,this);

progressDialog.setWindowTitle(tr("文件复制进度对话框"));

progressDialog.setWindowModality(Qt::ApplicationModal);

progressDialog.show();

for(int i = 0; i < 5000; i++){

progressDialog.setValue(i);

QApplication::processEvents();

if(progressDialog.wasCanceled()){

break;

}

}

progressDialog.setValue(5000);

}

因为循环很快完成，所以进度条一闪而逝，那么我们可以通过定时器的方式定时更新进度，这样就可以更清楚的看到进度条对话框更新情况了

void MainWindow::on\_pushButton\_8\_clicked()

{

//改用定时器

\_progressDialog = new QProgressDialog(tr("正在复制"),tr("取消复制"),0,5000,this);

\_progressDialog->setWindowTitle(tr("文件复制进度对话框"));

\_progressDialog->setWindowModality(Qt::ApplicationModal);

\_timer = new QTimer(this);

connect(\_timer, &QTimer::timeout, this, &MainWindow::on\_updateProgressDialog);

connect(\_progressDialog, &QProgressDialog::canceled, this, &MainWindow::on\_cancelProgressDialog);

\_timer->start(2);

qDebug("复制结束");

}

上面代码连接了定时器的timeout信号和主窗口的on\_updateProgressDialog函数, 该函数的主要作用就是定时更新进度条，逻辑如下

void MainWindow::on\_updateProgressDialog(){

\_count++;

if(\_count > 5000){

\_timer->stop();

delete \_timer;

\_timer = nullptr;

delete \_progressDialog;

\_progressDialog = nullptr;

\_count = 0;

return;

}

\_progressDialog->setValue(\_count);

}

另外我们需要在进度条满了的时候发消息通知主窗口关闭

void MainWindow::on\_cancelProgressDialog(){

\_timer->stop();

delete \_timer;

\_timer = nullptr;

delete \_progressDialog;

\_progressDialog = nullptr;

\_count = 0;

return;

}

这样进度条满了之后我们就可以回收定时器资源了。

接下来看到最后一个常用的对话框，**向导对话框**

这个相对来说比较复杂，存在多个层次。

最外层是一个QWizard容器，这个容器就是我们的向导对话框，其的创建也比较简单，可以直接使用无参的构造函数进行创建，不过最好还是制定一个父对象来进行挂载。

接下来看到这个容器里的基础组成元素，QWizardPage，这个是向导页，也是向导的基本组成单元，一个向导可以存在多个向导页，向导页的顺序与他们被压入向导对话框的顺序直接挂钩。在向导对话框中，已经设计好了相邻向导页之间的切换。对于向导页，我们可以通过QWizard的addPage方法进行向导页的添加。

在每一个向导页中，我们可以进行对其一定的设计，对于每个向导页的设计，我们需要一个QVBoxLayout对象，这些个对象将用于管理这里的向导页的设计。在这个对象中，我们又可以添加多个空间，只要他们是对应的Widget\*对象即可。对应的对象可以通过QVBoxLayout对象的addWidget方法进行添加。

当我们的QVBoxLayout对象设计好了之后，我们可以通过QWizardPage类的方法setLayout方法将我们设计好的QVBoxLayout对象压入到我们设计的向导页中，当我们全部的layout设计好，应用到对应的向导页中，然后我们这些个向导页又正确的压入我们的向导对话框中后，我们就完成了对于向导对话框的设计。此时我们就可以直接对于这个QWizard对象调用exec方法，这个方法将会阻断当前的窗口程序并显示对应的向导对话框。



对于一个向导对话框的个性化设计，我们可以通过继承对应的QWizardPage页来实现我们自己的向导页，接下来，在我们从QWizard继承的自己的向导对话框类中，我们可以在构造函数中进行对应的向导页的初始化。