Qt事件处理

徐枫

清华大学软件学院

feng-xu@tsinghua.edu.cn

课程主要内容

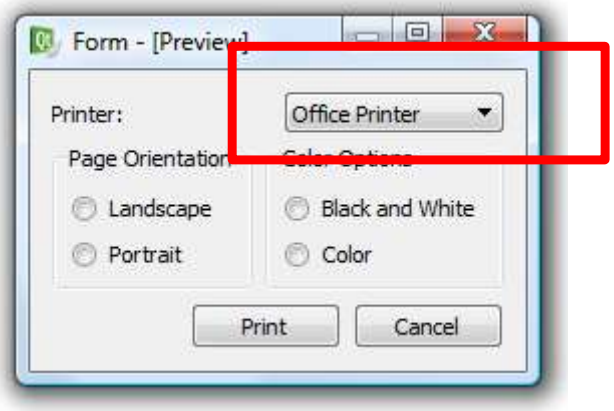
- 部件的尺寸策略
- 顶层窗体
- Qt图标
- Qt事件处理

部件的尺寸策略

尺寸策略

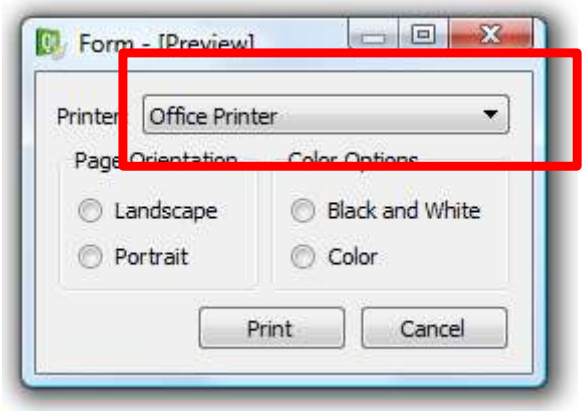
- 布局是在布局管理器和部件间进行协调的过程
- 布局管理器提供布局结构
  - 水平布局和垂直布局
  - 网格布局
  - 表格布局
- 部件则提供
  - 各个方向上的尺寸策略
  - 最大和最小尺寸

尺寸的策略



Void setSizePolicy(QSizePolicy::Policy horizontal, QSizePolicy::Policy vertical)

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);
```

顶层窗体

顶层窗体

- 没有父部件的部件自动成为顶层窗体
  - 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

- 任何部件都可成为顶层窗体
  - 没有父部件的部件自动成为顶层窗体
  - 拥有父部件的部件想成为顶层窗体，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 –非模式，可以和程序的其它窗体交互
  - WindowModal –窗体模式，程序在未处理完对话框时将阻止和对话框的父窗体、祖父窗体以及父窗体的兄弟姐妹窗体及其父窗体交互
  - ApplicationModal –应用程序模式，阻止和任何其它窗体进行交互

窗体属性

- 使用setWindowTitle设置窗体标题

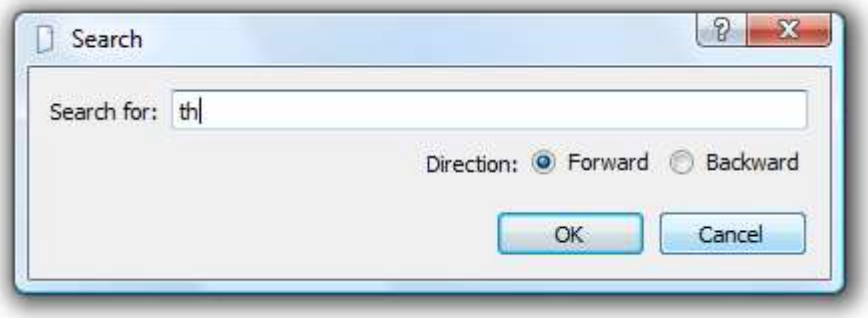
- QWidget构造函数和窗体标志位
  QWidget::QWidget(QWidget *parent, Qt::WindowFlags f=0)中f可以是：
  - Qt::Window – 生成一个顶层窗体
  - Qt::CustomizeWindowHint – 自定制，不用缺省设置
    - Qt::WindowMinimizeButtonHint
    - Qt::WindowMaximizeButtonHint
    - Qt::WindowCloseButtonHint
    - etc

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

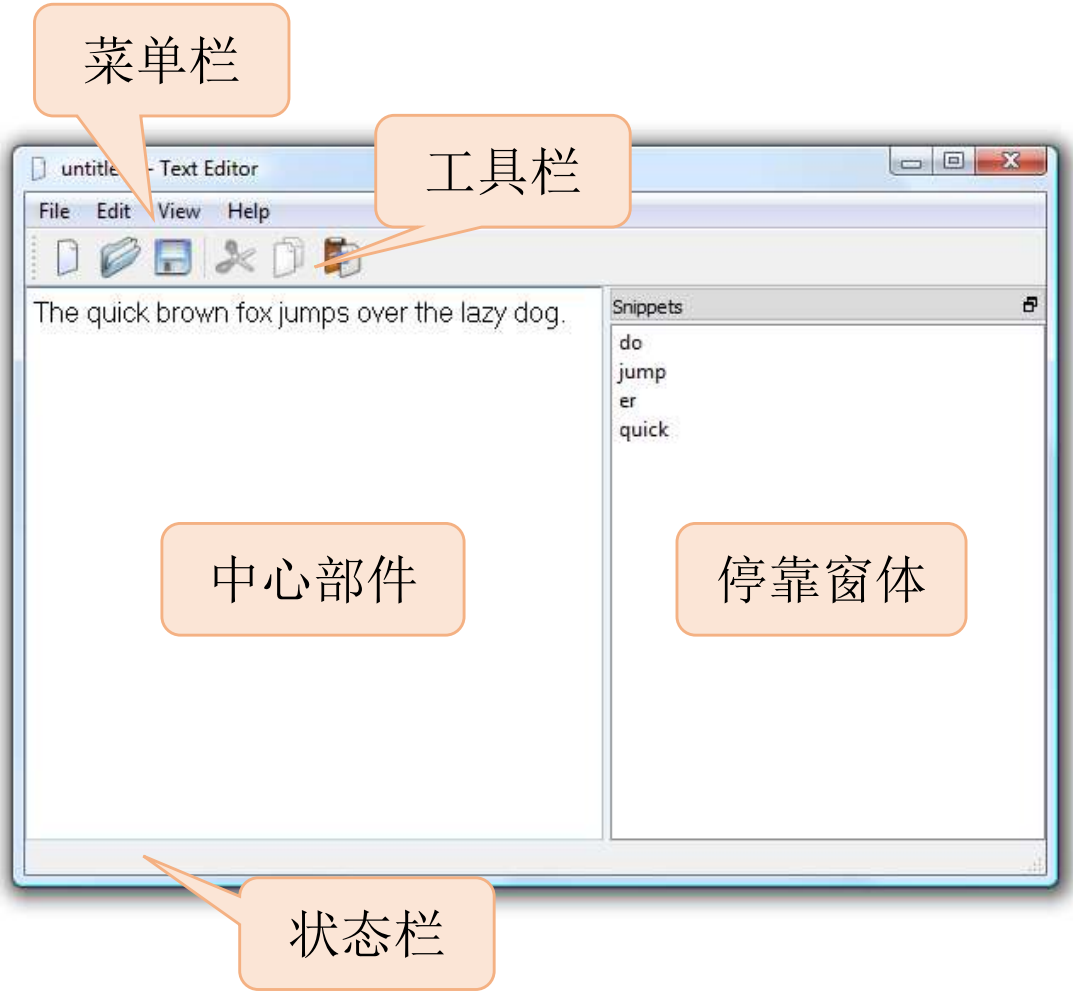
```
void MyWindow::myFunction()
{
    SearchDialog dlg(settings.value("searchText","").toString(),
                     settings.value("searchBackward", false).toBool(), this);

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

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

此**exec**和**Qapplication**的**exec**是对相同函数的动态继承，体现了**C++**编程的特点

使用QMainWindow

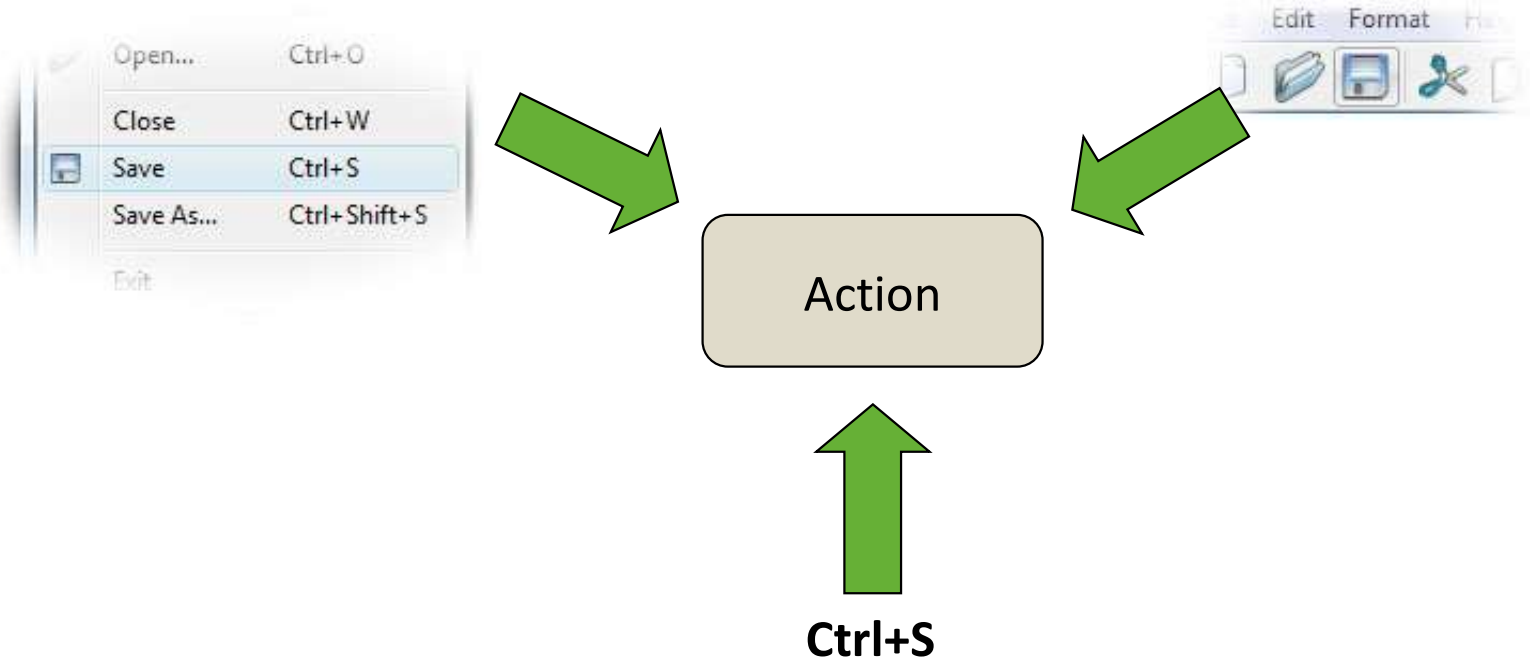
- QMainWindow 是普通桌面程序的文档窗体
  - 菜单栏(QMeunBar)
  - 工具栏(QToolBar)
  - 状态栏(QStatusBar)
  - 停靠窗体 (QDockWidget)
  - 中心部件(Central Widget)



QAction

QAction介绍

- 菜单栏和工具栏有相同的用户行为（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);
```

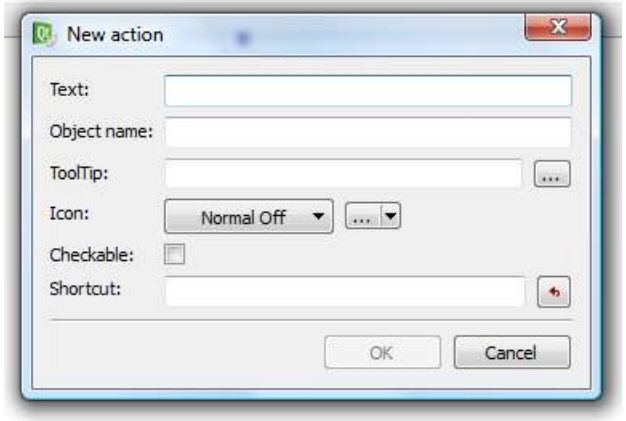
生成新的action

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

The QVariant class acts like a union for the most common Qt data types

- 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

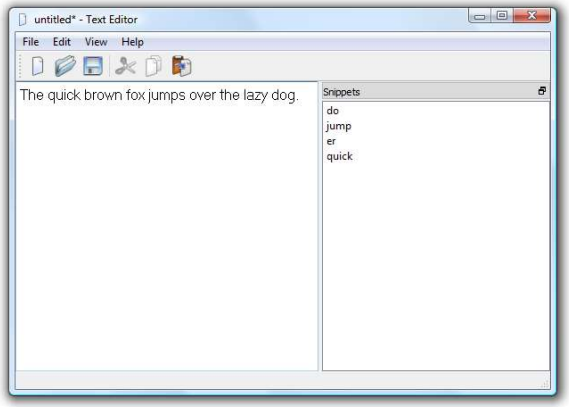
```
myMenu->addAction(action);
myToolBar->addAction(action);
```

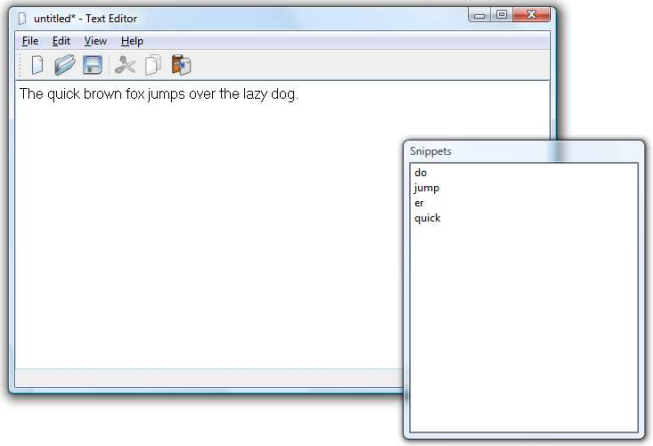
- 在设计器中，只需简单地将每一个动作拖放到工具栏或者菜单栏

Qaction的设计，体现了C++封装的性质，同一类对象，可以很好的应用到菜单栏、工具栏

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

    dock->setFeatures(QDockWidget::DockWidgetMovable |
                      QDockWidget::DockWidgetFloatable);
    dock->setAllowedAreas(Qt::LeftDockWidgetArea |
                          Qt::RightDockWidgetArea);
    dock->setWidget(actualWidget);


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

带标题的一个新dock

可以移动或者漂浮

可以停靠在边上

和用户进行交互的实际部件

最后将dock添加进窗体

图标

Qt 图标

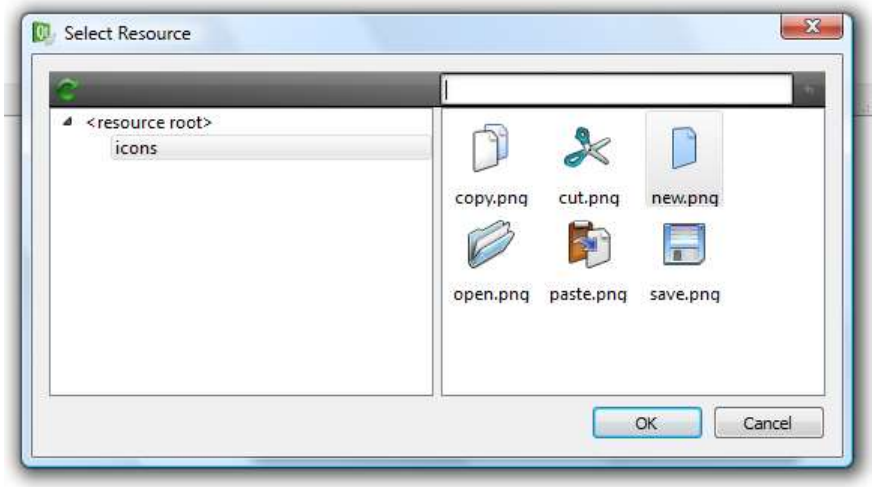
- 图标资源：将图标放进一个资源文件中，Qt会将它们内嵌进可执行文件
  - 避免部署多个文件
  - 不需要关心图标的路径位置
  - 一切都巧妙地在软件构建系统中自适应
  - ...
  - 可以将任何东西添加进资源文件中，不仅仅是图标

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中选择一个图标



Qt事件处理

Qt 事件机制

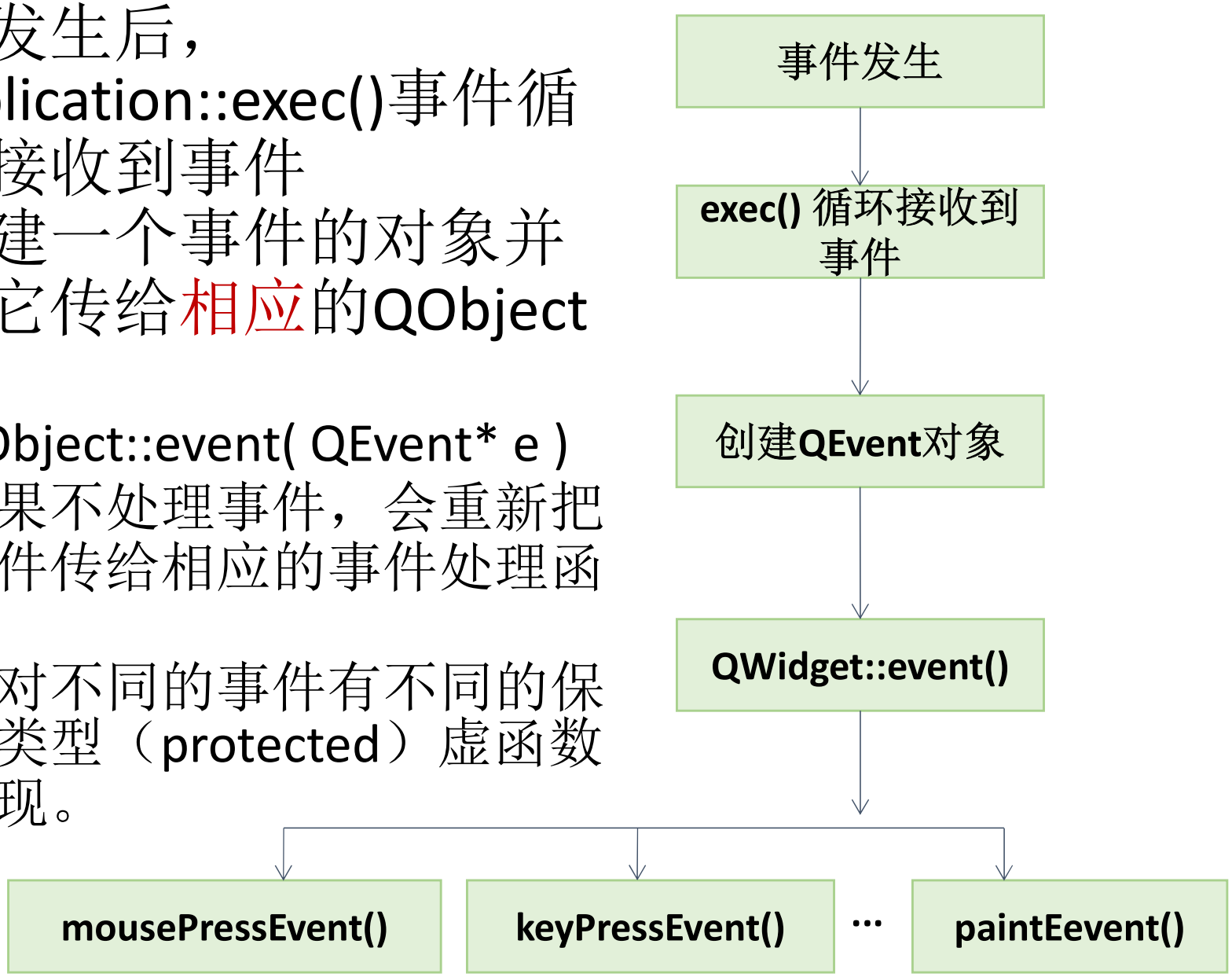
- 事件是窗口系统或者Qt对不同情况的响应。绝大多数被产生的事件都是对用户行为（鼠标、键盘操作）的响应，但是也有一些，比如定时器事件，由系统独立产生。
- 在Qt中，所有事件都发送到Qt事件队列中
- 在Qt中, 事件是一个被发送到事件处理函数的对象
  - QEvent类是所有事件类的基类。事件类包含事件参数。
  - QEvent的子类有QMouseEvent, QKeyEvent, QPaintEvent, QTimerEvent, etc.
  - 有些事件天然 变成signal，有些不是，比如任意的键盘或鼠标操作

Qt事件机制

- 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QMouseMoveEvent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。



传递原则

- 键盘的话应该是焦点对象，鼠标的话就是光标当前所在的对象；这个对象不处理，就朝父对象传；（事件过滤器允许某个对象提前截获其他对象的部分事件）

- 每个对象拿到事件后，先是event函数处理，将事件分发给特定的**mousePressEvent**等函数。

- 对于一个派生类对象，event函数（是虚函数）如果被overide，自然是派生类event函数被调用，否则则寻找其基类的函数实现，其他特定的**mousePressEvent**等函数也是一样。

Qt部件与事件处理

徐枫

清华大学软件学院

feng-xu@tsinghua.edu.cn

Open Question Points: 10

Setting

请列出如图对话框中使用布局管理器（QLayout）的地方

Open Question is only supported on Version 2.0 or newer.

Answer

测试

- 请列出如图对话框中可以使用布局管理器（QLayout）的地方

整体上左右两块是horizontal Layout

左侧，右侧分别是 virtual Layout

左侧上面三行是horizontal Layout

左侧第四行是Form Layout

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
- 重新实现特定的事件处理器
  - 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（作为函数的参数传入）
  - 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

方法1：重载虚函数event()

- 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); //派生类不处理的话，传给基类处理
}
// QEvent::key()函数返回QEvent::Type类型的枚举
```

方法2：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*)

  - 实现时钟事件，需要实现：

void timerEvent(QTimerEvent*)

  - 响应鼠标事件，需要实现：

void mousePressEvent(QMouseEve

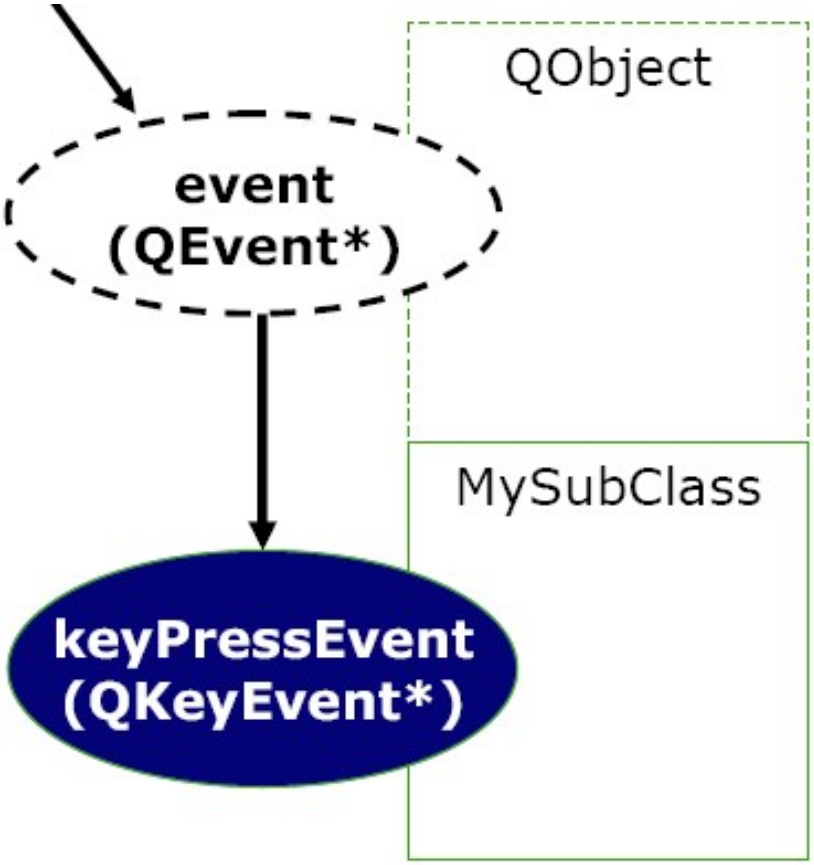
void mouseDoubleClickEvent

(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);
  }
}
```

方法3：在QObject中安装事件过滤器

- 监视对象是实现了eventFilter函数的QObject子类对象
  - 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)
  - MonitoredObj->installEventFilter(filterObj)
  - 可以将过滤器安装到任何QObject的子类对象上
  - 如果一个部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 QMainWindow::eventFilter(obj, event);
        }
    } 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());
```

singleShot：xx毫秒后，进入目标对象的一个slot函数；

QTimerEvent：定时器事件

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
    }
}
```

总结

- 用户界面部件介绍
- 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

谢谢！