



Qt部件与事件处理





课程主要内容



- ◆ 用户界面组件介绍
- ◆ 部件的布局管理
- ◆ 通用部件
- ◆ 部件的尺寸策略
- ◆ **Qt Designer**
- ◆ 顶层窗体
- ◆ **Qt图标**
- ◆ **Qt事件处理**



用户界面组件介绍



用户界面组件



- 用户界面由单个部件（**widget**）构成





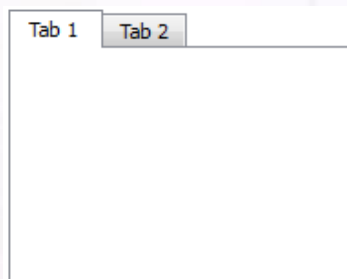
部件中的部件



- 部件被分层次放置



QGroupBox



QTabWidget

- 容器类提供可视化结构，但同时也是具有一定功能化的
 - 如 **QRadioButton**，需要用彼此间实现互斥，可以将多个**QRadioButton**放到一个**GroupBox**中



部件的特点



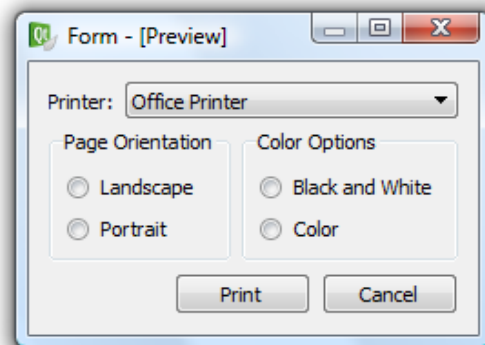
- 占据屏幕中一个方形的区域
- 从输入设备接收事件
- 当部件产生变化时，发出信号
- 多个部件以层次式的方法组合构建
- 一个部件中可以包含其他部件



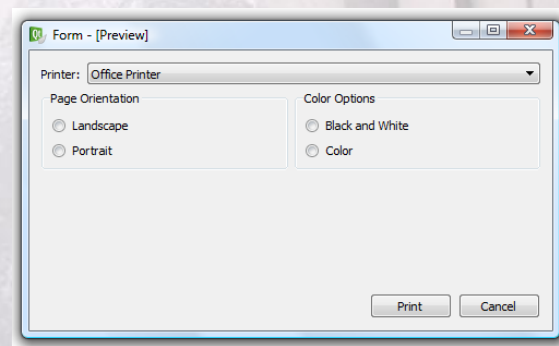
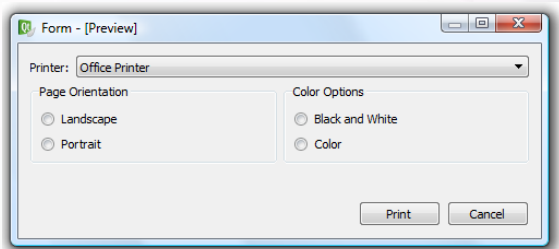
部件的布局管理



一个对话框例子



- 部件被放置在布局管理器中—使用户界面具有弹性易伸缩





弹性好在哪里？



- 让部件的大小适应内容

```
\home\john\Documents\Work\Project  
\home\john\Documents\Work\Project  
\home\john\Documents\Work\Project  
\home\john\Documents\Work\Project
```

```
\home\john\Documents\Work\Projects\Base  
\home\john\Documents\Work\Projects\Brainstorming  
\home\john\Documents\Work\Projects\Design  
\home\john\Documents\Work\Projects\Hardware
```

- 让部件适应翻译变化

News
Nyheter

Nyheter

- 让部件适应用户设置，如字体设置等

News

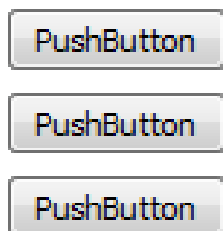
News



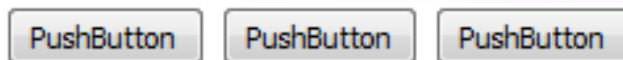
布局管理



- 几种可用的布局



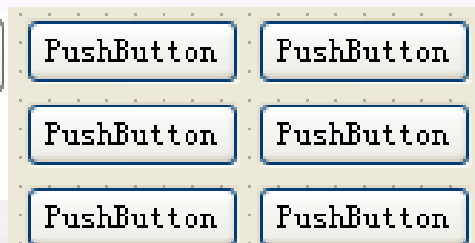
QVBoxLayout



QHBoxLayout



QGridLayout



QFormLayout

- 布局管理器和部件“协商”各个部件大小与位置
- 弹簧可以用来填充空白处

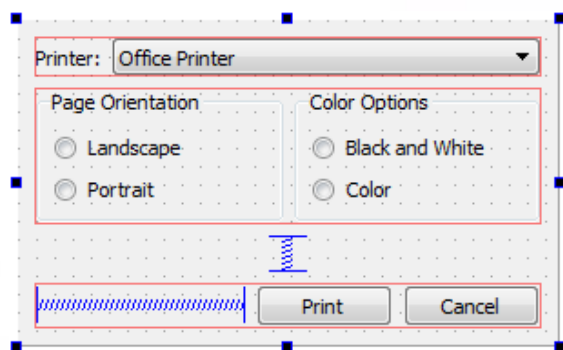




一个对话框例子



- 对话框由多个层次的布局管理器和部件组成



注意：布局管理器并不是其管理的部件的父对象

Object	Class
Form	QWidget
horizontalLayout	QHBoxLayout
label	QLabel
printerBox	QComboBox
horizontalLayout_2	QHBoxLayout
cancelButton	QPushButton
horizontalSpacer	Spacer
printButton	QPushButton
horizontalLayout_3	QHBoxLayout
groupBox	QGroupBox
landscapeButton	QRadioButton
portraitButton	QRadioButton
groupBox_2	QGroupBox
bwButton	QRadioButton
colorButton	QRadioButton
verticalSpacer	Spacer



对话框例子



```
QVBoxLayout *outerLayout = new QVBoxLayout(this);
```

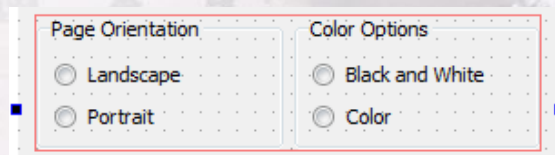
```
QHBoxLayout *topLayout = new QHBoxLayout();  
topLayout->addWidget(new QLabel("Printer:"));  
topLayout->addWidget(c=new QComboBox());  
outerLayout->addLayout(topLayout);
```



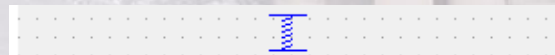
```
QHBoxLayout *groupLayout = new QHBoxLayout();
```

...

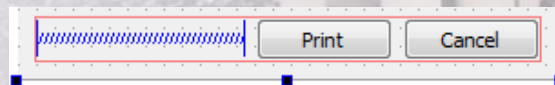
```
outerLayout->addLayout(groupLayout);
```



```
outerLayout->addSpacerItem(new QSpacerItem(...));
```



```
QHBoxLayout *buttonLayout = new QHBoxLayout();  
buttonLayout->addSpacerItem(new QSpacerItem(...));  
buttonLayout->addWidget(new QPushButton("Print"));  
buttonLayout->addWidget(new QPushButton("Cancel"));  
outerLayout->addLayout(buttonLayout);
```

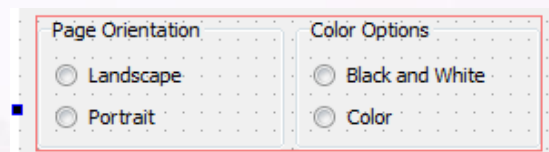




对话框例子

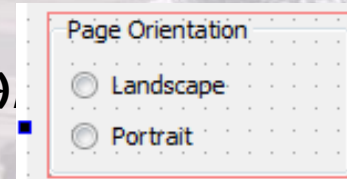


- Horizontal box, 包含 group boxes, vertical boxes, radio buttons

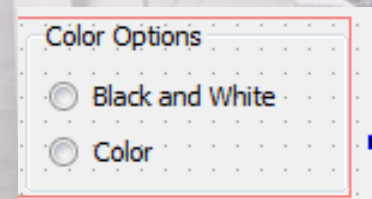


```
QHBoxLayout *groupLayout = new QHBoxLayout();
```

```
QGroupBox *orientationGroup = new QGroupBox();  
QVBoxLayout *orientationLayout = new QVBoxLayout(orientationGroup);  
orientationLayout->addWidget(new QRadioButton("Landscape"));  
orientationLayout->addWidget(new QRadioButton("Portrait"));  
groupLayout->addWidget(orientationGroup);
```



```
QGroupBox *colorGroup = new QGroupBox();  
QVBoxLayout *colorLayout = new QVBoxLayout(colorGroup);  
colorLayout->addWidget(new QRadioButton("Black and White"));  
colorLayout->addWidget(new QRadioButton("Color"));  
groupLayout->addWidget(colorGroup);
```





对话框例子



- 可以使用Qt设计器来建立同样的结构

Object	Class
Form	QWidget
horizontalLayout	QHBoxLayout
label	QLabel
printerBox	QComboBox
horizontalLayout_2	QHBoxLayout
cancelButton	QPushButton
horizontalSpacer	Spacer
printButton	QPushButton
horizontalLayout_3	QHBoxLayout
groupBox	QGroupBox
landscapeButton	QRadioButton
portraitButton	QRadioButton
groupBox_2	QGroupBox
bwButton	QRadioButton
colorButton	QRadioButton
verticalSpacer	Spacer



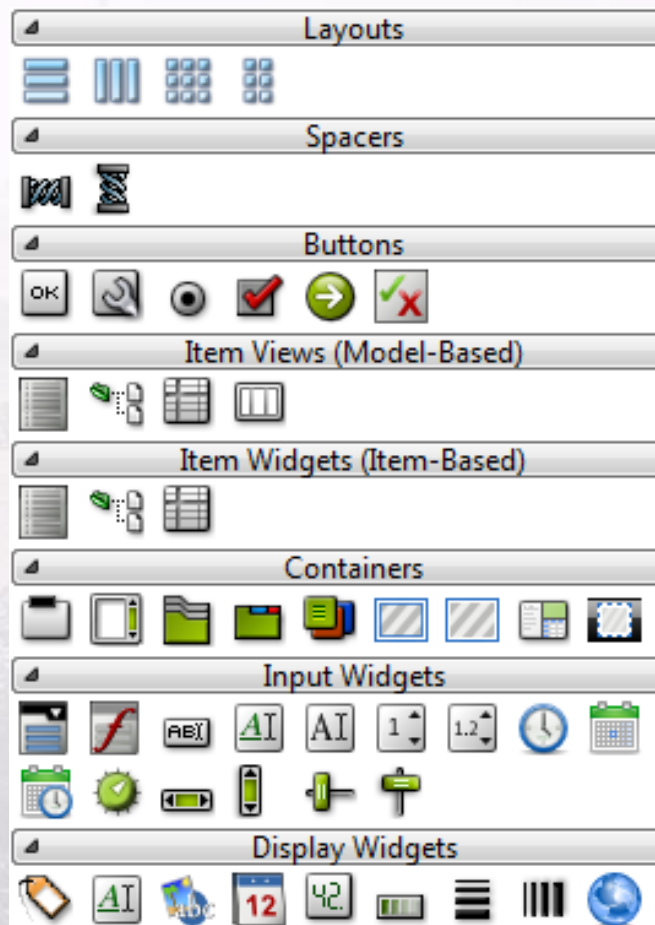
通用部件



通用部件



- Qt包含针对所有常见需求的大量通用部件
- Qt设计器中为部件组提供很好的概貌

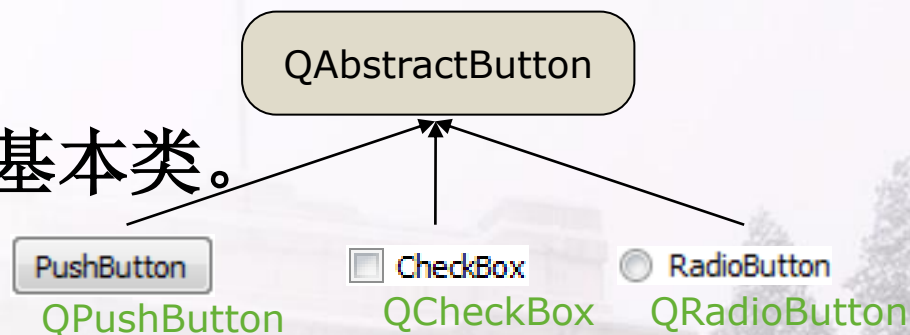




通用部件—按钮



- 所有按钮继承自 **QAbstractButton** 这个基本类。

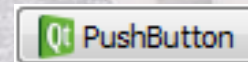


- 信号

- **clicked()** – 当按钮被按下（并弹起后）发出。
- **toggled(bool)** – 当按钮的状态发生改变时发出。

- 属性

- **checkable** – 当按钮可检查时为真。使按钮激活。
- **checked** – 当按钮被标记时为真。（用于复选或单选按钮）
- **text** – 按钮的文本。
- **icon** – 按钮的图标（可以和文本同时显示）。

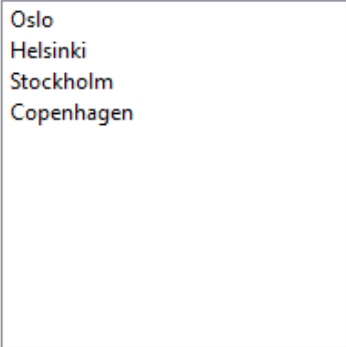




通用部件—列表项部件



- **QListWidget**用于显示列表项
- 添加项目
 - **addItem(QString)** – 将项目附加到列表末端
 - **insertItem(int row, QString)** – 将项目插入到指定行
- 选择项目
 - **selectedItems** – 返回**QListWidgetItem**的列表, 使用**QListWidgetItem::text**来形成文本
- 信号
 - **itemSelectionChanged** – 当选择状态改变时发出
- **QComboBox** 以更紧密的格式展示一个单选的项目列表。



Oslo
Helsinki
Stockholm
Copenhagen

QListWidget



QComboBox

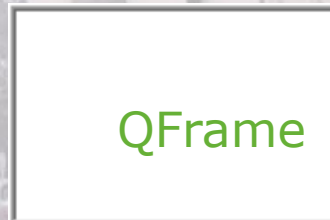
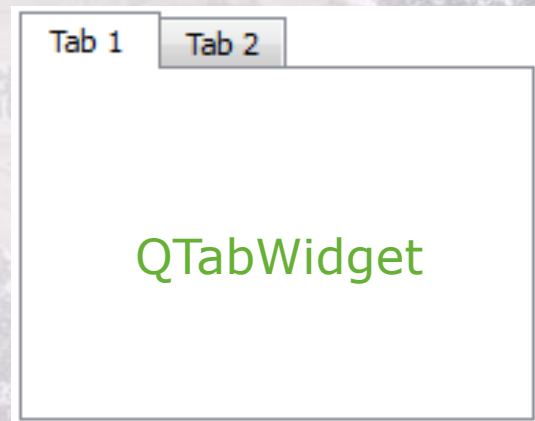


通用部件—容器



- 容器部件用来结构化用户界面
- 一个简单的 **QWidget** 对象可当做容器来使用
- 设计器：将部件放置在容器中并为容器提供一个布局管理器
- 代码：为容器创建一个布局管理器并将部件添加进布局管理器（布局管理器以容器为父对象）

```
QGroupBox *box = new QGroupBox();  
QVBoxLayout *layout = new QVBoxLayout(box);  
layout->addWidget(...);  
...
```





通用部件—输入部件



- 使用 **QLineEdit** 实现单行文本输入
- 信号
 - **textChanged(QString)** – 文本状态改变时发出
 - **editingFinished()** – 部件失去焦点时发出
 - **returnPressed()** – 回车键被按下时发出
- 属性
 - **text** – 部件的文本
 - **maxLength** – 限定输入的最大长度
 - **readOnly** – 设置为真时文本不可编辑（仍允许复制）

Hello World

QLineEdit



通用部件—输入部件



- 使用QTextEdit 和 QPlainTextEdit 实现多行文本输入

- Signals

- textChanged() -文本状态改变时发出

- 属性

- plainText – 无定义格式文本
 - html – HTML格式文本
 - readOnly – 设置为真时文本不可编辑



QTextEdit

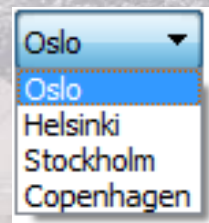
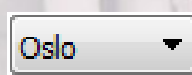
- QComboBox 通过editable属性使其可编辑

- Signals

- editTextChanged(QString) – 当文本正被编辑时发出

- 属性

- currentText – combo box的当前文本



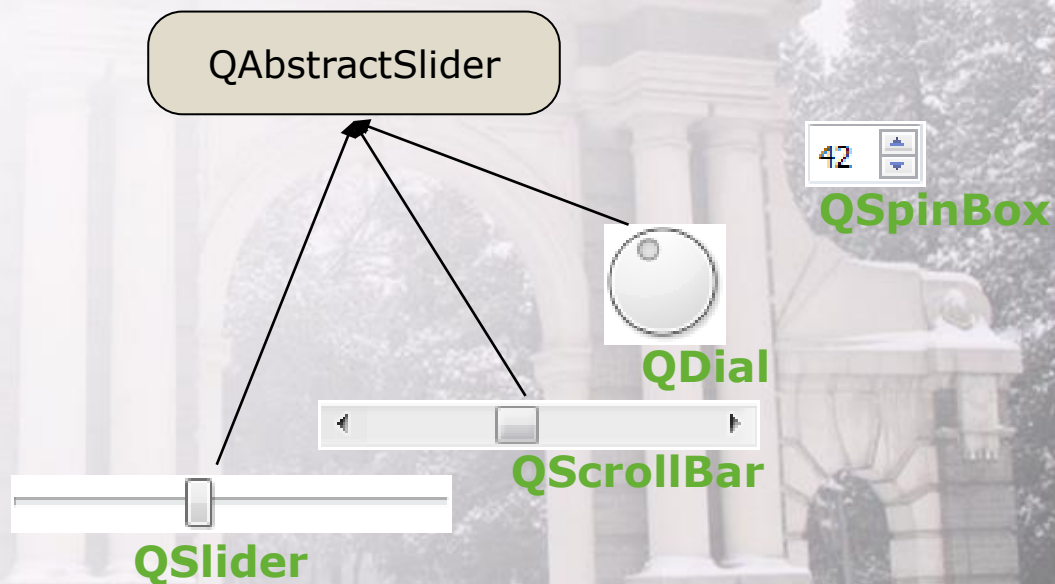
QComboBox



通用部件—输入部件



- 编辑整型数据有许多可选的输入部件
- 也有许多用于**double**，**time**和**date**类型的部件
 - 信号
 - **valueChanged(int)** – 当数值更新时发出
 - 属性
 - **value** – 当前值
 - **maximum** – 最大值
 - **minimum** – 最小值





通用部件—显示部件



- **QLabel** 部件显示文本或者图片

- 属性

- **text** – 标签文本
 - **pixmap** – 显示的图片

HelloWorld
QLabel



QLabel

- **QLCDNumber** 用于显示整形数值

- 属性

- **intValue** – 显示的数值(使用**display(int)**函数进行设置)



QLCDNumber



通用部件—属性



- 所有部件有一系列继承自**QWidget**类的共同属性

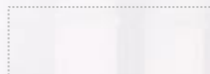
- **enabled** – 用户交互可用或不可用

PushButton

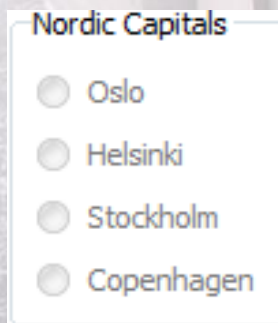
PushButton

- **visible** – 显示或不显示(**show** 或**hide**函数)

PushButton



- 这些属性同时影响到子部件
例如使一个容器部件不可用时:





QMessageBox



- ◆ 信息框是可以显示提示信息，并接受用户按钮输入的一种对话框
- ◆ 信息框使用方式一：静态函数
 - ⊕ **StandardButton QMessageBox::warning (QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton)**
 - ◆ **Parent:** 父组件指针
 - ◆ **Title:** 标题
 - ◆ **Text:** 提示文本
 - ◆ **Buttons:** 提示框中的按钮，可用或 (|) 运算添加多个按钮
 - ◆ **defaultButton:** 默认选中的按钮
 - ⊕ 类似函数还有**QMessageBox::information (...)**,
QMessageBox::critical(...), **QMessageBox::question(...)**,
QMessageBox::about(...), ...



QMessageBox



```
int ret = QMessageBox::warning(this, tr("My Application"), tr("The document  
has been modified.\n" "Do you want to save your changes?"),  
    QMessageBox::Save | QMessageBox::Discard | QMessageBox::Cancel,  
    QMessageBox::Save);
```

```
switch (ret) {  
    case QMessageBox::Save:  
        // Save was clicked  
        break;  
    case QMessageBox::Discard:  
        // Don't Save was clicked  
        break;  
    case QMessageBox::Cancel:  
        // Cancel was clicked  
        break;  
    default:  
        // should never be reached  
        break;  
}
```



QMessageBox



◆ 信息框使用方式二：构造函数

⊕ **QMessageBox::QMessageBox (Icon icon, const QString & title, const QString & text, StandardButtons buttons = NoButton, QWidget * parent = 0, Qt::WindowFlags f = Qt::Dialog | Qt::MSWindowsFixedSizeDialogHint)**

- ◆ **icon:** 图标，可取值为**MessageBox::NoIcon**, **QMessageBox::Question**, **QMessageBox::Information**, **QMessageBox::Warning**, **QMessageBox::Critical**
- ◆ **Title:** 标题
- ◆ **Text:** 提示文本
- ◆ **Buttons:** 提示框中的按钮，可用或（|）运算添加多个按钮
- ◆ **parent:** 父组件指针
- ◆ **F:** 窗口系统属性



QMessageBox



```
QMessageBox message(QMessageBox::NoIcon, tr("My Application"), tr("The  
document has been modified.\n" "Do you want to save your changes?"),  
QMessageBox::Save | QMessageBox::Discard | QMessageBox::Cancel);
```

```
switch (message.exec()) {  
    case QMessageBox::Save:  
        // Save was clicked  
        break;  
    case QMessageBox::Discard:  
        // Don't Save was clicked  
        break;  
    case QMessageBox::Cancel:  
        // Cancel was clicked  
        break;  
    default:  
        // should never be reached  
        break;  
}
```




部件的尺寸策略



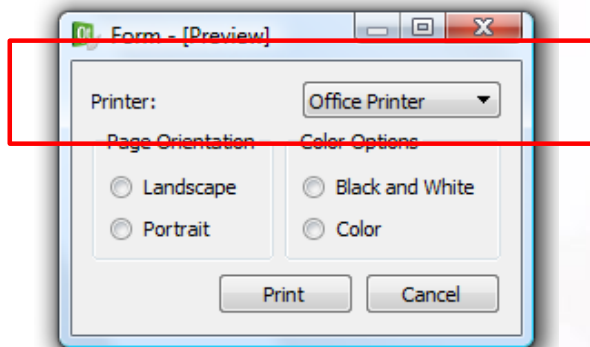
尺寸 (**size**) 的策略



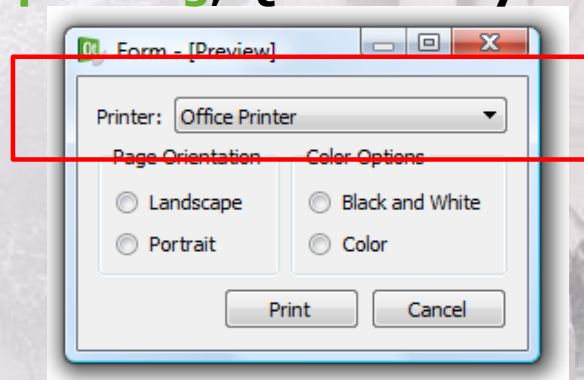
- 布局是在布局管理器和部件间进行协调的过程
- 布局管理器提供布局结构
 - 水平布局和垂直布局
 - 网格布局
- 部件则提供
 - 各个方向上的尺寸策略
 - 最大和最小尺寸



尺寸的策略



printerList->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed)





尺寸的策略



- 每一个部件都有一个尺寸大小的示意 (**hint**)，给出水平和垂直方向上的尺寸的策略
 - **Fixed** – 规定了 **widget** 的尺寸，固定大小（最严格）
 - **Minimum** – 规定了可能的最小值，可增长
 - **Maximum** – 规定可能的最大值，可缩小
 - **Preferred** – 给出最佳值，但不是必须的，可增长可缩小
 - **Expanding** – 同 **preferred**，但希望增长
 - **MinimumExpanding** – 同 **minimum**，但希望增长
 - **Ignored** – 忽略规定尺寸， **widget** 得到尽量大的空间



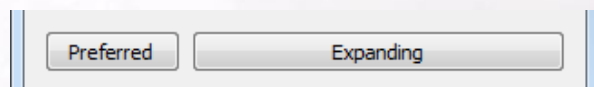
如果？



- 2个 preferred 相邻



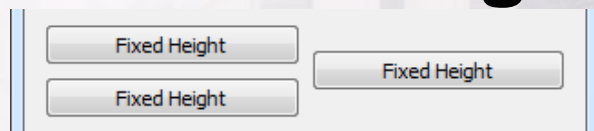
- 1个 preferred, 1个 expanding



- 2个 expanding 相邻



- 空间不足以放置widget (fixed)





关于尺寸的更多内容



- 可用最大和最小属性更好地控制所有部件的大小
- **maximumSize** –最大可能尺寸
- **minimumSize** –最小可能尺寸

```
ui->pushButton->setMinimumSize(100, 150);  
ui->pushButton->setMaximumHeight(250);
```



Qt Designer



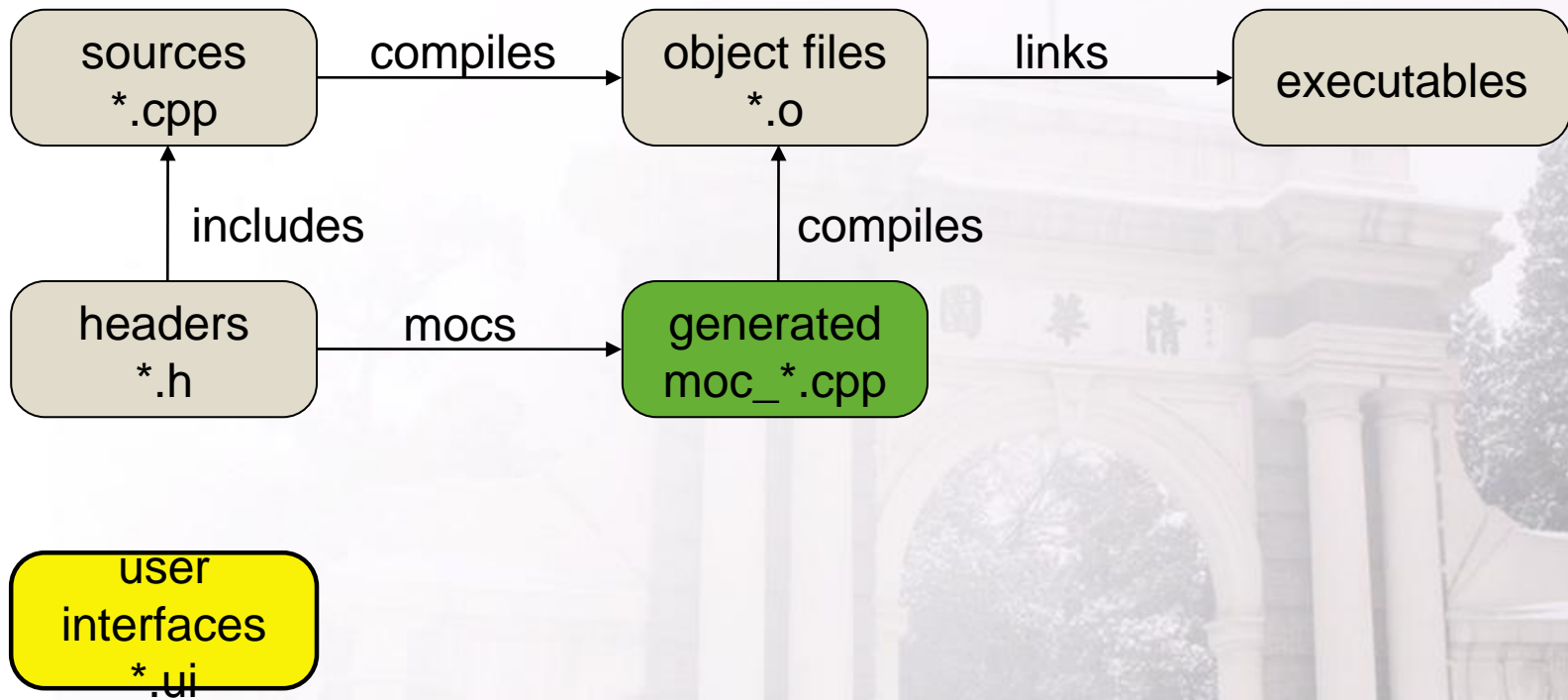
Qt Designer



- ◆ Qt应用程序除了使用手工编写代码的方式外，还可以用过Qt Designer来完成
- ◆ Qt Designer曾是一个独立的Qt桌面工具，现在集成于Qt Creator中
- ◆ 只需要拖动相应的控件
- ◆ 输出为.ui文件，内容其实就是XML
- ◆ Uic编译器把.ui 文件转换成.h文件
 - ⊕ myproject.ui -> ui_myproject.h

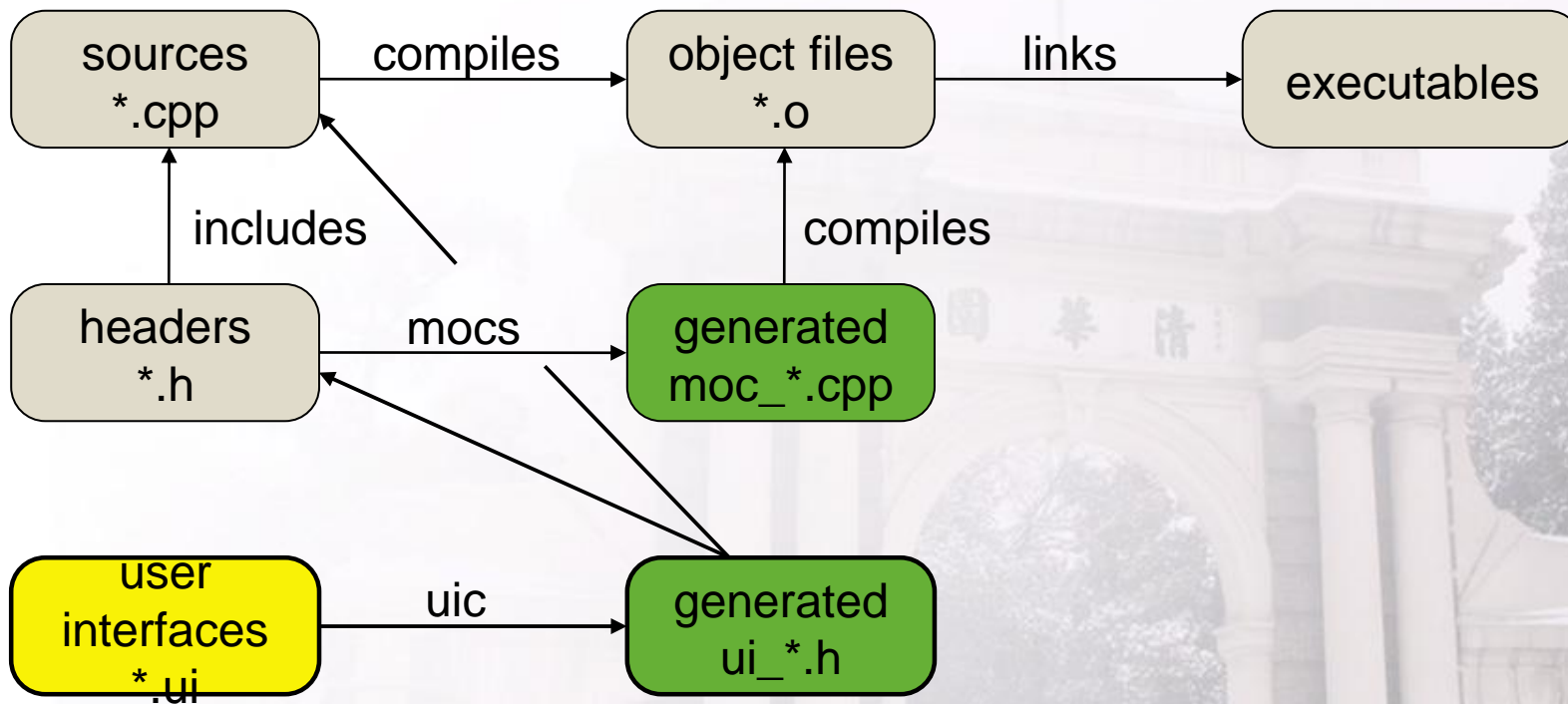


设计器介绍





设计器介绍





使用代码



Ui::Widget类的
前置声明

```
#ifndef WIDGET_H  
#define WIDGET_H
```

```
#include <QWidget>
```

```
namespace Ui {  
    class Widget;  
}
```

```
class Widget : public QWidget {  
    Q_OBJECT  
public:  
    Widget(QWidget *parent = 0);  
    ~Widget();
```

```
private:  
    Ui::Widget *ui;  
};
```

```
#endif // WIDGET_H
```

一个 **Ui::Widget**
类指针 **ui** ,
指向所有部件

基本上是一个标
准的 **QWidget**
派生类



使用代码



调用函数
setupUi, 生成所有父窗体 (**this**) 的子窗体部件

```
#include "widget.h"  
#include "ui_widget.h"
```

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

实例化类
Ui::Widget 为 **ui**

```
Widget::~~Widget()  
{  
    delete ui;  
}
```

删除 **ui** 对象



使用设计器



• 基本工作流程

- 粗略地放置部件在窗体上
- 从里到外进行布局，添加必要的弹簧
- 进行信号连接
- 在代码中使用

- 在整个过程中不断修改编辑属性

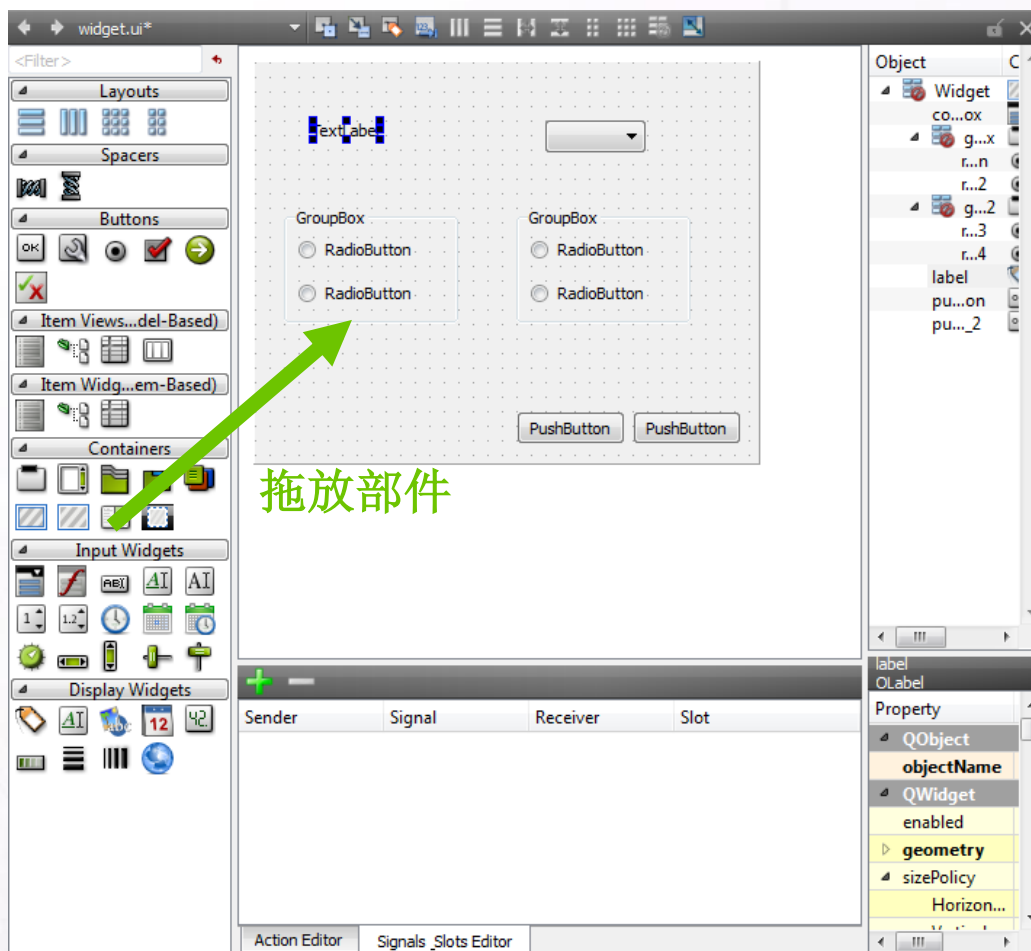
• 实践创造完美!



使用设计器



粗略地放置部件在窗体上



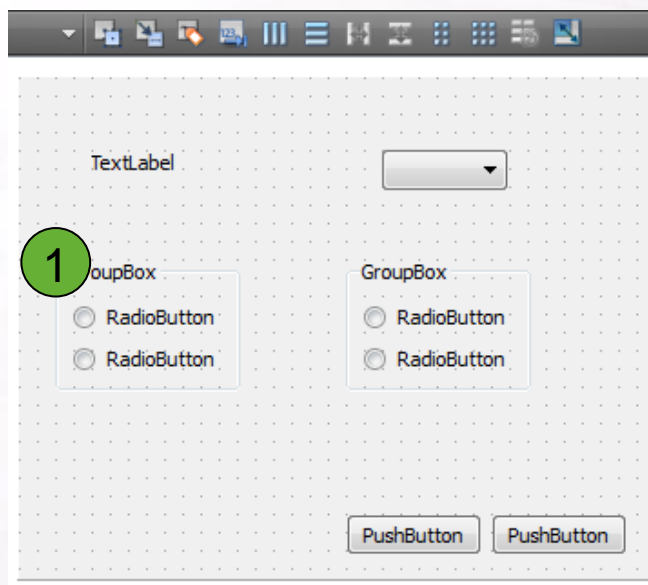
拖放部件



使用设计器



◆ 从里到外进行布局，添加必要的弹簧



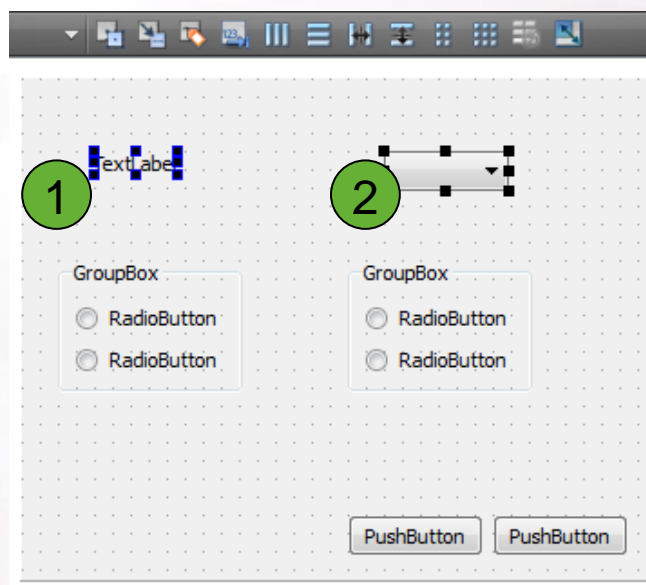
1. 选中每一个 group box, 2. 应用垂直布局管理



使用设计器



- ◆ 从里到外进行布局，添加必要的弹簧



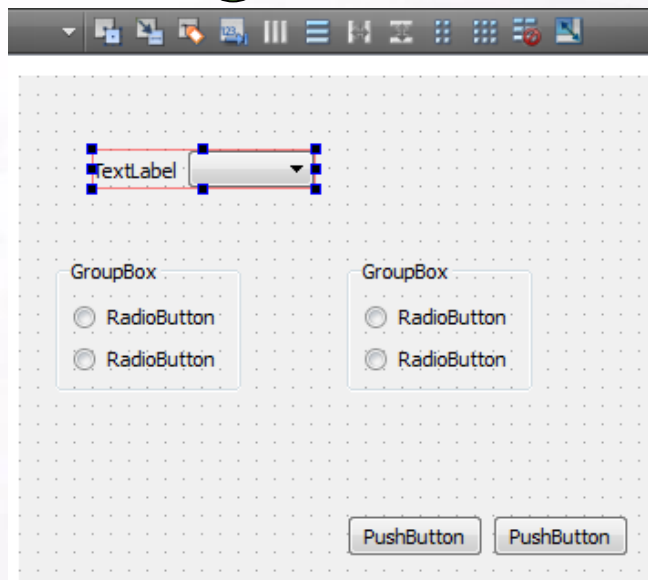
1. 选中label (click), 2. 选中combobox (Ctrl+click)



使用设计器



◆ 从里到外进行布局，添加必要的弹簧



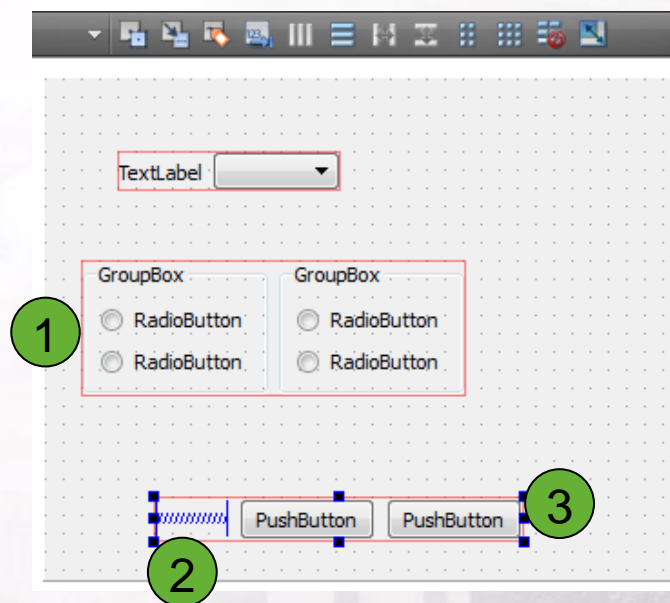
1. 应用一个水平布局管理



使用设计器



- ◆ 从里到外进行布局，添加必要的弹簧



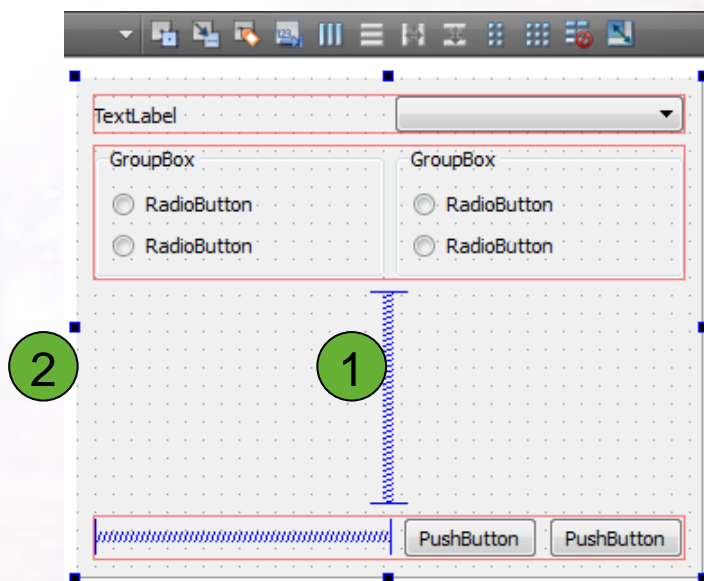
1. 选中2个group box并进行布局管理,
2. 添加一个水平弹簧,
3. 将弹簧和按钮放置进一个布局管理器中



使用设计器



◆ 从里到外进行布局，添加必要的弹簧



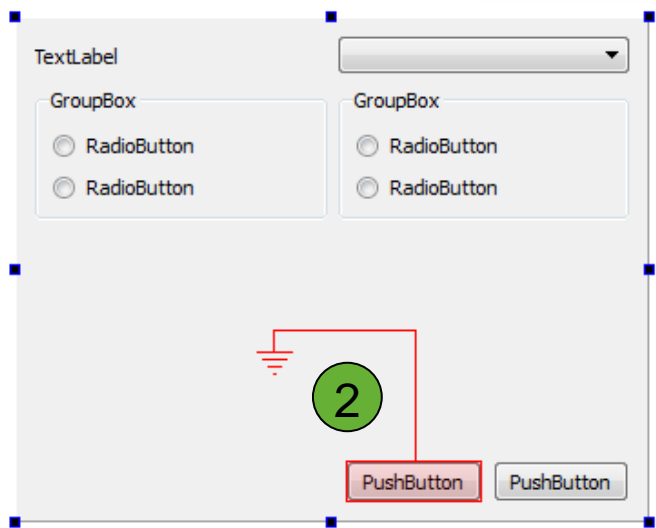
1. 添加一个垂直弹簧, 2. 选中窗体本身, 3. 应用一个垂直布局管理



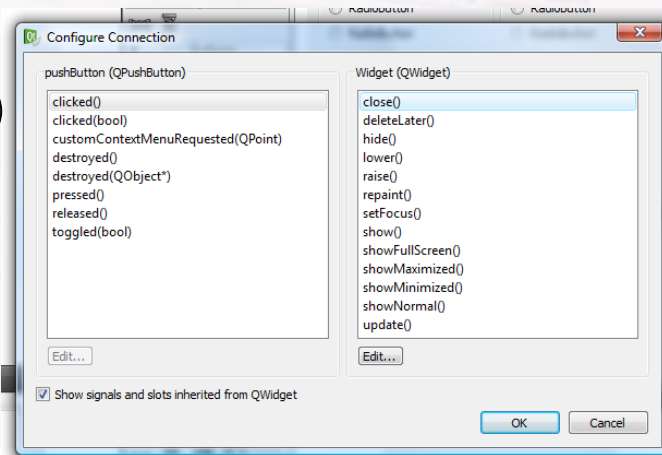
使用设计器

◆ 进行信号连接(部件之间)

1



3



4

Sender	Signal	Receiver	Slot
pushButton	clicked()	Widget	close()

1. 转到signals and slot 编辑模式,
2. 从一个部件拖放鼠标到另一个部件,
3. 选中signal and slot,
4. 在信号和槽编辑器中查看结果

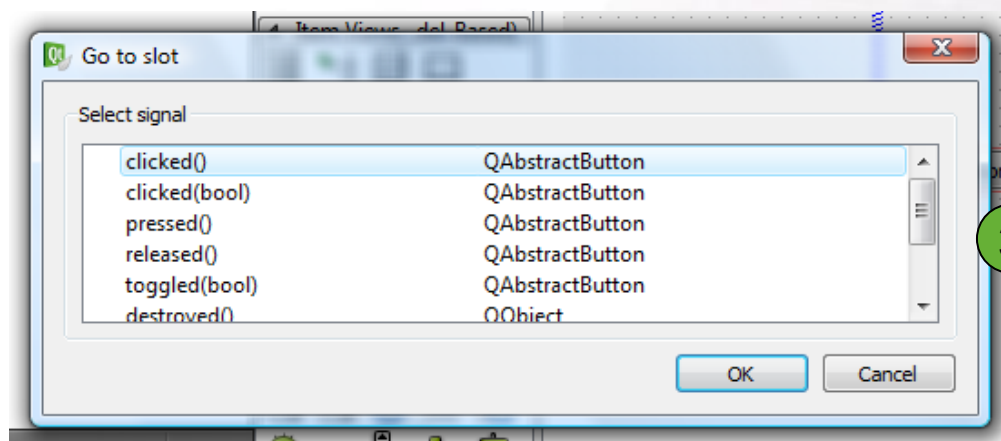


使用设计器



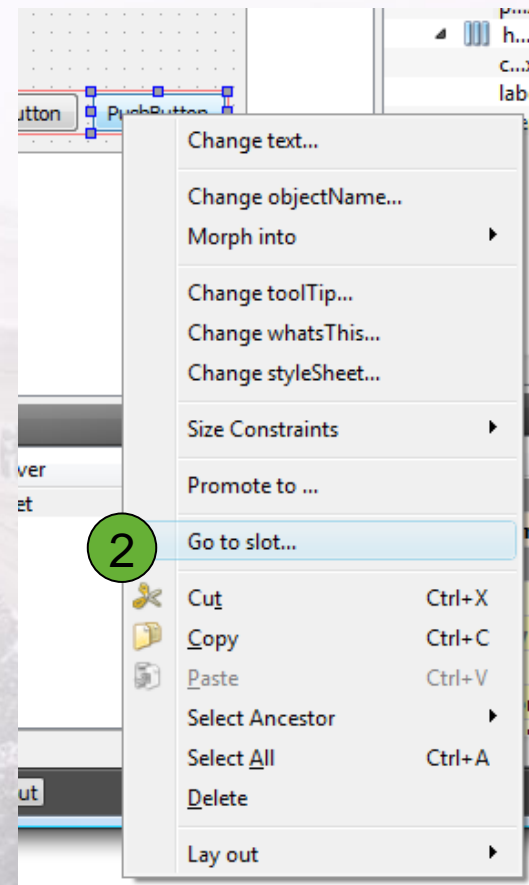
◆ 进行信号连接(到你的代码中)

1



3

2



1. 在widget editing 模式中
2. 右击一个部件并选择 *Go to slot...*
3. 选择一个信号来连接到你的代码



使用设计器



- ◆ 在代码中使用
 - 通过**ui**类成员使用访问其所有子部件

```
class Widget : public QWidget {  
    ...  
private:  
    Ui::Widget *ui;  
};
```

```
void Widget::memberFunction()  
{  
    ui->pushButton->setText(...);  
}
```



顶层窗体



顶层窗体



- 没有父部件的部件自动成为窗体
 - **QWidget** – 普通窗体，通常无模式
 - **QDialog** – 对话框，通常期望一个结果如 **OK**, **Cancel**等
 - **QMainWindow** – 应用程序窗体，有菜单，工具栏，状态栏等
- **QDialog** 和 **QMainWindow** 继承自 **QWidget**



使用QWidget作为窗体



- 任何部件都可成为窗体
 - 没有父部件的部件自动成为窗体
 - 拥有父部件的部件需要传递 **Qt::Window** 标志给 **QWidget** 构造函数
- 使用 **setWindowModality** 函数设定不同模式
 - **NonModal** – 所有窗体可立即使用
 - **WindowModal** – 父窗体阻塞
 - **ApplicationModal** – 所有其他窗体阻塞



窗体属性



- 使用 **setWindowTitle** 设置窗体标题
- **QWidget** 构造函数和窗体标志位
`QWidget::QWidget(QWidget *parent, Qt::WindowFlags f=0)`
 - **Qt::Window** – 生成一个窗体
 - **Qt::CustomizeWindowHint** – 自定义，不用缺省设置
 - **Qt::WindowMinimizeButtonHint**
 - **Qt::WindowMaximizeButtonHint**
 - **Qt::WindowCloseButtonHint**
 - **etc**

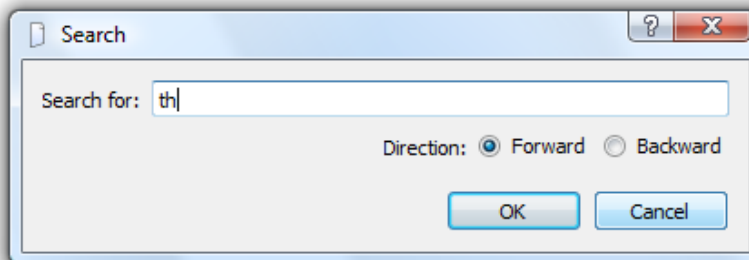
hint 这个单词很重要
不同的平台和窗体管理器对
这些设定有不同的影响



使用QDialog



- 搜索对话框是典型的自定义对话框



- 继承自 **QDialog**
- 使用设计器或代码来建立用户界面
 - **QLabel** 和 **QRadioButton** 是“输出”
 - **OK**, **Cancel**按钮



程序接口



```
class SearchDialog : public QDialog
{
    Q_OBJECT
public:
    explicit SearchDialog(const QString &initialText,
                          bool isBackward, QWidget *parent = 0);

    bool isBackward() const;
    const QString &searchText() const;

private:
    Ui::SearchDialog *ui;
};
```

在构造函数中初始化对话框

Getter 函数获取数据



实现



```
SearchDialog::SearchDialog(const QString &initialText,  
                           bool isBackward, QWidget *parent) :  
    QDialog(parent), ui(new Ui::SearchDialog)
```

```
{  
    ui->setupUi(this);
```

```
    ui->searchText->setText(initialText);  
    if(isBackward)  
        ui->directionBackward->setChecked(true);  
    else  
        ui->directionForward->setChecked(true);  
}
```

```
bool SearchDialog::isBackward() const  
{  
    return ui->directionBackward->isChecked();  
}
```

```
const QString &SearchDialog::searchText() const  
{  
    return ui->searchText->text();  
}
```

根据设置初始化对话框

getter函数



使用Dialog



- 软件接口已经被定义以使其更易于使用

```
void MyWindow::myFunction()
{
    SearchDialog dlg(settings.value("searchText", "").toString(),
                     settings.value("searchBackward", false).toBool(), this);

    if(dlg.exec() == QDialog::Accepted)
    {
        QString text = dlg.searchText();
        bool backwards = dlg.isBackward();
        ...
    }
}
```

QDialog::exec显示一个形式（阻塞）对话框并返回如同同意或拒绝的结果

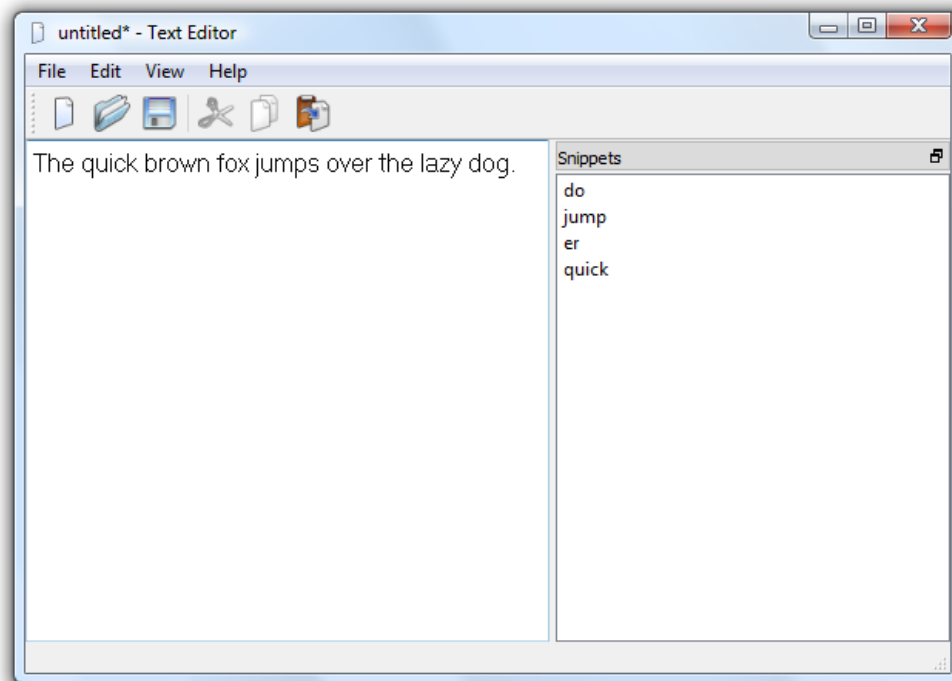


使用QMainWindow



- **QMainWindow** 是普通桌面程序的文档窗体

- 菜单栏
- 工具栏
- 状态栏
- 停靠窗体
- 中心部件





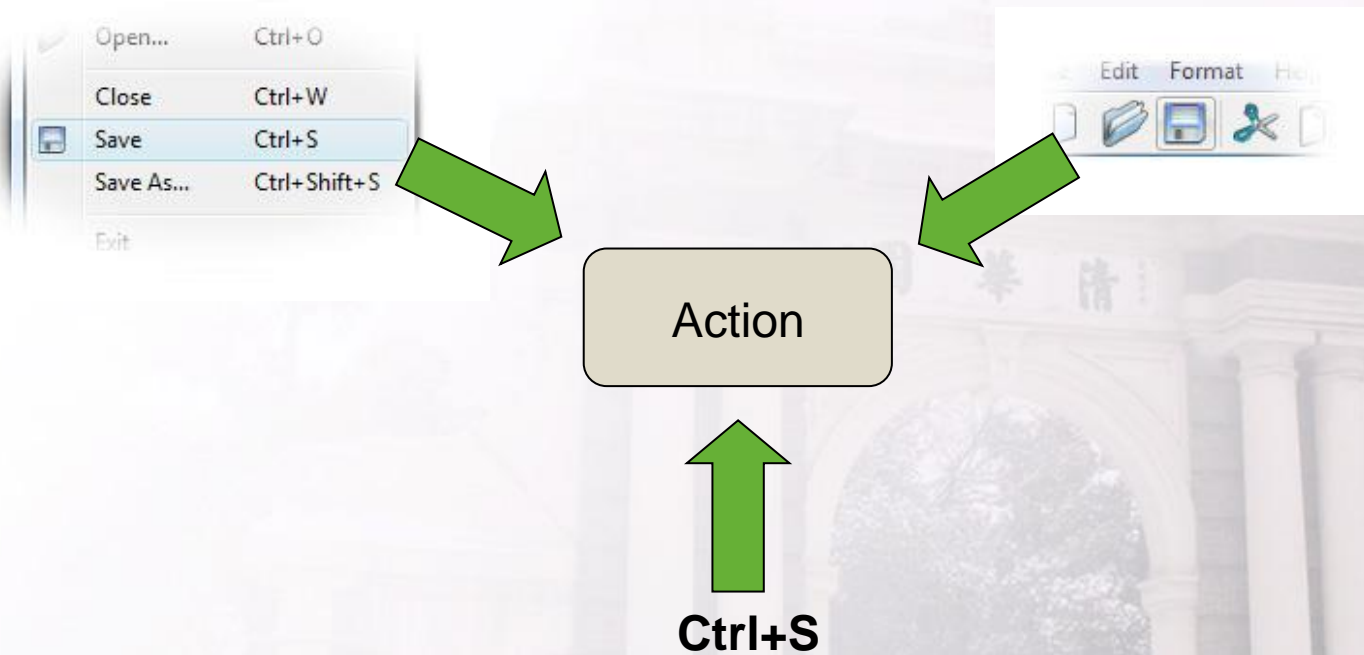
QAction



QAction介绍



- 许多用户界面元素有相同的用户行为（**action**）



- 一个**QAction**对象可以表示所有这些操作方式 – 并保持工具提示，状态栏提示等等。



QAction介绍



- 一个**QAction**封装所有菜单、工具栏和快捷键需要的设置
- 常用属性有
 - **text** – 各处所用的文本
 - **icon** – 各处用到的图标
 - **shortcut** – 快捷键
 - **checkable/checked** – 当前操作是否可选中以及是否已选中
 - **toolTip/statusTip** – 工具栏提示文本(鼠标停顿, 等待)和 状态栏提示文本(鼠标不用等待)



QAction介绍



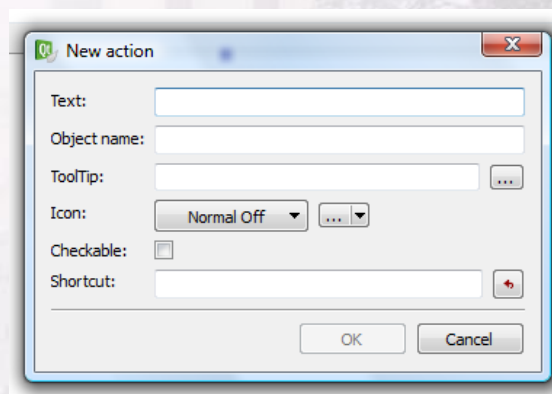
```
QAction *action = new QAction(parent);  
  
action->setText("text");  
action->setIcon(QIcon(":/icons/icon.png"));  
action->setShortcut(QKeySequence("Ctrl+G"));  
  
action->setData(myDataQVariant);
```

生成新的action

设置文本，图标和
快捷键

QVariant可以跟动作
关联，携带跟给定操
作相关联的数据

- 或者在设计器
中使用编辑器





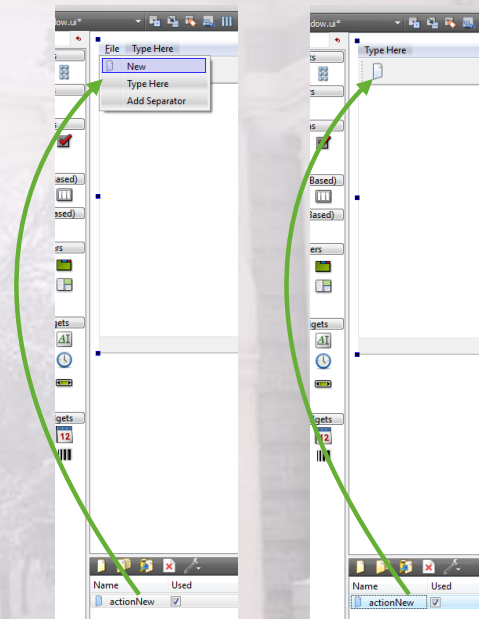
添加Action



- 向不同部分的用户接口添加动作就是调用**addAction**那么简单

```
myMenu->addAction(action);  
myToolBar->addAction(action);
```

- 在设计器中，只需简单地将每一个动作拖放到工具栏或者菜单栏

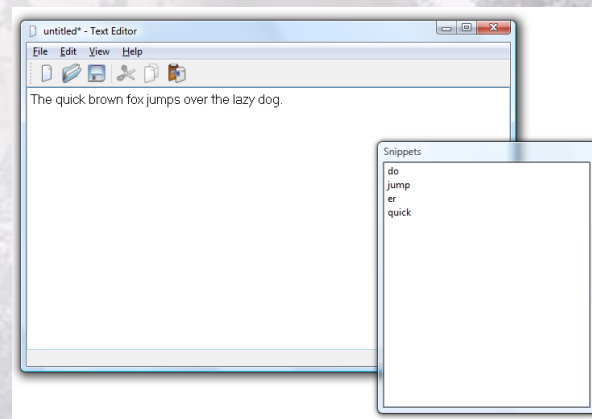
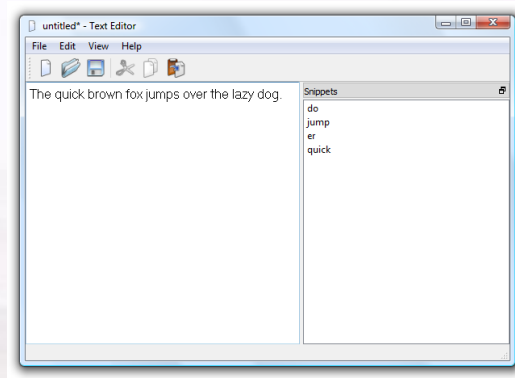




可停靠部件



- 可停靠部件是放置于 **QMainWindow** 边上的一些可拆分的部件
 - 便于使用和设置
- 只需简单地将部件放进 **QDockWidget** 中
- **QMainWindow::addDockWidget** 向窗体添加可停靠部件





可停靠部件



```
void MainWindow::createDock()  
{
```

带标题的一个
新dock

可以移动或者
漂浮

```
    QDockWidget *dock = new QDockWidget("Dock", this);
```

```
    dock->setFeatures(QDockWidget::DockWidgetMovable |  
                    QDockWidget::DockWidgetFloatable);
```

```
    dock->setAllowedAreas(Qt::LeftDockWidgetArea |  
                        Qt::RightDockWidgetArea);
```

```
    dock->setWidget(actualWidget);
```

和用户进行交互的
实际部件

```
    ...
```

可以停靠在边上

```
    addDockWidget(Qt::RightDockWidgetArea, dock);
```

最后将dock添加进窗体



Qt 图标



图标资源



- 将图标放进一个资源文件中，**Qt**会将它们内嵌进可执行文件
 - 避免部署多个文件
 - 不需要关心图标的路径位置
 - 一切都巧妙地在软件构建系统中自适应
 - ...
- 可以将任何东西添加进资源文件中，不仅仅是图标



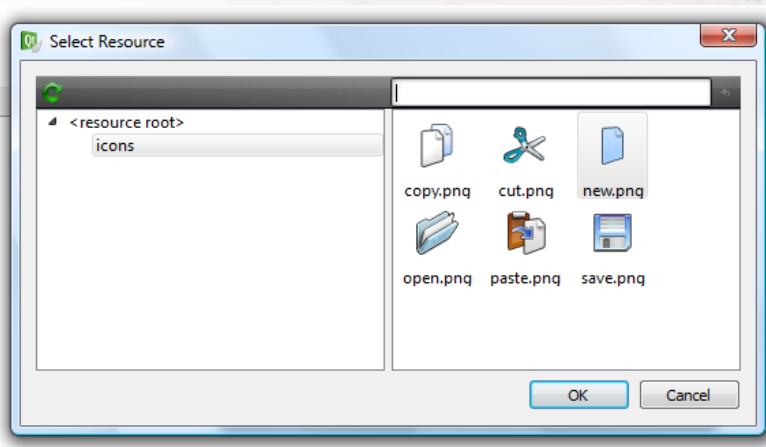
图标资源



- 可以轻松的在**QtCreator**中管理资源文件
- 在路径和文件名前添加 **:** 以使用资源

```
QPixmap pm(":/images/logo.png");
```

- 或者简单地在设计器的列表选择一个图标





Qt 事件处理



Qt事件机制（1）



- ◆ 事件是窗口系统或者Qt对不同情况的响应。绝大多数被产生的事件都是对用户行为（鼠标、键盘操作）的响应，但是也有一些，比如定时器事件，由系统独立产生。
- ◆ 在Qt中，所有事件都发送到Qt事件队列中
- ◆ 在Qt中，事件是一个被发送到事件处理函数的对象
 - ⊕ QEvent类是所有事件类的基类。事件类包含事件参数。
 - ⊕ QEvent的子类有QMouseEvent, QKeyEvent, QPaintEvent, QTimerEvent, etc.



Qt事件机制（2）



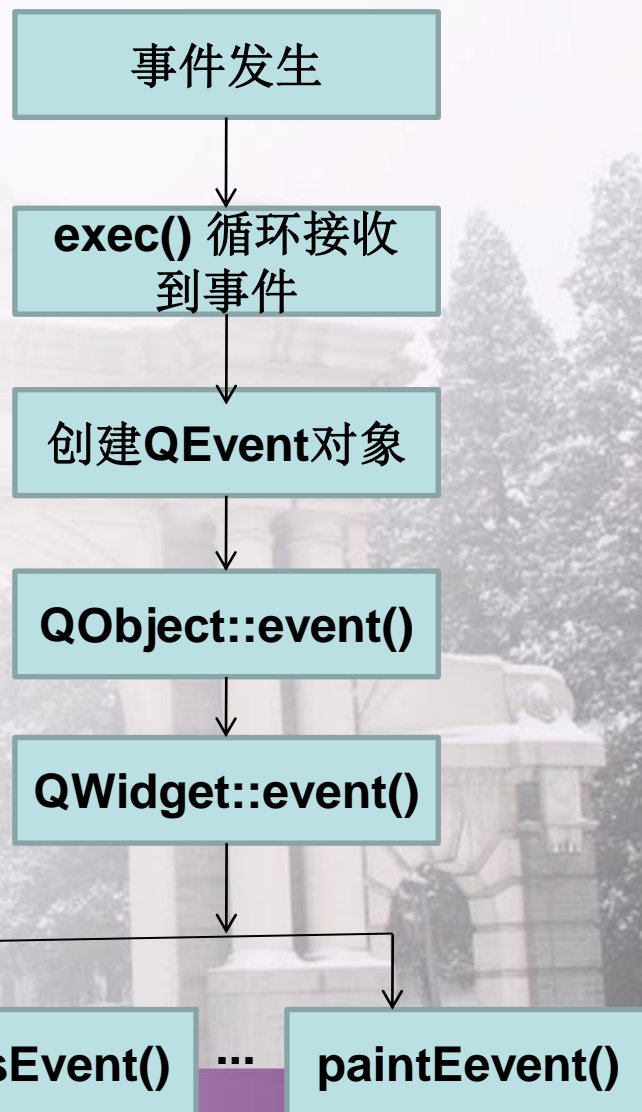
- ◆ Qt的主事件循环(**`QApplication::exec()`**)从事件队列中取得本地窗口系统的事件，并将它们转变成**`QEvent`**对象，然后发送给**`QObject`**对象处理
- ◆ 事件队列中的事件可能被合并
 - ⊕ 只有最后一个**`QMouseEvent`**被处理
 - ⊕ 多个**`QPaintEvent`**图形重绘要求可能被合并
- ◆ 当**`QObject`**对象收到一个事件时，**`QObject::event`**函数将被激活
 - ⊕ **`event`**函数可以接受或忽略这个事件
 - ⊕ 被忽略的事件依据对象继承层次传递出去



事件处理流程



- ◆ 事件发生后，**QApplication::exec()**事件循环会接收到事件
- ◆ Qt创建一个事件的对象并且把它传给相应的**QObject**对象
 - ⊕ **QObject::event(QEvent* e)**
 - ⊕ 如果不处理事件，会重新把事件传给相应的事件处理函数
 - ⊕ 针对不同的事件有不同的保护类型（**protected**）虚函数实现。





事件处理方式



- ◆ 重新实现 **QObject::event()** 或 **QWidget::event()**
 - ⊕ 此方法可以在事件到达特定事件处理器之前处理它们
- ◆ 重新实现 **特殊的事件处理器**
 - ⊕ **mousePressEvent()**, **keyPressEvent()**, ...
- ◆ 在 **QObject** 中安装 **事件过滤器**
 - ⊕ 通过对目标对象调用 **installEventFilter()** 来注册监视对象
 - ⊕ 在监视对象的 **eventFilter()** 中处理目标对象的事件
 - ⊕ 目标对象一旦通过函数 **installEventFilter()** 安装过滤器，目标对象的所有事件都会先发送给这个监视对象的 **eventFilter ()** 函数
 - ⊕ 如果目标对象安装多个事件过滤器，则会按照后安装先处理的顺序激活事件过滤器



用QObject::event()处理事件



- ◆ **QObject::event()** 函数主要用于事件的分发，重写该函数可以在事件分发之前做一些处理
- ◆ **event()**函数返回值是**bool**类型
 - ⊕ 如果传入的事件已被识别并且处理，返回**true**
 - ⊕ 否则返回**false**，分发下去处理



重新实现event()（例子）



◆ 例子：在窗口中的**tab**键按下时将焦点移动到下一组件，而不是让具有焦点的组件处理。

⊕ **MyWidget**是**QWidget**的子类，继承了**QObject**类的**event**函数

```
bool MyWidget::event(QEvent *event) {  
    if (event->type() == QEvent::KeyPress) {  
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);  
        if (keyEvent->key() == Qt::Key_Tab) {  
            // 处理Tab键，移动到下一个组件  
            return true;  
        }  
    }  
    return QWidget::event(event);  
}
```

// **QEvent::type()**函数返回**QEvent::Type**类型的枚举



特殊的事件处理器



◆ 子类化对象，并重新实现相应的保护类型的虚函数。例如：

⊕ 响应按键事件，需要实现：

void keyPressEvent(QKeyEvent*)

⊕ 实现时钟，需要实现：

void timerEvent(QTimerEvent*)

⊕ 响应鼠标事件，需要实现：

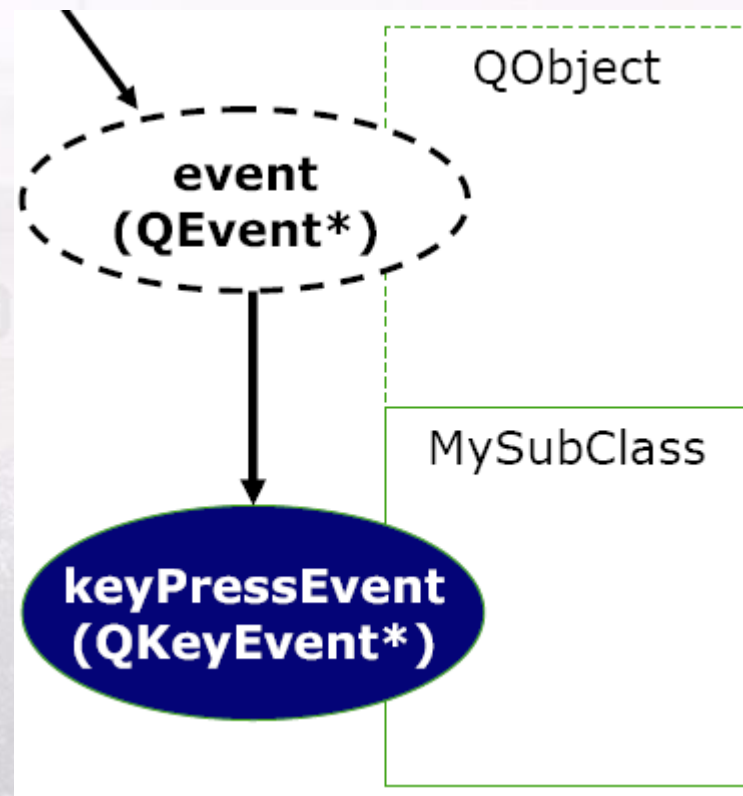
void mousePressEvent(QMouseEvent*)

void mouseDoubleClickEvent
(QMouseEvent * event)

⊕ 响应布局改变事件，需要实现：

void resizeEvent(QResizeEvent*)

void moveEvent(QMoveEvent*)





重新实现特殊的事件处理器（续）



```
void MyLabel::mousePressEvent(QMouseEvent * event)
{
    if(event->button() == Qt::LeftButton) {
        // do something
    } else {
        QLabel::mousePressEvent(event);
    }
}
```



在QObject中安装事件过滤器



- ◆ 过滤器（监视对象）是一个QObject子类对象，并实现了eventFilter函数
 - ⊕ virtual bool QObject::eventFilter (QObject * **target**, QEvent * event)
 - ⊕ 如果**target**对象（被监视对象或目标对象）安装了事件过滤器，这个函数会被调用并进行事件过滤
 - ⊕ 在重写这个函数时，如果需要过滤掉某个事件（如停止对这个事件的响应），则需要返回**true**
- ◆ 安装过滤器
 - ⊕ void QObject::installEventFilter (QObject * **filterObj**)
 - ⊕ 可以将监视对象安装到任何QObject的子类对象上
 - ⊕ 如果一个组件安装了多个过滤器，则最后一个安装的会最先调用，类似于堆栈的行为



在QObject中安装事件过滤器（实例）



```
bool MainWindow::eventFilter(QObject *obj, QEvent *event)
{
    if (obj == ui->textEdit) {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent = static_cast<QKeyEvent*>(event);
            qDebug() << "Ate key press" << keyEvent->key();
            return true;
        } else {
            return false;
        }
    } else {
        // pass the event on to the parent class
        return QMainWindow::eventFilter(obj, event);
    }
}
//....
MainWindow::MainWindow(...)... { ui->textEdit->installEventFilter(this); }
```




QTimer



◆ QTimer可以使用时钟生成事件

```
MyClass(QObject *parent) : QObject(parent)
{
    QTimer *timer = new QTimer(this);
    timer->setInterval(5000);
    connect(timer, SIGNAL(timeout()), this, SLOT(doSomething()));
    timer->start();
}
```

5000ms, 即5s

◆ 或用于延迟一个动作

```
QTimer::singleShot(1500, dest, SLOT(doSomething()));
```



关闭窗口事件



- ◆ 通过拦截关闭窗口消息，可以弹出警告窗口，即使用户确认退出操作

- ⊕ 可以实现如下函数 **void QWidget::closeEvent (QCloseEvent * event) [virtual protected]**

```
#include <QCloseEvent>
```

```
void MainWindow::closeEvent(QCloseEvent * event) {  
    int ret = QMessageBox::warning(0, tr("PathFinder"), tr("您真的想要退出? "), QMessageBox::Yes | QMessageBox::No);  
    if (ret == QMessageBox::Yes) {  
        event->accept(); //确认关闭  
    } else {  
        event->ignore(); //不关闭  
    }  
}
```



谢谢！