第5章-事件

author：岳石磊 copyright：科林明伦 内部资料禁止外泄

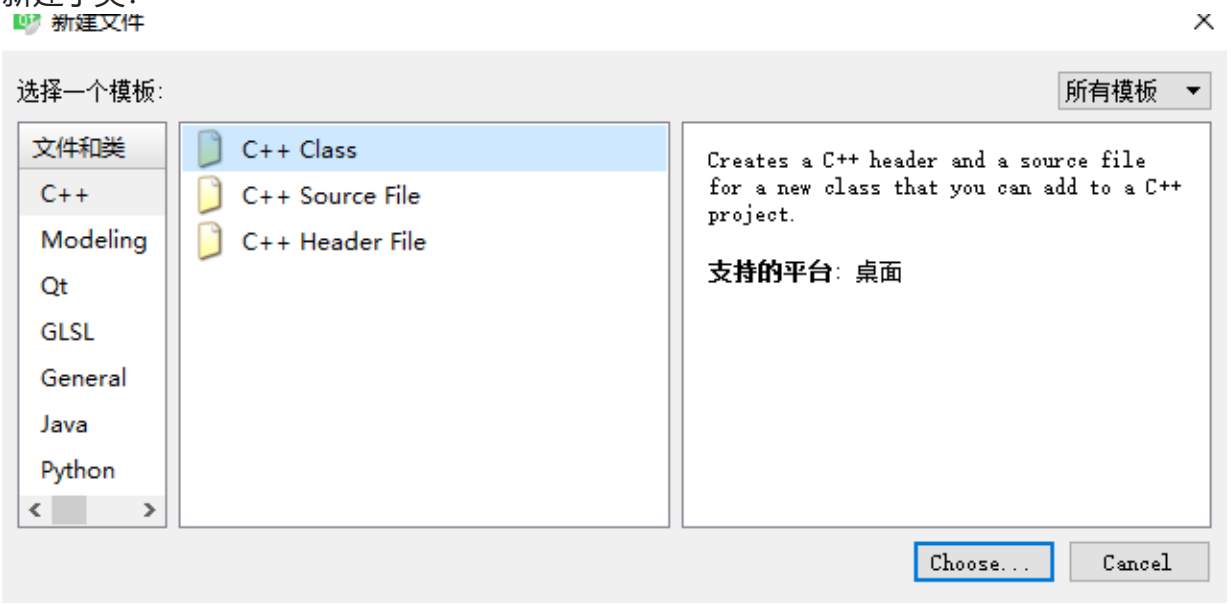
1. 事件

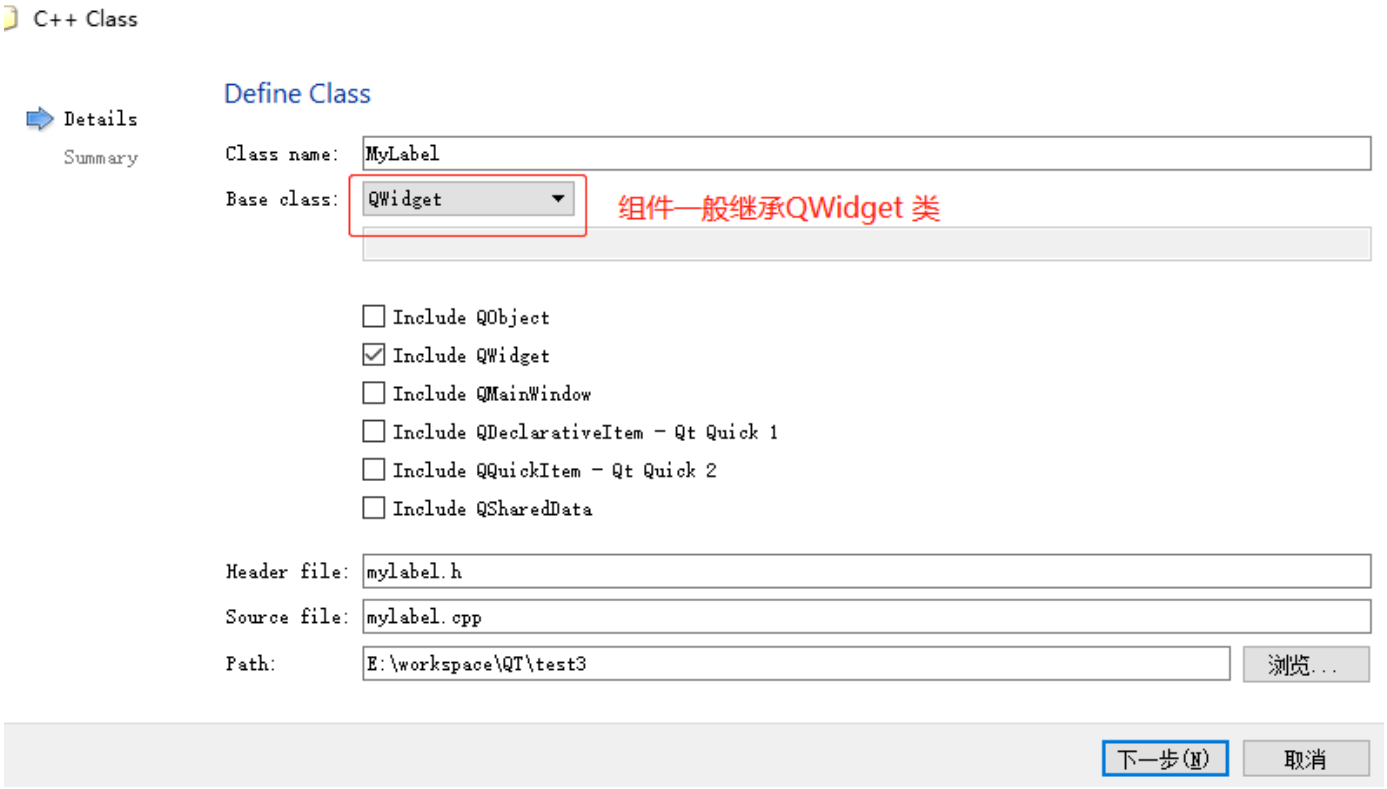
事件(event)是由系统或 Qt 本身在不同的时刻发出的。比如，当用户按下鼠标，敲下键盘，或窗口需要重新绘制的时候，都会发出一个相应的事件。一些事件是在对用户操作做出 响应的时候发 出，如键盘事件等；另一些事件则是由系统自动发出，如定时器事件。

Qt 的事件和信号槽很容易混淆，事件其实也就是所谓的事件驱动，`signal` 由具体对象发出，然后会马上交给由 `connect` 函数连接的 `slot` 进行处理。而对于事件，Qt 使用一个**事件队列(windowSystemEventQueue)**对所有发出的事件进行维护，当新的事件产生时，会被追加到事件队列的尾部，前一个事件完成后，取出后面的事件进行处理。但是，必要的时候，Qt的事件也是可以不进入事件队列，而是直接处理的。
如果我们使用组件，我们关心的是信号槽；如果我们自定义组件，我们关心的是事件。

2. 重写事件

一般我们重写某个组件的事件，需要自定义类，继承对应的组件类，重写感兴趣的事件。
新建子类：




C++ Class
Details
Summary
Define Class
Class name:
MyLabel
Base class:
QWidget
组件一般继承QWidget 类
Include QObject
Include QWidget
Include QMainWindow
Include QDeclarativeItem - Qt Quick 1
Include QQuickItem - Qt Quick 2
Include QSharedData
Header file:
mylabel.h
Source file:
mylabel.cpp
Path:
E:\workspace\QT\test3
浏览...
下一步(N)
取消

改成父类 QLabel

```h
#ifndef MYLABEL_H
#define MYLABEL_H

//#include <QWidget>
#include<QLabel>

class MyLabel : public QLabel
{
    Q_OBJECT
public:
    explicit MyLabel(QWidget *parent = 0);

signals:

public slots:
};

#endif // MYLABEL_H
```

```cpp
#include "mylabel.h"

MyLabel::MyLabel(QWidget *parent) : QLabel(parent)
{

}
```

想知道有哪些事件，我们需要转到父类中，模糊搜索event，事件多为虚函数，供我们重写，定义自己的实现规则。

```
protected:
    bool event(QEvent *e) Q_DECL_OVERRIDE;
    void keyPressEvent(QKeyEvent *ev) Q_DECL_OVERRIDE;
    void paintEvent(QPaintEvent *) Q_DECL_OVERRIDE;
    void changeEvent(QEvent *) Q_DECL_OVERRIDE;
    void mousePressEvent(QMouseEvent *ev) Q_DECL_OVERRIDE;
    void mouseMoveEvent(QMouseEvent *ev) Q_DECL_OVERRIDE;
    void mouseReleaseEvent(QMouseEvent *ev) Q_DECL_OVERRIDE;
    void contextMenuEvent(QContextMenuEvent *ev) Q_DECL_OVERRIDE;
    void focusInEvent(QFocusEvent *ev) Q_DECL_OVERRIDE;
    void focusOutEvent(QFocusEvent *ev) Q_DECL_OVERRIDE;
    bool focusNextPrevChild(bool next) Q_DECL_OVERRIDE;
```

我们关注鼠标相关的事件：分别将 `mousePressEvent`, `mouseMoveEvent`,`mouseReleaseEvent` 三个虚函数(在qt中表现为斜体)重写。

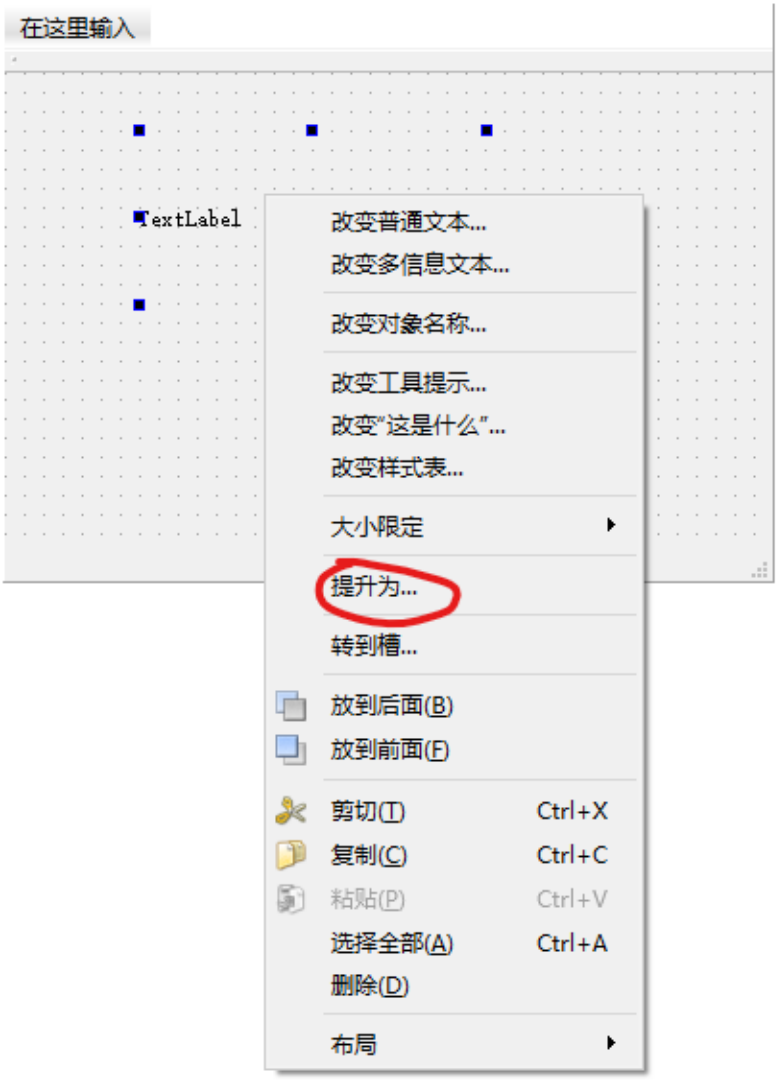
```
class MyLabel : public QLabel
{
    Q_OBJECT
public:
    explicit MyLabel(QWidget *parent = 0);

signals:

public slots:

public:
    void mousePressEvent(QMouseEvent *ev) Q_DECL_OVERRIDE;
    void mouseMoveEvent(QMouseEvent *ev) Q_DECL_OVERRIDE;
    void mouseReleaseEvent(QMouseEvent *ev) Q_DECL_OVERRIDE;
};
```

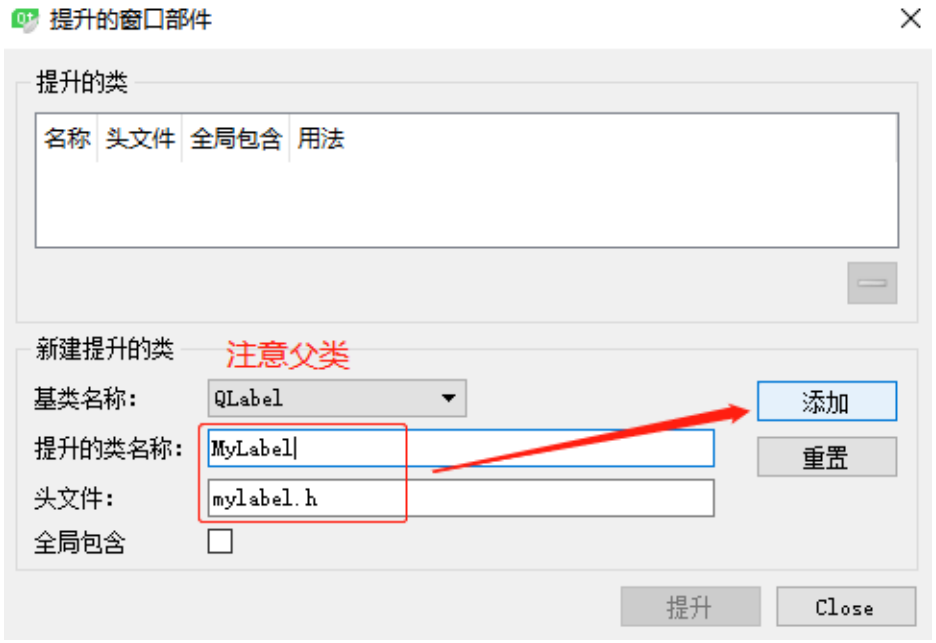
可以在函数的声明后面 加上宏 `Q_DECL_OVERRIDE` 或关键字 `override` 进行校验当前的虚函数是否为**重写父类**的，如果不是则会报错。
在重写的三个虚函数中，我们需要跟踪鼠标左键在Label组件的状态。

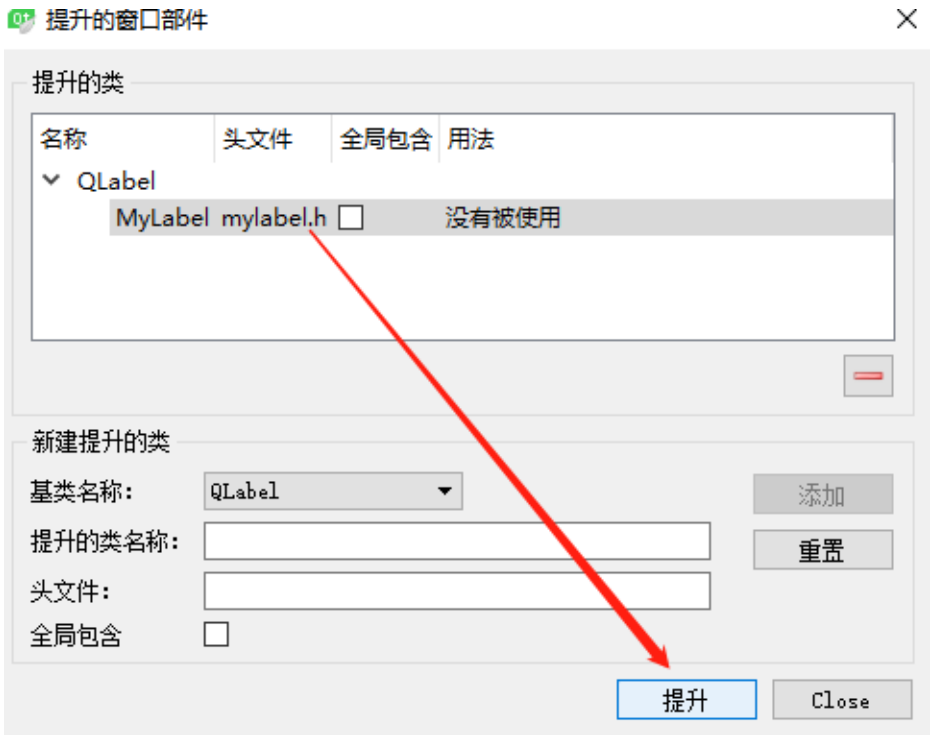
```
void MyLabel::mousePressEvent(QMouseEvent *ev){
    if( ev->button()  == Qt::LeftButton ){  //如果是鼠标左键按下:
        //获取按下的点的坐标
        //ev->x();   ev->y();
        QString str = "鼠标左键按下: ";
        str+=QString::number(ev->x())+","+QString::number(ev->y());
        this->setText(str);  //将文字显示到label组件上
    }
}
void MyLabel::mouseMoveEvent(QMouseEvent *ev){
```

```
11      //button: 触发当前事件的按钮（mouseMoveEvent 中返回的永远是NoButton）
12      //buttons:  当前事件发生时，有哪些按钮是按下的状态
13      if(ev->buttons() == Qt::LeftButton){
14          QString str = "鼠标左键按下移动：";
15          str+=QString::number(ev->x())+","+QString::number(ev->y());
16          this->setText(str);
17      }else{
18          this->setText(QString("没有按下做鼠标左键移动：")+QString::number
    (ev->x())+","+QString::number(ev->y()));
19      }
20  }
21  void MyLabel::mouseReleaseEvent(QMouseEvent *ev){
22      if(ev->button() == Qt::LeftButton){
23          QString str = "鼠标左键抬起：";
24          str+=QString::number(ev->x())+","+QString::number(ev->y());
25          this->setText(str);
26      }
27  }
```

自定义的MyLabel 类 与主窗口上的Label 组件绑定。

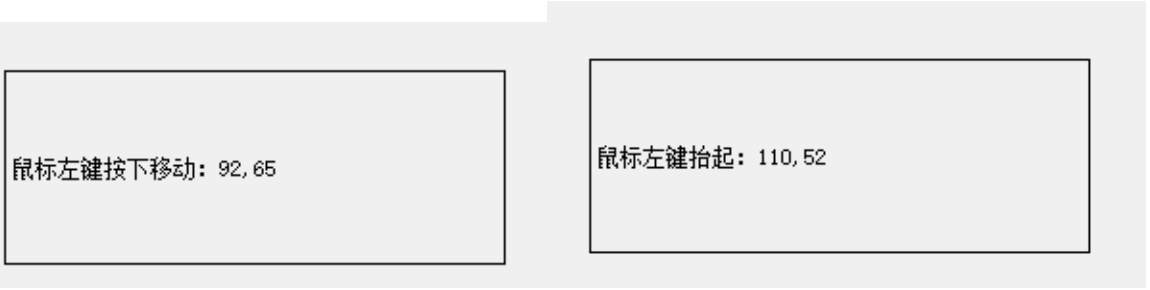






重构后 `ui_mainwindow.h` 中变为我们自定义的类。

```
#include "mylabel.h"

QT_BEGIN_NAMESPACE

class Ui_MainWindow
{
public:
    QWidget *centralWidget;
    MyLabel *label;
    QMenuBar *menuBar;
    QToolBar *mainToolBar;
```



当我们并未按下鼠标任何按键移动的时候，并未有文字进行提示。这说明当直接移动时并未追踪鼠标，在此情况下至少有一个鼠标按键按下时才会响应鼠标移动事件，我们使用函数`setMouseTracking`来决定是否追踪鼠标，默认为`false`不追踪，设置为`true`追踪鼠标。
在 MyLabel的构造函数中，设置追踪鼠标：

```
MyLabel::MyLabel(QWidget *parent) : QLabel(parent)
{
    this->setMouseTracking(true);  //设置默认追踪鼠标
}
```

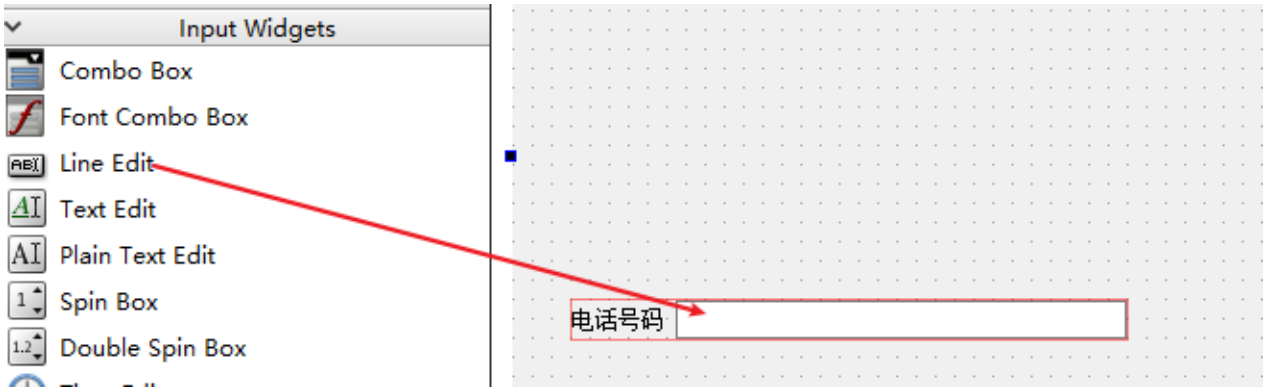
3. 事件分发

事件对象创建完毕后，Qt将这个事件传递给QObject::event() 函数，event()函数主要用于事件的分发，一般情况下并不直接处理事件，而是将这些事件对象按照它们不同的类型，分发给不同的事件处理器（event handler）。
如果想在事件分发之前做一些额外的操作或屏蔽掉某些事件，我们也可以重写event()函数。通过

event->type() 来确定事件的类型，事件的类型在 QEvent 类中定义的枚举：

```
class Q_CORE_EXPORT QEvent                  // event ba
{
    Q_GADGET
    QDOC_PROPERTY(bool accepted READ isAccepted
public:
    enum Type {
        /*
          If you get a strange compiler error on
          it's probably because you're also incl
          which #define the symbol None.  Put the
          the Qt includes to solve this problem.
        */
        None = 0,
        Timer = 1,
        MouseButtonPress = 2,
        MouseButtonRelease = 3,
        MouseButtonDblClick = 4,
        MouseMove = 5,
        KeyPress = 6,
        KeyRelease = 7,
        FocusIn = 8,
        FocusOut = 9,
        FocusAboutToChange = 23,
        Enter = 10,
```

举例：在窗口添加组件 【Line Edit】我们约定其为电话号码，只能输入数字且最多可输入11位。



“只能输入数字”这个约束条件，我们在事件分发中去做，

```
//事件分发
bool MyLineEdit::event(QEvent * ev){
    if(ev->type() == QEvent::KeyPress){  //如果事件的类型是一个键盘按下的类型
        QKeyEvent*pKey  = (QKeyEvent*)ev;   //将事件强转为具体的键盘事件类型
```

```
5
6           //pKey->key();  //获取按下的哪个键
7           if((Qt::Key_0<=pKey->key()  &&  pKey->key() <= Qt::Key_9)){  //
    如果按下的键是0~9
8               qDebug()<<"分发key = "<<pKey->key();
9               return QLineEdit::event(ev);  //正常分发
10          }else{
11              qDebug()<<"拦截key = "<<pKey->key();
12              return true;  //当前的事件已经处理了，  不需要继续分发了
13          }
14      }
15      return QLineEdit::event(ev);   //不关心的事件，正常去分发
16  }
```

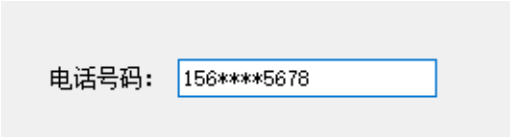
注意，event 函数 返回 `true` 代表我们已经对事件进行识别处理了，不再进行转发了，而会继续处理事件队列中的其他事件。
如果直接返回 `false`，那么代表当前类放弃事件的转发了，但事件并未得到处理，所以一般情况下我们不要直接返回 `false`，而是调用父类的 event 继续处理。

对于事件处理器，我们选择键盘按下事件`void keyPressEvent(QKeyEvent *event)`，增加一个用于存储电话号码的类成员属性，

```
class MyLineEdit : public QLineEdit
{
    Q_OBJECT
public:
    explicit MyLineEdit(QWidget *parent = 0);

signals:

public slots:

public:
    //事件分发函数
    bool event(QEvent *) Q_DECL_OVERRIDE;

    //event handler
    virtual void keyPressEvent(QKeyEvent *event) override;
public:
    QString m_tel;  //用于存储电话号码的
};
```

电话号码的展示我们做了一些处理，中间4位数字我们采用*号代替，达到一种保密的效果。

```
void MyLineEdit::keyPressEvent(QKeyEvent *event) {
    qDebug()<<"keyPressEvent=="<<event->key();
    if(m_tel.size()<= 10){
         m_tel+= QString::number(event->key()-Qt::Key_0);    //拼接字符串

         if(m_tel.size()<=3){
             this->setText(m_tel);
         }else if(3<m_tel.size()&&m_tel.size()<=7 ){
             QString tel = m_tel.left(3);
             for( int i=3;i< m_tel.size();i++ ){
                 tel+="*";
             }
             this->setText(tel);
         }else{
             QString tel = m_tel.left(3)+"****"+m_tel.right(m_tel.size()-7);
             this->setText(tel);
         }
     }
}
```




```
分发key =  49
keyPressEvent== 49
分发key =  53
keyPressEvent== 53
分发key =  54
keyPressEvent== 54
```

通过函数 key 获取并不是对应的数字，而是数字字符对应的ASCII码值，想要获取真正的值需要 event->key()-Qt::Key_0。

如果不小心输入错了，想要删除我们需要适当放开一些条件——Backspace 按键。

```
//分发
bool MyLineEdit::event(QEvent * ev){
    if(ev->type() == QEvent::KeyPress){  //如果事件的类型是一个键盘按下的类型
        QKeyEvent*pKey  = (QKeyEvent*)ev;   //将事件强转为具体的键盘事件类型
        //pKey->key();  //获取按下的哪个键

        if((Qt::Key_0<=pKey->key()  &&  pKey->key() <= Qt::Key_9)||
                pKey->key()==Qt::Key_Backspace){  //如果按下的键是0~9 或是 退格键

            return QLineEdit::event(ev);  //正常分发
        }else{
            qDebug()<<"key=="<<pKey->key();
            return true;  //当前的事件已经处理了，  不需要继续分发了
        }
    }
    return QLineEdit::event(ev);   //不关心的事件，正常去分发
}
```

删除电话号码中最后一位，通过字符串的截取来实现。

```
//event handler
void MyLineEdit::keyPressEvent(QKeyEvent *event) {
    qDebug()<<"keyPressEvent=="<<event->key();
    if(event->key()== Qt::Key_Backspace){
        m_tel = m_tel.left(m_tel.size()-1);   //左边截取n-1 位
        this->setText(this->text().left(this->text().size()-1));
    }else if(m_tel.size()<= 10  ){
        m_tel+= QString::number(event->key()-Qt::Key_0);   //拼接字符串
        if(m_tel.size()<=3){
            this->setText(m_tel);
        }else if(3<m_tel.size()&&m_tel.size()<=7 ){
            QString tel = m_tel.left(3);
            for( int i=3;i< m_tel.size();i++ ){
                tel+="*";
            }
            this->setText(tel);
        }else{
            QString tel = m_tel.left(3)+"****"+m_tel.right(m_tel.size()-7);
            this->setText(tel);
        }
    }
}
```

最后想查看完整的电话号码，可以在分发时做一些简单的处理：

```
//分发
bool MyLineEdit::event(QEvent * ev){
    if(ev->type() == QEvent::KeyPress){  //如果事件的类型是一个键盘按下的类型
        QKeyEvent*pKey  = (QKeyEvent*)ev;   //将事件强转为具体的键盘事件类型

        //pKey->key();  //获取按下的哪个键
        if( (Qt::Key_0<=pKey->key()  &&  pKey->key() <= Qt::Key_9)||
                pKey->key()==Qt::Key_Backspace){  //如果按下的键是0~9
            qDebug()<<"分发key = "<<pKey->key();
            return QLineEdit::event(ev);  //正常分发
        }else{
            qDebug()<<"拦截key = "<<pKey->key();
            if(pKey->key()==Qt::Key_Return){
                QMessageBox::information(this,"电话号码",m_tel);
            }
            return true;  //当前的事件已经处理了，  不需要继续分发了
        }
    }
    return QLineEdit::event(ev);   //不关心的事件，正常去分发
}
```

对于回车来说有两个，Key_Return：表示字母区的回车（shift键上面的那个），Key_Enter：表示数字小键盘中的回车。

```
enum Key {
    Key_Escape = 0x01000000,                // misc
    Key_Tab = 0x01000001,
    Key_Backtab = 0x01000002,
    Key_Backspace = 0x01000003,
    Key_Return = 0x01000004,
    Key_Enter = 0x01000005,
    Key_Insert = 0x01000006
```

4.事件过滤

event()函数是一个 protected 的函数，这意味着我们要想重写 event()，必须继承一个已有的组件类，一一重写其 event()函数。event()函数的确有一定的控制，不过有时候我的需求更严格一些：我希望那些组件根本看不到这种事件。event()函数虽然可以拦截，但其实也是接收到了事件。我连让它收都收不到。这样做的好处是，模拟一种系统根本没有那个事件的效果，所以其它组件根本不会收到这个事件，也就无需修改自己的事件处理函数。所以我们可以使用事件过滤器，事件过滤器给我们一种能力，让我们能够完全移除某种事件。事件过滤器可以安装到任意 QObject 类型上面，并且可以安装多个。

我们需要用到2个函数：
QObject::installEventFilter：**安装过滤器**

```
1 | void installEventFilter(QObject *filterObj)
```

filterObj：监控者， 包含 eventFilter 事件过滤器的对象，当 this 发生事件时，会先执行 filterObj 对象中的过滤器，再分发事件。

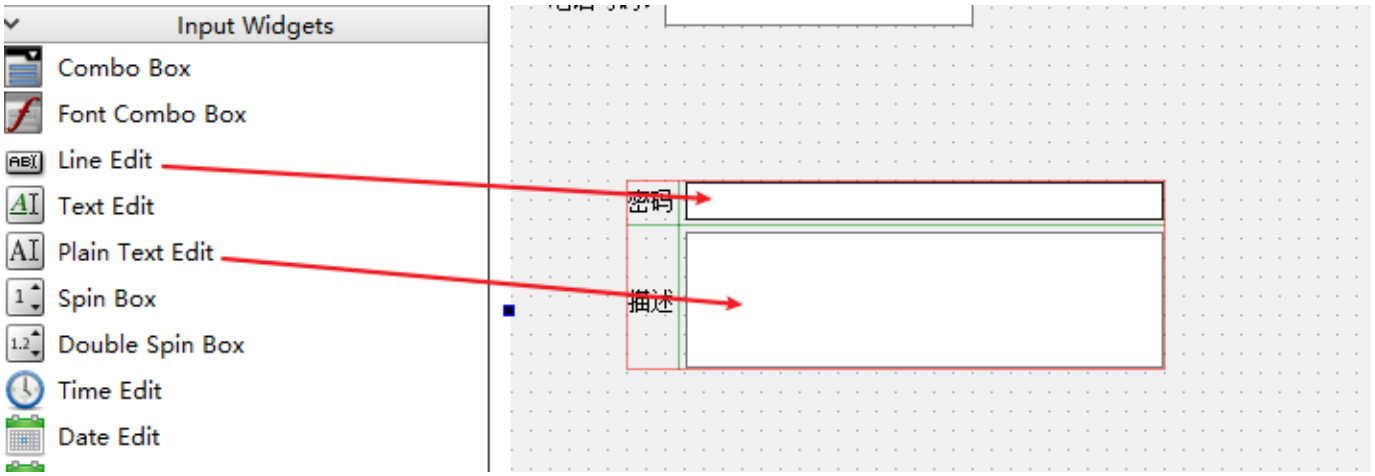
QObject::eventFilter：**过滤器函数**

```
1 | virtual bool eventFilter(QObject *watched, QEvent *event);
```

watched 被过滤器监视的对象，event： 发生的事件，当watched对象发生事件时，会先调用过滤，在进行event()分发。
返回 true 代表，拦截成功，事件将不会再继续传递。
返回 false 代表，放行。

函数执行顺序： eventFilter -> event -> event Handler。

例子：在窗口界面上添加【line Edit】和【Plain Text Edit】组件并label 约定为密码和描述。
针对于密码输入组件，我们规定只能输入"数字"和"字母"，对于描述的内容，我们约定滑轮中键按下且滚动时，为放大或缩小文本。



对于上述要求，我们使用事件的重写，分发完全可以做到，但是需要我们自定义类、继承组件类并提升。如果界面的组件有很多每一个都需要自定义组件将导致增加很多类，带来代码管理上的麻烦。

首先被监控的组件需要“主动”安装过滤器，将监控的权利交给其他人。在主窗口的构造函数中调用 `installEventFilter`。

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    //单行文本输入框，安装事件过滤，由主窗口（this）监控
    ui->lineEdit_pass->installEventFilter(this);
    //多行文本输入框，安装事件过滤，由主窗口（this）监控
    ui->plainTextEdit->installEventFilter(this);
```

这样对于这两个组件发生的所有事件都会交由父窗口处理，我们需要一个统一的处理函数 eventFilter，虚函数需要我们重写，

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
public:
    virtual bool eventFilter(QObject *watched, QEvent *event);
```

```
1  //监控组件后的处理函数
2  // watched:  哪一个组件发生了事件 , event: 当前组件发生了什么事件
3  bool MainWindow::eventFilter(QObject *watched, QEvent *event){
4      if(watched == ui->lineEdit_pass){  //如果是 密码输入框组件发生了事件
5          if( event->type() == QEvent::KeyPress){  //如果发生的事件时键盘按
   下的事件
6              QKeyEvent* pKey = (QKeyEvent*)event;  //强转为键盘的事件
7
8              if(  (Qt::Key_0 <= pKey->key()  && pKey->key()<=Qt::Key_9)
   ||
9                   (Qt::Key_A <= pKey->key()  && pKey->key()<=Qt::Key_Z))
   {  //如果是数字和字母
10                 //return false;  //不过滤，放行
11                 return QMainWindow::eventFilter(watched,event);  //不过
   滤，放行
12             }else{
13                 qDebug()<<"key = "<<pKey->key();
14                 return true;   //过滤，不能继续走了
15             }
16         }
17     }else if(watched == ui->plainTextEdit){  //多行文本输入框
18         if( event->type() == QEvent::Wheel){  //如果是滑轮事件
```

```
            QWheelEvent* pWheel = (QWheelEvent*)event;  //强转为滑轮的事件
            if( pWheel->buttons()== Qt::MidButton){  //判断鼠标中键是否按下
                qDebug()<<"x = "<<pWheel->angleDelta().x();    //alt+滑轮上 =120   alt+滑轮下 =-120
                qDebug()<<"y = "<<pWheel->angleDelta().y();    //滑轮上 =120       滑轮下=-120

                if(pWheel->angleDelta().y()>0){  //滑轮为上滚动
                    ui->plainTextEdit->zoomIn();  //放大文本
                }else{
                    ui->plainTextEdit->zoomOut(); //缩小文本
                }
            }
        }
    }
    return QMainWindow::eventFilter(watched,event);  //其他类型的事件，我们放行
}
```

其中的120 = 8*15，滚轮每滚动1度，相当于移动了8度，而常见的滚轮鼠标拨动一下的滚动角度为15度。

对于lineEdit输入组件，默认是支持中文输入法的，这样就会输入除了数字和字母外的其他字符，我们可以使其禁用中文输入法。

```
//禁止中文输入法
ui->lineEdit_pass->setAttribute(Qt::WA_InputMethodEnabled,false);
```

5. 自定义事件

自定义事件类型(id) 在 User ~ MaxUser 之间。

```
//qcoreevent.h
QEvent::User    = 1000,                 // first user event id
QEvent::MaxUser = 65535                 // last user event id
```

自定义事件信息需要继承QEvent，才能被发送、转化、解析。

发送自定义事件 postEvent 和 sendEvent

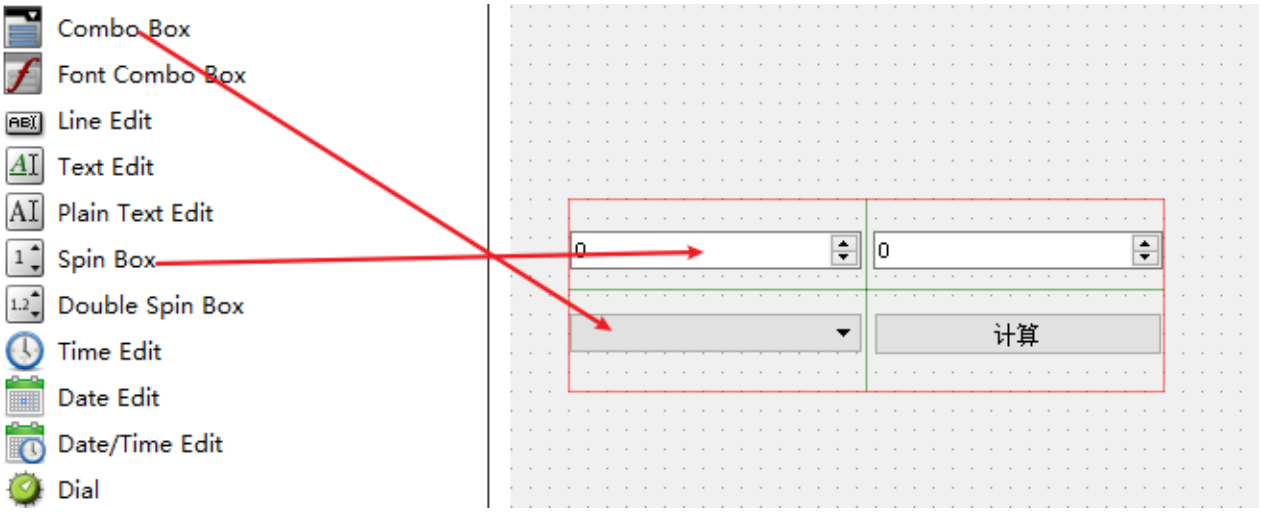
```
static bool QCoreApplication::sendEvent(
QObject *receiver,//事件的接收者（发送给谁）
QEvent *event); //发送一个什么事件

static void QCoreApplication::postEvent(QObject *receiver,
QEvent *event,
int priority = Qt::NormalEventPriority); //事件的优先级
```

sendEvent：事件直接调用对应的处理器，阻塞，事件多为栈区对象。
postEvent：进入事件队列，非阻塞，事件对象一般new出来。

自定义事件处理器：

```
virtual void QObject::customEvent(QEvent *event);
```

举例：在窗口界面上添加两个【Spin Box】用作计算的两个数，【Combo Box】作为计算的规则，【Push Button】将计算的结果以事件的形式发送出去。



准备工作：计算规则我们需要添加加、减、乘、除规则。在MainWindow 的构造函数中添加对应的Item

```
#define ADD    "+"
#define SUB    "-"
#define MUL    "*"
#define DIV    "/"
```

```
    QStringList strList;
    strList.push_back(ADD);
    strList.push_back(SUB);
    strList.push_back(MUL);
    strList.push_back(DIV);
    ui->comboBox->addItems(strList);    //添加下拉框数据

    QObject::connect(ui->pushButton,SIGNAL(clicked()),this,SLOT(slot_PushButton()));
```

计算按钮，我们添加自定义的槽函数 slot_PushButton，用来发送点击时的自定义事件。

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    void slot_PushButton();
```

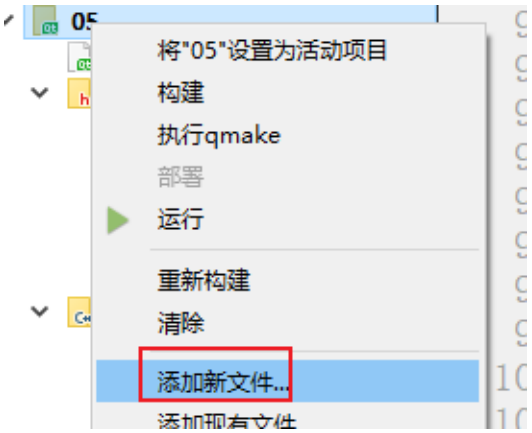
在槽函数的实现中，我们获取需要计算的两个数字 和 计算规则，

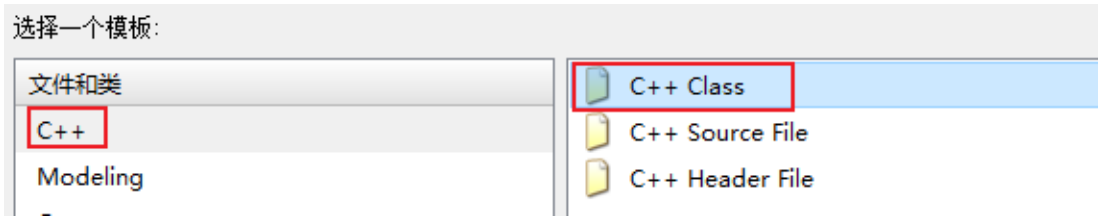
```
// 点击计算按钮
void MainWindow::slot_PushButton(){

    //获取数字输出框上的值
    int num1 = ui->spinBox->value();
    int num2 = ui->spinBox_2->value();

    int ret = 0;

    //获取计算规则
    QString strRule = ui->comboBox->currentText();
    if(strRule==ADD){
        ret = num1+num2;
    }else if(strRule == SUB){
        ret = num1-num2;
    }else if(strRule == MUL){
        ret = num1*num2;
    }else if(strRule == DIV){
        ret = num1/num2;
    }
```

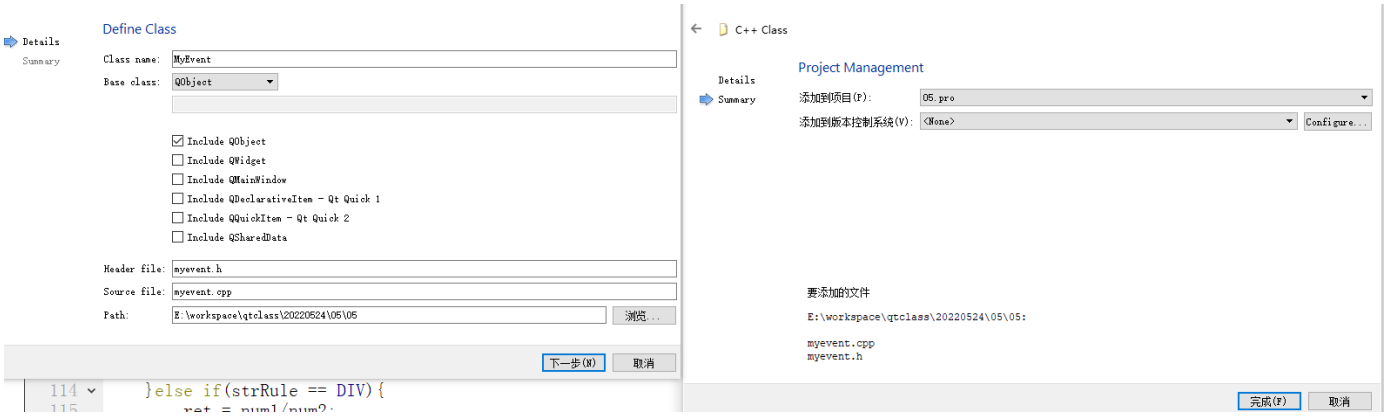
准备事件：首先确定好自定义事件的类型（id），

```
1  //自定义事件的ID  User~MaxUser
2  QEvent::Type  typeId  =  QEvent::User;
```

自定义事件类MyEvent







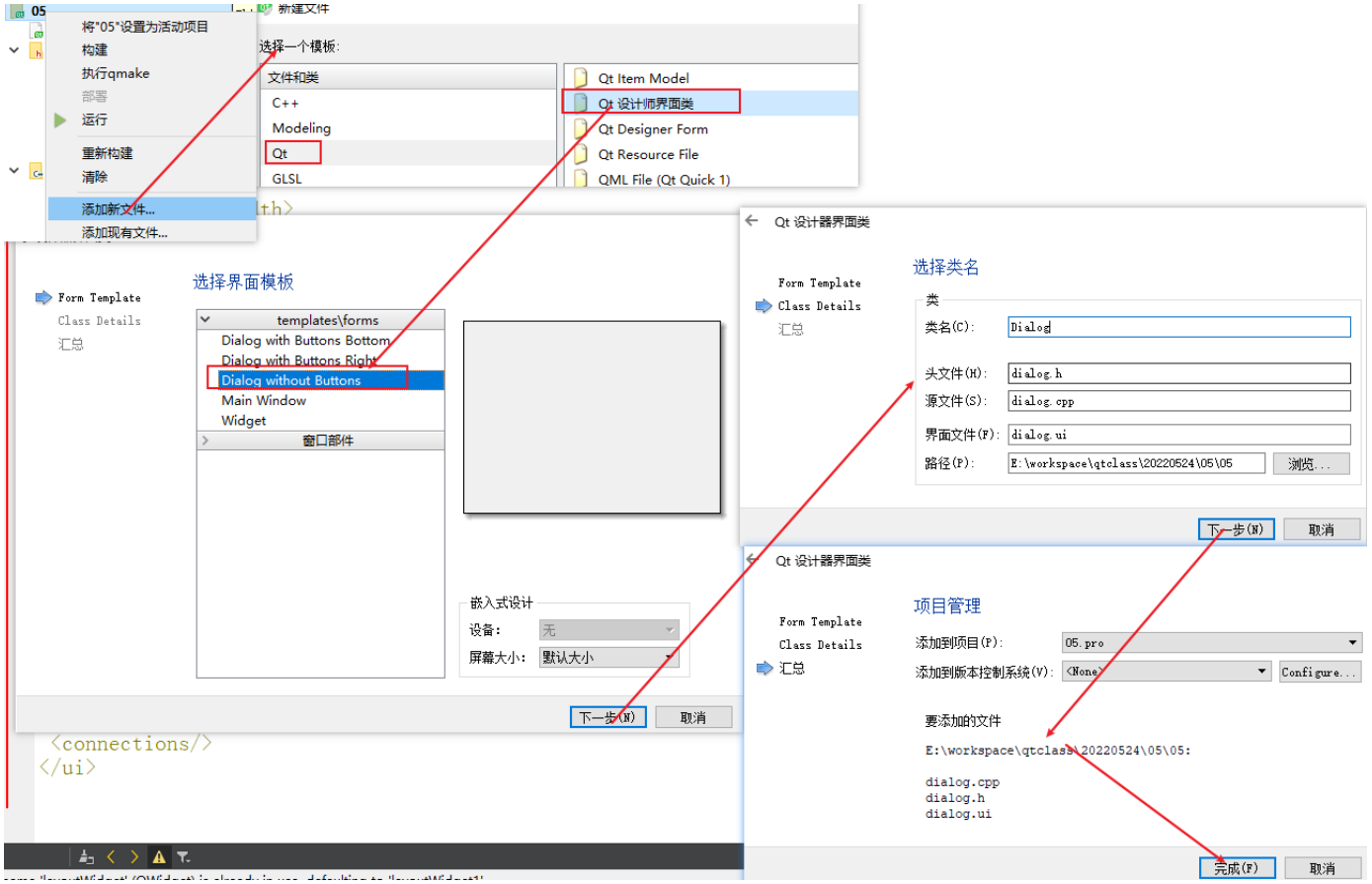
继承父类修改为 QEvent，构造函数参数改为事件的类型，添加事件携带的信息

int ret; //计算的结果 QString strRule; //计算的规则

```
#ifndef MYEVENT_H
#define MYEVENT_H

#include <QEvent>
#include<QString>

class MyEvent : public QEvent
{
public:
    explicit MyEvent(Type type);

    int ret;
    QString strRule;
};

#endif // MYEVENT_H
```

```
#include "myevent.h"

MyEvent::MyEvent(Type type) : QEvent(type)
{

}
```

我们将事件发送给你另一个窗口，所以新建Dialog窗口



在dialog 中我们需要定义接收自定义事件的处理函数：

`void customEvent(QEvent *event)` 虚函数，需要我们重写。

```
class Dialog : public QDialog
{
    Q_OBJECT

public:
    explicit Dialog(QWidget *parent = 0);
    ~Dialog();

private:
    Ui::Dialog *ui;

public:
    virtual void customEvent(QEvent *event);
```

```
void Dialog::customEvent(QEvent *event){
    extern QEvent::Type  typeId;
    if( event->type() ==  typeId){  //如果是我的这个typeId自定义事件

        MyEvent*pEvent = (MyEvent*)event;
        //将事件携带的信息显示到窗口label标签上
        ui->label->setText(QString("规则是：")+pEvent->strRule+",结果为："+QString::number(pEvent->ret));
        this->show();  //显示窗口
    }
    qDebug()<<"customEvent";
}
```

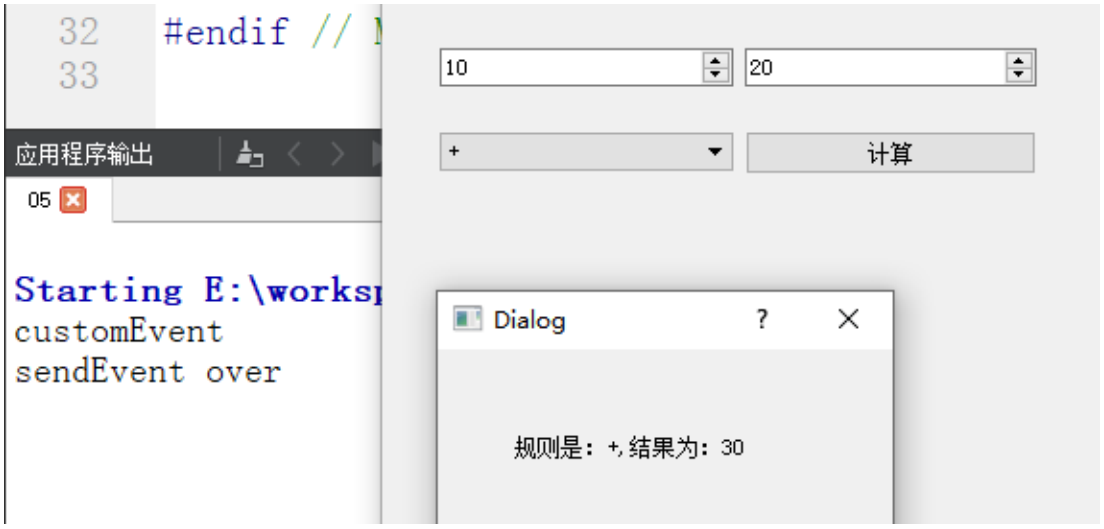
在主函数中定义对象，并添加全局的指针。至此接收者准备完毕。

```
#include"dialog.h"

Dialog *pDia = nullptr; //方便其他源文件中使用


int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    Dialog dia;    //另一个窗口（接收自定义事件）
    pDia = &dia;


    return a.exec();
}
```

使用sendEvent 发送自定义事件。

```
// 点击计算按钮
void MainWindow::slot_PushButton(){
    //获取数字输出框上的值
    int num1 = ui->spinBox->value();
    int num2 = ui->spinBox_2->value();

    int ret = 0;

    //获取计算规则
    QString strRule = ui->comboBox->currentText();
    if(strRule==ADD){
        ret = num1+num2;
    }else if(strRule == SUB){
        ret = num1-num2;
    }else if(strRule == MUL){
        ret = num1*num2;
    }else if(strRule == DIV){
        ret = num1/num2;
    }

    //定义自定义事件的对象
    MyEvent event(typeId);
    event.ret = ret;
    event.strRule = strRule;

    extern Dialog *pDia;

    //发送，是一个阻塞的发送事件，直到接收方接收到事件且处理完毕，sendEvent 才会返回，类似于打电话
    QCoreApplication::sendEvent(pDia,&event);
    qDebug()<<"sendEvent over";
```

看输出结果发现，sendEvent阻塞发送事件，直到接收方接收到事件且处理完毕，sendEvent 才会返回。



除此之外还可以使用postEvent 发送

```
extern Dialog *pDia;

MyEvent* pEvent = new MyEvent(typeId);  //new 堆区
pEvent->ret = ret;
pEvent->strRule = strRule;
//发送自定义事件,非阻塞的函数，发送完事件立即返回，执行下面的代码，类似于发短信（要求发送事件new堆区）
QCoreApplication::postEvent(pDia,pEvent);
qDebug()<<"postEvent over";
```

从输出结果看，postEvent非阻塞的函数，发送完事件立即返回。