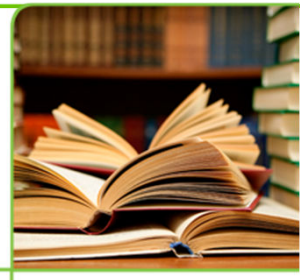
Qt入门基础

- Qt简介
- Qt的应用
- Qt的使用
- Qt深入理解

什么是Qt?



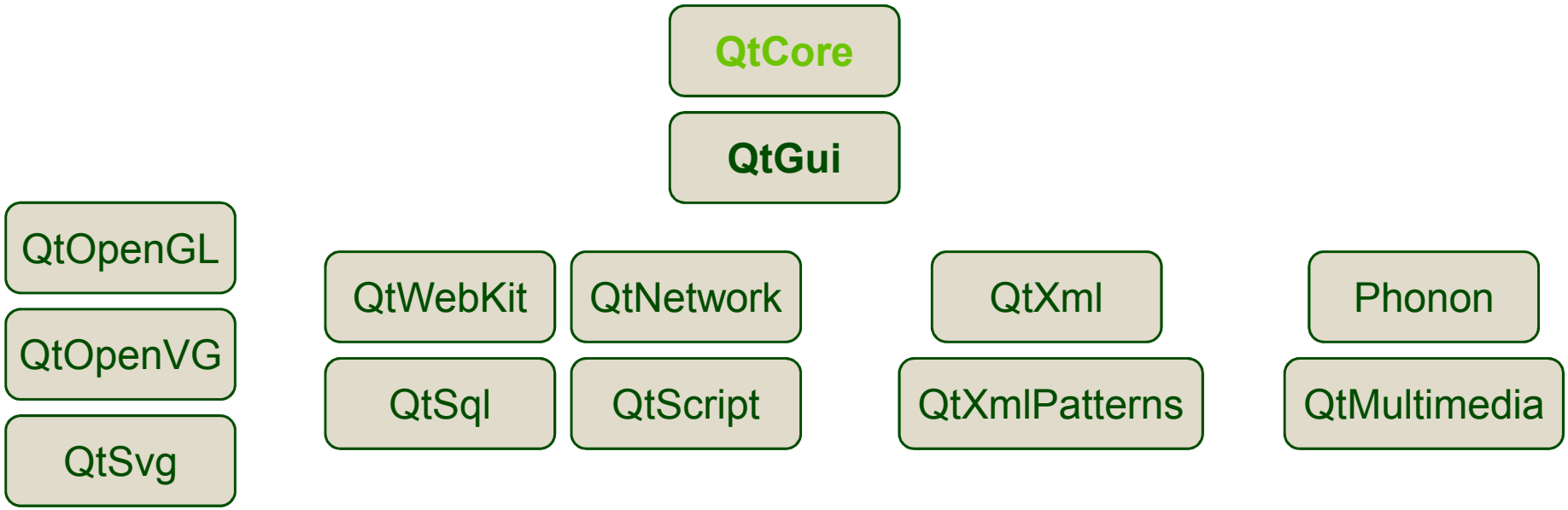
Qt 是一个用C++编写的跨平台开发框架.

原来用作用户界面开发，现可用作所有的开发

例如：Databases, XML, WebKit, multimedia, networking, OpenGL, scripting, non-GUI...

什么是Qt?

- Qt由模块构建



什么是QT?

- Qt用宏（macros）和内省（introspection）扩展了C++

```
foreach (int value, intList) { … }
```

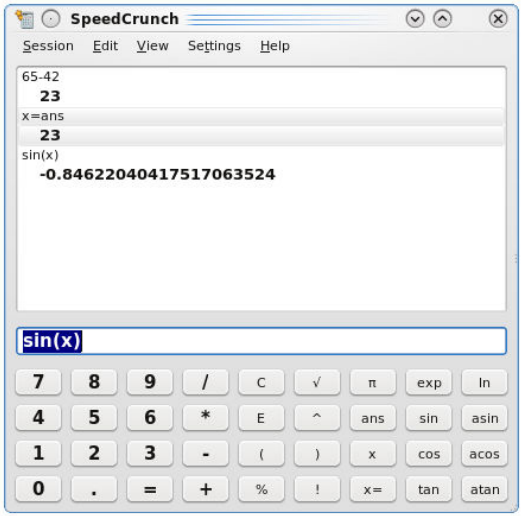
```
QObject *o = new QPustButton;
o->metaObject()->className(); // 返回 ”QPushButton”
```

```
connect(button, SIGNAL(clicked()), window, SLOT(close()));
```

- 所有的代码仍然是简明C + +

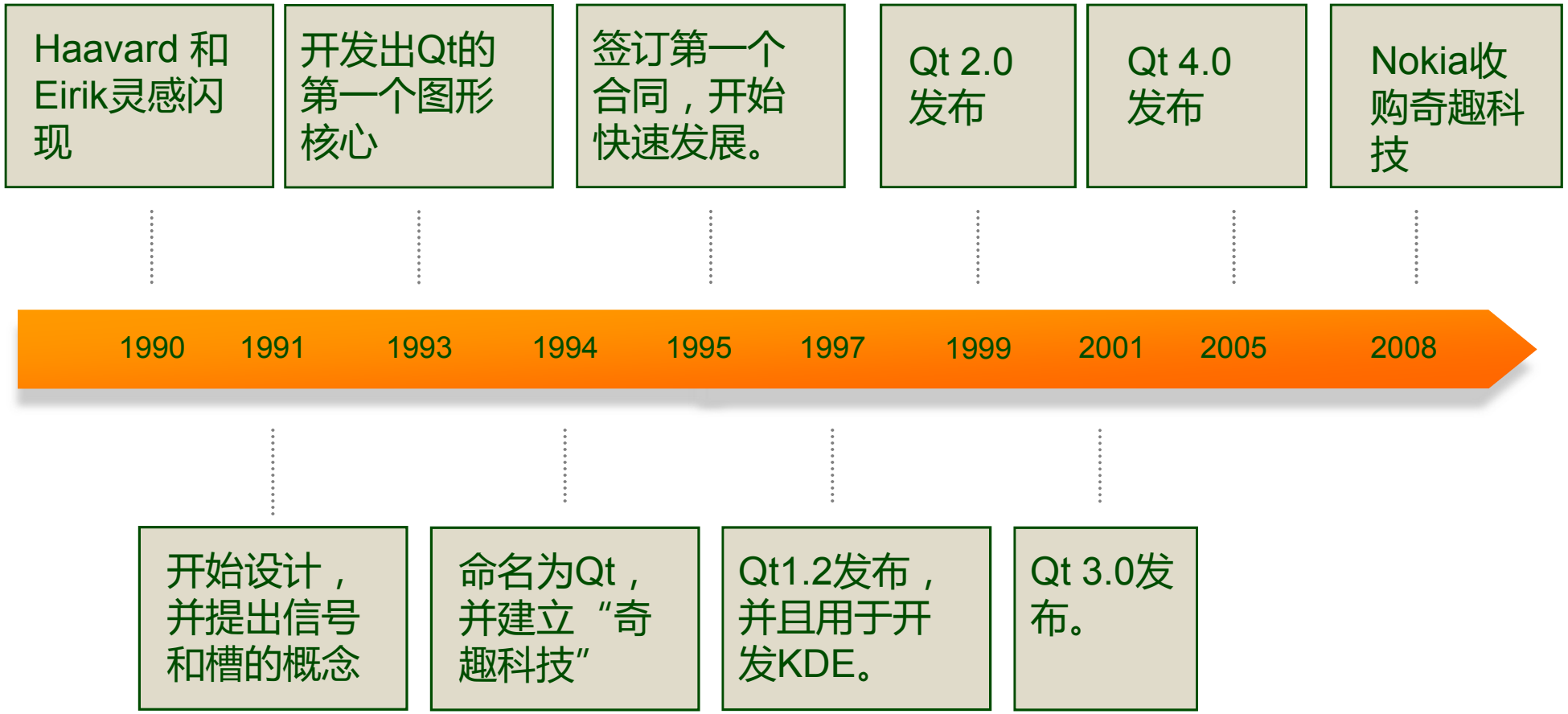
Qt的目的

- 一次编写，到处编译
- 根据不同平台的本地观感生成相应的本地应用



- 简单地使用API，高开发效率，开放性，使用有趣

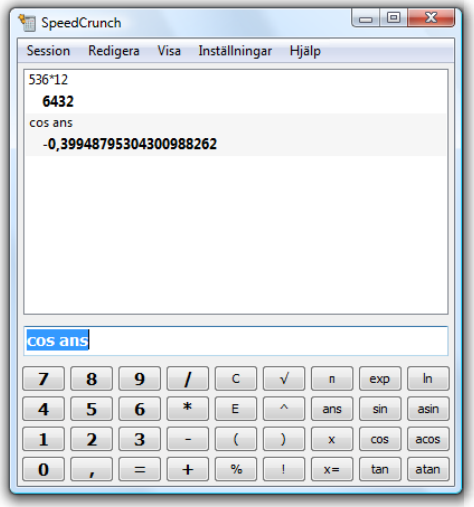
Qt的历史



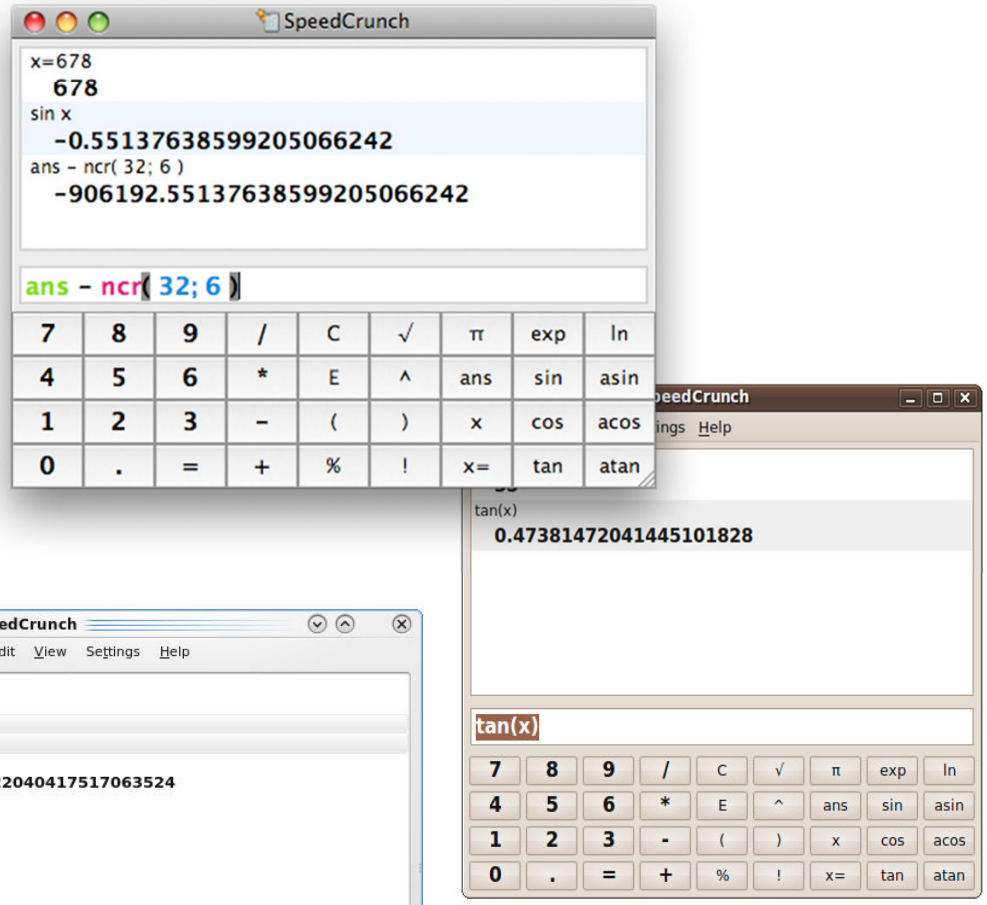
- 2012年诺基亚将Qt部门卖给Digia
- 2016年Qt 公司独立于Digia，并在纳斯达克赫尔辛基独立上市

Qt的跨平台—桌面平台

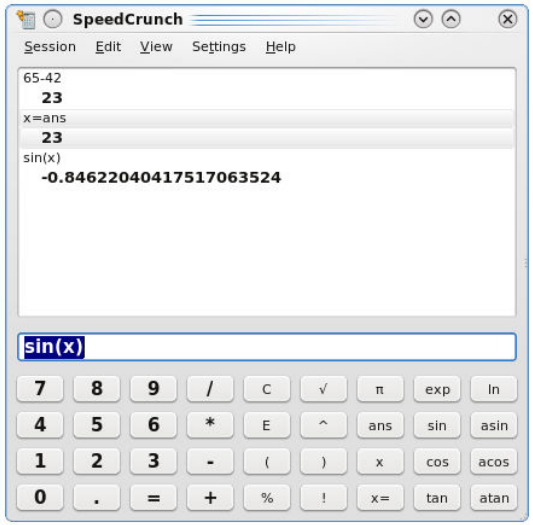
- Windows



- Mac OS X



- Linux/Unix X11



Qt的跨平台—嵌入式平台

- Windows CE
- Symbian
- Maemo
- 嵌入式Linux

Qt的授权

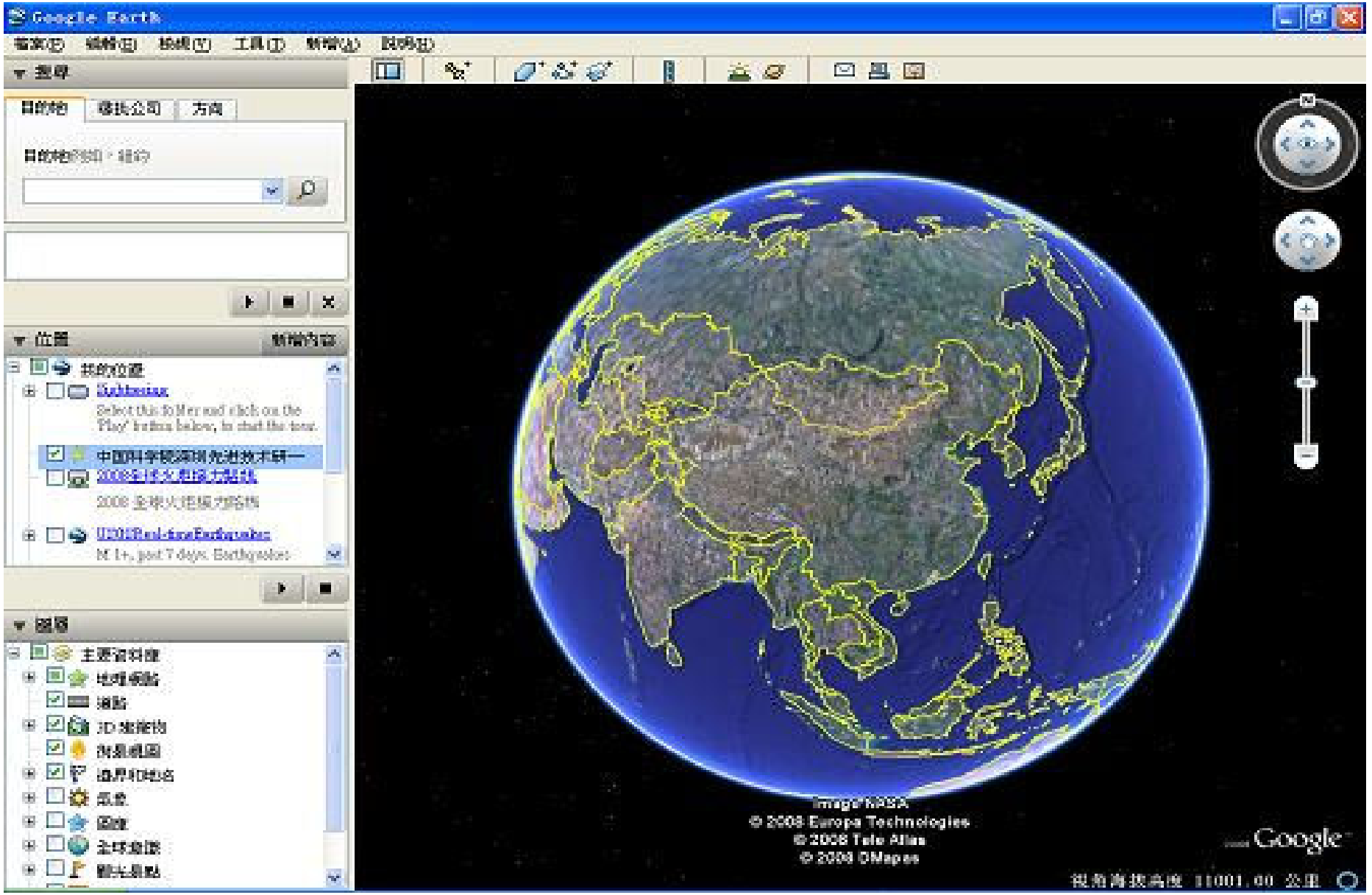
- LGPL – 免费
  - 你的应用程序可以是开源的或者是不开源的
  - 对Qt的修改，必须反馈到社区
- GPL – 免费
  - 你的应用程序必须是开源的
  - 对Qt的修改，必须反馈到社区
- 商业的 – 收费
  - 你的应用程序可以是不开源的
  - 对Qt的修改，可以不开源

- Qt简介
- Qt的应用
- Qt的使用
- Qt深入理解

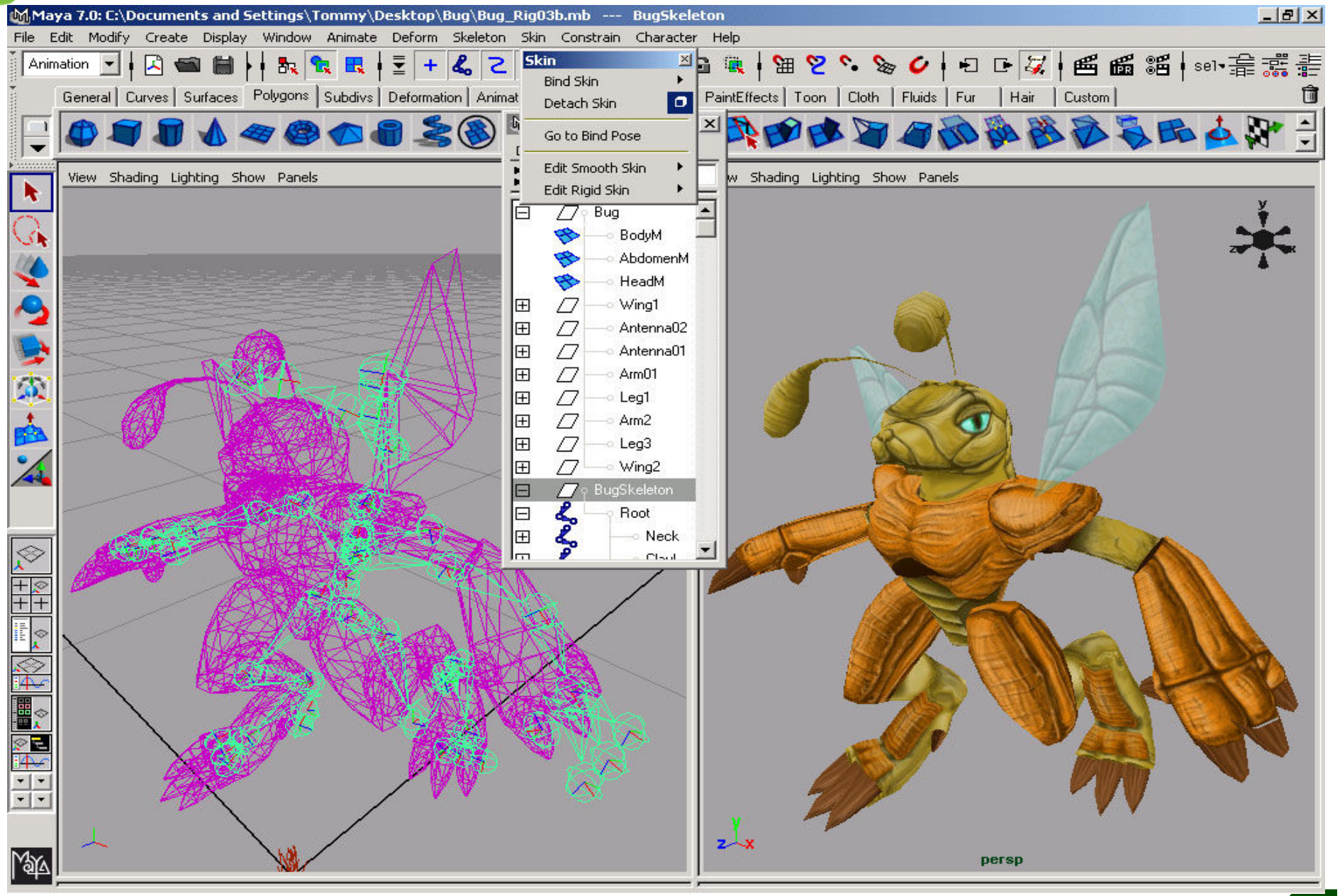
基于Qt开发的软件—KDE(1)

基于Qt开发的软件—KDE(2)

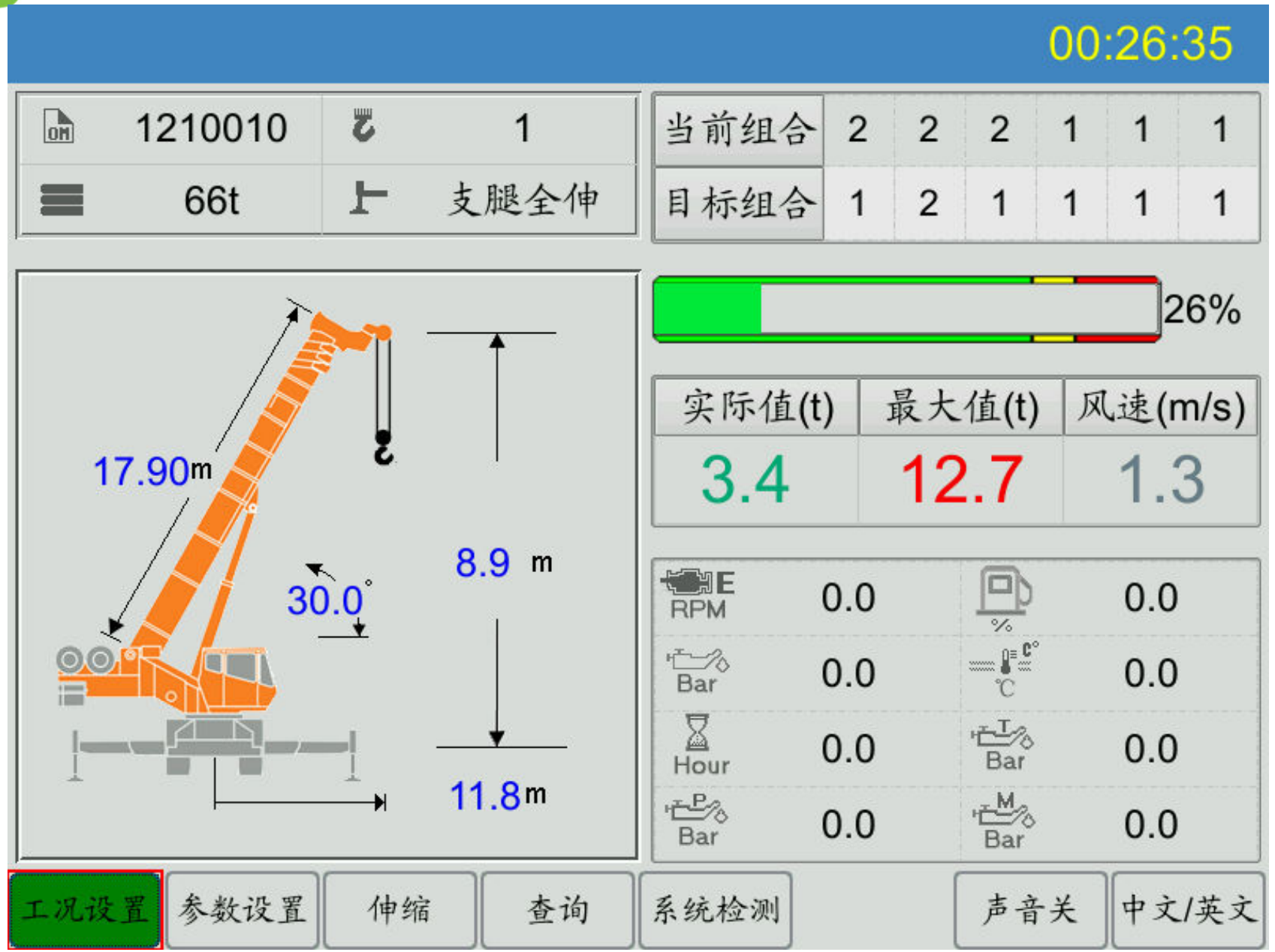
基于Qt开发的软件—Google Earth



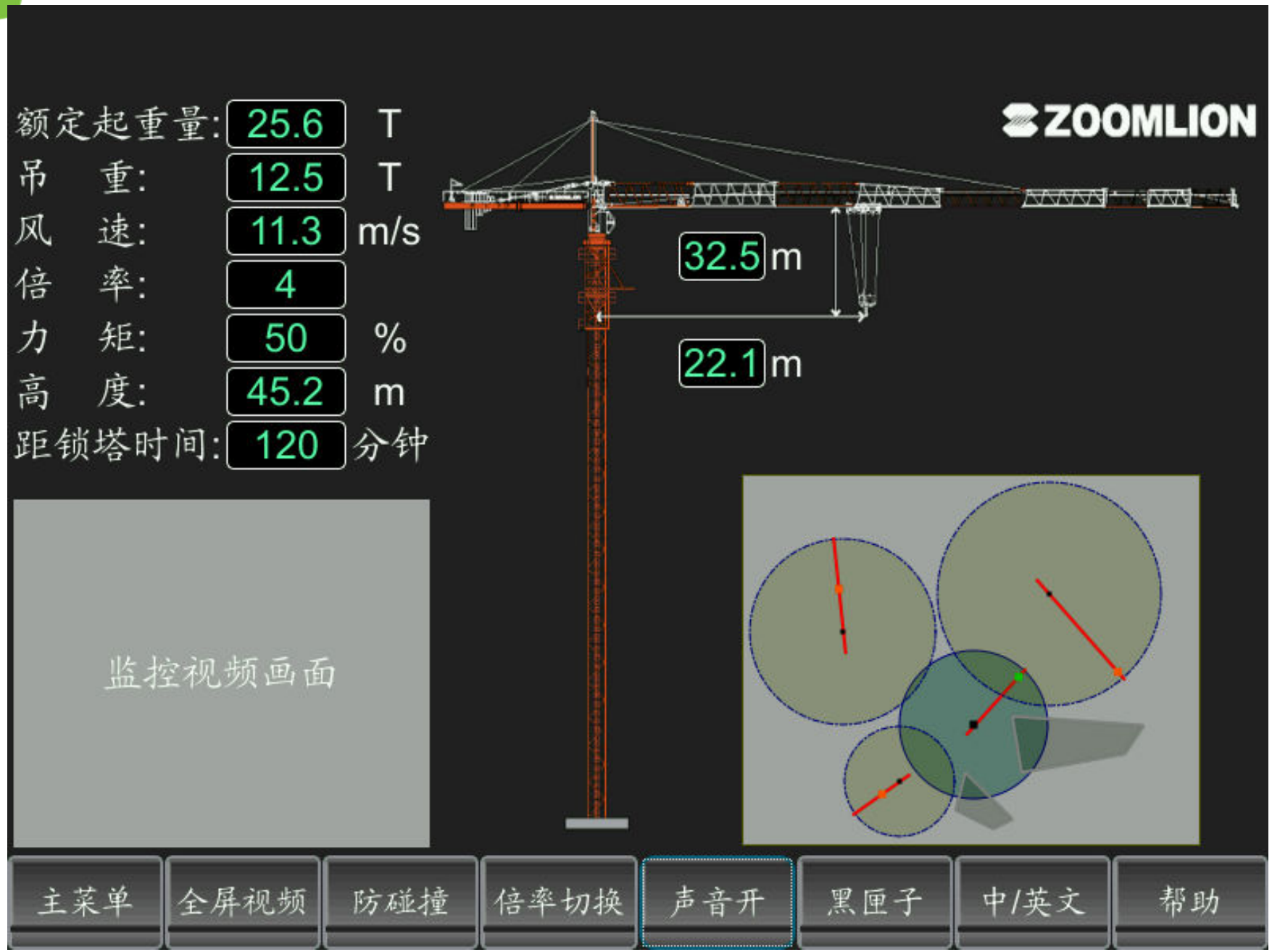
基于Qt开发的软件—MAYA



基于Qt开发的软件-大吨位力矩限制器

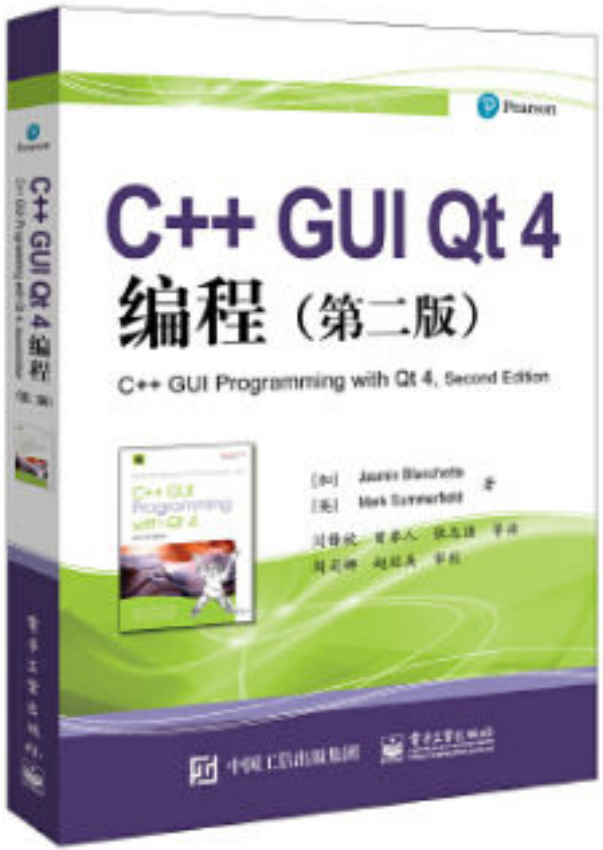


基于Qt开发的软件-塔机智能监控系统



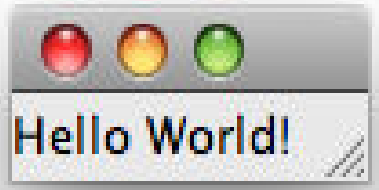
Qt资源推荐

1. 《C++ GUI Programming with Qt 4》，Second Edition，Prentice Hall
2. Qt Assisant—Qt帮助文档
3. Qt Demo



- Qt简介
- Qt的应用
- Qt的使用
- Qt深入理解

Qt的代码结构--Hello World



Qt的代码结构--Hello World

```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```

- 源码特点：**简明**（面向对象编程）、**代码量少**

Qt的代码结构--Hello World

```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```

- 所包含的文件：**文件名为类的名字，无“.h”结尾**

Qt的代码结构--Hello World

```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```

- 任何GUI应用程序，必须始终有一个QApplication对象，必须由main函数明确进行实例化，管理主事件循环、应用程序的初始化、管理全局设置等。对于非GUI应用程序，则使用QCoreApplication替代。

Qt的代码结构--Hello World

```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```

- QLabel 是一个标签部件， 文本在部件显现之前被传递到构造函数。每一样事物都是由部件建造。部件可以是标签、按钮、滚动条、组框、窗口 等等。

Qt的代码结构--Hello World

```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```

- 调用exec启动事件循环，这使得一切都运行起来。当最后一个窗口结束时（可以是关闭），事件循环结束。
- 事件循环一旦开始，你必须改变思维观念。这里一切都是事件驱动的，可以是用户交互（按键或鼠标），网络或计时器。

Qt开发工具集

1. **Qt Creator**（集成开发环境，含 Designer和Assisant）
2. Qt Designer（界面工具）
3. Qt Assisant（帮助文档）
4. Qt Linguist（翻译工具）
5. Qt Demos（示例程序）

信号(Signal)与槽(Slot)初探

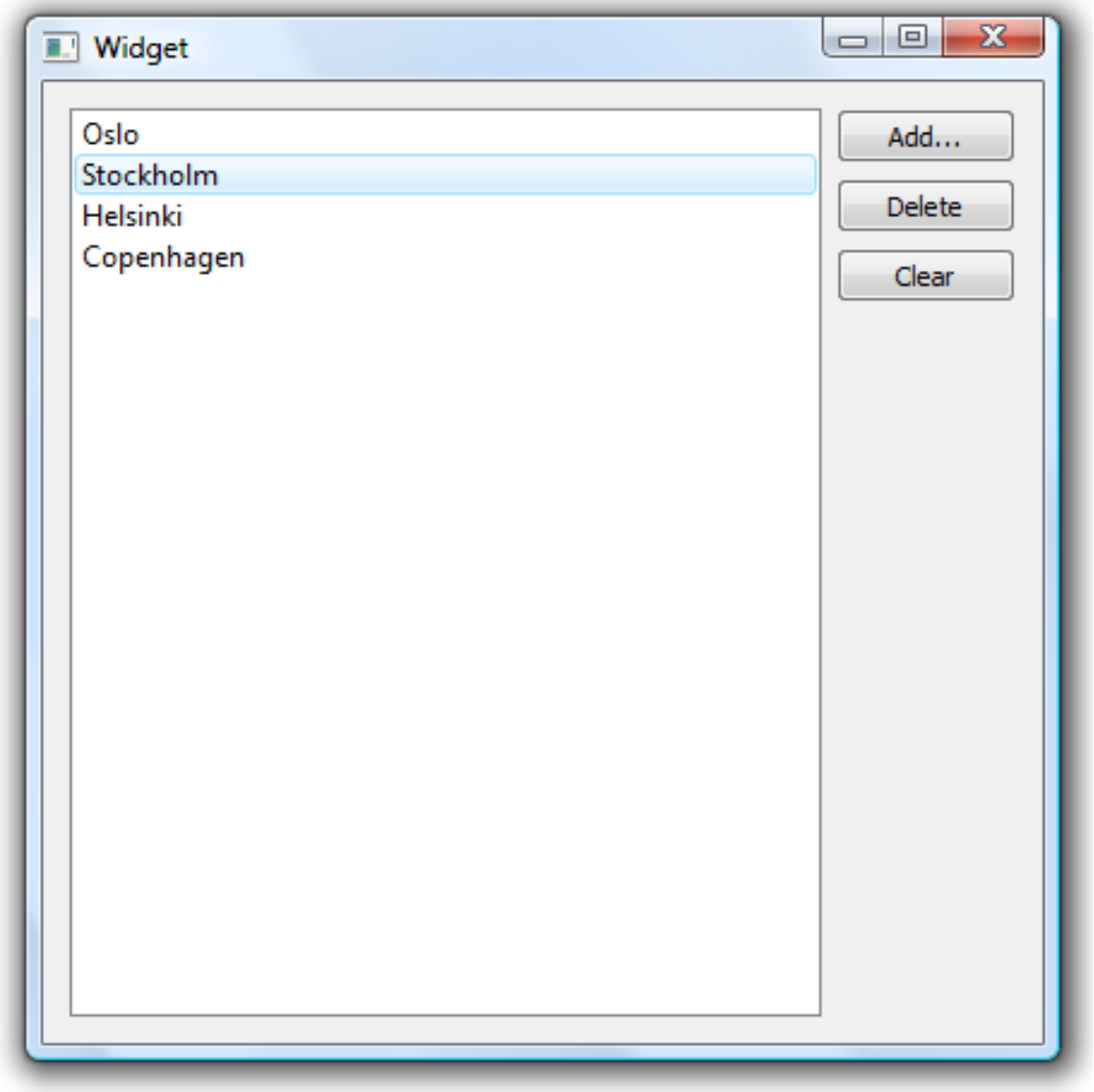
- QT提供的一种在对象间进行通讯的技术：**信号和槽机制**，它是Qt运行的关键技术。
- 它是一种动态并松散地以反应方式绑定事件和状态更改的机制。

- 动态地 = 在运行时
- 松散地 = 发送者和接收者并不知晓对方
- 事件 = 定时器时间，点击等等，不会被实际事件（QEvent）打乱
- 状态更改 = 尺寸更改，文本更改，值更改等等
- 反应 = 实际做事的代码

信号(Signal)与槽(Slot)初探

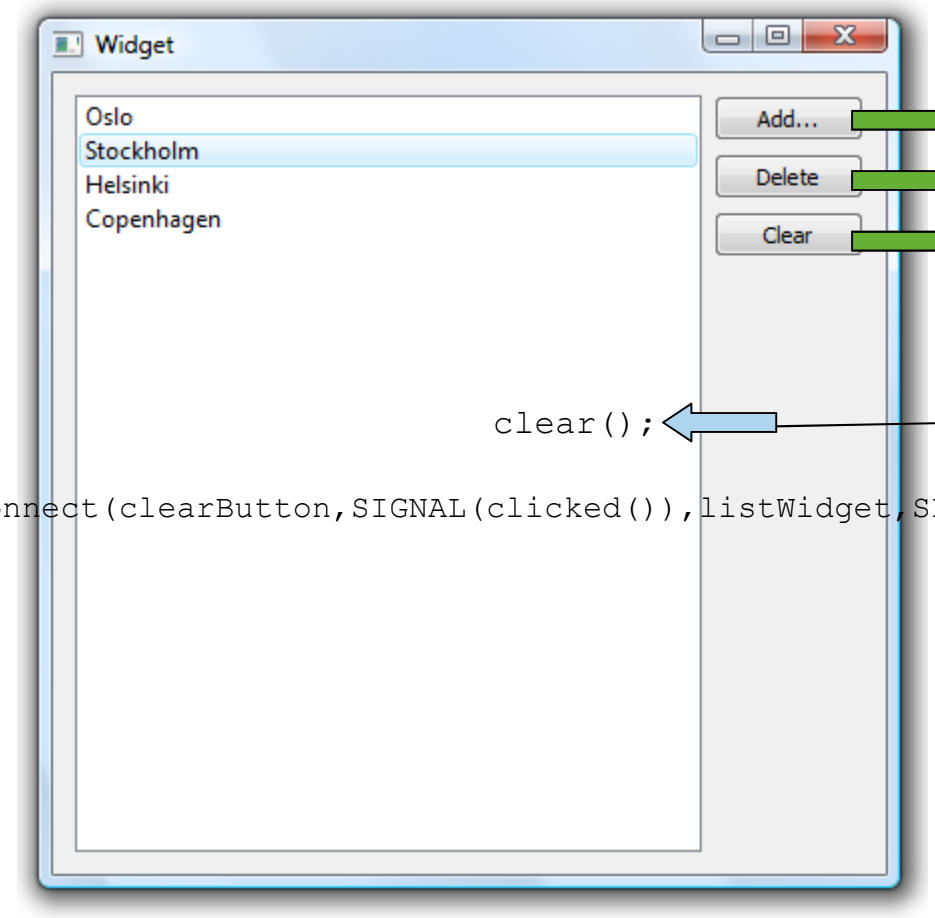
- ✓ 类似于windows中的消息和消息响应
- ✓都是通过C++类成员函数实现的
- ✓信号和槽是通过连接实现相互关联的
- ✓包含信号或槽的类必须从Qobject类继承

信号(Signal)与槽(Slot)初探




```
emit clicked();
```

信号(Signal)与槽(Slot)初探



2x `connect(addButton,SIGNAL(clicked()),this,SLOT(...));`

```
private slots:
    void on_addButton_clicked();
    void on_deleteButton_clicked();
```

`clear();`

`connect(clearButton,SIGNAL(clicked()),listWidget,SLOT(clear()));`

信号(Signal)与槽(Slot)初探


```
{
    ...
    emit clicked();
    ...
}
```

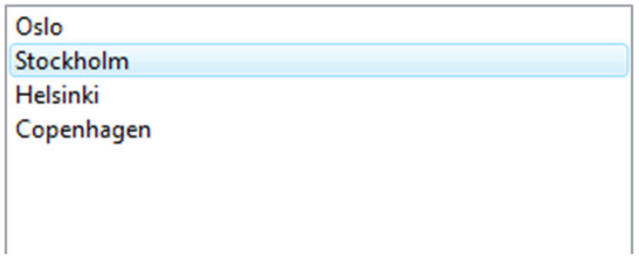
```
{
    QString newText =
        QInputDialog::getText(this,
                              "Enter text", "Text:");

    if( !newText.isEmpty() )
        ui->listWidget->addItem(newText);
}
```


```
{
    ...
    emit clicked();
    ...
}
```

```
{
    foreach (QListWidgetItem *item,
             ui->listWidget->selectedItems())
    {
        delete item;
    }
}
```


```
{
    ...
    emit clicked();
    ...
}
```

```
clear();
```

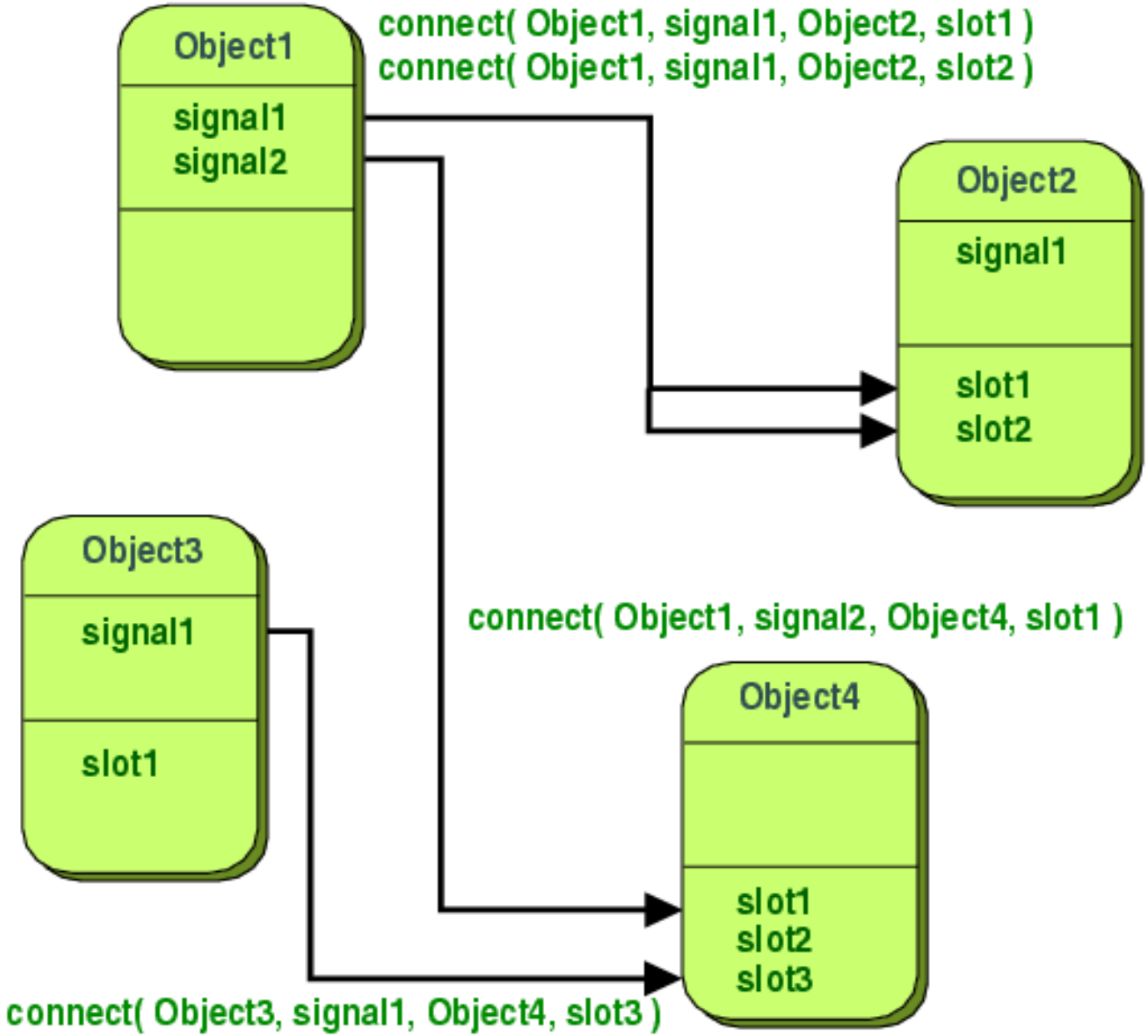


信号(Signal)与槽(Slot)初探

信号和槽的声明：

```
class Employee : public QObject
{
    Q_OBJECT
    public:
        Employee();
        int salary() const;
    public slots:
        void setSalary(int newSalary);
    signals:
        void salaryChanged(int newSalary);
    private:
        int mySalary;
};

emit salaryChanged(50);
```

信号(Signal)与槽(Slot)初探



信号(Signal)与槽(Slot)初探

- 信号与槽机制只能用在继承于QObject的类。
- 槽可以返回值，但通过联接返回时不能有返回值，槽以一个普通的函数实现，可以作为普通函数调用。
- **信号总是返回空**，信号总是不必实现
- 一个信号可以连接到多个槽，但槽的调用顺序不确定。
- **信号和槽需要具有相同的参数列表**；如果信号的参数比槽多，那么多余的参数会被忽略；如果参数列表不匹配，Qt会产生运行时错误信息。

信号和槽 vs 回调

回调（callback）是一个函数指针，当一个事件发生时被调用，任何函数都可以被安排作为回调。

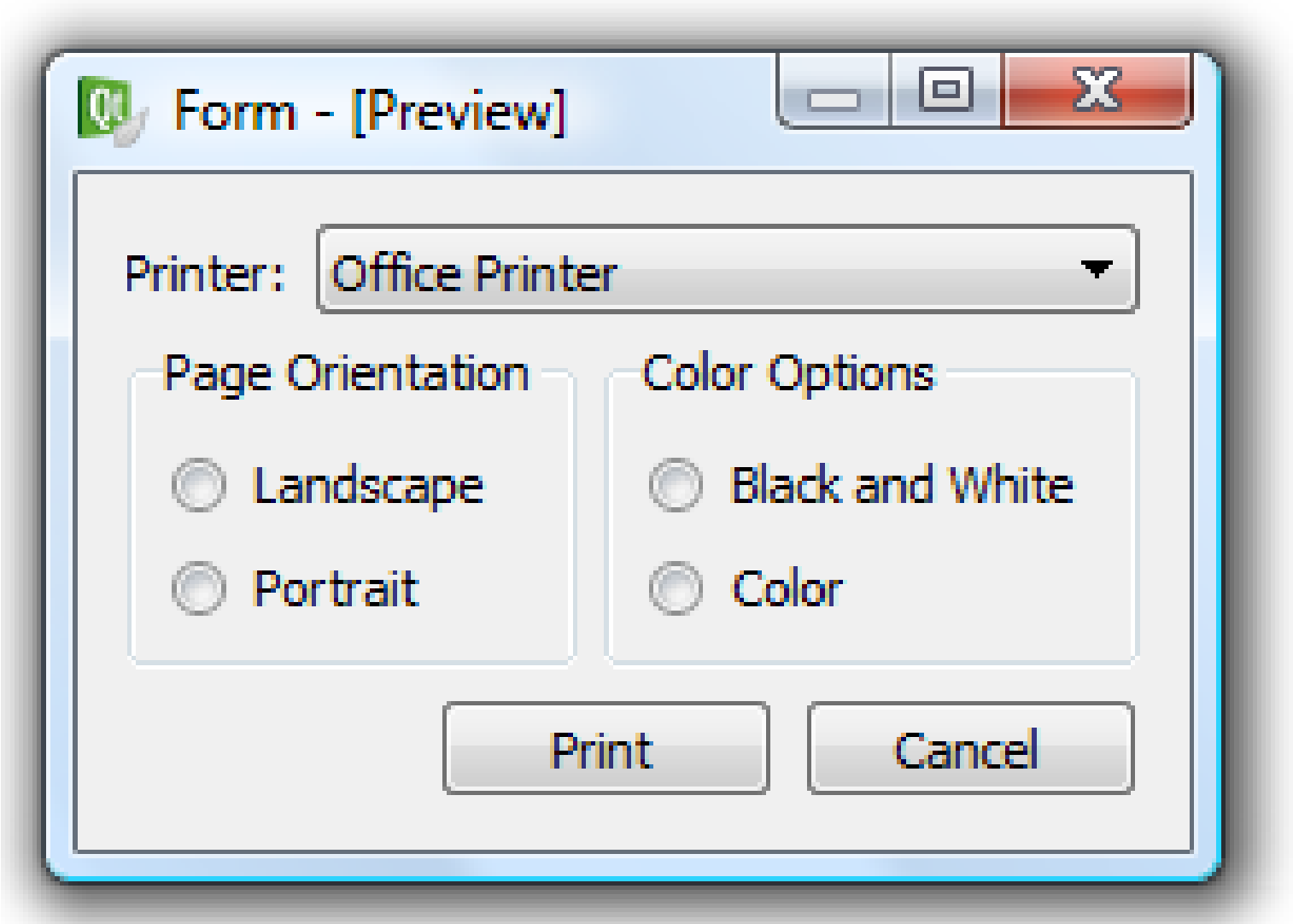
- 没有类型安全
- 总是以直接调用方式工作（服务面向过程编程）

信号和槽的方式更加动态

- 一个更通用的机制
- 更容易互连两个已存在的类（服务面向对象编程）
- 相关类之间涉及更少的知识共享

用户界面设计

用户界面由特定的部件（widget）构建



用户界面设计—三种方式

1.绝对定位(absolute positioning)

- 用户不能改变界面的大小
- 对部件的大小、位置进行硬编码，在翻译或者改变风格的时候，会出现字体溢出等问题

2. 手工布局(manual layout)

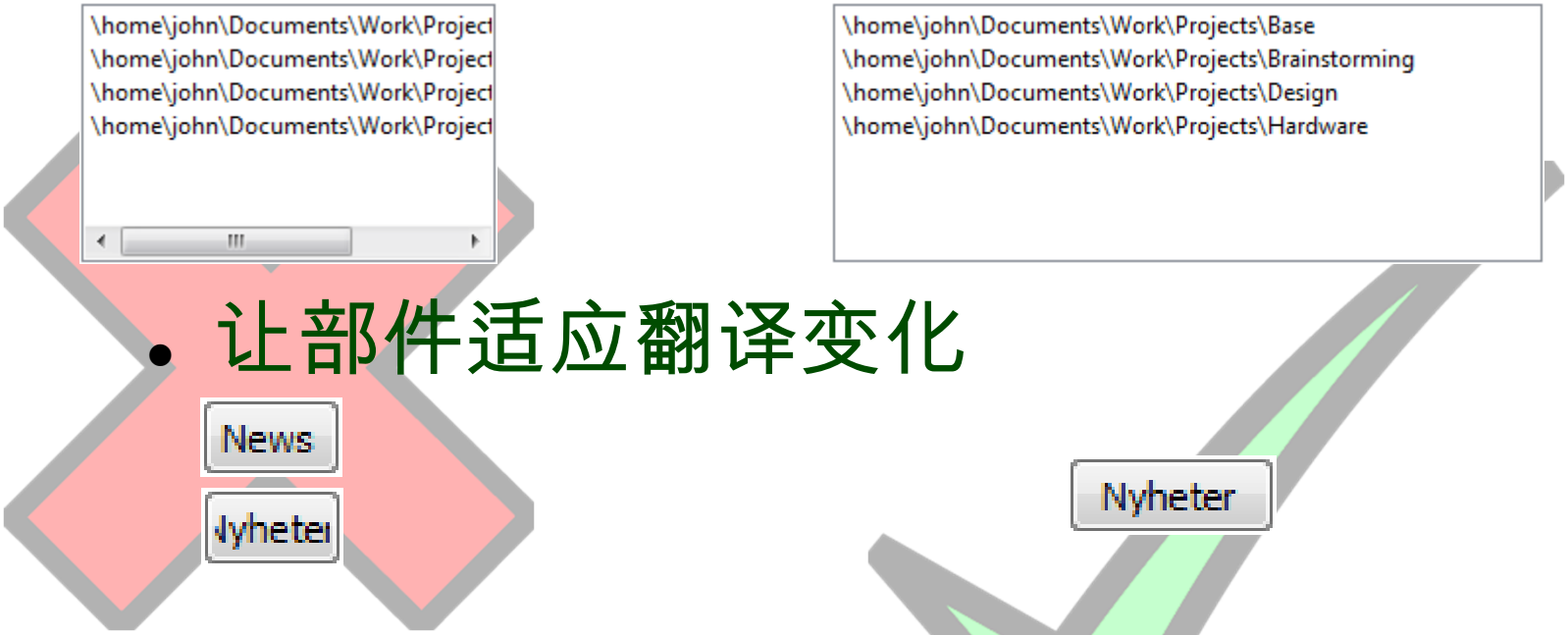
- 绝对位置，但通过resizeEvent()方法改变大小
- 仍然需要大量的硬编码

3.布局管理器(layout managers)

- 部件放置在布局管理器中，使界面更具弹性。
- 当界面大小、风格、语言等发生变化时，均可以自动调整大小，而且不需要编码。

布局管理器的优点？

- 让部件适应内容



- 让部件适应翻译变化

- 让部件适应用户设置

布局管理

- 几种可用的布局

QVBoxLayout

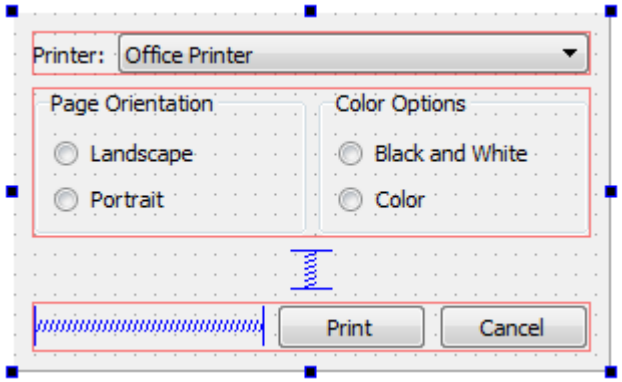
QHBoxLayout

QGridLayout

- 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 

布局管理器示例

- 对话框由多层的布局管理器和部件组成



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

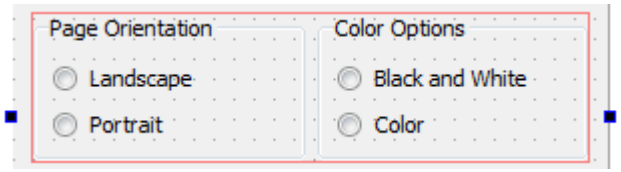
注意：布局管理器并不是其管理的部件的父对象

- 两种实现方式：代码实现，使用设计器

布局管理器—代码实现

```
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);
```



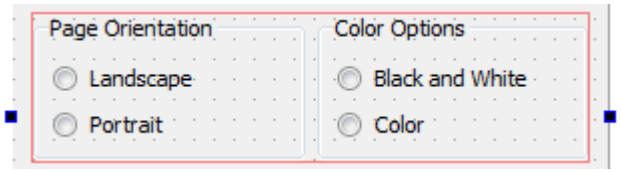

```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

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

布局管理器—代码实现

```
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);
```



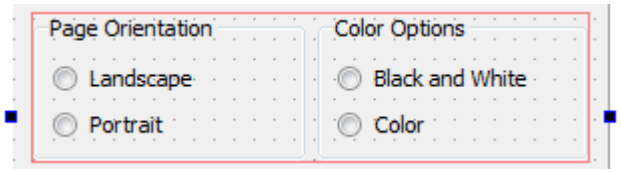

```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

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

布局管理器—代码实现

```
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);
```



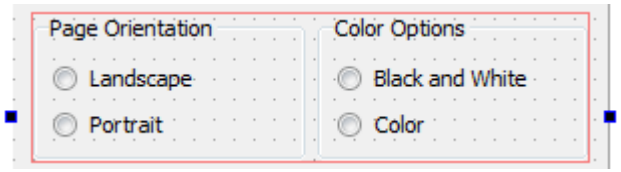

```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

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

布局管理器—代码实现

```
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);
```



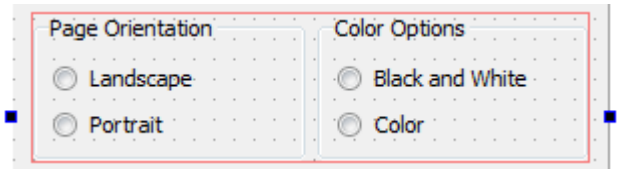

```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

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

布局管理器—代码实现

```
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);
```

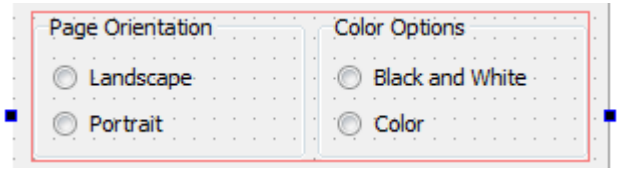



```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

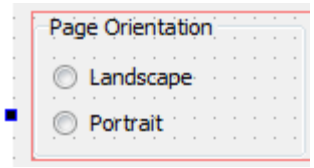
```
QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);
```

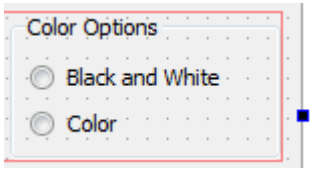
布局管理器—代码实现

- 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);
```





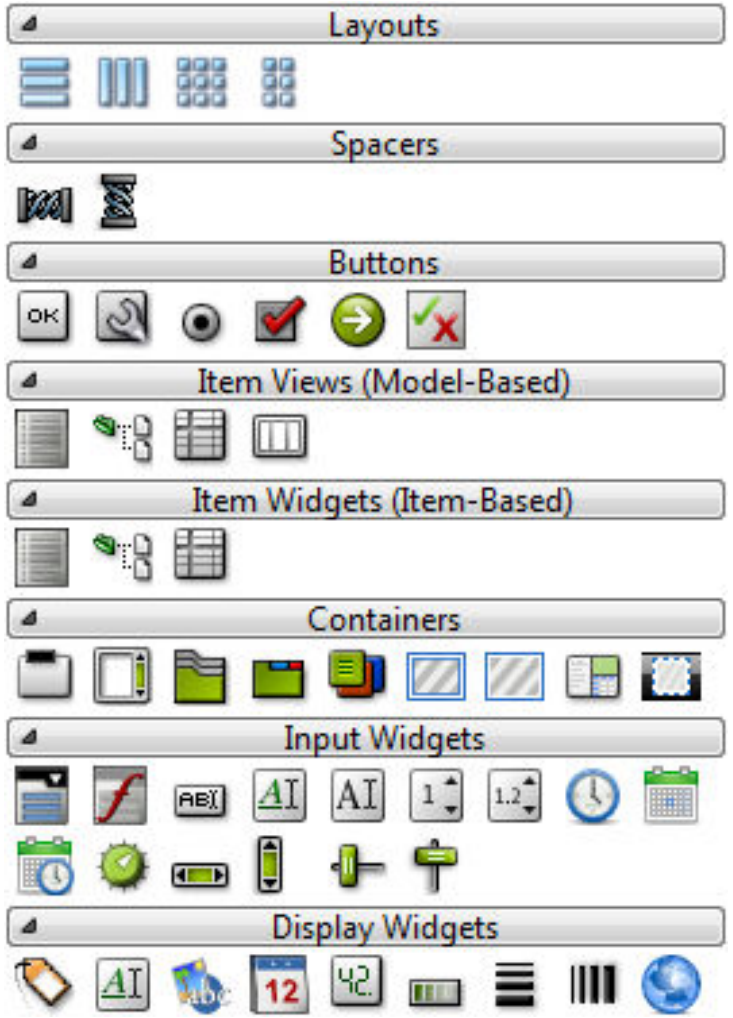
布局管理器—设计器

- 可以使用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

- Qt包含针对所有情形的大量通用部件；
- 第三方控件，如QWT
- 自定义控件

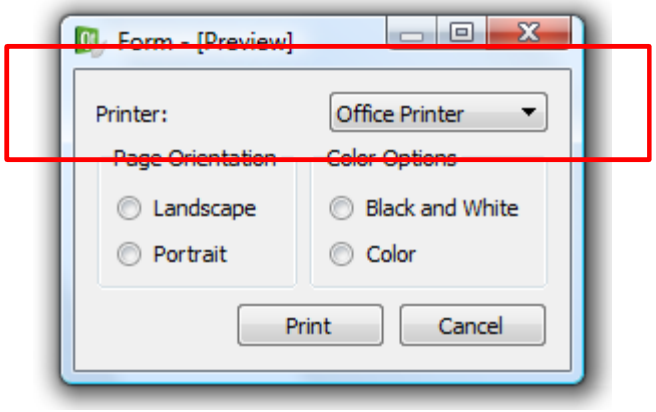


尺寸（size）的策略

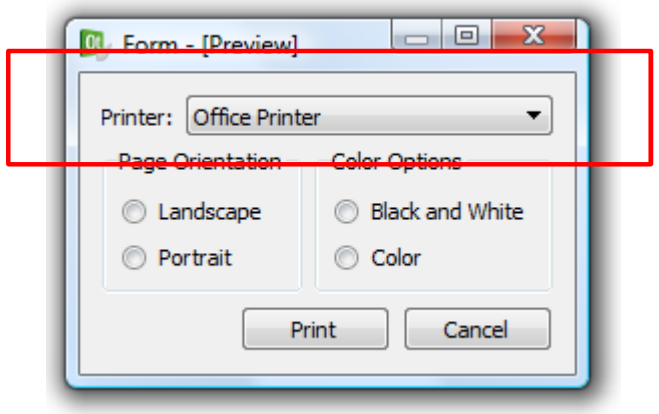
- 布局管理器是在空间和其他布局管理器之间进行协调
- 布局管理器提供布局结构
  - 水平布局和垂直布局
  - 网格布局
- 部件则提供
  - 各个方向上的尺寸策略
  - 最大和最小尺寸

尺寸的策略

- 例子未完成!




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



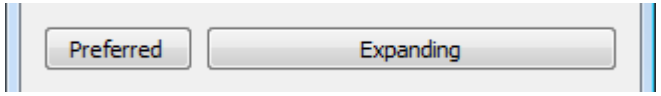
尺寸的策略

- 每一个widget有一个大小的示意，它给出了各个方向上尺寸的策略
  - `Fixed` –规定了widget的尺寸
  - `Minimum` – 规定了可能的最小值
  - `Maximum` – 规定可能的最大值
  - `Preferred` – 给出最好的值但不是必须的
  - `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

不同尺寸属性的竞争

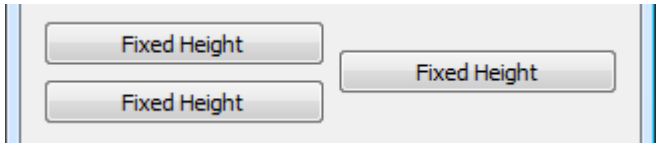
- 2个 preferred 相邻

- 1个 preferred, 1个 expanding



- 2个 expanding 相邻

- 空间不足以放置widget (fixed)



关于尺寸的更多信息

- 可用最大和最小属性更好地控制widget的大小

- `maximumSize` –最大可能尺寸
- `minimumSize` –最小可能尺寸

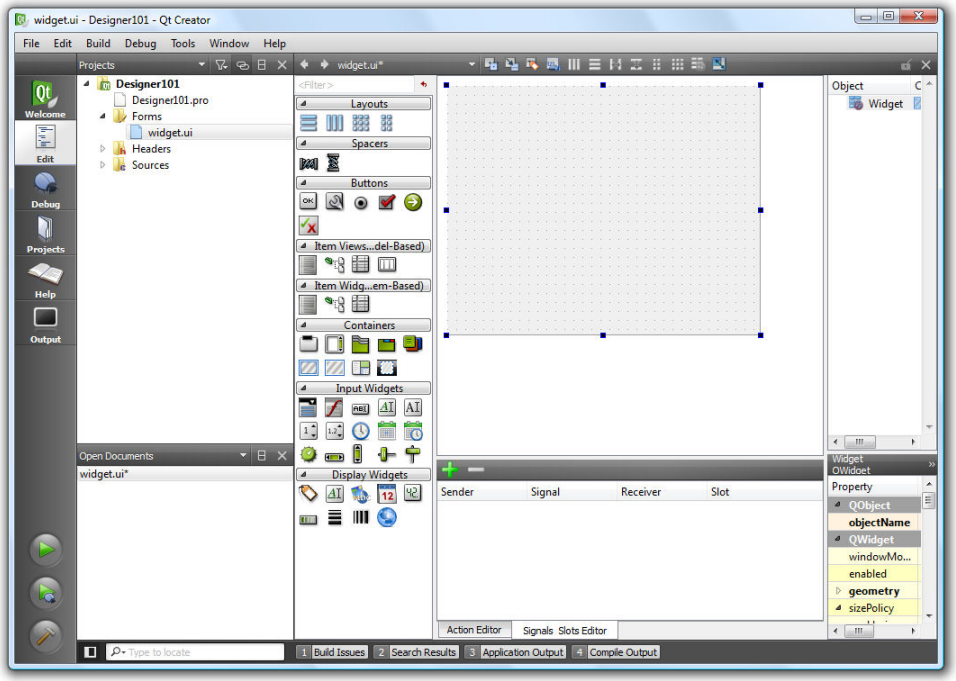
```
ui->pushButton->setMinimumSize(100, 150);
ui->pushButton->setMaximumHeight(250);
```

伸缩因子

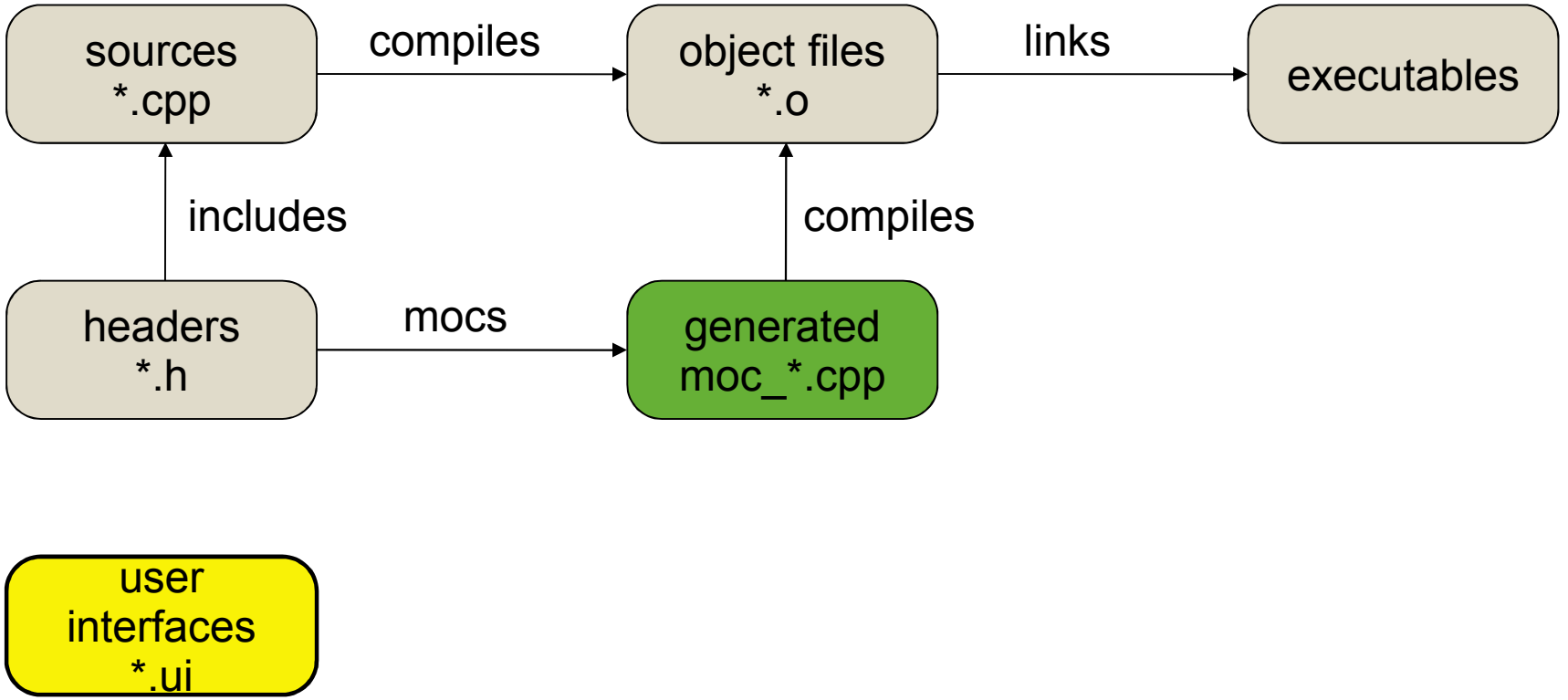
控制缩放时，各控件的缩放比例。

设计器介绍

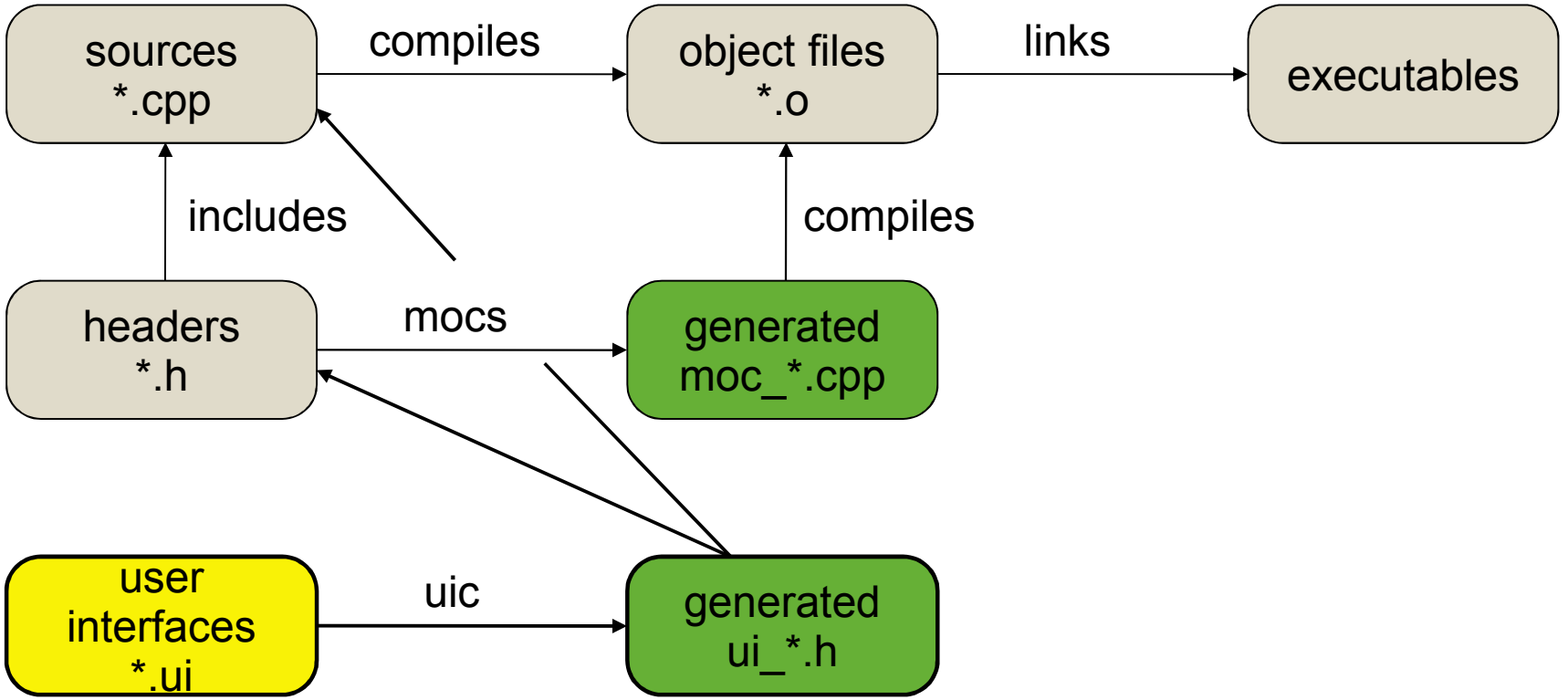
- 以前设计器（Designer）是一个独立的工具，但现在是QtCreator的一个组成部分
- 可视化窗体编辑器
- 拖放部件
- 安排布局
- 进行信号连接



设计器介绍


sources
*.cpp
compiles
object files
*.o
links
executables
includes
compiles
headers
*.h
mocs
generated
moc_*.cpp
user
interfaces
*.ui

设计器介绍


sources
*.cpp
compiles
object files
*.o
links
executables
includes
compiles
headers
*.h
mocs
generated
moc_*.cpp
user
interfaces
*.ui
uic
generated
ui_*.h

使用代码

```
#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
```

Ui::Widget类的前置声明

一个 Ui::Widget 类指针 ui ，指向所有部件

基本上一个标准的 QWidget 派生类

使用代码

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

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

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

调用函数 setupUi，生成所有父窗体 (this) 的子窗体部件

实例化类 Ui::Widget 为 ui

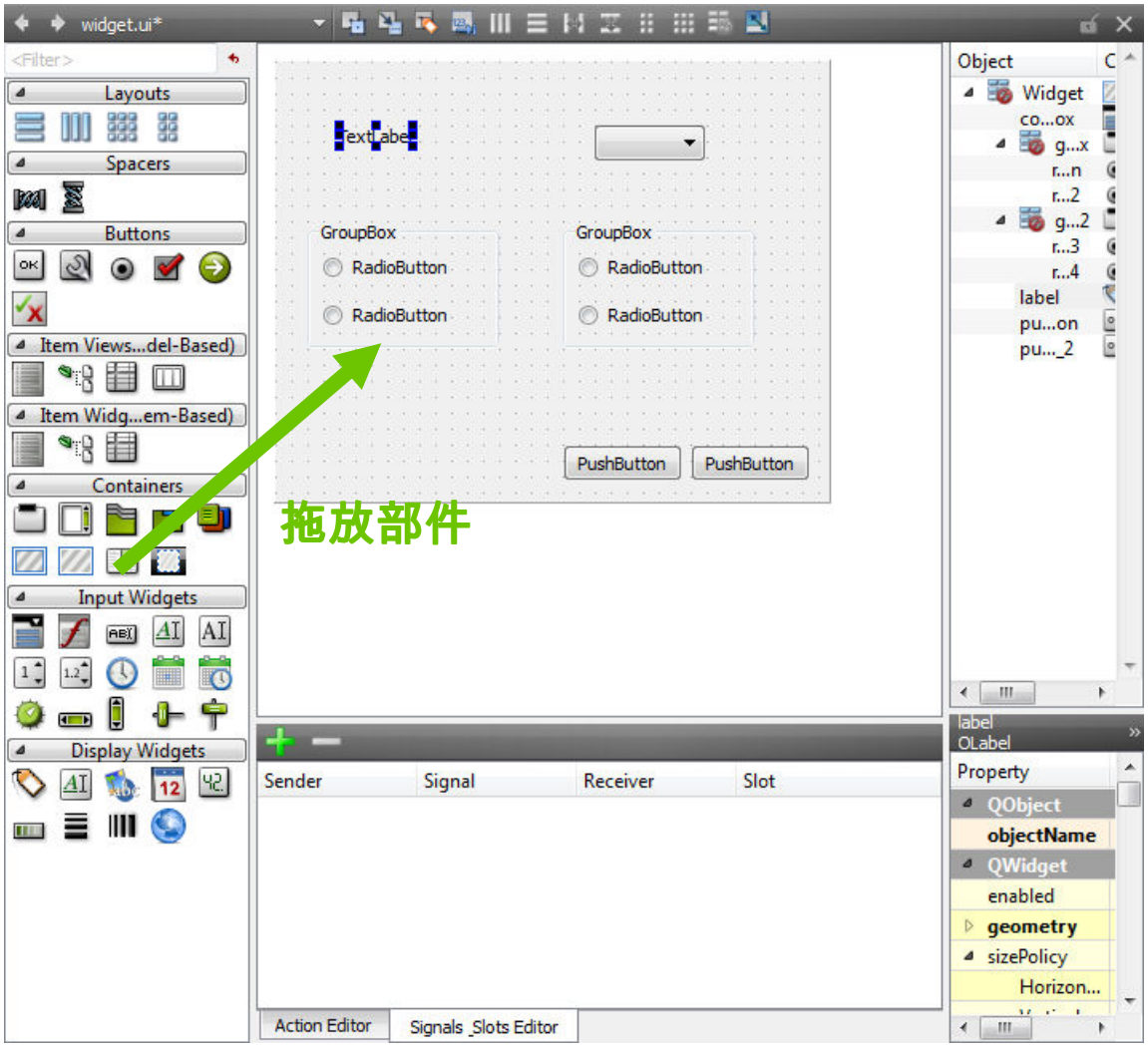
删除 ui对象

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!

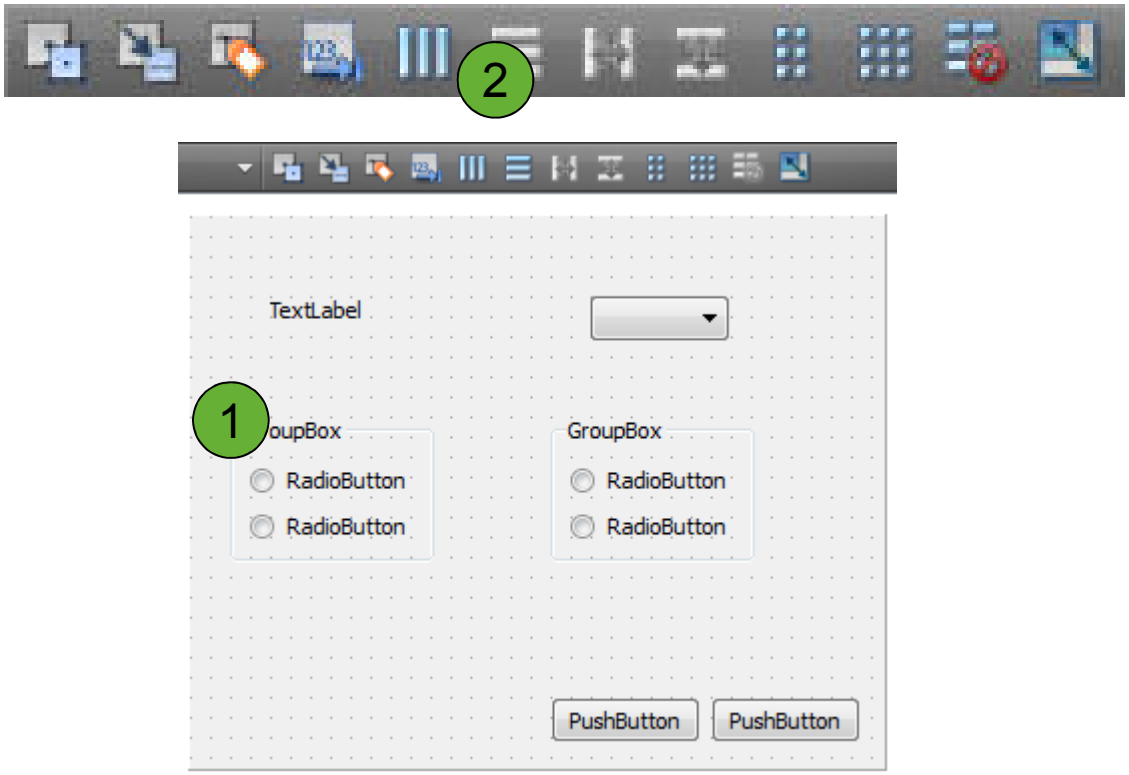
使用设计器

粗略地放置部件在窗体上



使用设计器

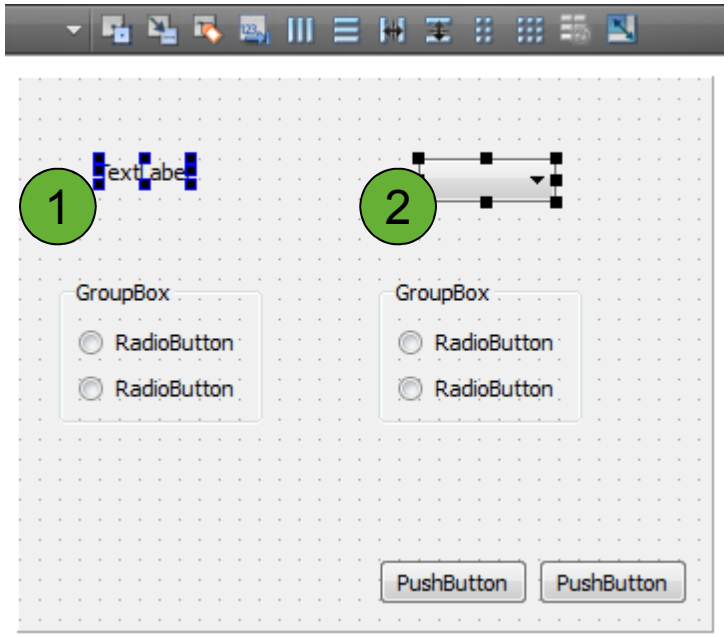
从里到外进行布局，添加必要的弹簧



1. 选中每一个 group box, **2**. 应用垂直布局管理

使用设计器

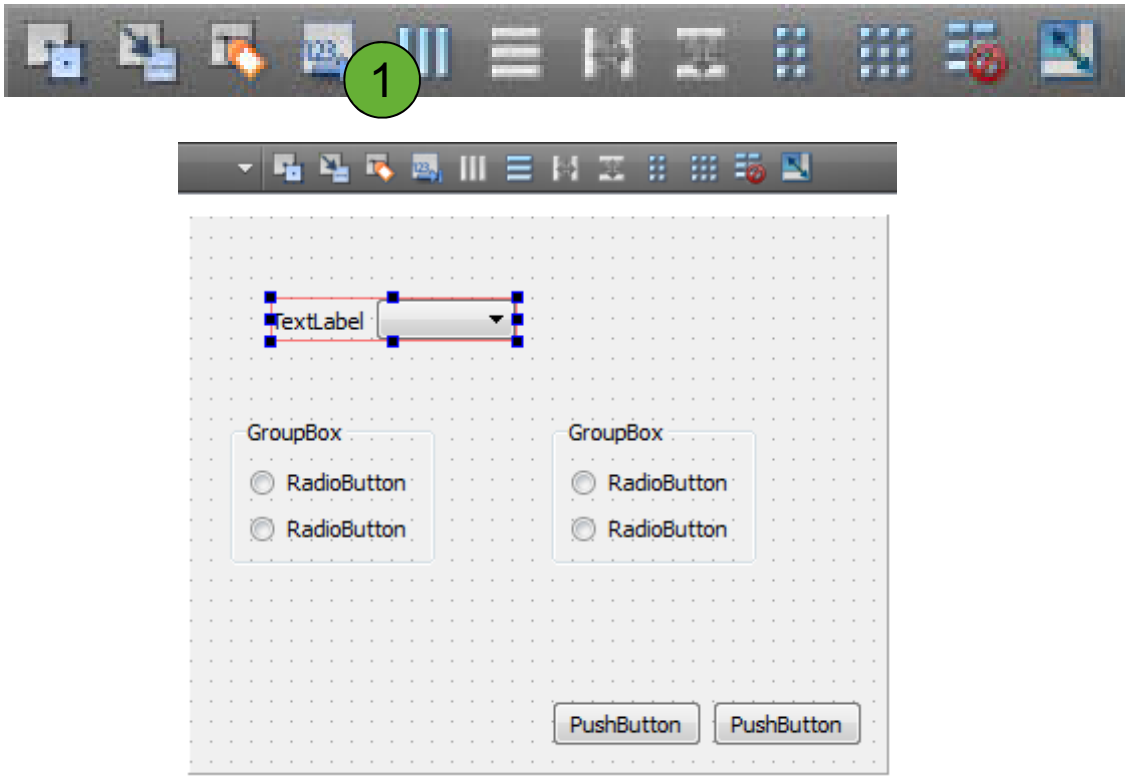
从里到外进行布局，添加必要的弹簧



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

使用设计器

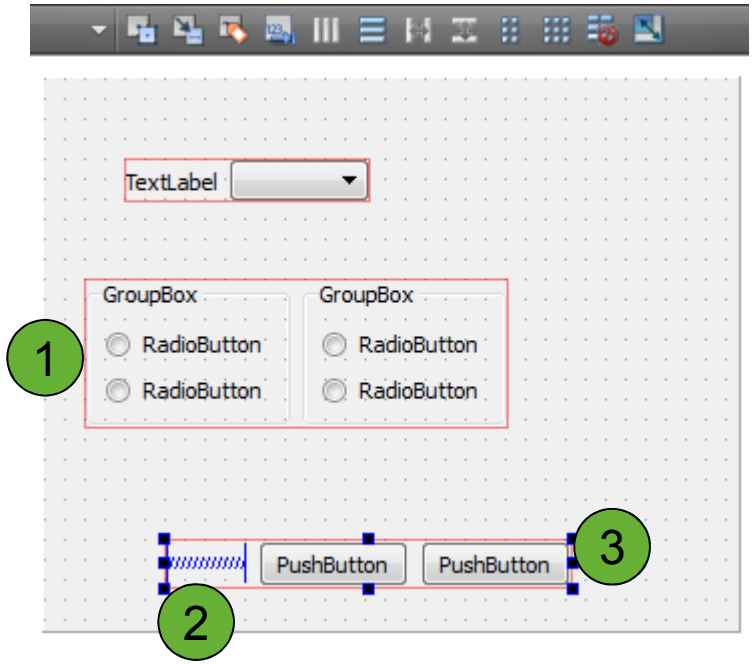
从里到外进行布局，添加必要的弹簧



1. 应用一个水平布局管理

使用设计器

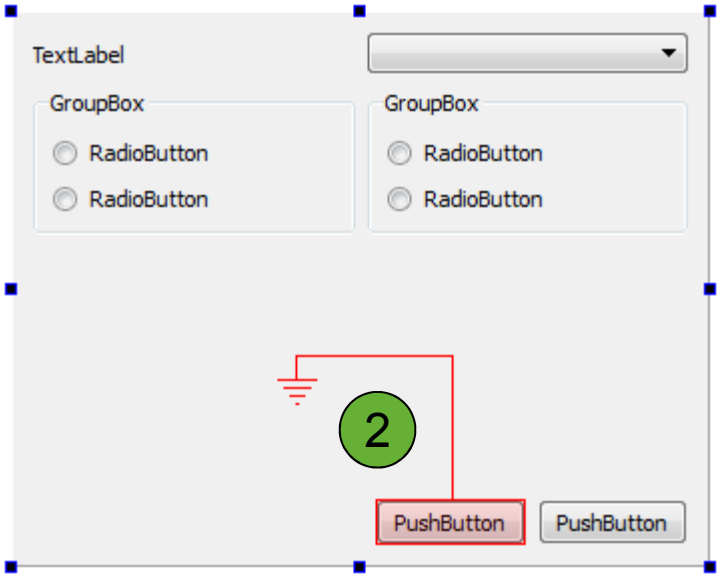
从里到外进行布局，添加必要的弹簧

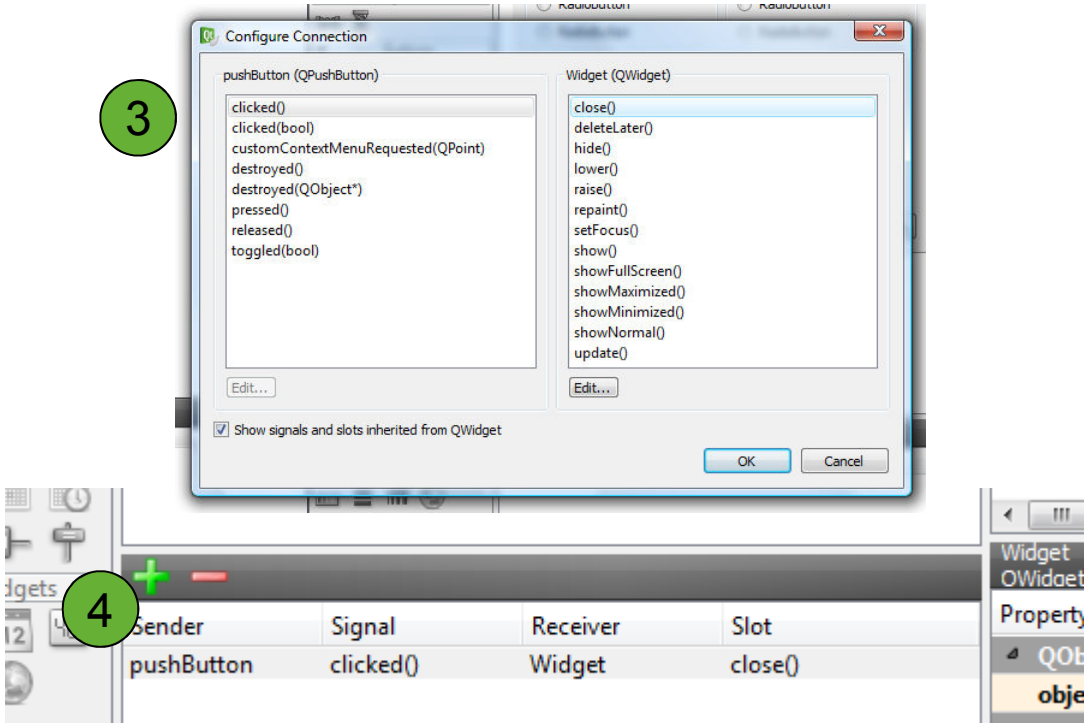


1. 选中2个group box并进行布局管理, **2**. 添加一个水平弹簧,
3. 将弹簧和按钮放置进一个布局管理中

使用设计器

进行信号连接(部件之间)

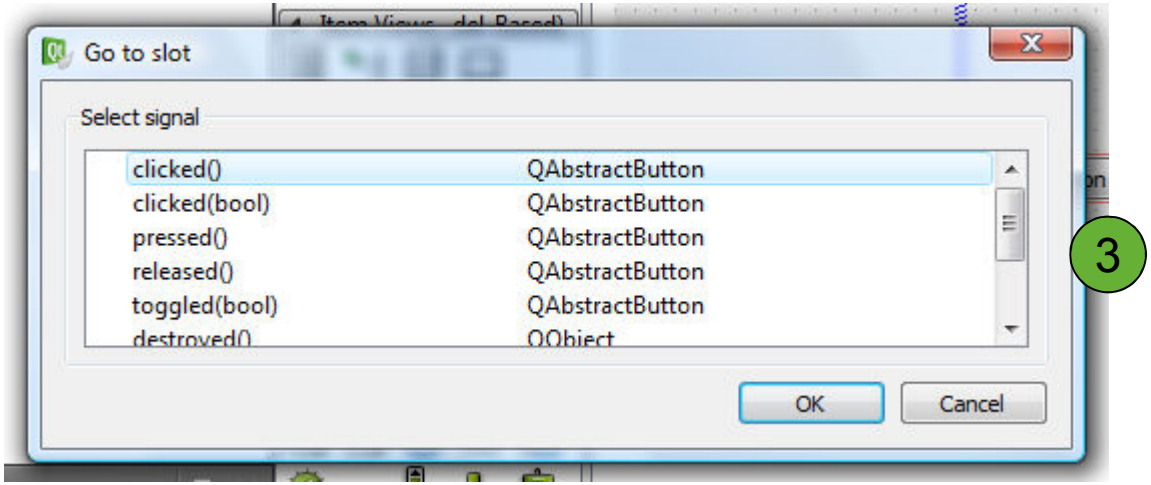


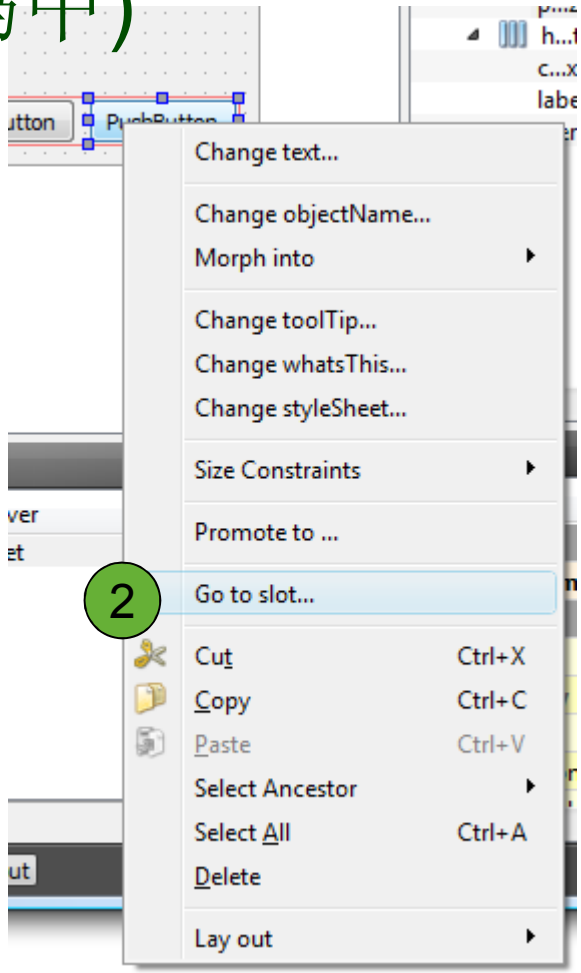


1. 转到signals and slot 编辑模式, **2**. 从一个部件拖放鼠标到另一个部件,
3. 选中signal and slot, **4**. 在connections‘ dock中查看结果

使用设计器

进行信号连接(到你的代码中)





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类成员使用所有部件

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

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

界面美化

1.子类化已有的控件类，重新实现paintEvent()、MouseEvent()等方法；

2.子类化QStyle，或者使用已经定义了的QWindowStyle等。

3.使用StyleSheet。

样式表(StyleSheet)

- 所有的 `QWidget` 类都有一个 `styleSheet` 属性以支持跨平台样式
- 样式表是受启发自CSS的
- 它们可以用来进行高亮处理并进行许多小的修改

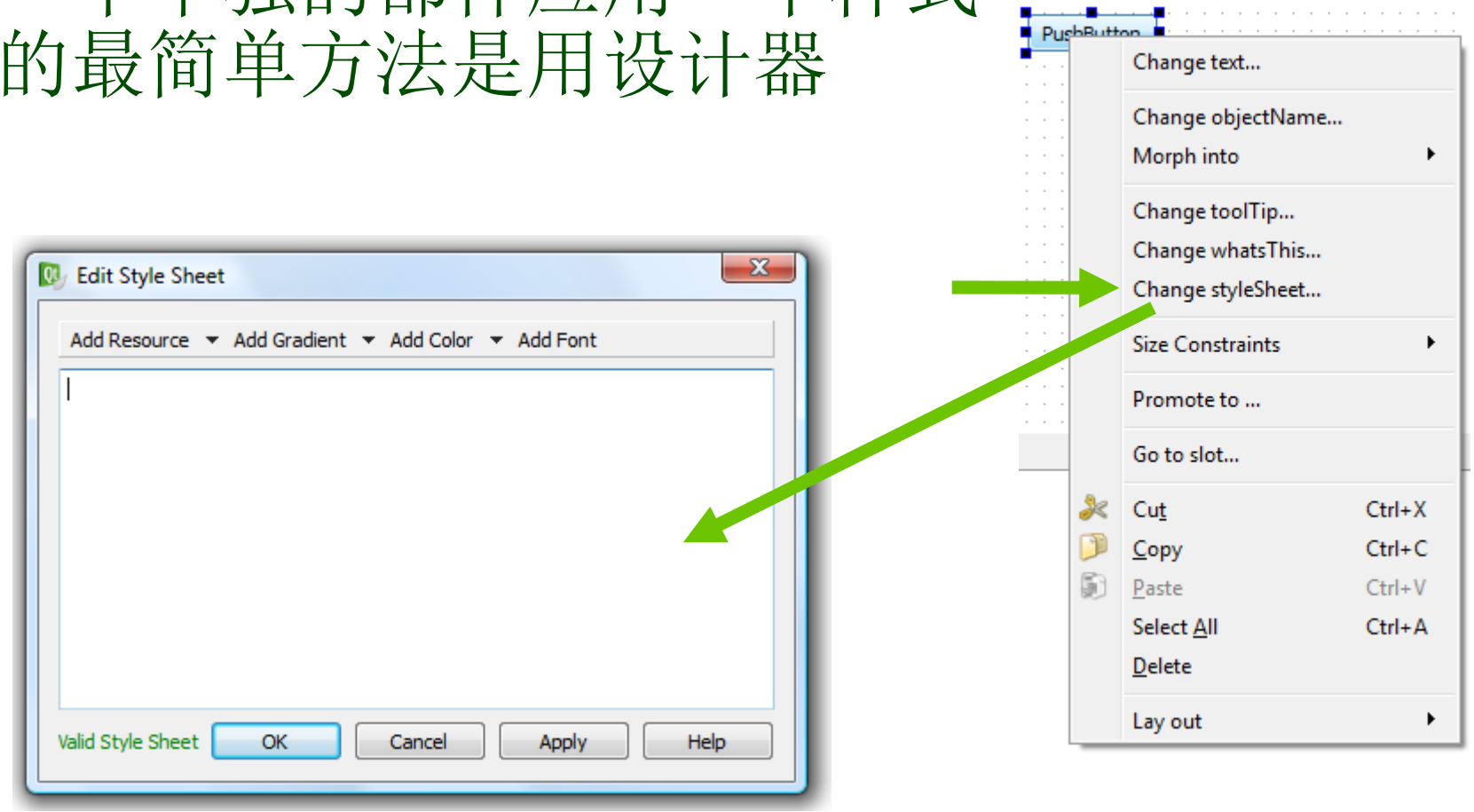
Hello World　　Hello World

- 当然也可以用于用户界面的整体修改

PushButton

样式表

- 为一个单独的部件应用一个样式表的最简单方法是用设计器



样式表

- 想为整个应用程序设定样式，可以使用 `QApplication::setStyleSheet`

选择一个类

```
QLineEdit { background-color: yellow }
QLineEdit#nameEdit { background-color: yellow }

xtEdit, QListView {
    background-color: white;
    background-image: url(draft.png);
    background-attachment: scroll;
}

QGroupBox {
    background-color: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
                                      stop: 0 #E0E0E0, stop: 1 #FFFFFF);
    border: 2px solid gray;
    border-radius: 5px;
    margin-top: 1ex;
}
```

通过对象名选择一个对象

使用图片

在Designer的编辑器中建立这些

资源文件(qrc)

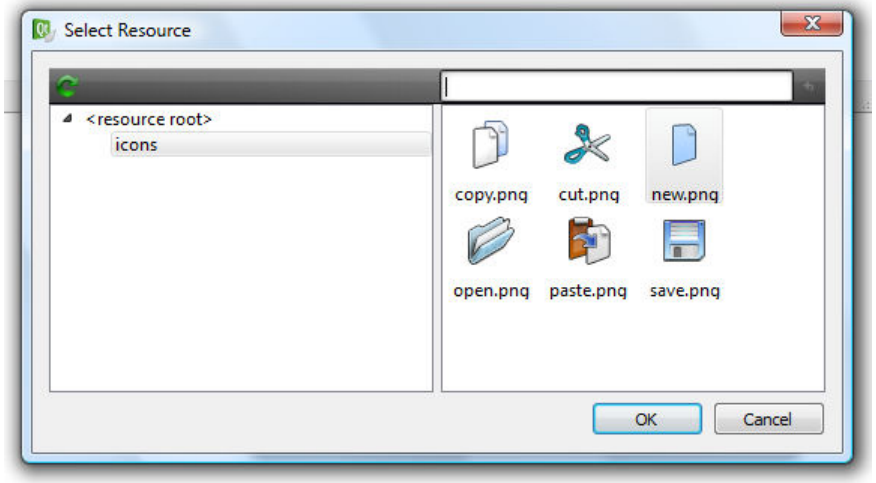
- 将图标放进一个资源文件中，Qt会将它们内嵌进可执行文件
  - 避免调用多文件
  - 不需要尝试确定每个特定安装风格下的图标的路径
  - 一切都巧妙地在软件构建系统中自适应
  - 避免部署的时候出现文件丢失的错误
  - 可以将任何东西添加进资源文件中，不仅仅是图标，但一般是不需要修改的文件。

资源文件(qrc)

- 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的国际化

1. 确保应用程序是可翻译的：
   - 所用用户可见的字符串都使用tr()修饰
   - 根据不同的目标语言加载不同的qm的文件。
2. 即使应用程序目前不需要翻译，也应该为以后的需求留出余地。

Qt国际化—步骤

1.在代码中使用tr()修饰用户可见的字符串；

2.lupdate提取需要翻译的字符串；

```
TRANSLATIONS = spreadsheet_cn.ts \
               spreadsheet_en.ts
```

3.使用linguist工具翻译；

4.在程序开始时加载正确的qm文件。

Qt国际化—加载qm文件

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
     ...
    QTranslator appTranslator;
    appTranslator.load("myapp_" + QLocale::system().name(), qmPath);
    app.installTranslator(&appTranslator);
    ...
    return app.exec();
}
```

Qt国际化—动态语言切换

额外的工作：

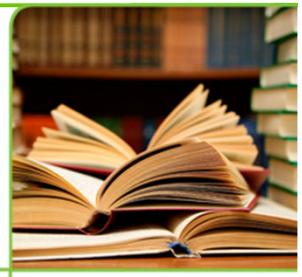
1. 语言切换的途径（菜单、按钮等）；

2. 在统一的方法(RetranslateUI())内处理用户可见字符串，并在语言切换时调用该方法。

3. 实现changeEvent(QEvent *event) 方法

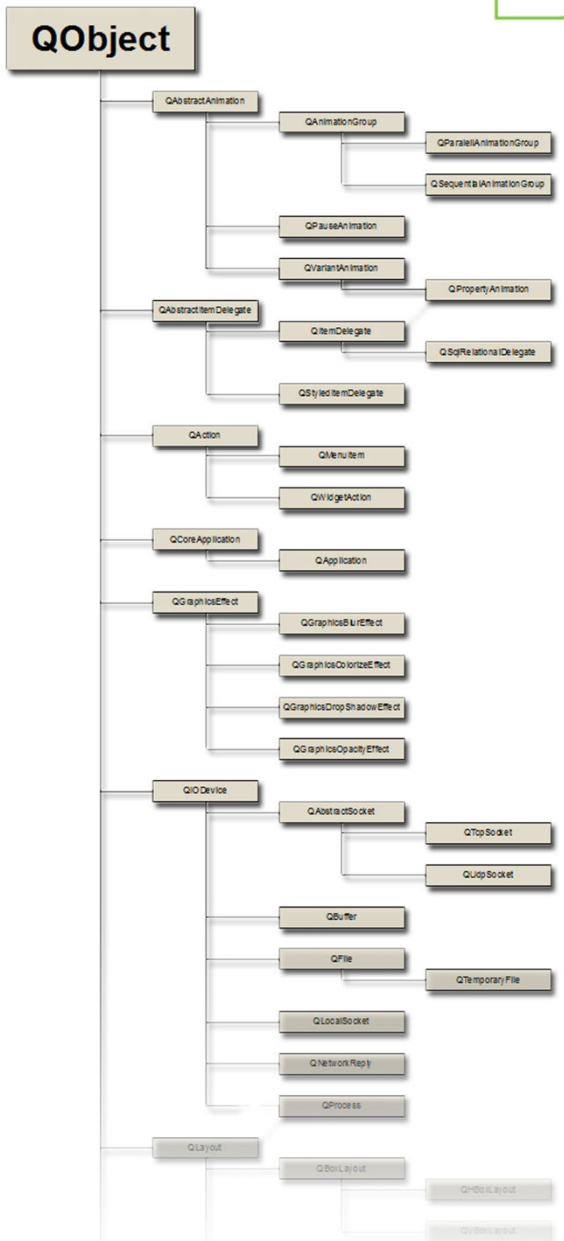
```
void JournalView::changeEvent(QEvent *event)
 {
      if (event->type() == QEvent::LanguageChange)
          retranslateUi();
     QTableWidget::changeEvent(event);
 }
```

- Qt简介
- Qt的应用
- Qt的使用
- Qt深入理解



QObject类

QObject是几乎所有Qt类和所有部件(widget)的基类。



它包含很多组成Qt的机制

- 事件
- 信号和槽
- 属性
- 内存管理

QObject类

QObject是大部分Qt 类的基类

例外的例子是:

类需要作为轻量级的类，例如图元（graphical primitives） QPen、QBrush。

数据容器(QString, QList, QChar等)。

需要可复制的类，因为QObject类是无法被复制的。

QObject类

“QObject 的实例是单独的! 不能复制！”

它们可以拥有一个名字 (`QObject::objectName`)

addButton, lineEdit_Password…. 复制的话就同名了。

它们被放置在QObject实例的一个层次上，无法复制上下文

它们可以有到其他 `QObject` 实例的联接，无法复制联接。

对象数据存储(1)

C++中定义数据变量的一般方法：

```
class Person
{
private:
    string mszName; // 姓名
    bool mbSex;    // 性别
    int mnAge;     // 年龄
};
```

对象数据存储(2)

Qt定义数据变量(Qt 2.x)：

```
// File name:  person.h

// 声明私有数据成员类型

struct PersonalDataPrivate;

class Person

{

public:

    Person ();   // constructor
     virtual ~Person ();  // destructor
     void setAge(const int);
     int getAge();

private:

    PersonalDataPrivate* d;

};
```

```
// File name:  person.cpp

struct PersonalDataPrivate  // 定义私有数据成员类型
      {
         string mszName; // 姓名
         bool mbSex;    // 性别
         int mnAge;     // 年龄
      };

// constructor
      Person::Person ()
      {
         d = new PersonalDataPrivate;
      };

// destructor
      Person::~Person ()
      {
         delete d;
      };

void Person::setAge(const int age)
      {
         if (age != d->mnAge)
            d->mnAge = age;
      };

int Person::getAge()
      {
         return d->mnAge;
      };
```

元对象系统(Meta-Object System)

- QObject 类
  作为每一个需要利用元对象系统的类的基类。
- Q_OBJECT 宏，
  定义在每一个类的私有数据段，用来启用元对象功能，比如，动态属性，信号和槽。
- 元对象编译器moc (the Meta Object Complier)

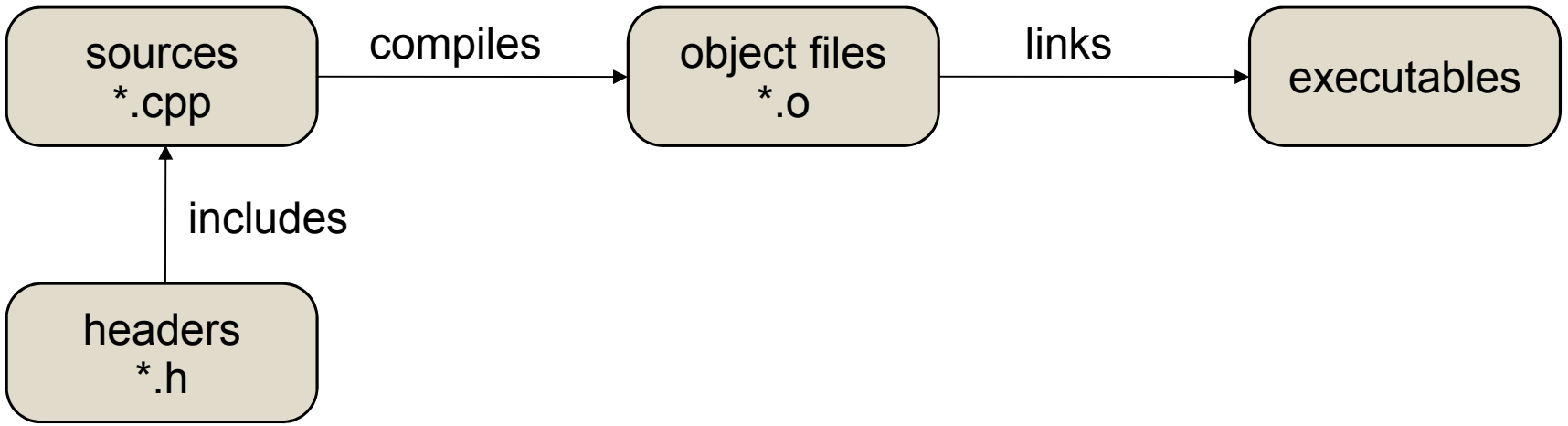
元对象系统(Meta-Object System)

元对象系统的功能:

- 元数据(`QObject::metaObject`)
- 类名 (`QObject::className`)
- 继承 (`QObject::inherits`)
- 属性(`setProperty`和 `QObject::property`)
- 信号和槽(`Signal and slot`)
- 普通信息(`QObject::classInfo`)
- 国际化(`tr()`, `QObject::trUtf8()`)

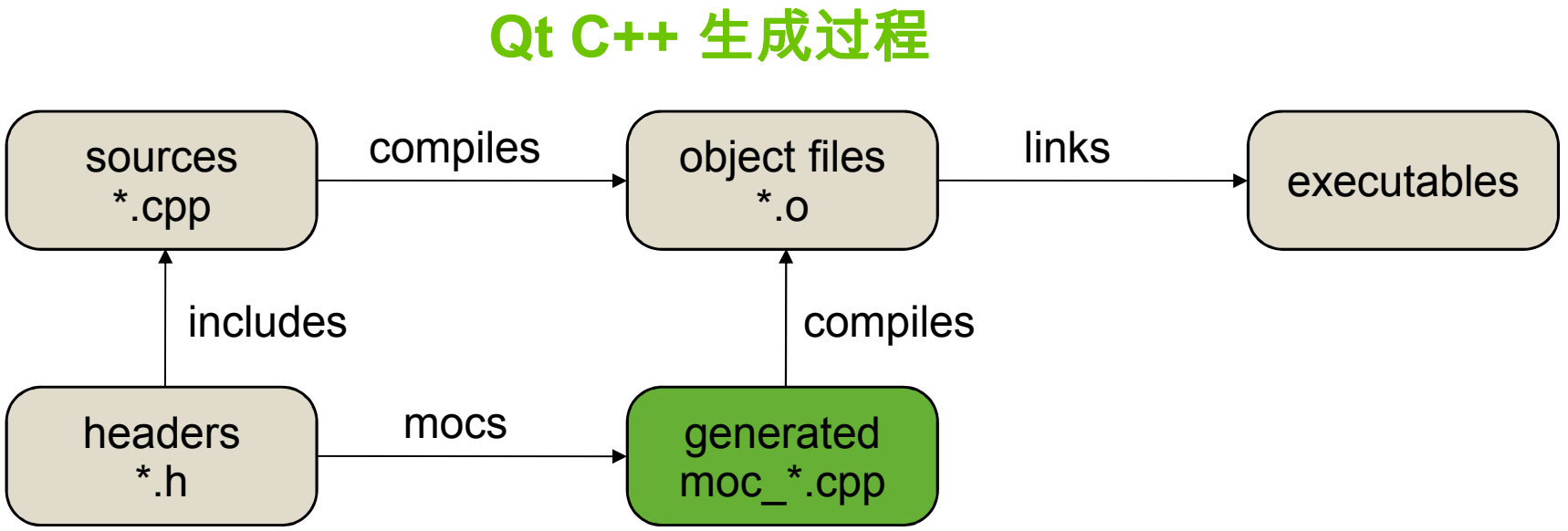
元对象系统(Meta-Object System)

普通的C++生成过程



元对象系统(Meta-Object System)

元数据通过元对象编译器(moc)在编译时组合在一起，元对象编译器用来处理Qt 的C++扩展。



moc从头文件里面获得数据。

元数据

moc 找什么？

首先确认该类继承自 QObject（可能是间接）

Q_OBJECT 宏，通常是第一步

类的一般信息

Qt 关键字

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_CLASSINFO("author", "John Doe")

public:
    MyClass(const Foo &foo, QObject *parent=0);

    Foo foo() const;

public slots:
    void setFoo( const Foo &foo );

signals:
    void fooChanged( Foo );

private:
    Foo m_foo;
};
```

内省(Introspection)

类在运行时了解它们自己的信息

```
if (object->inherits("QAbstractItemView"))
{
    QAbstractItemView *view = static_cast<QAbstractItemView*>(widget);
    view->...
```

能够实现动态转换而不需要运行时类型检查(RTTI)

```
enum CapitalsEnum { Oslo, Helsinki, Stockholm, Copenhagen };

int index = object->metaObject()->indexOfEnumerator("CapitalsEnum");
object->metaObject()->enumerator(index)->key(object->capital());
```

元对象了解细节

例子:它可以将枚举值转换成更容易阅读和保存的字符串

对实现脚本和动态语言的绑定有很好的支持。

属性(Properties)

QObject有getter和setter函数属性

```
class QLabel : public QFrame
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)
public:
    QString text() const;
public slots:
    void setText(const QString &);
};
```

Setter, 返回空,
将值当成唯一参数

Getter, 常量,返回值,
没有参数

命名策略: `color, setColor`

对于布尔: `isEnabled, setEnabled`

属性

为什么使用setter 函数？

可以验证设置

```
void setMin( int newMin )
{
    if( newMin > m_max )
    {
        qWarning("Ignoring setMin(%d) as min > max.", newMin);
        return;
    }
    ...
```

对可能的变化作出反应

```
void setMin( int newMin )
{
    ...

    m_min = newMin;
    updateMinimum();
}
```

属性Properties

为什么使用getter 函数？

间接的属性

```
QSize size() const
{
    return m_size;
}

int width() const
{
    return m_size.width();
}
```

属性

```
Q_PROPERTY(type name
           READ getFunction
           [WRITE setFunction]
           [RESET resetFunction]
           [NOTIFY notifySignal]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
           [USER bool]
           [CONSTANT]
           [FINAL])
```

使用属性

直接获取

```
QString text = label->text();
label->setText("Hello World!");
```

通过元信息和属性系统

```
QString text = object->property("text").toString();
object->setProperty("text", "Hello World");
```

在运行时发现属性

```
int QMetaObject::propertyCount();
QMetaProperty QMetaObject::property(i);

QMetaProperty::name/isConstant/isDesignable/read/write/...
```

动态属性

在运行时给对象增加属性

```
bool ret = object->setProperty(name, value);
```

真：如果属性经过 Q_PROPERTY 定义

假：如果只是动态增加

```
QObject::dynamicPropertyNames() const
```

返回一个动态属性的列表

可以用来“标识”对象，等等。

创建自定义属性

宏，描述属性

```
class AngleObject : public QObject

    Q_OBJECT
    Q_PROPERTY(qreal angle READ angle WRITE setAngle)

public:
    AngleObject(qreal angle, QObject *parent = 0);

    qreal angle() const;

    void setAngle(qreal);

private:
    qreal m_angle;
};
```

初始化值

Getter

Setter

私有状态

创建自定义属性

```
AngleObject::AngleObject(qreal angle, QObject *parent) :
        QObject(parent), m_angle(angle)
{
}

qreal AngleObject::angle() const
{
    return m_angle;
}

void AngleObject::setAngle(qreal angle)
{
    m_angle = angle;
    doSomething();
}
```

初始化值

Getter 简单返回值。这里
你可以计算复杂的值。

更新内部状态,
对变化作出反应。

自定义属性 - 枚举

```
class AngleObject : public QObject
{
    Q_OBJECT
    Q_ENUMS(AngleMode)
    Q_PROPERTY(AngleMode angleMode READ ...)

public:
    enum AngleMode {Radians, Degrees};

    ...
};
```

宏通知Qt AngleMode 是一个枚举类型。

普通枚举声明。

属性使用枚举作为类型。

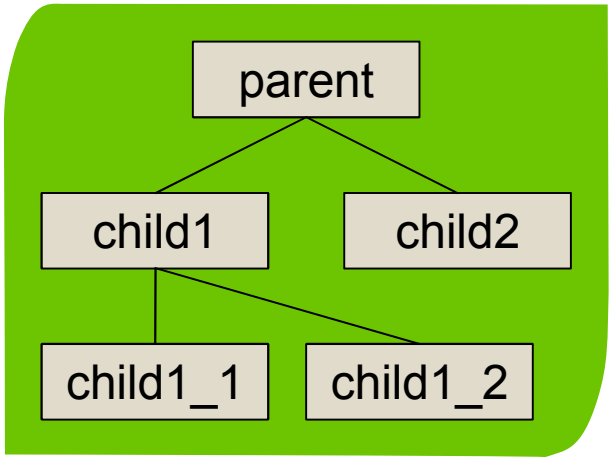
内存管理

QObject 可以有父对象和子对象

当一个父对象被删除，它的子对象也同样被删除。

```
QObject *parent = new QObject();
QObject *child1 = new QObject(parent);
QObject *child2 = new QObject(parent);
QObject *child1_1 = new QObject(child1);
QObject *child1_2 = new QObject(child1);

delete parent;
```

parent 删除 child1 和 child2
child1 删除 child1_1 和 child1_2



内存管理

当需要实现视觉层级时使用到它。

```
QDialog *parent = new QDialog();
QGroupBox *box = new QGroupBox(parent);
QPushButton *button = new QPushButton(parent);
QRadioButton *option1 = new QRadioButton(box);
QRadioButton *option2 = new QRadioButton(box);

delete parent;
```

parent 删除 box 和 button
box 删除 option1 和 option2

使用方法

使用 this指针指向最高层父对象

```
Dialog::Dialog(QWidget *parent) : QDialog(parent)
{
    QGroupBox *box = QGroupBox(this);
    QPushButton *button = QPushButton(this);
    QRadioButton *option1 = QRadioButton(box);
    QRadioButton *option2 = QRadioButton(box);
    ...
```

在栈上分配父对象空间

```
void Widget::showDialog()
{
    Dialog dialog;

    if (dialog.exec() == QDialog::Accepted)
    {
        ...
    }
}
```

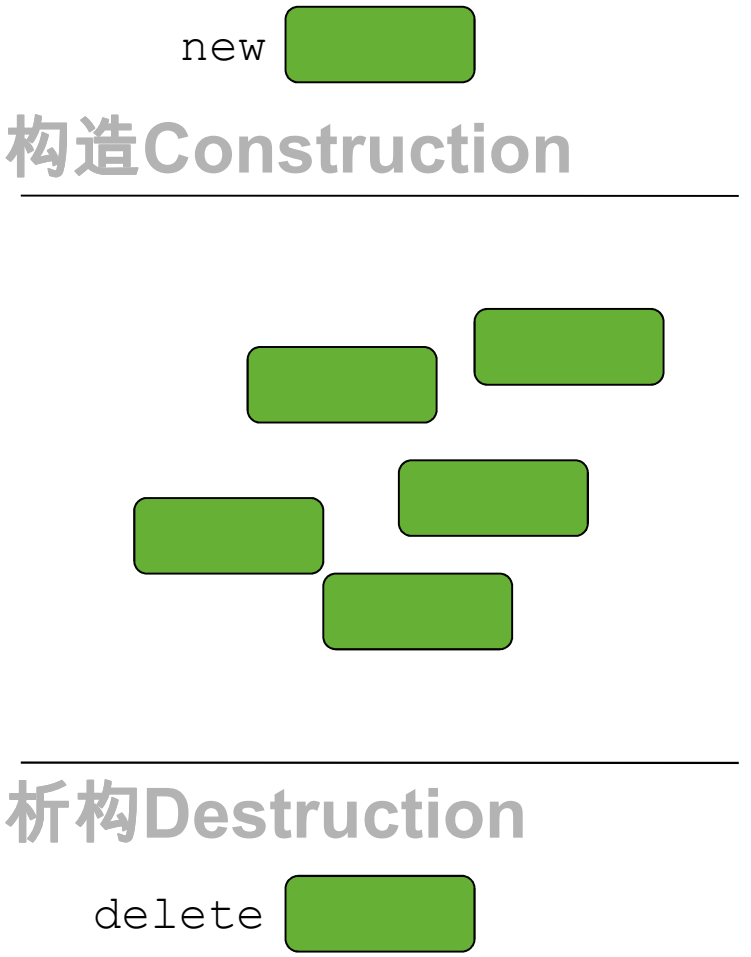
dialog 在作用范围结束时被删除

堆（Heap）

当使用 new 和 delete时，内存在堆中分配。

堆内存空间必须通过 delete 完全释放，以防止内存泄漏。

只要有需要，分配在堆上的对象可以一直存活下去。

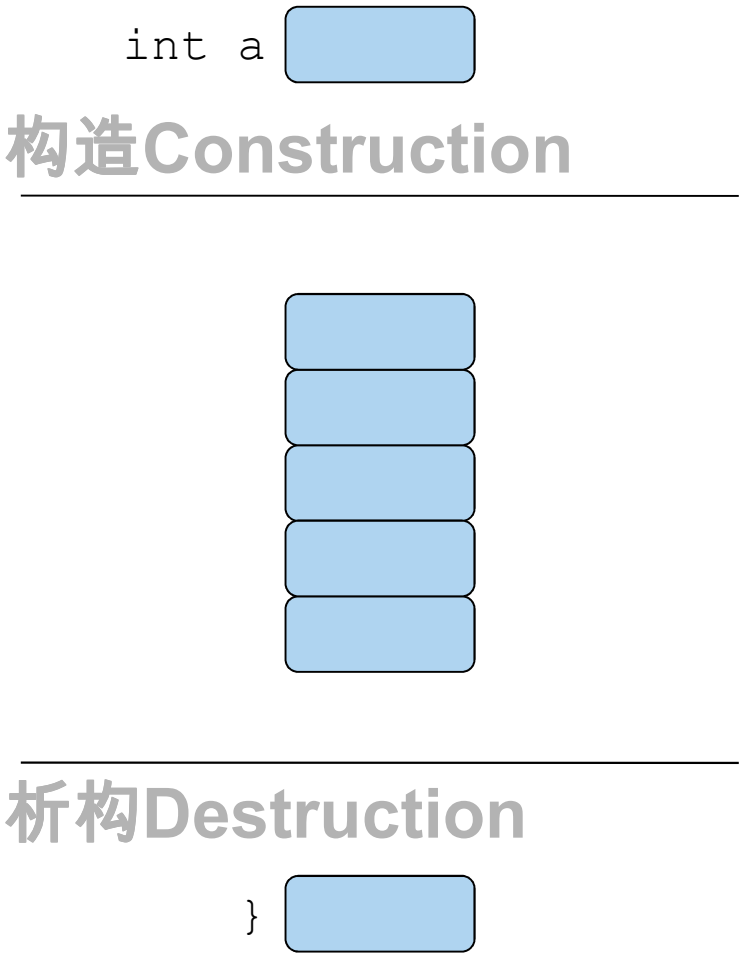


栈（Stack）

局部变量在栈上分配。

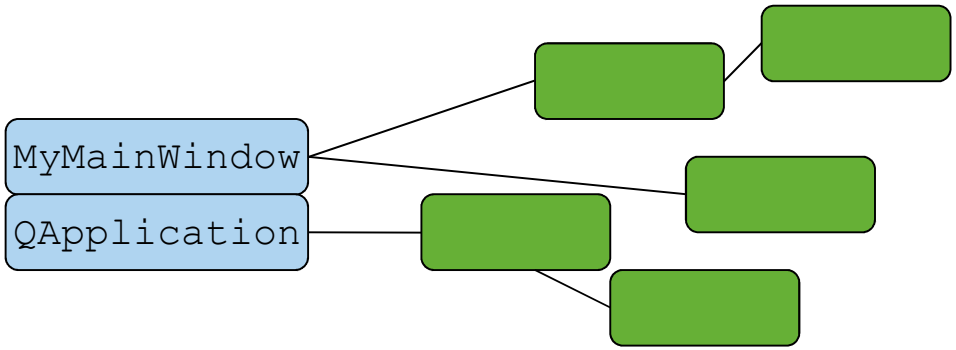
栈变量超过作用范围时会自动释放。

分配在栈中的对象在超出作用范围时总是会被析构。



堆 和 栈

想要自动内存管理，只有父对象需要在栈上分配。




```
int main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyMainWindow w;
    w.show();
    return a.exec();
}
```

```
MyMainWindow::MyMainWindow(...
{
    new QLabel(this);
    new ...
}
```

改变所有者

QObject可以修改它所属的父对象。

```
obj->setParent(newParent);
```

父对象知道何时子对象被删除

```
delete listWidget->item(0); // 删除第一个item(不安全)
```

一系列函数实现返回指针，从其所有者“拿走”释放的数据，把它留给拿取者处理

```
QLayoutItem *QLayout::takeAt(int);
QListWidgetItem *QListWidget::takeItem(int);

// Safe alternative
QListWidgetItem *item = listWidget->takeItem(0);
if (item) { delete item; }
```

item列表本质上并不是子对象，而是拥有者。

这个例子进行了说明。

构造规范

几乎所有的 QObject 都有一个默认为空值的父对象。

```
QObject(QObject *parent=0);
```

Qwidget 的父对象是其它 QWidget

类为了方便,倾向于提供多种构造（包括只带有父对象的一种）

```
QPushButton(QWidget *parent=0);
QPushButton(const QString &text, QWidget *parent=0);
QPushButton(const QIcon &icon, const QString &text, QWidget *parent=0);
```

父对象通常是带缺省值的第一个参数。

```
QLabel(const QString &text, QWidget *parent=0, Qt::WindowFlags f=0);
```

构造规范

当创建自己的 Qobject时, 需考虑：

1）总是允许父对象 parent 为 0 (null)

2）有一个只接受父对象的构造函数

3）parent 是带默认值的第一个参数

4）提供几种构造函数，避免空值、无效值 (e.g. QString())作为参数。

动手实践！