1 Hello world

2017年1月29日 21:37

```
// main.cpp
#include <QApplication>   // QApplication类
#include <QLabel>   // QLabel类
int main(int argc, char *argv[]){
    /*
     * main函数一般以创建QApplication实例开始
     * GUI程序为QApplication类，非GUI程序为QCoreApplication类，前者为后者的子类
     * 该对象用于管理QT程序的生命周期，开启事件循环
     */
    QApplication app(argc, argv);
    // 让QLabel显示Hello world
    QLabel label("Hello world");
    label.show();
   return app.exec();  // 开启事件循环，防止程序被直接退出
}
```

QLabel可以通过new创建，但并不建议；
因为当main函数退出后QLabel并不会自动销毁，虽然程序结束后操作系统会进行垃圾回收，但这种编程习惯是不可取的；

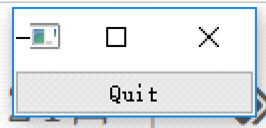
在控制台打印调试信息——
qt编程中没有cout，但是可以用qDebug来代替，如：`qDebug() << "Discard changes!";`
此时需要包含头文件QDebug

2 信号槽

2017年1月29日 22:14

信号槽：实际就是观察者模式——当某个事件发生后（比如一个按钮被点击），它就会广播一个信号（signal），如果有哪个对象对这个信号感兴趣，它就会使用连接（connect）函数（即一个槽）来处理这个信号。

```
#include <QApplication>
#include <QPushButton>  // QPushButton类
int main(int argc, char *argv[]){
    QApplication app(argc, argv);
    QPushButton button("Quit");      // 创建一个显示文本为Quit的按钮
    // 将按钮的点击事件连接到app的quit函数上（即点击按钮后退出本程序）
    // 最常用的connect版本——connect( const Qobject * sender, PointerToMemberFunction signal, Functor slot )
    QObject::connect(&button, &QPushButton::clicked, &QApplication::quit);
    button.show();
    return app.exec();
}
```



connect的基本形式：connect(sender, signal, receiver, slot)

- 返回值为QMetaObject::Connection类型，暂且不关心这个值
- sender为信号槽的发送者，为const QObejct *类型
- signal为信号类型即事件类型，可以是字符串const char *，可以是const QMetaMethod &，也可以是PointerToMemberFunction
- receiver为信号槽的接收者，可有可无（默认为this），类型同sender，即const Qobject *
- slot为信号槽即回调函数，类型可以是signal的那三种类型之外，还可以是Functor类型（static函数、全局函数、Lambda表达式）

信号槽要求：

信号signal与槽slot的参数类型和顺序一致（允许slot参数较少，此时忽略信号的末尾几个参数）

3 自定义信号槽

2017年1月30日 11:10

报纸与订阅：信号槽机制可以比作报纸与订阅，订阅者（观察者）可以订阅报纸（被观察者），订阅者会将自身注册到报纸的一个容器当中，当报纸更新时它就会逐一去通知容器中的所有订阅者

```
////////// newspaper.h
#include <QObject>
class Newspaper : public QObject{  // 发送类sender，只有继承QObject的类才具有信号槽能力
    /*
     * 凡是QObject类及其子类第一行都必须是Q_OBJECT
     * 它将展开为该类提供信号槽机制、国际化机制以及QT提供的不基于RTTI的反射能力
     * 该宏命令只在头文件生效，如果需要在cpp文件定义QObject的子类，那么需要额外的操作
     */
    Q_OBJECT
public:
    Newspaper(const QString & name) : m_name(name){}
    void send(){
        // emit是qt定义的关键字，即为发出（信号）
        // 可以把它作为一个语句来使用，这里表示调用signals的newPaper来发出信号
        emit newPaper(m_name);
    }
// signals是qt定义的关键字，这里的newPaper只声明而没有定义
// Q_OBJECT将自动实现相应的函数体
signals:
    void newPaper(const QString &name);     // 信号函数signal
private:
    QString m_name;
};
```

```
////////// reader.h
#include <QObject>
#include <QDebug>
class Reader : public QObject{  // 接收类receiver
    Q_OBJECT
public:
    Reader() {}
    void receiveNewspaper(const QString & name){    // 槽函数slot
        // qDebug类似cout，用于在控制台打印信息
        //使用qDebug时需要在pro文件中添加一句“QMAKE_CXXFLAGS += -std=c++0x”
        qDebug() << "Receives Newspaper: " << name;
    }
};
```

```
////////// main.cpp
#include <QCoreApplication>
#include "newspaper.h"
#include "reader.h"
int main(int argc, char *argv[]){
    QCoreApplication app(argc, argv);
    Newspaper newspaper("Newspaper A");
    Reader reader;
    // connect相当于把reader注册到newspaper的一个容器当中
```

```
    QObject::connect(&newspaper, &Newspaper::newPaper,
                     &reader,    &Reader::receiveNewspaper);
    newspaper.send();   // newspaper发送信号，感兴趣的reader将接收到信号并处理
    return app.exec();
}
```

- 槽的无默认参数的参数数量必须小于等于信号参数数量
  信号的参数实际上是“返回值”，用来作为槽的参数；
  可以忽略部分“返回值”，但绝对不能引用不存在的“返回值”；
- 对于重载的槽函数或信号函数

```
// 如两个信号函数——
void newPaper(const QString &name, const QDate &date);
void newPaper(const QString &name) const;
```

```
// 如果只告知函数名，编译器将分不清你用的是哪个版本的函数从而报错
QObject::connect(&newspaper, &Newspaper::newPaper,
                     &reader,    &Reader::receiveNewspaper);
```

```
// 方法一：借助SIGNAL宏和SLOT宏来处理信号（槽）函数，告诉编译器是用具体的哪个版本的函数
QObject::connect(&newspaper, SIGNAL(newPaper(QString, QDate)),
                 &reader,    SLOT(receiveNewspaper(QString, QDate)));
```

```
// 方法二：借助函数指针来指明函数版本
void (Newspaper:: *newPaperNameDate)(const QString &, const QDate &) =
&Newspaper::newPaper;
QObject::connect(&newspaper, newPaperNameDate,
                 &reader,    &Reader::receiveNewspaper);
```

```
// 方法三：直接通过强制类型转换指明函数版本
//        （C语法，不建议）
QObject::connect(&newspaper,
                  (void (Newspaper:: *)(const QString &, const QDate
&))&Newspaper::newPaper,
                 &reader,
                 &Reader::receiveNewspaper);
//        （C++语法，推荐）
QObject::connect(&newspaper,
                 static_cast<void (Newspaper:: *)(const QString &, const QDate
&)>(&Newspaper::newPaper),
                 &reader,
                 &Reader::receiveNewspaper);
```

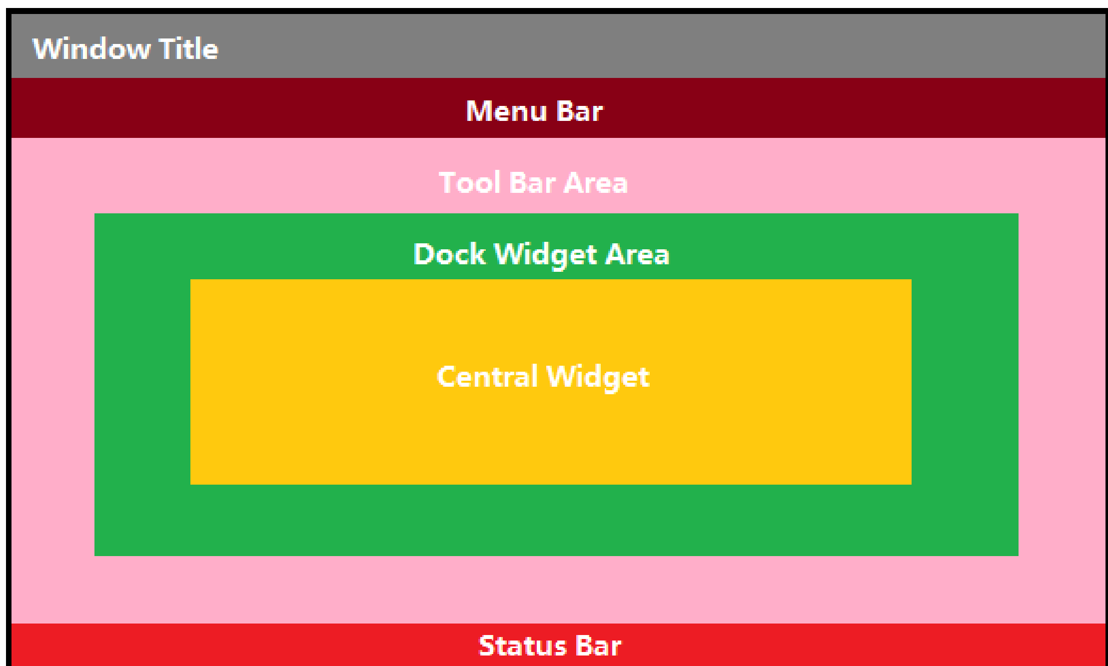
方法一为QT4语法，方法二、三为QT5新语法；
如果是有默认参数的槽函数，方法二、三则无法使用，因为参数默认值只能适用于函数调用中，当用指针取地址时，默认值是不可见的，编译器无法分辨出这个参数是不是有默认值；此时只能采用方法一，如果还坚持使用QT5语法，则可以借助如下形式的Lambda函数（支持QT5的编译器必定支持Lambda函数）

```
QObject::connect(&newspaper,
                 static_cast<void (Newspaper:: *)(const QString &)>
(&Newspaper::newPaper),
                 [=](const QString &name) { /* Your code here. */ });
```

4 MainWindow

2017年1月30日　　14:27

- 窗口的基本组成



  - Window title：标题栏
  - Menu bar：菜单栏
  - Status bar：状态栏
  - Tool bar area：工具条区域
  - Dock widget area：停靠窗口区域（如侧边栏等可以浮动的区域）
  - Central widget：主窗口

通常我们的程序会继承自QMainWindow，以便获得其提供的各种便利的函数

5 动作

2017年1月30日 14:42

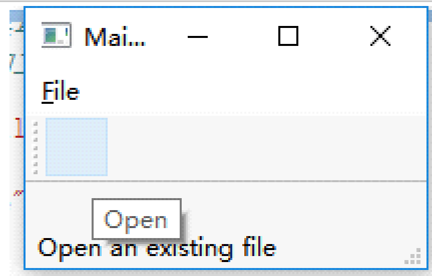
- QAction类
  其对象中可以设置动作对应的图标、菜单文字、快捷键、状态栏文字、浮动帮助等信息；
  可以将该动作对象添加到菜单栏、工具栏等地方中去，此时由QT来选择显示哪些属性而无须我们关心；
  同时使用同一QAction对象的动作将实现同步更新

```
// ========== mainwindow.h
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>  // QMainWindow
class MainWindow : public QMainWindow{  // 自定义一个主窗口函数并继承自QMainWindow
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    void open();              // 动作所连接的slot函数
    QAction *openAction;      // 声明一个open的动作
};
#endif // MAINWINDOW_H
```

```
// ========== mainwindow.cpp
#include <QAction>      // QAction
#include <QMenuBar>     // QMenu
#include <QMessageBox>  // QMessageBox
#include <QStatusBar>   // statusBar
#include <QToolBar>     // QToolBar
#include "mainwindow.h"
MainWindow::MainWindow(QWidget *parent) :  QMainWindow(parent){
    // tr函数用于将字符串国际化，其中文本常用英文
    // setWindowTitle函数用于设置窗口标题栏
    setWindowTitle(tr("Main Window"));
    /*
     * 创建一个QAction的对象
     *   QIcon图标的参数是一个字符串，该字符串对应资源文件（后缀为.qrc）的一段路径
     *   第二个参数的字符串为显示的文本，前缀&表明这有一个快捷键
     * setShortcuts用来设置前述的快捷键
     *   这里QKeySequence提供了很多内置的快捷键，当然也可以直接设置为tr("Ctrl+O"),
     *   但不同平台的按键有所区别，比如mac键盘没有ctrl;
     *   而QKeySequence会根据平台的不同来设定相应的快捷键
     * setStatusTip用来设置鼠标滑过该图标时状态栏的提示内容
     */
    openAction = new QAction(QIcon(":/images/doc-open"), tr("&Open..."), this);
    openAction->setShortcuts(QKeySequence::Open);
    openAction->setStatusTip(tr("Open an existing file"));
    connect(openAction, &QAction::triggered, this, &MainWindow::open);
    /*
     * 菜单栏、工具栏、状态栏设置——
     * menuBar、toolBar、statusBar分别对应菜单栏、工具栏、状态栏
     * 将前边设置的QAction对象添加到菜单栏和工具栏当中
     */
```

```
    QMenu *file = menuBar()->addMenu(tr("&File"));
    file->addAction(openAction);
    QToolBar *toolBar = addToolBar(tr("&File"));
    toolBar->addAction(openAction);
    statusBar() ;
}
// 空析构
MainWindow::~MainWindow(){}
// 用作slot的open函数，打开一个信息框
void MainWindow::open(){
    QMessageBox::information(this, tr("Information"), tr("Open"));
}
```

```
// main.cpp
#include <QApplication>
#include "mainwindow.h"
int main(int argc, char *argv[]){
    QApplication app(argc, argv);
    MainWindow win;
    win.show();
    return app.exec();
}
```



6 资源文件

2017年1月31日　　13:52

- 创建资源文件（文件和类 - QT - Qt资源文件，后缀.qrc）
- 添加前缀（如/images）
- 为前缀添加文件，如123.png，此时可以通过:/images/123.png来使用这一资源，
  也可以并且建议为文件设置别名，如img，此时可以通过:/images/img.png，当文件名字改变时无需改动代码；
- 属性中的语言可以对资源进行国际化，比如设置为fr，则当本地化信息为fr（即Qlocal::system().name()返回fr_FR）则优先使用fr语言下的资源，否则使用默认的即没有设置语言的资源

7 对象模型

2017年1月31日 15:42

- QT在C++编译器预处理前提前对QT代码进行一次moc预处理（Meta Object Compiler）
  这就是为什么代码中会有些函数只做了声明而没有定义
- QT的moc特性（摘抄）：
  - 信号槽机制，用于解决对象之间的通讯，这个我们已经了解过了，可以认为是 Qt 最明显的特性之一；
  - 可查询，并且可设计的对象属性；
  - 强大的事件机制以及事件过滤器；
  - 基于上下文的字符串翻译机制（国际化），也就是 tr() 函数，我们简单地介绍过；
  - 复杂的定时器实现，用于在事件驱动的 GUI 中嵌入能够精确控制的任务集成；
  - 层次化的可查询的对象树，提供一种自然的方式管理对象关系。
  - 智能指针（QPointer），在对象析构之后自动设为 0，防止野指针；
  - 能够跨越库边界的动态转换机制。
  - moc的代码更加灵活，虽然效率不如C++的模板（大概一个moc相当于四个模板）
- QObject是以对象树的形式组织的
  - QObject的构造函数都有一个parent的父对象指针
  - 当父对象析构时，子对象也会被析构
  - 对象树的对象是没有顺序的，也就是说，析构时没有顺序的

```
1 {
2     QPushButton quit("Quit");
3     QWidget window;
4 
5     quit.setParent(&window);
6 }
```

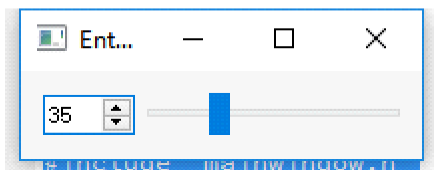
    - 上述代码中，由于C++的机制，先析构window再析构quit
    - 而由于QT的对象树机制，window是quit的父对象，析构window前会先去析构子对象quit
    - 这就导致quit发生两次析构，C++不允许对同一对象析构两次，于是程序将崩溃
    - 因此，在QT中要尽量在构造时指明parent对象，同时大胆地在堆上创建（new）

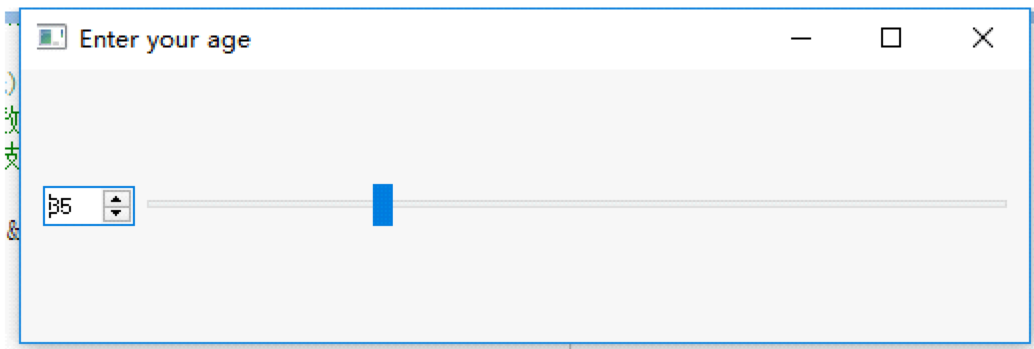
8 布局管理器

2017年1月31日　　16:36

- QT提供两种定位机制
  - 绝对定位：给定组件具体的坐标和大小
  - 布局定位：由布局管理器智能管理

```
#include <QApplication>
#include <QWidget>
#include <QSpinBox>
#include <QSlider>
#include <QHBoxLayout>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    window.setWindowTitle("Enter your age");
    // QSpinBox是一个带有上下箭头的数字输入框
    QSpinBox *spinBox = new QSpinBox(&window);
    // QSlider是一个带有滑块的滑竿
    QSlider *slider = new QSlider(Qt::Horizontal, &window);
    // 为两个组件设置范围
    spinBox->setRange(0, 130);
    slider->setRange(0, 130);
    // 当滑块变化时改动输入框的值
    QObject::connect(slider, &QSlider::valueChanged, spinBox, &QSpinBox::setValue);
    /*
     * 当输入框的值变化时改变滑块
     * 注意这里QSpinBox::valueChanged有两个版本——
     *     void valueChanged(int)
     *     void valueChanged(const QString &)
     * 直接对函数取地址，编译器无法知道是函数的哪一个版本而出错
     * 因此这里定义一个明确的指针，避免出现歧义
     */
    void (QSpinBox:: *spinBoxSignal)(int) = &QSpinBox::valueChanged;
    QObject::connect(spinBox, spinBoxSignal, slider, &QSlider::setValue);
    spinBox->setValue(35);
    // QHBoxLayout是一个布局管理器
    QHBoxLayout *layout = new QHBoxLayout;
    // 将滑竿和输入框加入到布局管理器中
    layout->addWidget(spinBox);
    layout->addWidget(slider);
    // 为窗口设置布局管理器
    window.setLayout(layout);
    window.show();
    return app.exec();
}
```





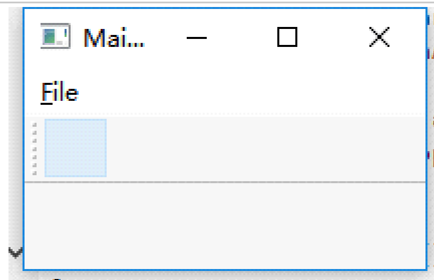
- QT提供的几种布局管理器
  - QHBoxLayout：水平布局（从左到右）
  - QVBoxLayout：竖直布局（从上到下）
  - QGridLayout：网格布局（类似html的table）
  - QFormLayout：表格布局，每行一段文本带一个组件（类似html的form）
  - QstackedLayout：层叠布局，允许第三维度的布局

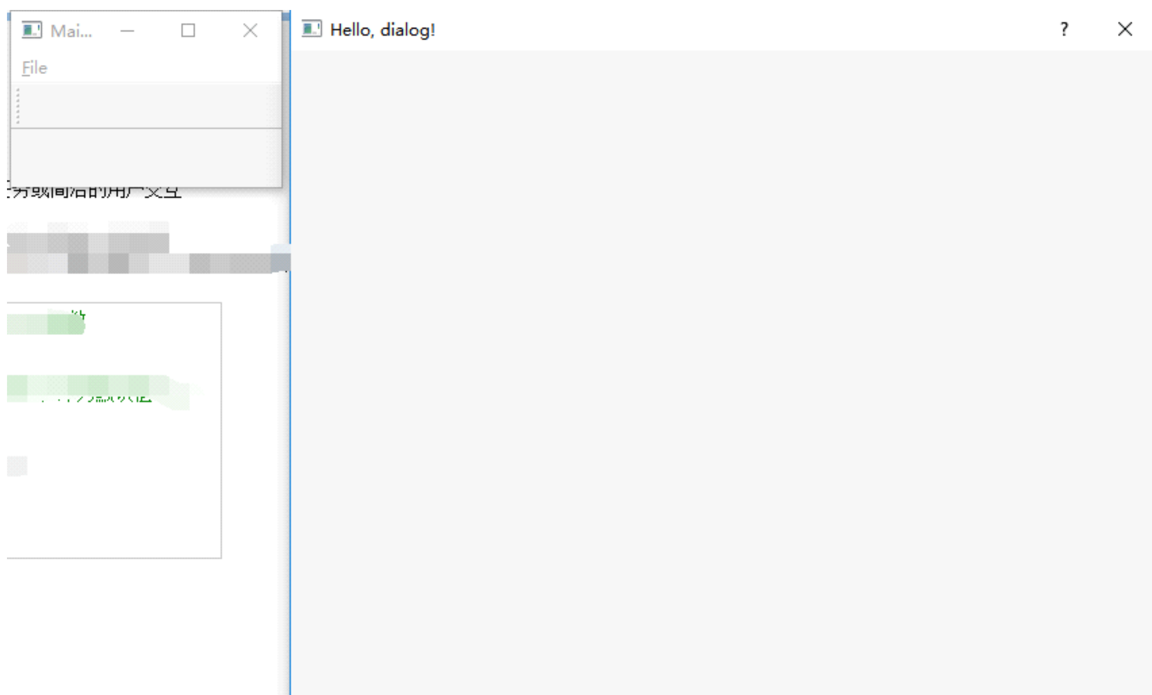
9 对话框

2017年1月31日　　23:05

- 对话框通常是一个顶层窗口，用于实现短期任务或简洁的用户交互
- QDialog类
  parent指针为NULL时为顶层窗口，此时任务栏会占用独立的位置；
  parent指针非NULL时，创建的对话框会出现在parent组件的中心，而在任务栏中与父组件共享位置

```
// 借用《5 动作》中的代码，仅改动MainWindow::open函数
void MainWindow::open()
{
    // 创建一个QDialog的实例（此处没有指明父组件，即为默认值NULL）
    QDialog dialog;
    // 设置对话框标题
    dialog.setWindowTitle(tr("Hello, dialog!"));
    // 运行（打开）对话框
    dialog.exec();
}
```

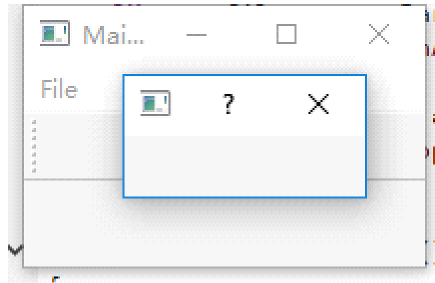


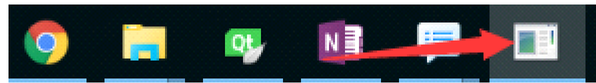


可以看到对话框与主窗口在任务栏中的位置独立（因为是同一个进程，win10将其折叠了起来）

- 如果将QDialog的实例的父组件指定为主窗口MainWindow

即——QDialog dialog(this);   // open是MainWindow的成员函数，this即为MainWindow的实例
此时对话框将在主窗口的中心创建，同时在任务栏中与父组件共享位置





- 模态对话框：阻塞同一程序的其他窗口
  - 应用程序级别的模态对话框：阻塞同一程序的所有其他的窗口，QDialog::*exec*();
  - 窗口级别的模态对话框：仅仅阻塞与对话框相关的窗口，QDialog::*open*();

  非模态对话框：不阻塞其他窗口，QDialog::show()
- 注意：因为非模态对话框不阻塞，所以当在栈上建立对话框，程序并不会在show处阻塞而会继续执行，此时如果退出了函数，它有可能就会被回收；因此，最好是将非模态对话框建立在堆上~
  但又有一个问题，建立在堆上的对话框只会在父组件被销毁时才会被自动销毁，如果父组件长时间存在而且打开了多个对话框，那么就存在内存泄漏的问题；解决方法——
  设置属性 dialog->setAttribute(Qt::WA_DeleteOnClose);
  设置该属性后当对话框被关闭之后，该对话框的实例就会被delete掉；
  也可以通过Qobject::deleteLater()函数来解决（但是需要额外开启一个事件循环）

- 数据传递
  对于模态对话框（以应用程序级别模态为例）
  - QDialog::exec()将返回QDialog::Accepted或QDialog::Rejected，分别表示用户点击了“确定”或“取消”
  - 可以直接借助其返回值来实现对话框的数据传递

非模态对话框因为程序没有堵塞，因此无法直接借助返回值来实现数据传递，此时应当借助信号槽机制；

```
/*
 * UserAgeDialog继承了QDialog类
 * QDialog类有accept()、reject()、done()三个相关的成员函数
 *      分别表示对话框被点击了“确认”、“取消”、“确认或取消”
 *      当对话框的相应按钮被点击时，对应的函数也会被触发
 */
void UserAgeDialog::accept(){
    // 调用signals标签下的userAgeChanged函数来发出信号
    emit userAgeChanged(newAge);
    // 此处重载了QDialog的accept函数，因此再次调用父类的accept来确保原本的操作被执行
    QDialog::accept();
}

void MainWindow::showUserAgeDialog(){
    // 创建对话框实例
    UserAgeDialog *dialog = new UserAgeDialog(this);
    // 连接信号槽，当对话框的“确认”被按下时，设置主窗口的成员数据，从而实现数据传递
```

```
    connect(dialog, &UserAgeDialog::userAgeChanged, this,
&MainWindow::setUserAge);
    // 非模态对话框
    dialog->show();
}

// 槽函数，接收信号中的数据，设置成员函数，完成数据传递
void MainWindow::setUserAge(int age){
    userAge = age;
}
```

10 标准对话框QMessageBox

2017年2月2日　　23:10

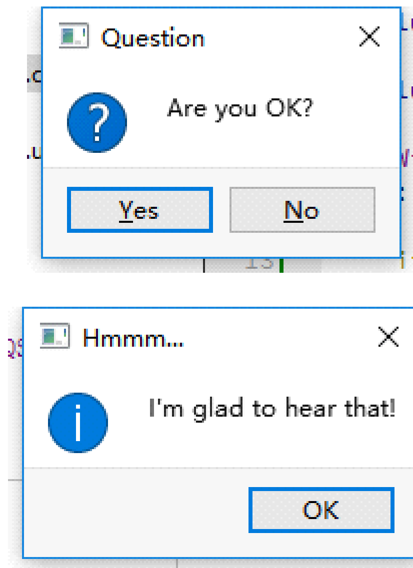
内置的对话框类型

- QColorDialog：颜色选择
- QFileDialog：文件或目录选择
- QFontDialog：字体选择
- QInputDialog：用户输入一个值，将其返回
- QMessageBox：模态对话框，显示信息、询问问题等
- QPageSetupDialog：打印相关的纸张选项
- QPrintDialog：打印机配置
- QPrintPreviewDialog：打印预览
- QProgressDialog：显示操作过程

QMessageBox模态对话框（几个已有的static函数模板）

- void about(QWidget * parent, const QString & title, const QString & text)
  最简单的对话框，参数为父组件、标题、内容和一个OK按钮
- void aboutQt(QWidget * parent, const QString & title = QString())
  显示有关QT的信息
- StandardButton critical(QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton)
  严重错误对话框，显示一个红色的错误符号，buttons为要显示的按钮（默认为一个ok按钮）
- StandardButton information(QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton)
  类似critical函数，但显示的是普通的信息图标
- StandardButton question(QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = StandardButtons(Yes | No), StandardButton defaultButton = NoButton)
  类似critical函数，显示的是一个问号图标，按钮为“是”和“否”
- StandardButton warning(QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton)
  类似critical函数，显示的是一个感叹号图标
- 使用示例

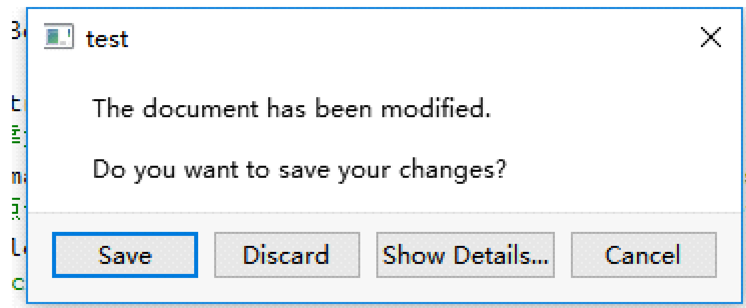
```
if (QMessageBox::Yes == QMessageBox::question(this,
                                              tr("Question"),
                                              tr("Are you OK?"),
                                              QMessageBox::Yes |
QMessageBox::No,
                                              QMessageBox::Yes)) {
    QMessageBox::information(this, tr("Hmmm..."), tr("I'm glad to hear
that!"));
} else {
    QMessageBox::information(this, tr("Hmmm..."), tr("I'm sorry!"));
}
```

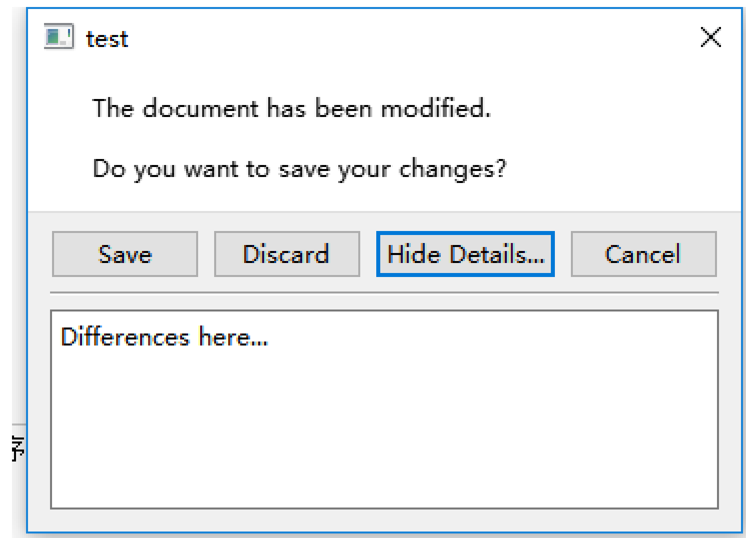


定制QMessageBox对话框

```
// 在栈上建立对话框对象
QMessageBox msgBox;
// 设置正文文本
msgBox.setText(tr("The document has been modified."));
// 设置提示文本（靠近底部的文本）
msgBox.setInformativeText(tr("Do you want to save your changes?"));
// 文本细节（通过点击按键来展开/隐藏这一段），这个API设置包含一个Show Details按钮
msgBox.setDetailedText(tr("Differences here..."));
// 补充Save, Discard, Cancel三个按钮
msgBox.setStandardButtons(QMessageBox::Save
                          | QMessageBox::Discard
                          | QMessageBox::Cancel);
// 设置默认按钮
msgBox.setDefaultButton(QMessageBox::Save);
// 把这个对话框加入事件循环并取得返回值
int ret = msgBox.exec();
switch (ret) {
case QMessageBox::Save:
    qDebug() << "Save document!";
    break;
case QMessageBox::Discard:
    qDebug() << "Discard changes!";
    break;
case QMessageBox::Cancel:
    qDebug() << "Close document!";
    break;
}
```

注意上面并没有指定按钮的顺序，QT会根据操作系统的习惯进行顺序调整，如果需要手动指定按钮的排列顺序，可以通过QDialogButtonBox类来实现


test
The document has been modified.
Do you want to save your changes?
Save
Discard
Show Details...
Cancel


test
The document has been modified.
Do you want to save your changes?
Save
Discard
Hide Details...
Cancel
Differences here...

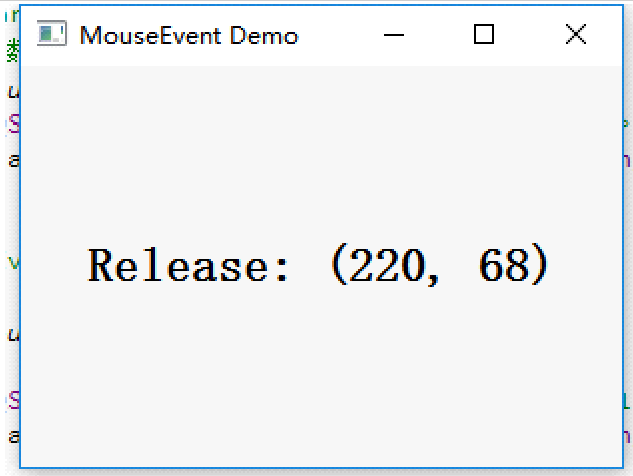
11 事件

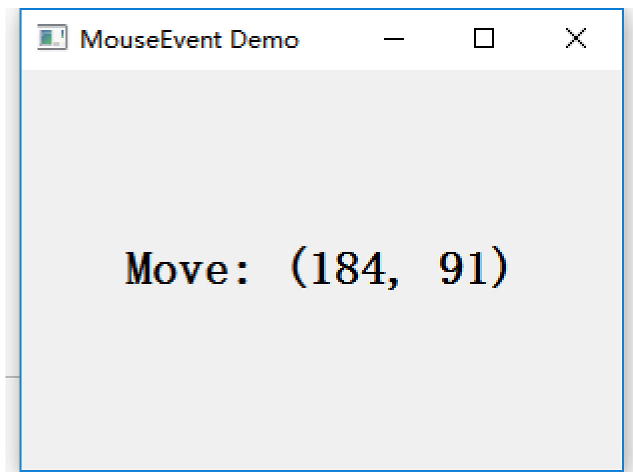
2017年2月9日 16:53

- 事件和信号槽
  - 信号槽由具体对象发出，然后马上交给相应的槽进行处理
    事件用一个事件队列来维护，当新事件产生时，会被追加到事件队列的尾部，也可以被直接处理
  - 事件可以用事件过滤器来过滤，对部分事件作出额外处理，或者忽略部分事件
  - 使用组件时，关心信号槽
    自定义组件时，关心事件
- 事件机制
  - QT在main函数创建一个QCoreApplication对象，然后调用它的exec函数开始事件循环监听应用程序事件
  - 事件发生后QT将创建一个事件对象（继承于QEvent类）并将其传递给QObject对象的event函数
  - event函数不直接处理事件，而将按照事件类型分发给特定的事件处理函数（event handler）
- 所有组件的父类QWidget中，定义了很多protected virtual的事件处理回调函数，可以在子类中进行重载

```
// 继承QLabel，并重载其中的三个函数
class EventLabel : public QLabel{
protected:
    void mouseMoveEvent(QMouseEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
};

// QLabel支持HTML语法
// QString有成员函数arg可以替换占位符
// event的x()、y()函数分别返回鼠标的x、y坐标
void EventLabel::mouseMoveEvent(QMouseEvent *event){
    this->setText(QString("<center><h1>Move: (%1, %2)</h1></center>")
                .arg(QString::number(event->x()), QString::number(event->y())));
}

// .....mousePressEvent的定义省略

// QString还有类似C语言sprintf函数可以用来格式化字符串
void EventLabel::mouseReleaseEvent(QMouseEvent *event)
{
    QString msg;
    msg.sprintf("<center><h1>Release: (%d, %d)</h1></center>",
                event->x(), event->y());
    this->setText(msg);
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    EventLabel *label = new EventLabel;
    label->setWindowTitle("MouseEvent Demo");
```

```
    label->resize(300, 200);
    label->show();

    return a.exec();
}
```





该程序中鼠标需要先点击一下程序，程序才能开始追踪鼠标；
这是因为label有一个属性mouseTracking，只有当其为true时才会追踪鼠标，其属性默认值为false，当鼠标点击一次之后才会变成true从而触发mouseMoveEvent事件，当然也可以设置label->setMouseTracking(true);让程序一开始就追踪鼠标

- 事件的接受与忽略
  - 事件event拥有两个成员函数accept()和ignore()，分别表示接受事件和忽略事件
  - 如果忽略了事件，那么事件将继续传播给父组件（注意不是父类）并寻找其他接受者，而且不会继续执行ignore之后的语句；如果接受则不会进一步传播事件
  - 事件对象默认采用accept，只有所有组件的父类QWidget会采用ignore
  - 有一个特殊的情况必须使用accept和ignore

```
// ...
textEdit = new QTextEdit(this); // 富文本文本框
setCentralWidget(textEdit);
// []表示lambda表达式，=表示以传值的形式捕获变量，&表示以引用的形式捕获变量
connect(textEdit, &QTextEdit::textChanged, [=]() {
    // 开启内容变动通知
    this->setWindowModified(true);
});

setWindowTitle("TextPad [*]");
// ...
```

```
void MainWindow::closeEvent(QCloseEvent *event){
    if (isWindowModified()) {   // 如果关闭窗口时检测到内容变动
        bool exit = QMessageBox::question(this,
                                          tr("Quit"),
                                          tr("Are you sure to quit this
application?"),
                                          QMessageBox::Yes | QMessageBox::No,
                                          QMessageBox::No) ==
QMessageBox::Yes;
        if (exit) { // 当点击Yes时，exit为true
            event->accept();    // 接受事件，窗口关闭
        } else {
            event->ignore();    // 拒绝事件，拒绝窗口的关闭
        }
    } else {
        event->accept();    // 拒绝事件，拒绝窗口的关闭
    }
}
```

- QWidget::event()
  如前所述，在子类中该函数一般负责事件的分发工作，如果需要在分发之前完成一些工作，那么可以重写这个函数，但务必不要忘了在重写的函数中调用父类的event函数

```
// CustomWidget是自定义的一个QWidget的子类
// 这里重写了event函数（返回bool，参数是一个QEvent指针）
//    如果传入的事件已经处理了，那么返回true，且事件对象设置了accept，
//       此时QT认为事件已经处理，然后继续处理队列中的下一个事件
//    如果事件不处理，则返回false，
//       此时QT认为事件还未处理，然后将其发送给其他对象进行处理
bool CustomWidget::event(QEvent *e){
    // type()返回事件的类型，他是QEvent::Type类型的一个枚举
    if (e->type() == QEvent::KeyPress) {
        // 将QEvent类的指针转换为其子类QKeyEvent类的指针（方便引用其中的成员）
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(e);
        if (keyEvent->key() == Qt::Key_Tab) {
            qDebug() << "You press tab.";
            return true;    // 事件处理完毕，返回true
        }
    }
    // 如果事件没处理完，则调用父类的event继续尝试分发事件
    return QWidget::event(e);
}
```

  特殊情况：如果要屏蔽不需要的事件处理器，可以不调用父类的event，例如创建一个组件只让他响应tab按键
- 事件过滤器
  - 有时候对象需要查看、拦截发送给另外对象的事件，拦截操作可以重写event函数来实现，但是要重写每一个组件的event函数很麻烦，更不要说还得重写时还得小心一堆问题，此时用事件过滤器来实现会更加合理

```
class MainWindow : public QMainWindow{
 public:
    MainWindow();
 protected:
    // 重载protected virtual的eventFilter函数，相当于创建了过滤器
    bool eventFilter(QObject *obj, QEvent *event);
 private:
    // 富文本框
```

```
    QTextEdit *textEdit;
};

// 主窗口创建一个富文本框，同时安装过滤器
MainWindow::MainWindow(){
    textEdit = new QTextEdit;
    setCentralWidget(textEdit);

    // 为富文本框组件安装一个过滤器
    // 任何QObject对可以作为过滤器，但只有重写了eventFilter函数的才是有效的
    // 此处是把MainWindow作为过滤器
    // installEventFilter是安装过滤器，removeEventFilter则是移除过滤器
    // installEventFilter可以多次调用以安装多个过滤器，过滤器是栈结构的，后进
的先被执行
    textEdit->installEventFilter(this);
}

/*
 * 过滤器函数
 * 返回值为bool，返回true时代表已经处理了事件，将阻止事件的继续转发；否则返回
false
 * 第一个参数为目标对象，过滤器将在目标对象接收到事件之前被执行；
 * 第二个参数即为接收到的事件
 */
bool MainWindow::eventFilter(QObject *obj, QEvent *event){
    // MianWindow的富文本组件
    if (obj == textEdit) {
        // 该组件的KeyPress事件
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
            qDebug() << "Ate key press" << keyEvent->key();
            return true;   // 富文本组件的KeyPress事件，返回true直接忽略事
件！！
        } else {
            return false;  // 富文本组件但非KeyPress事件，返回false使事件能顺
利的转发给obj
        }
    } else {
        // 对于非富文本组件，有可能还存在着其他过滤器，此处重新调用父类的过滤
器会比较可靠
        return QMainWindow::eventFilter(obj, event);
    }
}
```

    - 过滤器可以直接作用在整个应用程序上，非常方便，但这也将严重降低整个程序的分发效率，除非不得不使用，否则应尽量避开它
    - 如果在过滤器中delete了某个组件，那么务必返回true以阻止事件进一步传播，否则有可能事件会被传播到已经delete掉的组件上从而导致程序崩溃
    - 组件与对应的过滤器必须在同一线程上才是有效的
- QT事件处理机制的五个层次
    - 重写事件处理函数
    - 重写分发的event()函数
    - 在特定对象上安装过滤器
    - 在QCoreApplication::instance()上安装过滤器（该过滤器将作用于全部对象的全部事件，仅限于主线程）
    - 重写QCoreApplication::notify()函数（拥有最高控制权，且能作用于任何线程）
- 自定义事件

- 注册事件
  - 需要继承QEvent类，并且提供一个整型的type，QT保留0-999的值，并且提供两个边界常量，QEvent::User值为1000，QEvent::MaxUser值为65535
  - 但是这个type值不方便维护，用事件注册函数QEvent::registerEventType()会更加合理
    ```
    static int QEvent::registerEventType ( int hint = -1 );
    ```
    传入一个整型的hint，函数将尝试用这个数值去注册一个自定义事件，如果成功则返回这个数值，否则分配另一个合法的数值并返回
- 发送事件（两种方法）
  - `static bool QCoreApplication::sendEvent(QObject *receiver, QEvent *event);`
    直接将事件event发送给receiver，使用的是QCoreApplicaton::notify()函数，其返回值即为事件处理函数的返回值，事件发送时不会销毁event对象，通常在栈上创建这个event对象，如——
    ```
    QMouseEvent event(QEvent::MouseButtonPress, pos, 0, 0, 0);
    QApplication::sendEvent(mainWindow, &event);
    ```

  - `static void QCoreApplication::postEvent(QObject *receiver, QEvent *event);`
    将事件event和接收者receiver追加到事件队列，函数立即返回；
    post之后事件队列将持有事件对象，因此此处的event对象需要创建在堆上，出队之后才会被delete
    当控制权返回主线程后，所有事件会通过notify()发送出去，一般会按入队顺序进行处理，如果需要改变顺序，可以借助Qt::NormalEventPriority指定优先级
  - 其他发送方式：QCoreApplication::sendPostedEvents()、processEvent()
- 处理事件
  与内置事件的处理方式相同