

# Qt编程

**QT PROGRAMMING**

**DAY01**

# 内容

上午	09:00 ~ 09:30	Qt简介
	09:30 ~ 10:20	Qt环境与工具链
	10:30 ~ 11:20	第一个Qt程序
	11:30 ~ 12:20	使用中文
下午	14:00 ~ 14:50	信号和槽
	15:00 ~ 15:50	容器窗口与事件同步
	16:00 ~ 16:50	
	17:00 ~ 17:30	总结和答疑



# Qt简介

---

Qt简介

Qt简介

Qt是图形用户程序框架

Qt的由来和发展

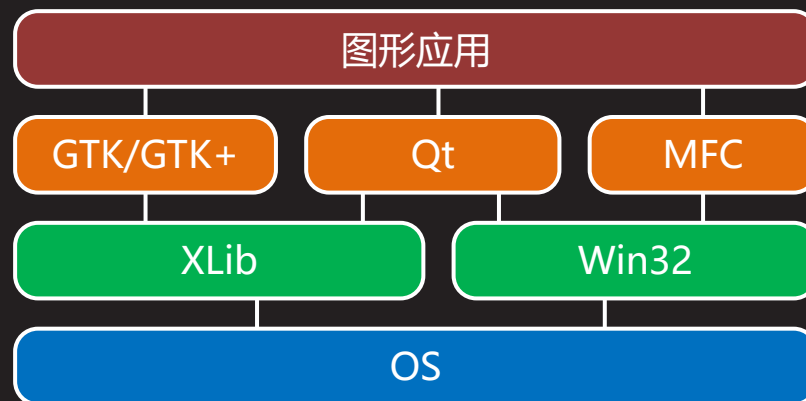
为什么选择Qt

# Qt简介



# Qt是图形用户界面框架

- Qt是对底层**图形应用编程接口**(APIs)面向对象化的封装
  - 一套基于C++语言的类库
- Qt专注但**不局限于**图形用户界面(GUI)的开发
  - 系统调用、网络编程、数据库编程、2D/3D图形处理
- Qt是**跨平台**的应用编程框架
  - Linux、Windows、Mac OS X、Android、iOS
- Qt堪称**艺术级别**的图形开发工具
  - 同时为最终用户和开发团队带来高品质的使用体验



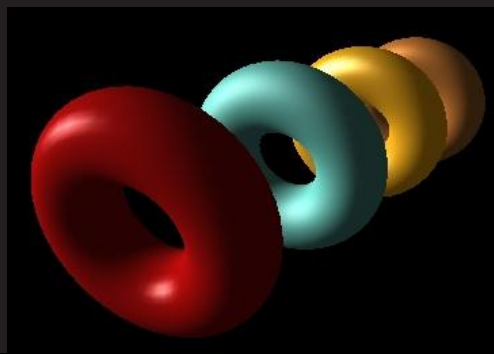
# Qt的由来和发展

- 诞生——1991
  - Haavard Nord和Eirik Chambe-Eng合作编写了最初的Qt
- 成长——1994
  - Haavard和Eirik创立了Troll Tech(奇趣科技)公司
- 开源——2005
  - Qt4.0发布，奇趣科技被诺基亚收购，更名为Qt Software，并于2009年宣布开源
- 新生——2012
  - 诺基亚将全部Qt业务和知识产权出售给Digia公司
  - Digia公司于2014年成立The Qt Company全资子公司，专注于Qt技术的开发和拓展



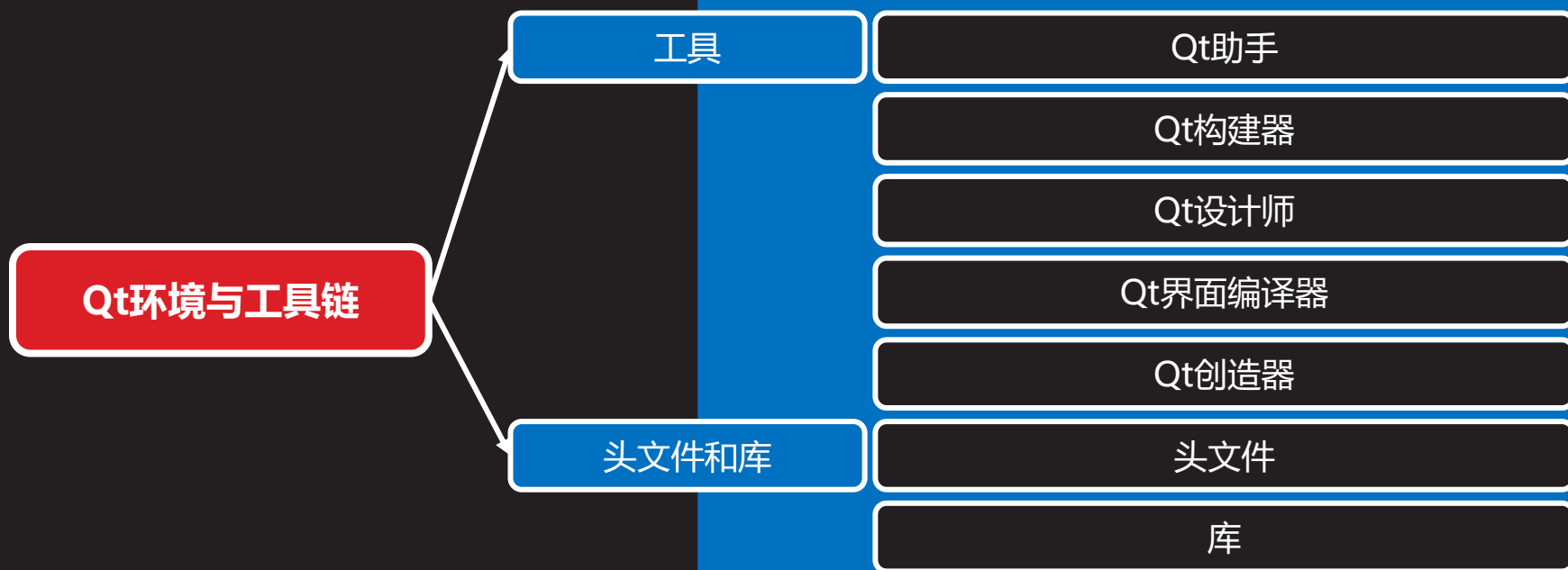
# 为什么选择Qt

- 基于C++语言，简单、易用、面向对象
- 优良的跨平台特性
- 架构健壮，功能强大，性能卓越
- 基于Qt的开发简便而高效
- Free! Free! Free!



# Qt环境与工具链

---



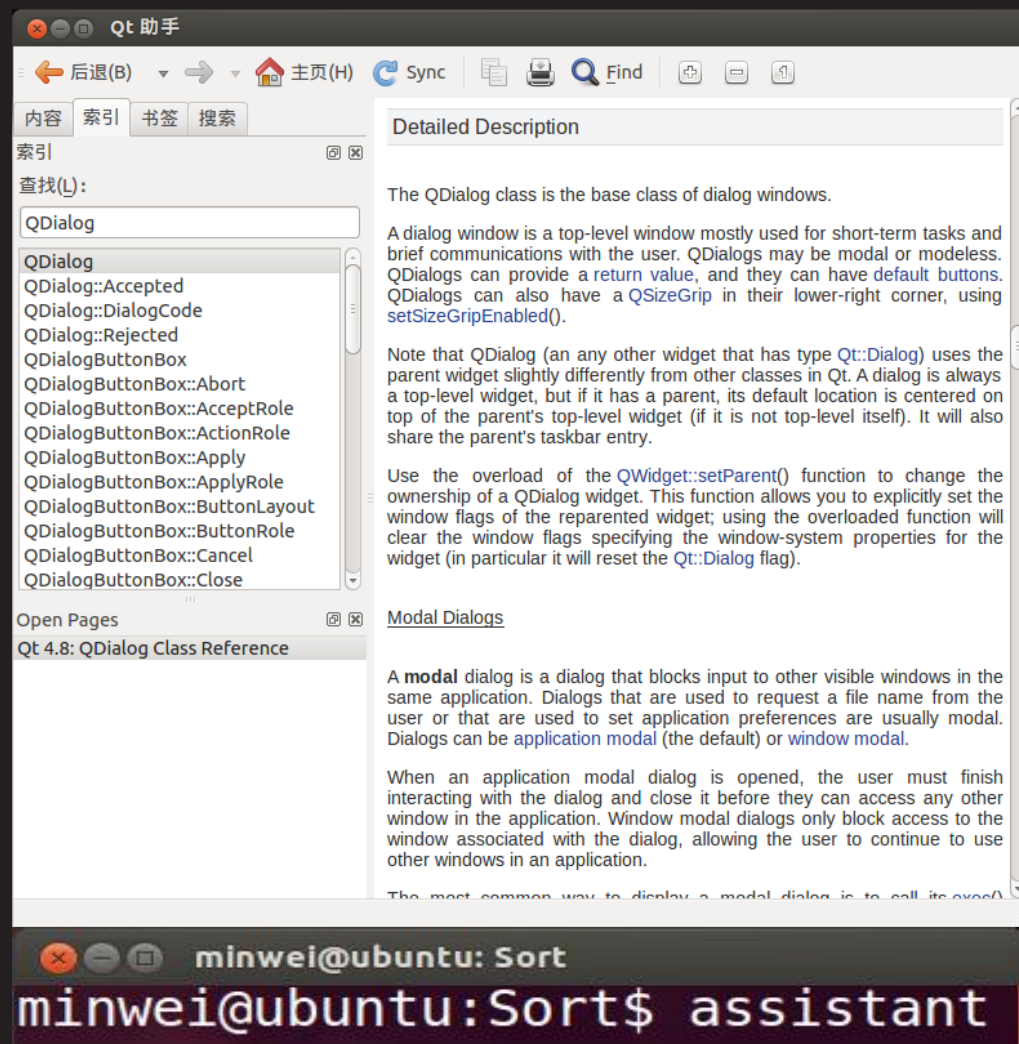


# 工具



# Qt助手

- Qt的参考文档涉及了Qt中的每一个类和函数，是Qt开发人员的必备手册
- Qt助手是和Qt一起发布的帮助浏览器软件，除了可以阅读Qt参考文档外，Qt助手还提供强大的查询和检索功能
- 启动Qt助手只需在命令行终端上输入assistant命令即可



# Qt构建器

- 构建器在很大程度上简化了有多个头文件、源文件和库文件所组成的，复杂工程的编译和链接工作
- Qt构建器是和Qt一起发布的，针对Qt应用的脚本生成器
  - **qmake -version**  
查看版本信息
  - **qmake -project**  
根据工程目录生成平台无关的工程文件(.pro)
  - **qmake**  
根据平台无关的工程文件(.pro)生成平台相关的Makefile
- qmake包含了调用Qt内置代码生成工具(moc、uic和rcc)的逻辑规则，并对编译和链接选项中与Qt有关的头文件和库文件进行了处理，甚至可以指定目标平台和编译器



# 利用qmake生成工程文件和构建脚本

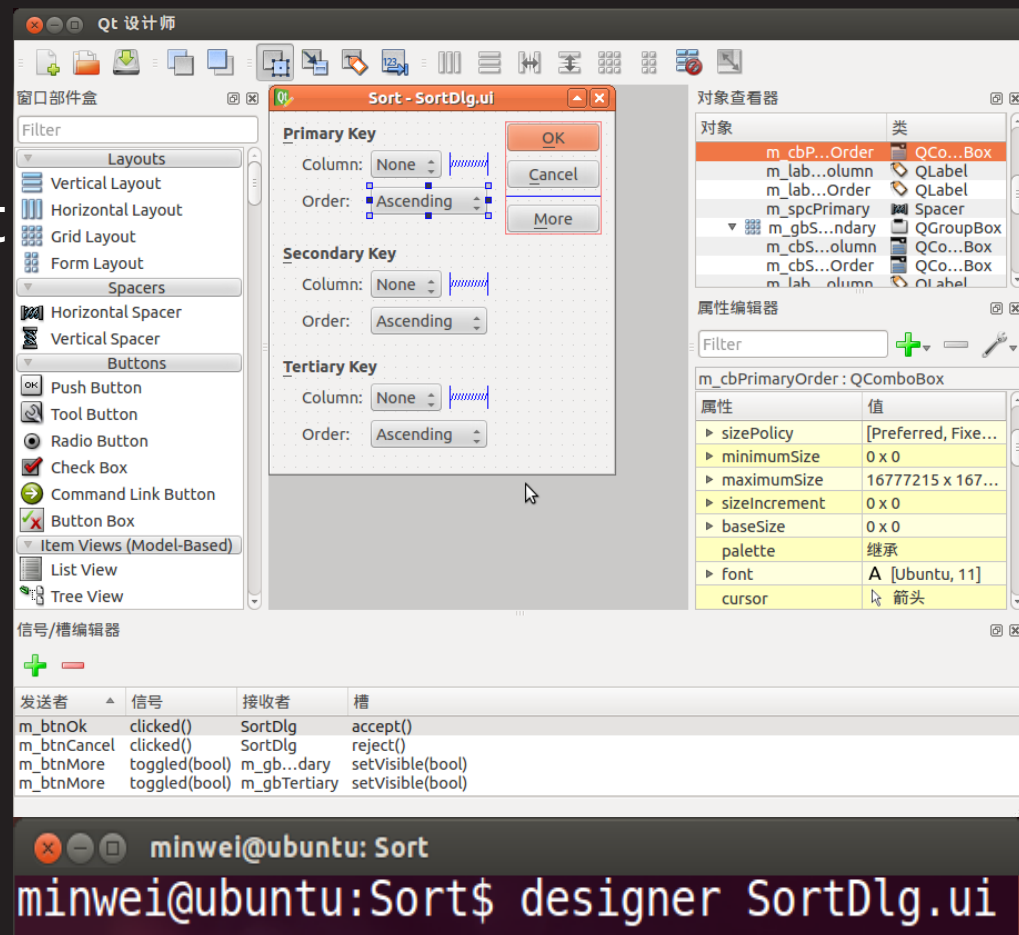
【参见：TTS COOKBOOK】

- 利用qmake生成工程文件和构建脚本



# Qt设计师

- 对于程序员来说，纯粹通过编写C++源代码来开发整个Qt应用程序并非不可能，但是人类的本性还是更希望通过可视化的方法，以所见即所得(WYSIWYG, What You See Is What You Get)的方式进行快速、高效且简洁的开发
- Qt设计师为程序员提供了这样一种可视化的设计能力，只需在命令行终端上输入designer命令即可



# Qt界面编译器

- Qt设计师以所见即所得的方式，允许程序员进行可视化的界面设计，但它所输出的并非通常意义上的C++代码，而是一个扩展名为.ui的XML文件，谓之**界面描述文件**
- 为了得到与界面描述文件所描述的界面具有相同视觉效果C++代码，还需要借助**Qt界面编译器**将XML格式的.ui文件编译为符合标准C++语法的.h文件，以类的方式描述界面描述文件中通过XML语法所描述的界面元素

```
minwei@ubuntu: Sort
minwei@ubuntu:Sort$ uic SortDlg.ui -o ui_SortDlg.h
```

```
minwei@ubuntu: Sort
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ui version="4.0">
3   <class>SortDlg</class>
4   <widget class="QDialog" name="SortDlg">
5     <property name="geometry">
6       <rect>
7         <x>0</x>
8         <y>0</y>
9         <width>284</width>
10        <height>301</height>
11      </rect>
12    </property>
```

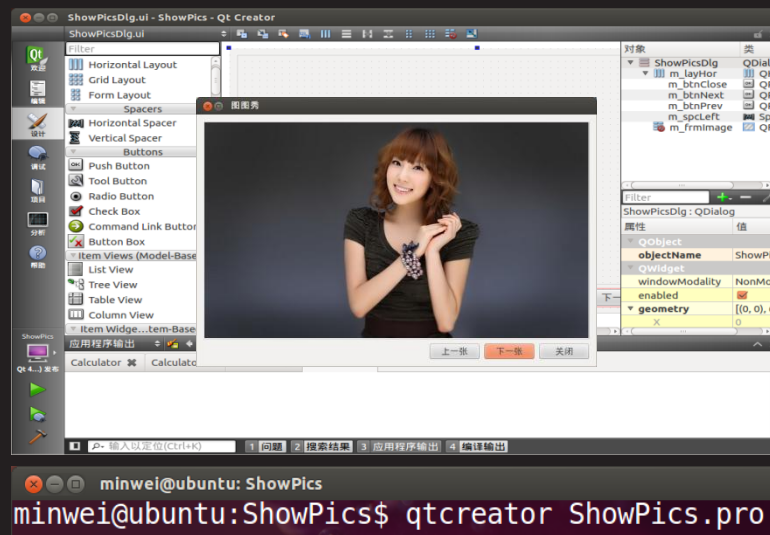
uic

```
minwei@ubuntu: Sort
29 class Ui_SortDlg
30 {
31 public:
32   QGridLayout *m layGrid;
33   QGroupBox *m gbPrimary;
34   QGridLayout *m layPrimary;
35   QLabel *m labPrimaryColumn;
36   QComboBox *m cbPrimaryColumn;
37   QSpacerItem *m spcPrimary;
38   QLabel *m labPrimaryOrder;
39   QComboBox *m cbPrimaryOrder;
40   QVBoxLayout *m layButtons;
```



# Qt创造器

- 在文件编辑器中编写C++代码，在Qt设计师中进行界面设计，并通过命令行工具uic将其编译为C++类，最后借助qmake生成构建脚本，并执行make命令编译链接。其间抑或还需要资源编译、断点调试等过程。每一个环节都需要专门的工具，每一个步骤都要执行专门的命令
- 如果这一切已令你感到目不暇接，甚至有些手忙脚乱，Qt创造器将为你提供一个集编辑、设计、编译、链接、调试，甚至在线帮助于一身的集成开发环境(IDE)。启动这个环境，只需在命令行终端上输入qtcreator命令即可



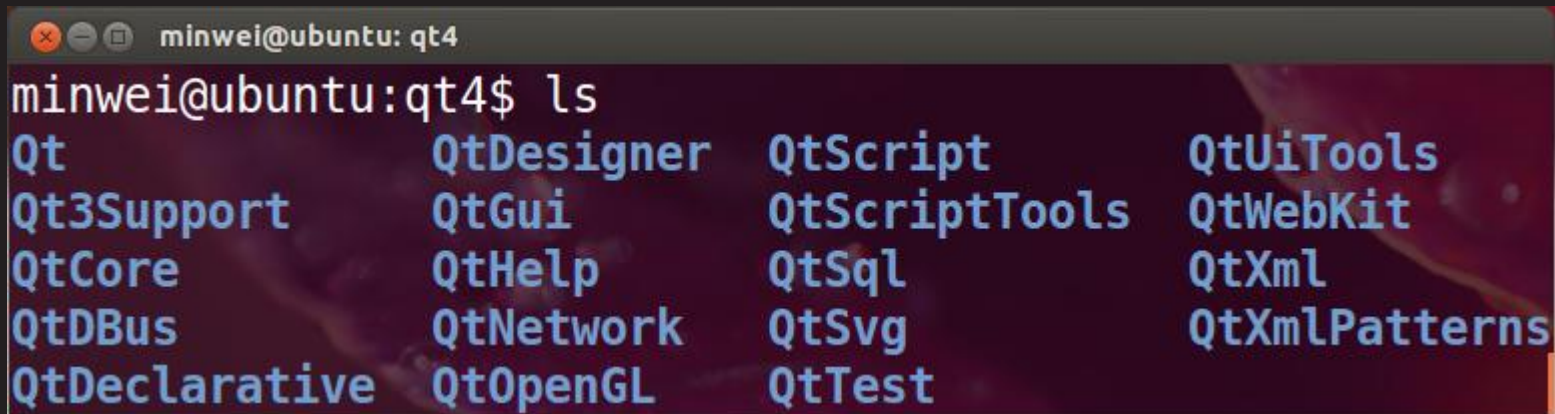
# 头文件和库





# 头文件

- 将Qt安装到系统中以后，会默认在/usr/include目录下创建一个名为qt4的子目录，该目录又按功能被划分为更多的子目录，其中每个子目录都包含相应功能的头文件



```
minwei@ubuntu: qt4
minwei@ubuntu:qt4$ ls
Qt          QtDesigner  QtScript    QtUiTools
Qt3Support  QtGui       QtScriptTools QtWebKit
QtCore      QtHelp      QtSql       QtXml
QtDBus      QtNetwork   QtSvg       QtXmlPatterns
QtDeclarative QtOpenGL    QtTest
```

- Qt的头文件不带.h扩展名，并且每个单词的首字母大写
  - #include <QPushButton>
- 用gcc/g++编译时需要用-I选项指定Qt头文件的路径
  - g++ ... -I/usr/include/qt4/QtGui ...

# 库

- 将Qt安装到系统中以后，会默认在/usr/lib/i386-linux-gnu目录下存放若干以libQt为前缀的.so文件，它们就是按功能划分的Qt共享库，为Qt应用程序提供运行时支持

```
minwei@ubuntu: i386-linux-gnu
minwei@ubuntu:i386-linux-gnu$ ls libQt*.so
libQt3Support.so          libQtOpenGL.so
libQtCLucene.so          libQtScript.so
libQtCore.so             libQtScriptTools.so
libQtDBus.so            libQtSql.so
libQtDeclarative.so     libQtSvg.so
libQtDesignerComponents.so libQtTest.so
libQtDesigner.so        libQtWebKit.so
libQtGui.so             libQtXmlPatterns.so
libQtHelp.so            libQtXml.so
libQtNetwork.so
```

- 链接时需要通过-l和-L选项指定所连的库及该库所在路径
  - g++ ... -L/usr/lib/i386-linux-gnu -lQtGui ...

# 第一个Qt程序

第一个Qt程序

Hello, World !

查阅文档

编辑源代码

生成构建脚本

编译链接和运行

Public Types

Properties

Public Functions

Public Slots

Signals

Static Public Members

Protected Functions

Macros

Detailed Description

# Hello, Qt!



# 编辑源代码

- 使用vi、Emacs或任何一款文本编辑器编写如下代码：

```

1 #include <QApplication>
2 #include <QLabel>
3 int main (int argc, char* argv[]) {
4     QApplication app (argc, argv);
5     QLabel lab ("Hello, Qt!");
6     lab.show ();
7     return app.exec ();
8 }
    
```

- 第1-2行

- 包含**QApplication**和**QLabel**两个类的定义头文件。每个Qt类都有一个与该类同名的头文件，包含对该类的定义



# 编辑源代码（续1）

- 第4行
  - 创建一个QApplication对象，负责管理整个应用程序的资源，同时接收Qt自己能够处理的命令行参数
- 第5行
  - 创建一个显示"Hello, Qt!"的QLabel窗口部件(widget)
- 第6行
  - 使前面创建的QLabel部件可见，QLabel部件默认为隐藏
- 第7行
  - 将应用程序的控制权交给Qt，使程序进入事件循环，等待用户的动作，比如单击鼠标、敲击键盘、点击按钮或菜单等，并做出相应的响应，最后返回Qt的返回值



# 生成构建脚本

- 按照Qt对项目管理的约定，最好为每个项目建立一个以**项目名称**命名的目录，该项目的代码都存放在此目录下
  - 例如：/home/tarena/**Hello**/Hello.cpp
- 在工程目录下通过Qt构建器生成平台无关的工程文件
  - 例如：\$ **qmake -project**
  - 得到：Hello.pro
- 在工程目录下通过Qt构建器根据工程文件生成构建脚本
  - 例如：\$ **qmake Hello.pro**
  - 得到：Makefile
- 如果工程目录下只有一个工程文件，也可以省略文件名
  - 例如：\$ **qmake**





# 编译链接和运行

- 在工程目录下直接执行make命令，根据构建脚本Makefile执行编译和链接，得到可执行程序
  - \$ make -f Makefile
  - \$ make
- 如果make进入死循环，那么请把系统时间调至当前时间

```
minwei@ubuntu: Hello
minwei@ubuntu:Hello$ make
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_WE
BKIT -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB
-DQT_SHARED -I/usr/share/qt4/mkspecs/linux-g++
-I. -I/usr/include/qt4/QtCore -I/usr/include/
qt4/QtGui -I/usr/include/qt4 -I. -I. -o Hello.
o Hello.cpp
g++ -Wl,-O1 -o Hello Hello.o -L/usr/lib/i38
6-linux-gnu -lQtGui -lQtCore -lpthread
minwei@ubuntu:Hello$ Hello
```

```
Hello
Hello, Qt!
```

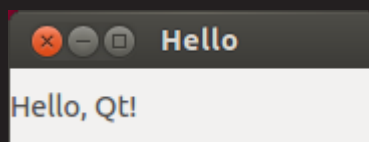




# 显示"Hello, Qt!"标签

【参见：TTS COOKBOOK】

- 显示"Hello, Qt!"标签

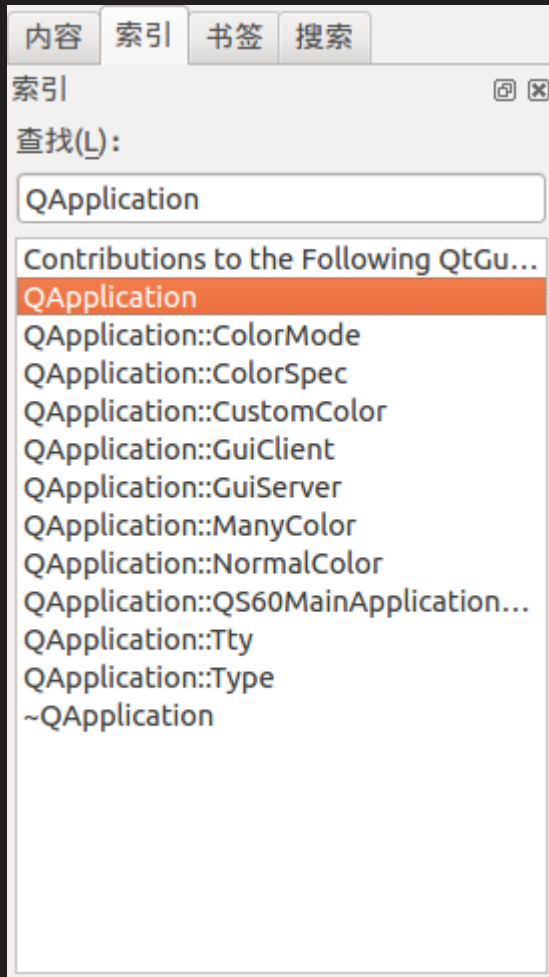


# 查阅文档



# Public Types

- 执行assistant命令，启动Qt助手，“索引/查找”输入QApplication，可以看到QApplication类的参考信息



## QApplication Class Reference

```
#include <QApplication>
```

Inherits: [QCoreApplication](#).

其中，除了列出头文件和基类以外，还描述了该类的具体细节、代码示例和相关主题，例如Public Types即指其**公有成员类型**，诸如ColorSpec、Type，等等.....

### Contents

- [Public Types](#)
- [Properties](#)
- [Public Functions](#)
- [Public Slots](#)
- [Signals](#)
- [Static Public Members](#)
- [Protected Functions](#)
- [Macros](#)
- [Detailed Description](#)



# Properties

- 对象的属性，其语义类似于C++语法中的成员变量

Properties

- `autoMaximizeThreshold` : int
- `autoSipEnabled` : bool
- `cursorFlashTime` : int
- `doubleClickInterval` : int
- `globalStrut` : QSize
- `keyboardInputInterval` : int
- `layoutDirection` : Qt::LayoutDirection
- `quitOnLastWindowClosed` : bool
- `startDragDistance` : int
- `startDragTime` : int
- `styleSheet` : QString
- `wheelScrollLines` : int
- `windowIcon` : QIcon

- 4 properties inherited from `QCoreApplication`
- 1 property inherited from `QObject`

- 属性多被定义为私有，访问它们需要专门的set/get方法
  - `int wheelScrollLines (void)` // 获取wheelScrollLines
  - `void setWheelScrollLines (int)` // 设置wheelScrollLines



# Public Functions

- 公有函数，表示对象的外观行为，既有主动行为也有被动行为，甚至包括构造函数、析构函数，当然也包括从基类继承的函数，和对基类覆盖的函数

## Public Functions

```

QApplication ( int & argc, char ** argv )
QApplication ( int & argc, char ** argv, bool GULenabled )
QApplication ( int & argc, char ** argv, Type type )
QApplication ( Display * display, Qt::HANDLE visual = 0, Qt::HANDLE colormap = 0 )
QApplication ( Display * display, int & argc, char ** argv, Qt::HANDLE visual = 0, Qt::HANDLE colormap = 0 )
QApplication ( QApplication::QS60MainApplicationFactory factory, int & argc, char ** argv )
virtual ~QApplication ()
virtual void commitData ( QSessionManager & manager )
QInputContext * inputContext () const
virtual void saveState ( QSessionManager & manager )
QString sessionId () const
int x11ProcessEvent ( XEvent * event )
    
```

- 4 public functions inherited from **QCoreApplication**
- 29 public functions inherited from **QObject**



# Public Slots

- 公有槽函数通常并不需要在程序中直接调用(虽然可以这样做), 而是被默认或人为地与某个或某些特定的信号连接起来, 以表示对触发该信号的动作的响应
- 槽函数可以从基类继承, 也可以被子类覆盖, 当然后者必须在基类中被声明为虚函数

Public Slots

```

void aboutQt ()
int autoMaximizeThreshold () const
bool autoSipEnabled () const
void closeAllWindows ()
void setAutoMaximizeThreshold ( const int threshold )
void setAutoSipEnabled ( const bool enabled )
void setStyleSheet ( const QString & sheet )

```

- 1 public slot inherited from `QCoreApplication`
- 1 public slot inherited from `QObject`



# Signals

- 如果对象的内部**状态**发生**变化**，比如按钮被点击或者窗口失去焦点，这种变化可能会对它的客户或拥有者具有一定的意义，那么这个对象就可以**抛出信号**，而与之相连的**槽**就会被立即**执行**，以应对上述变化所造成的影响
- 信号与槽的这种联动机制**独立**于任何图形界面事件循环

## Signals

```
void aboutToReleaseGpuResources ()
void aboutToUseGpuResources ()
void commitDataRequest ( QSessionManager & manager )
void focusChanged ( QWidget* old, QWidget* now )
void fontDatabaseChanged ()
void lastWindowClosed ()
void saveStateRequest ( QSessionManager & manager )
```

- 1 signal inherited from `QCoreApplication`
- 1 signal inherited from `QObject`



# Static Public Members

- 静态公有成员函数与具体对象无关，管理该类对象的共享资源

## Static Public Members

```

QWidget* activeModalWidget ()
QWidget* activePopupWidget ()
QWidget* activeWindow ()
void alert ( QWidget* widget, int msec = 0 )
QWidgetList allWidgets ()
void beep ()
void changeOverrideCursor ( const QCursor & cursor )
QClipboard* clipboard ()
int colorSpec ()
int cursorFlashTime ()
QDesktopWidget* desktop ()
bool desktopSettingsAware ()
int doubleClickInterval ()
int exec ()
QWidget* focusWidget ()
QIcon windowIcon ()
    
```

- 38 static public members inherited from `QCoreApplication`
- 7 static public members inherited from `QObject`





# Protected Functions

- 当从Qt类中派生自己的子类，以扩充或者定制化Qt基类的功能时，基类中的保护成员将成为除公有成员以外的可利用资源，在不破坏对象封装性的前提下，为子类提供更多复用基类实现的可能，部分保护成员支持重实现

## Protected Functions

```
virtual void childEvent ( QChildEvent * event )
virtual void connectNotify ( const char * signal )
virtual void customEvent ( QEvent * event )
virtual void disconnectNotify ( const char * signal )
int receivers ( const char * signal ) const
QObject * sender () const
int senderSignalIndex () const
virtual void timerEvent ( QTimerEvent * event )
```



# Macros

- 宏定义在本质上与类的作用域无关，也不受访问控制属性的约束，放在此类的参考中描述，只为强调该宏与此类的联系相对**紧密**，或者该宏就表示此类**对象**的一个引用或指针

- Qt定义了一组特殊的宏，以辅助其**元**

## Macro Documentation

### qApp

A global pointer referring to the unique application object. It is equivalent to the pointer returned by the `QCoreApplication::instance()` function except that, in GUI applications, it is a pointer to a `QApplication` instance.

Only one application object can be created.

See also `QCoreApplication::instance()`.

**对象编译器**(moc)将扩展C++语法编译为标准C++代码

- SIGNAL/SLOT
- Q\_OBJECT

# Detailed Description

- 细节描述，详细说明类的用法、调用上下文和注意事项

## Detailed Description

The `QApplication` class manages the GUI application's control flow and main settings.

`QApplication` contains the main event loop, where all events from the window system and other sources are processed and dispatched. It also handles the application's initialization, finalization, and provides session management. In addition, `QApplication` handles most of the system-wide and application-wide settings.

For any GUI application using Qt, there is precisely **one** `QApplication` object, no matter whether the application has 0, 1, 2 or more windows at any given time. For non-GUI Qt applications, use `QCoreApplication` instead, as it does not depend on the `QtGui` library.

The `QApplication` object is accessible through the `instance()` function that returns a pointer equivalent to the global `qApp` pointer.

`QApplication`'s main areas of responsibility are:

- It initializes the application with the user's desktop settings such as `palette()`, `font()` and `doubleClickInterval()`. It keeps track of these properties in case the user changes the desktop globally, for example through some kind of control panel.
- It performs event handling, meaning that it receives events from the underlying window system and dispatches them to the relevant widgets. By using `sendEvent()` and `postEvent()` you can send your own events to widgets.
- It parses common command line arguments and sets its internal state accordingly. See the [constructor documentation](#) below for more details.
- It defines the application's look and feel, which is encapsulated in a `QStyle` object. This can be changed at runtime with `setStyle()`.
- It specifies how the application is to allocate colors. See `setColorSpec()` for details.



# 使用中文

---

使用中文

UTF-8码中文字符

字符编码

内外有别

解码器与翻译器

# UTF-8码中文字符



# 字符编码

- 字符编码

- 任何字符在计算机内部都是用**数字**表示的，即字符编码
- 不同国家和地区都为自己的语言定义了**不同**的编码标准
  - ✓ 英语国家：基本ASCII，128个字符
  - ✓ 欧洲国家：扩展ASCII，256个字符
  - ✓ 中国大陆：**GBK**，21003个字符
  - ✓ 港台地区：BIG5，13060个字符

- 统一编码

- 国际标准化组织制定的ISO/IEC 10646标准是一种可以支持世界上所有语言文字的字符编码标准，称为**Unicode**
- 根据字长的不同，Unicode又被分为两个字符集：
  - ✓ UCS-2：**双**字节统一字符集，65000个字符
  - ✓ UCS-4：**四**字节统一字符集，99089个字符



# 字符编码（续1）

- UCS转换格式(UCS Transformation Format, UTF)
  - 无论是UCS-2还是UCS-4，作为Unicode字符集，它们都只是规定了如何对一个字符编码，但并没有规定如何传输、保存这个编码，即在程序中如何表示这个编码，而UTF所要解决的就是这个问题。根据转换算法的不同可被分为：
    - ✓ UTF-8：用1到4个字节表示一个Unicode字符
    - ✓ UTF-16：用2或4个字节表示一个Unicode字符
    - ✓ UTF-32：用4个字节表示一个Unicode字符
  - 例如，字符"汉"在UCS-2字符集中的字符编码是0x6C49，其在小端机器的内存中，从低地址到高地址依次为：
    - ✓ UTF-8：0xE6, 0xB1, 0x89
    - ✓ UTF-16：0x49, 0x6C
    - ✓ UTF-32：0x49, 0x6C, 0x00, 0x00



# 内外有别

- 内部编码
  - Qt应用程序编程接口及其内部实现所使用的字符，都是以UTF-16格式表示的，因此UTF-16就是Qt字符的内部编码
- 外部编码
  - 程序源代码中使用的字面值形式的字符和字符串、用户通过程序界面输入的字符和字符串，以及程序通过文件、网络、进程间通信或其它媒介读取的字符和字符串，受系统环境等因素的影响，通常会形形色色的编码格式，一般将其统称为外部编码
- 当外部编码和内部编码不一致时，对于输入操作，Qt会尽可能地将外部编码转换为内部编码，而对于输出操作，则执行相反的转换。程序员的工作就是要确保转换正确

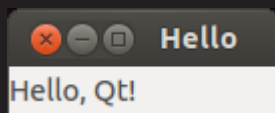




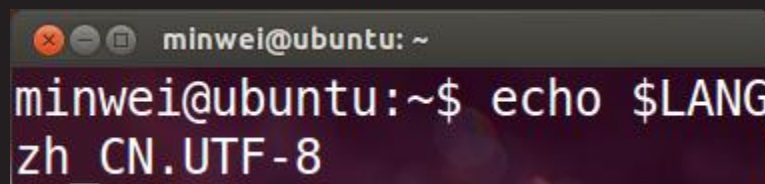
# 解码器与翻译器

- 默认情况下，Qt可以正确理解ASCII格式的外部编码，并将其正确转换为UTF-16格式的内部编码，反之亦然

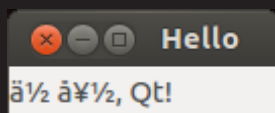
- `QLabel lab ("Hello, Qt!");`



- 但如果所提供的字符串中包含中文字符，多数操作系统会将其表示为UTF-8格式。Qt无法理解UTF-8格式的外部编码，因此也就无法将其转换为正确的内部编码



- `QLabel lab ("你好, Qt!");`



# 解码器与翻译器（续1）

- 为了让Qt能够正确理解UTF-8格式的汉字字符，并将其正确地转换为内部编码，需要在程序中完成三步操作：

- 第一步：创建可以理解UTF-8格式外部编码的解码器

```
QTextCodec* codec =
    QTextCodec::codecForName ("utf-8");
```

- 第二步：将前面创建的解码器交给的翻译器

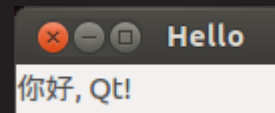
```
QTextCodec::setCodecForTr (codec);
```

- 第三步：将外部编码字符串翻译成内部编码的字符串

```
QString str = QObject::tr ("你好, Qt!");
```

- 例如

- QTextCodec::setCodecForTr (
 QTextCodec::codecForName ("utf-8"));
 QLabel lab (QObject::tr ("你好, Qt!));



# 显示中文的下压按钮

【参见：TTS COOKBOOK】

- 显示中文的下压按钮



# 信号和槽

---

信号和槽

信号和槽

信号

槽

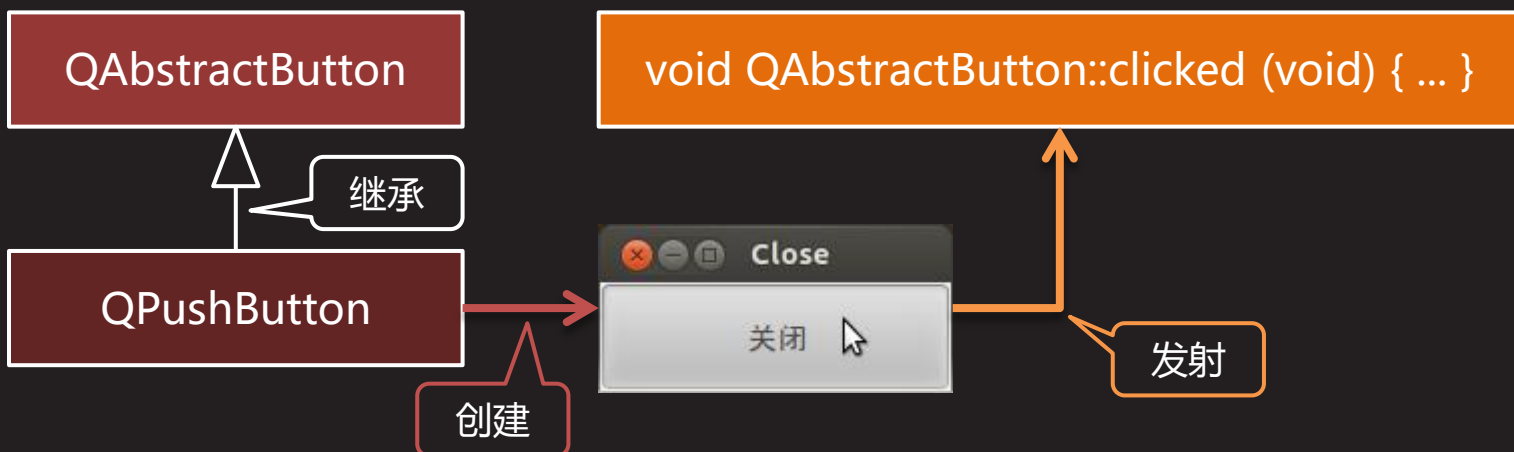
连接信号和槽

# 信号和槽



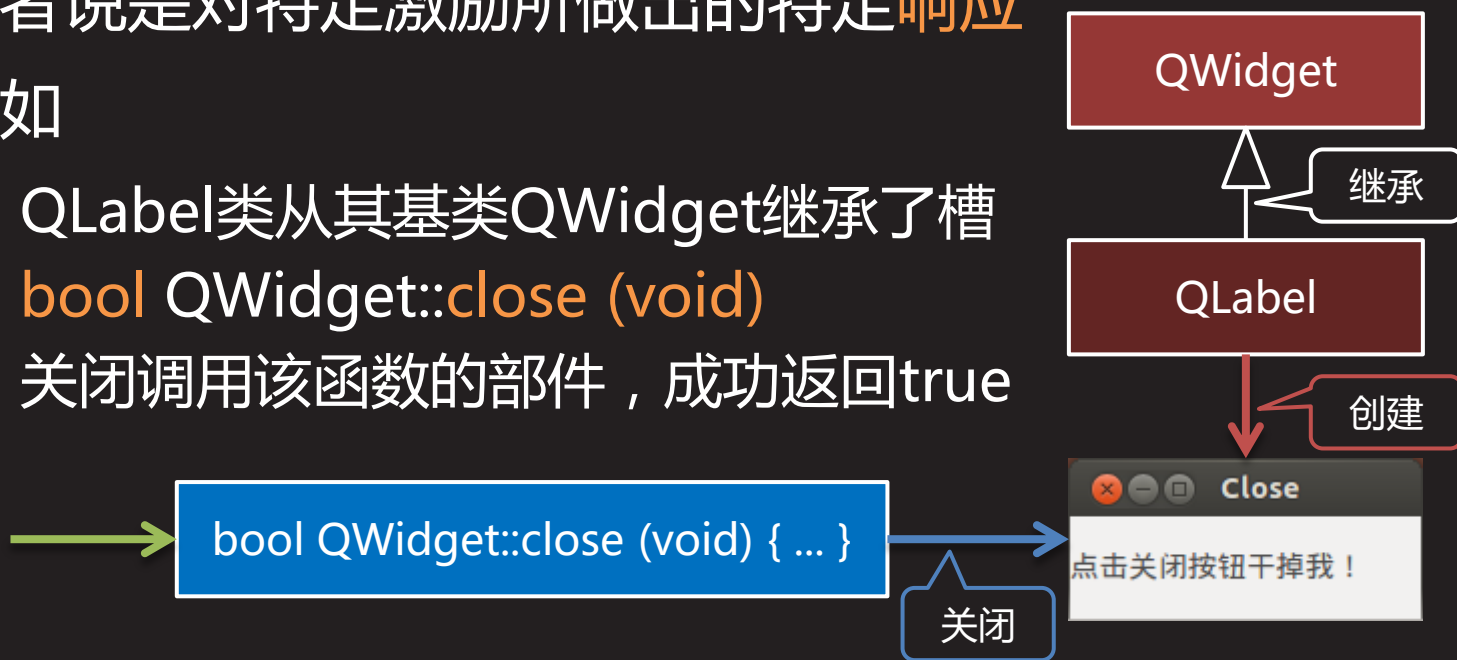
# 信号

- 当用户或系统触发了一个**动作**，导致某个窗口部件的状态发生了改变，该部件就会发射一个信号，即调用其类中一个特定的**成员函数**，同时还可能携带有必要的参数
- 例如
  - QPushButton类从其基类QAbstractButton继承了信号  
**void QAbstractButton::clicked (void)**  
当鼠标光标在按钮窗口范围内被按下并释放时发射此信号



# 槽

- 槽几乎就是一个普通的类**成员函数**——可以是公有的、保护的或私有的，可以被重载，也可以被覆盖，其参数可以是任意类型，并可以在其它函数中被调用
- 槽函数与普通成员函数的差别并不在于其语法特性，而在于其功能。槽函数更多体现为对某种特定信号的处理，或者说是对于特定激励所做出的特定**响应**
- 例如
  - QLabel类从其基类QWidget继承了槽  
**bool QWidget::close (void)**  
关闭调用该函数的部件，成功返回true



# 连接信号和槽

- 既然**信号**表达了由用户或系统所触发的某种**激励**，而**槽**体现了窗口部件对该激励所做出的**响应**，那么剩下的工作就是把表示激励的信号和表示响应的槽**连接**起来，这一步操作是通过**QObject**类的**connect()**成员函数完成的

```
- bool QObject::connect (  
    const QObject* sender,    // 信号发送对象  
    const char* signal,      // 信号函数签名  
    const QObject* receiver, // 信号接收对象  
    const char* method);    // 槽函数的签名
```

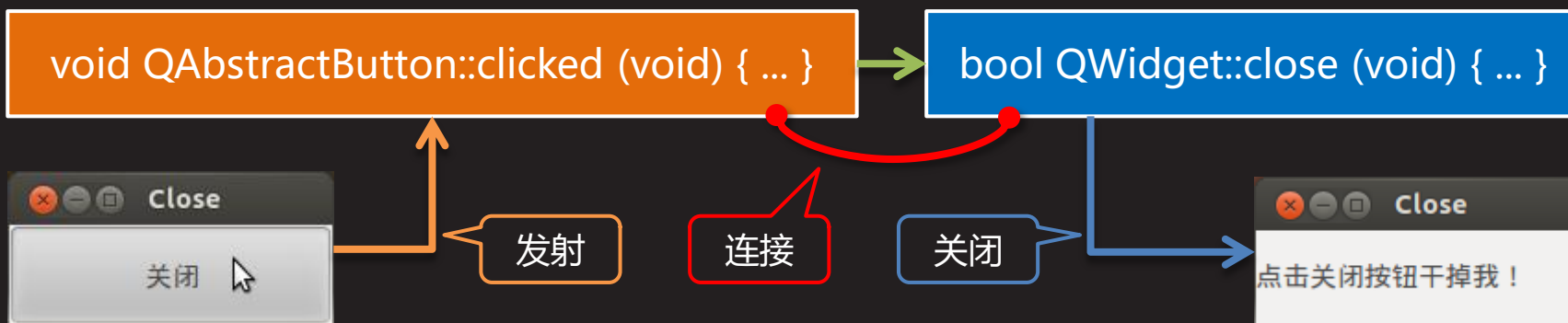
- 其中信号函数和槽函数的**签名**需要借助两个预定义宏
  - ✓ **SIGNAL()** - 将信号函数签名表示成字符串
  - ✓ **SLOT()** - 将槽函数的签名表示成字符串





# 连接信号和槽 (续1)

- 例如
  - `QObject::connect (&btn, SIGNAL (clicked (void)), &lab, SLOT (close (void)));`
  - 将btn对象的clicked信号与lab对象的close槽连接起来
  - 当btn对象因用户点击而发射clicked信号时，lab对象的close槽函数将被调用，于中完成关闭lab窗口部件的操作
  - C++函数可以通过差别化参数表实现重载，必须将函数名和参数表放在一起才能唯一标识一个函数，此即函数签名



# 点击"关闭"按钮，关闭标签

【参见：TTS COOKBOOK】

- 点击"关闭"按钮，关闭标签



# 容器窗口与事件同步

---

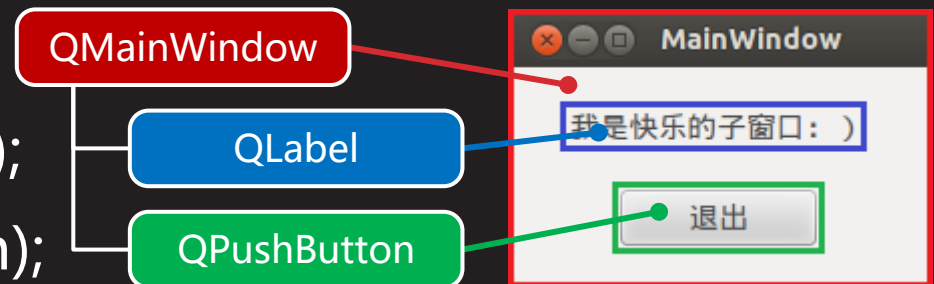


# 容器窗口



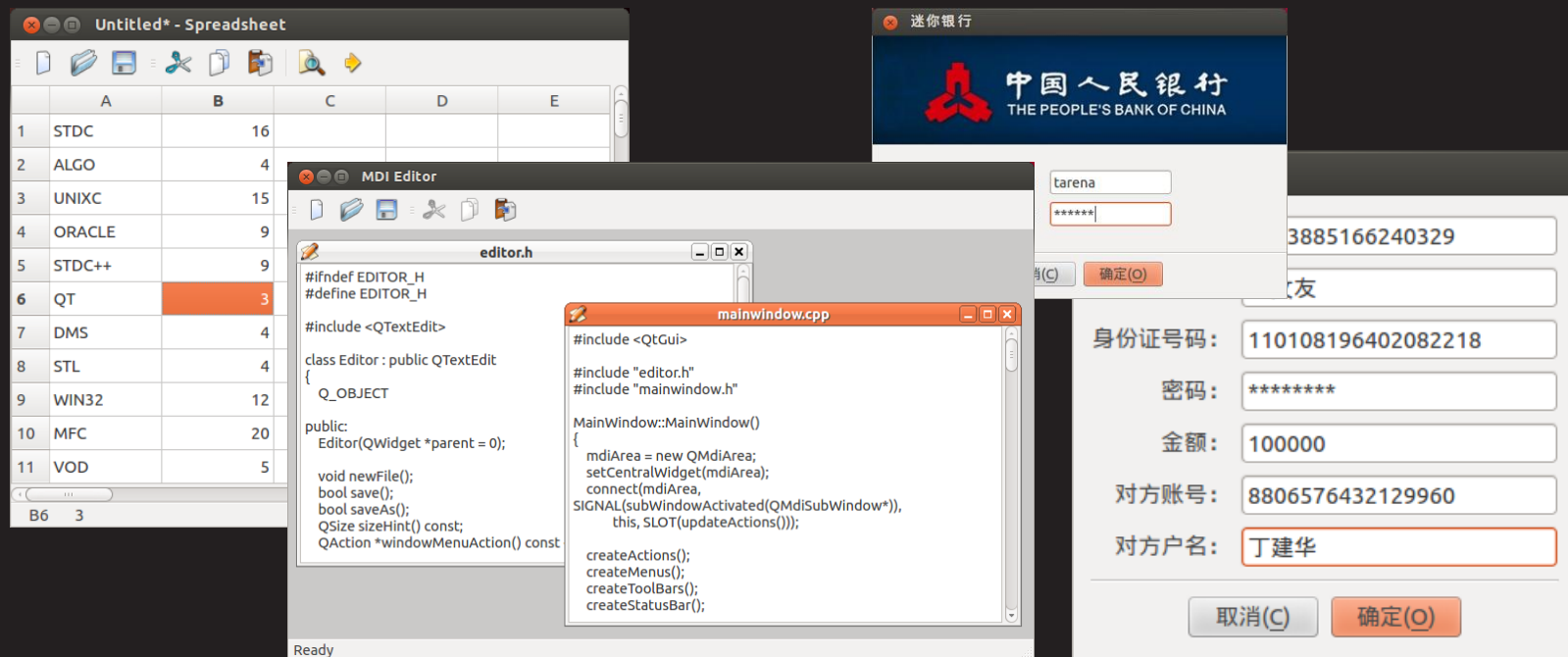
# 父窗口和子窗口

- 如果把一个窗口作为某个容器窗口的子窗口，那么该窗口将被**束缚**在其父窗口的内部，并伴随父窗口一起移动、隐藏、显示和关闭，否则该窗口将作为独立窗口显示在屏幕上，且**游离**于其它窗口之外
- 只有**QWidget**及其子类的对象可以作为其它窗口的容器
- 父窗口的**析构**函数负责**销毁**其所有的子窗口对象，因此即使子窗口对象是通过new运算符动态创建的，也无需因为没有显式地delete它们而担心内存泄漏的风险
- 设置窗口的**位置**和**大小**
  - void move (int x, int y);
  - void resize (int w, int h);



# 常见容器窗口类

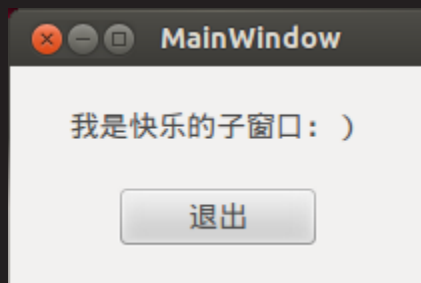
- Qt中常见的容器窗口通常都可被用作应用程序的主框架
  - 主窗口，**QMainWindow**，通常由标题栏、菜单栏、工具条、状态栏以及中央客户区组成
  - 多文档，**QMdiArea**，管理多个多文档子窗口的中央部件
  - 对话框，**QDialog**，管理多个不同种类的交互式部件



# 主窗口

【参见：TTS COOKBOOK】

- 主窗口



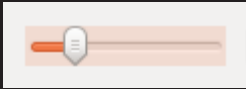
# 事件同步





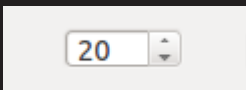
# 滑块和微调框

- QSlider // 滑块类



- void **setRange** (int min, int max); // 设置滑动范围
- void **setPageStep** (int); // 设置页步距
- void **setValue** (int); // 设置当前值
- void **valueChanged** (int value) [signal]; // 值改变信号

- QSpinBox // 微调框类



- void **setRange** (int min, int max); // 设置微调范围
- void **setValue** (int); // 设置当前值
- void **valueChanged** (int value) [signal]; // 值改变信号

# 值改变和改变值

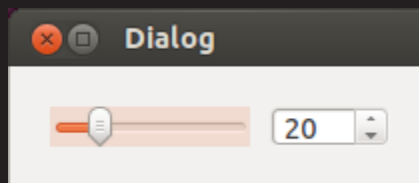
- 滑块和微调框虽然是两个独立的窗口部件，但它们显示的内容确是一个——学生的年龄，因此无论用户拖拽滑块还是翻转微调框，另一个部件都必须做出**相同**的反应
  - QObject::connect (slider, SIGNAL (valueChanged (int)), spin, SLOT (setValue (int)));
  - QObject::connect (spin, SIGNAL (valueChanged (int)), slider, SLOT (setValue (int)));
  - 无论滑块还是微调框，只要其中之一的数值发生变化，就会触发valueChanged信号，而该信号被连接到**另一个**部件的setValue槽上，信号和槽函数的参数就是变化后的值
  - valueChanged信号只有在滑块或微调框中的数值确实发生**变化**的情况下才会触发，因此不必担心无限循环的发生



# 年龄对话框

【参见：TTS COOKBOOK】

- 年龄对话框



# 总结和答疑

