

经常有人问哪里有学习 Qt 的资料，Qt 的教程，怎么才能入门等等，或者抱怨说中文的信息太少。其实网上有很多关于 Qt 的学习资料，今天在这里总结一下，希望各位想学习 Qt 的同学，各取所需，早日从入门到精通！

关于本教程：

本教程是基于您已经了解 C++ 的基础之上，毕竟 Qt 是一个 C++ 库。否则您必须先学习 C++ 后再来看本教程，C++ 入门比较好的书籍是《C++ 程序设计》（谭浩强）。某些人可能觉得《C++ Primer》好，但我认为这本书不适合 C++ 入门，所以还是推荐一下谭老的这本通俗易懂的书吧。

教程由 3 部分组成，第一部分为新手上路，是 Qt 的官方教程。其实把它放在第一部分并不合适，因为对于一个初学 Qt 的人来说一上来就将最好是从一个 Hello World 程序开始学习比较好。所以推荐初学者从第二部分 Qt 学习之路开始阅读。第三部分为 Qt 深入编程，对 Qt 很熟悉的读者可以深入研究研究。教程中的所有代码必须联网才能下载，给您造成的不便之处本人感到非常抱歉。

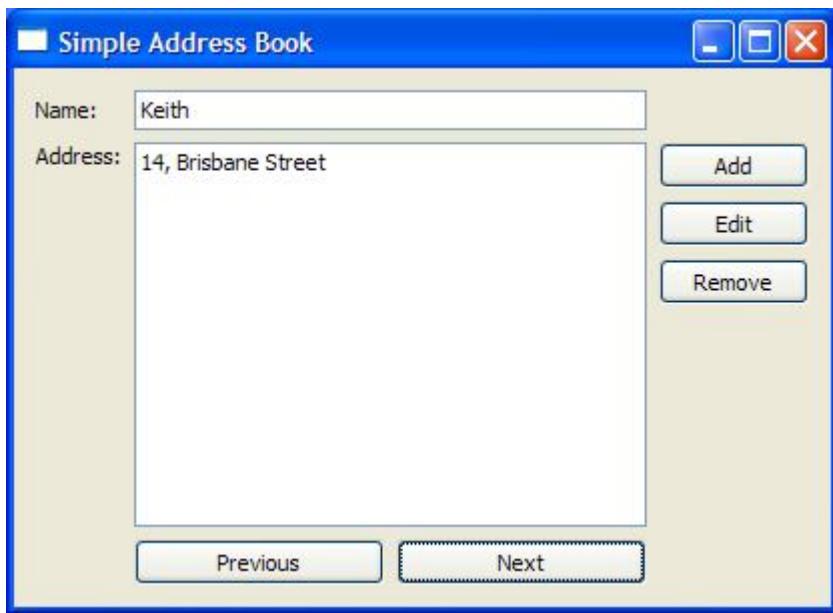
郑重声明：本教程所有文章及代码均非本人所写，教程中尽可能的标明文章的出处。如果侵犯了您的版权可以联系本人，本人将立即删除侵犯版权的内容。E-Mail：yangt1100@163.com

Part 1: 新手上路

Qt 官方学习教程

地址簿教程

本教程介绍了使用 Qt 跨平台框架的 GUI 编程。



在学习过程中，我们将了解部分 Qt 基本技术，如

- Widget 和布局管理器
- 容器类
- 信号和槽
- 输入和输出设备

如果您完全不了解 Qt，请阅读[如何学习 Qt](#)（如果您还未阅读）。

教程的源代码位于 Qt 的 examples/tutorials/addressbook 目录下。

教程章节：

1. 设计用户界面
2. 添加地址
3. 浏览地址簿条目
4. 编辑和删除地址
5. 添加查找功能

6. 加载和保存

7. 附加功能

虽然这个小型应用程序看起来并不象一个成熟的现代 GUI 应用程序，但它使用多种用于更复杂应用程序的基本技术。在您完成学习之后，我们建议您查看一下 [应用程序示例](#)，它提供带有菜单、工具栏、状态栏等项目的大型 GUI 应用程序。

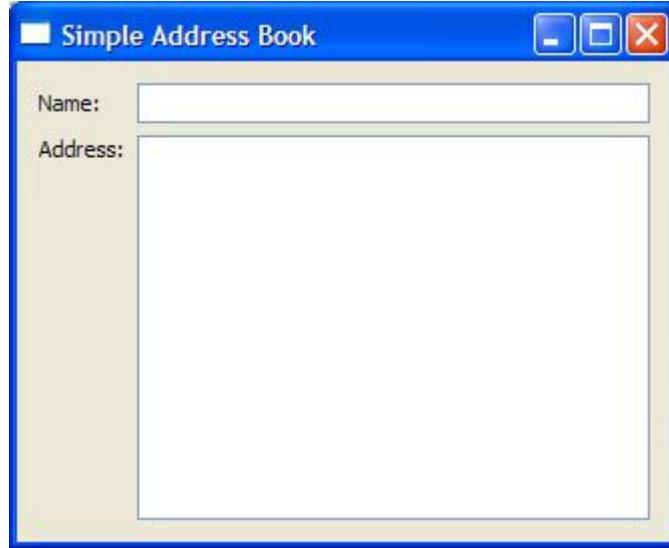
地址簿 1 — 设计用户界面

文件：

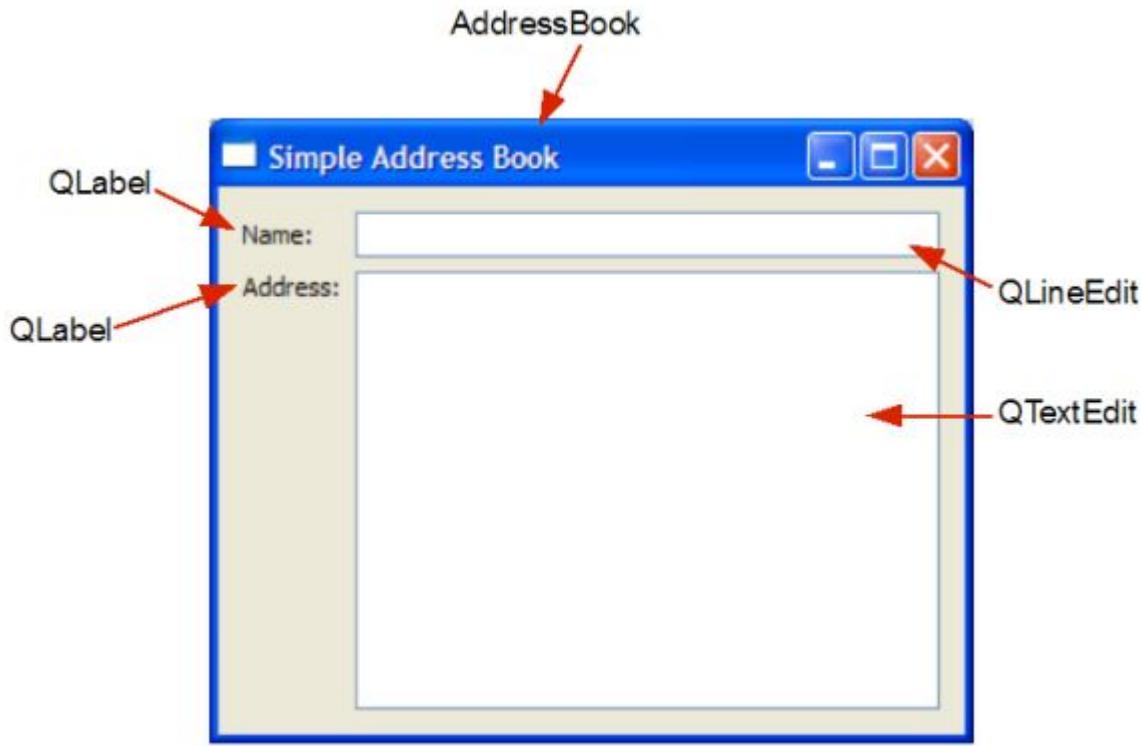
- [tutorials/addressbook/part1/addressbook.cpp](#)
- [tutorials/addressbook/part1/addressbook.h](#)
- [tutorials/addressbook/part1/main.cpp](#)
- [tutorials/addressbook/part1/part1.pro](#)

本教程的第一部分讲述了用于地址簿应用程序的基本图形用户界面（GUI）的设计。

创建 GUI 程序的第一步就是设计用户界面。在本章中，我们的目标是设置应用基本地址簿应用程序所需的标签和输入字段。下图为期望输出的屏幕截图。



我们需要使用两个 [QLabel](#) 对象：nameLabel 和 addressLabel，以及两个输入字段：[QLineEdit](#) 对象 nameLine 和 [QTextEdit](#) 对象 addressText，这样用户才能输入联系人的姓名和地址。使用的 widget 及其位置如下图所示。



要应用地址簿需使用三个文件：

- addressbook.h — AddressBook 类的定义文件，
- addressbook.cpp — AddressBook 类的执行文件，以及
- main.cpp — 包含 main() 函数并带有 AddressBook 实例的文件。

Qt 编程 — 使用子类

在编写 Qt 程序时，我们通常使用 Qt 对象子类来添加功能。这是创建定制 widget 或标准 widget 集合的基本概念之一。使用子类扩展或改变 widget 的操作具有以下优势：

- 我们可以编写虚函数或纯虚函数应用，以得到我们确切所需的功能，并在需要时再使用基本的类应用。
- 这样我们就可以在类中封装部分用户界面，应用程序的其他部分也就无需了解用户界面中单独使用的 widget。
- 可使用子类在同一应用程序或库中创建多个定制 widget，这样子类的代码可在其他项目重复使用。

由于 Qt 未提供特定的地址簿 widget，我们在标准的 Qt widget 类中使用子类，然后添加功能。我们在本教程中创建的 AddressBook 类在需要使用基本地址簿 widget 的情况下可重复使用。

定义 AddressBook 类

`addressbook.h` 文件用于定义 AddressBook 类。

我们从定义 AddressBook 为 `QWidget` 子类和声明构造器开始入手。我们还使用 `Q_OBJECT` 宏表明该类使用国际化功能与 Qt 信号和槽功能，即使在本阶段不会用到所有这些功能。

```
class AddressBook : public QWidget
{
    Q_OBJECT

public:
    AddressBook(QWidget *parent = 0);

private:
    QLineEdit *nameLine;
    QTextEdit *addressText;
};
```

该类包含了 `nameLine` 和 `addressText` 的声明、上文提到的 `QLineEdit` 和 `QTextEdit` 的私有实例。在以后章节中，您会看到储存在 `nameLine` 和 `addressText` 中的数据在地址簿的许多功能中都会用到。

我们不必包含要使用的 `QLabel` 对象的声明，这是因为在创建这些对象后我们不必对其进行引用。在下一部分中，我们会说明 Qt 记录对象所属关系的方式。

`Q_OBJECT` 宏本身应用了部分更高级的 Qt 功能。我们暂时把 `Q_OBJECT` 宏理解为使用 `tr()` 和 `connect()` 函数的快捷方式，这种理解对我们的学习更有用。

我们现已完成 `addressbook.h` 文件，接下来我们来执行对应的 `addressbook.cpp` 文件。

应用 AddressBook 类

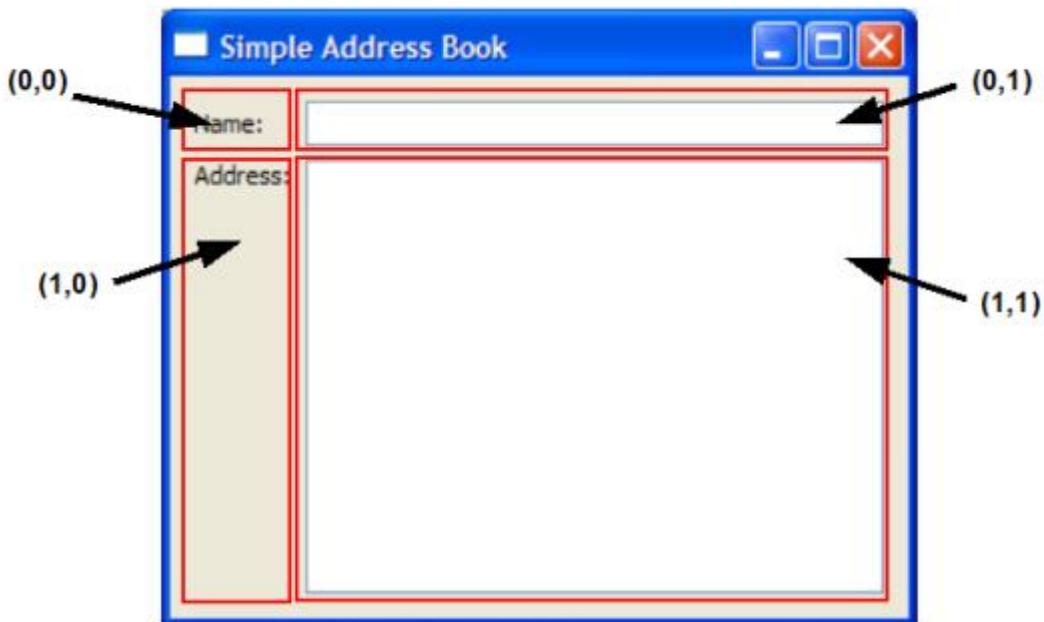
AddressBook 的构造器接收 `QWidget` 参数 `parent`。按惯例，我们将参数传递给基本类的构造器。这种父项可有一个或多个子项的所属概念对 Qt 中的 widget 分组十分有用。例如，如果删除父项，也会删除其所有子项。

```
AddressBook::AddressBook(QWidget *parent)
    : QWidget(parent)
{
    QLabel *nameLabel = new QLabel(tr("Name:"));
```

```
nameLine = new QLineEdit;  
  
QLabel *addressLabel = new QLabel(tr("Address:"));  
addressText = new QTextEdit;
```

在该构造器中，我们声明并通过实例来表示两个局部 `QLabel` 对象 `nameLabel` 和 `addressLabel`，以及 `nameLine` 和 `addressText`。如果字符串已进行转换，则 `tr()` 函数返回已转换的字符串，否则返回字符串本身。我们可以将此函数理解 `<insert translation here>` 标识来标记要进行转换 `QString` 对象。在以后章节和 `Qt Examples` 中，您会看到只要使用了可转换的字符串就是使用该函数。

使用 Qt 编程时，了解布局是如何起作用的会对您很有帮助。Qt 提供三个主要布局类 `QHBoxLayout`、`QVBoxLayout` 和 `QGridLayout` 来处理 widget 的位置。



我们使用 `QGridLayout` 以结构化的方式放置标签和输入字段。`QGridLayout` 将可用空间分为网格，并将 widget 放置在指定了行列号的单元格中。上面的图表显示了布局单元格和 widget 的位置。我们通过以下代码指定这种排列方式：

```
QGridLayout *mainLayout = new QGridLayout;  
mainLayout->addWidget(nameLabel, 0, 0);  
mainLayout->addWidget(nameLine, 0, 1);  
mainLayout->addWidget(addressLabel, 1, 0, Qt::AlignTop);  
mainLayout->addWidget(addressText, 1, 1);
```

请注意，`addressLabel` 是使用作为附加参数的 `Qt::AlignTop` 来排放位置。这可确保其不会纵向放置在单元格 (1, 0) 中央。有关 Qt 布局的基本简介，请参见[布局类](#)文档。

要在 `widget` 上安装布局对象，必须调用 `widget` 的 `setLayout()` 函数：

```
setLayout(mainLayout);
setWindowTitle(tr("Simple Address Book"));
}
```

最后，我们将 `widget` 标题设置为“简单地址簿”。

运行应用程序

`main()` 函数使用单独的文件 `main.cpp`。在该函数中，我们实例化了 `QApplication` 对象 `app`。`QApplication` 负责管理多种应用范围的资源（如默认字体和光标），以及运行事件循环。因此，在每个使用 Qt 的 GUI 应用程序中都会有一个 `QApplication` 对象。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    AddressBook *addressBook = new AddressBook;
    addressBook->show();

    return app.exec();
}
```

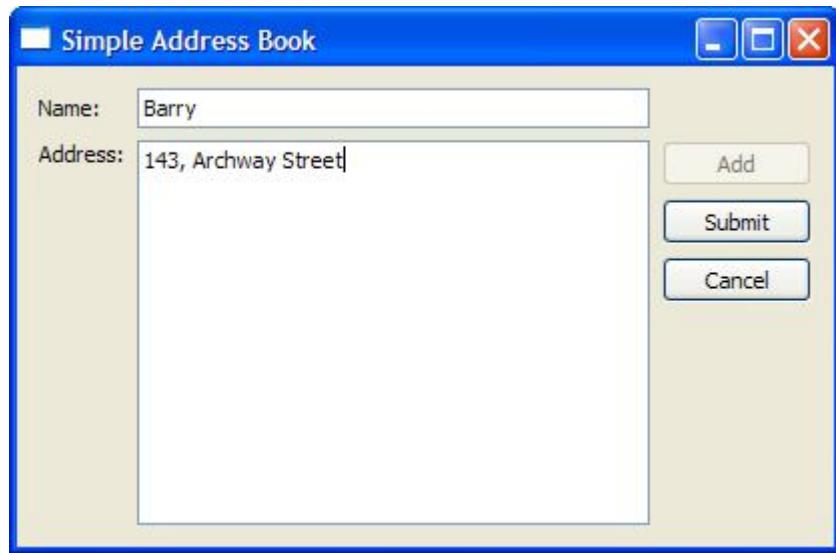
我们使用 `new` 关键字在堆中构造一个新的 `AddressBook` `widget`，然后调用 `show()` 函数对其进行显示。不过，该 `widget` 只有在应用程序事件循环开始时才会显示。我们通过调用应用程序的 `exec()` 函数开始事件循环。该函数返回的结果作为 `main()` 函数的返回值。

地址簿 2 — 添加地址

文件：

- `tutorials/addressbook/part2/addressbook.cpp`
- `tutorials/addressbook/part2/addressbook.h`
- `tutorials/addressbook/part2/main.cpp`
- `tutorials/addressbook/part2/part2.pro`

创建基本地址簿应用程序的下一步是添加少许用户互动操作。



我们将提供一个按钮，用户可点击该按钮来添加新联系人。此外，还需要对数据结构进行限定，以便有序地储存这些联系人。

定义 AddressBook 类

由于已经设置了标签和输入字段，我们只需添加按钮就可完成添加联系人这一步骤。也就是说，在 addressbook.h 文件中已经声明了三个 `QPushButton` 对象和三个对应的公共槽。

```
public slots:  
void addContact();  
void submitContact();  
void cancel();
```

槽是对特殊信号进行响应的函数。我们将在应用 `AddressBook` 类时进一步详细说明这一概念。如需有关 Qt 信号和槽概念的简介，请参见[信号和槽](#)文档。

三个 `QPushButton` 对象分别是 `addButton`、`submitButton` 和 `cancelButton`，已与要在上一章中说明的 `nameLine` 和 `addressText` 一同包含在私有变量声明中。

```
private:  
QPushButton *addButton;  
QPushButton *submitButton;  
QPushButton *cancelButton;  
QLineEdit *nameLine;
```

```
QTextEdit *addressText;
```

我们需要一个容器来储存地址簿联系人，这样才能搜索和显示联系人。 `QMap` 对象 `contacts` 就可实现此功能，因为其带有一个键-值对：联系人姓名作为键，而联系人地址作为值。

```
QMap<QString, QString> contacts;  
QString oldName;  
QString oldAddress;  
};
```

我们还会声明两个私有 `QString` 对象：`oldName` 和 `oldAddress`。这些对象用来保留在用户点击添加时最后显示的联系人姓名和地址。这样，当用户点击取消时，我们就可以返回至上一个联系人的详细信息。

应用 AddressBook 类

在 `AddressBook` 构造器中，我们将 `nameLine` 和 `addressText` 设置为只读，这样就可仅显示而不必编辑现有联系人的详细信息。

```
...  
nameLine->setReadOnly(true);  
...  
addressText->setReadOnly(true);
```

然后，我们实例化以下按钮： `addButton`、 `submitButton` 和 `cancelButton`。

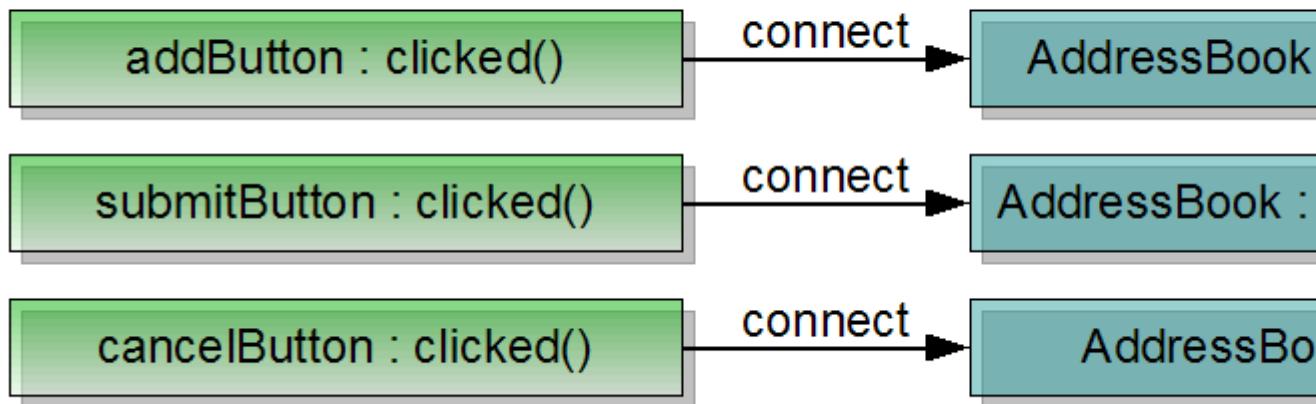
```
addButton = new QPushButton(tr("&Add"));  
addButton->show();  
submitButton = new QPushButton(tr("&Submit"));  
submitButton->hide();  
cancelButton = new QPushButton(tr("&Cancel"));  
cancelButton->hide();
```

显示 `addButton` 是通过调用 `show()` 函数实现的，而隐藏 `submitButton` 和 `cancelButton` 则需调用 `hide()`。这两个按钮仅当用户点击添加时才会显示，而此操作是通过在下文中说明的 `addContact()` 函数处理的。

```
connect(addButton, SIGNAL(clicked()), this, SLOT(addContact()));  
connect(submitButton, SIGNAL(clicked()), this, SLOT(submitContact()));  
connect(cancelButton, SIGNAL(clicked()), this, SLOT(cancel()));
```

我们将按钮的 `clicked()` 信号与其相应的槽关联。下面的图表说明了此过程。

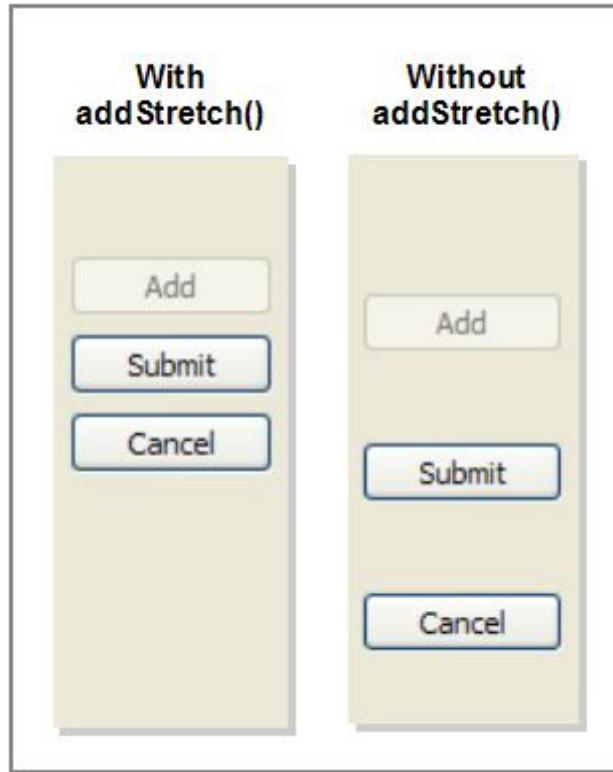
Signals



接下来,我们将按钮整齐的排列在地址簿 widget 的右侧, 使用 `QVBoxLayout` 将其进行纵向排列。

```
QVBoxLayout *buttonLayout1 = new QVBoxLayout;
buttonLayout1->addWidget(addButton, Qt::AlignTop);
buttonLayout1->addWidget(submitButton);
buttonLayout1->addWidget(cancelButton);
buttonLayout1->addStretch();
```

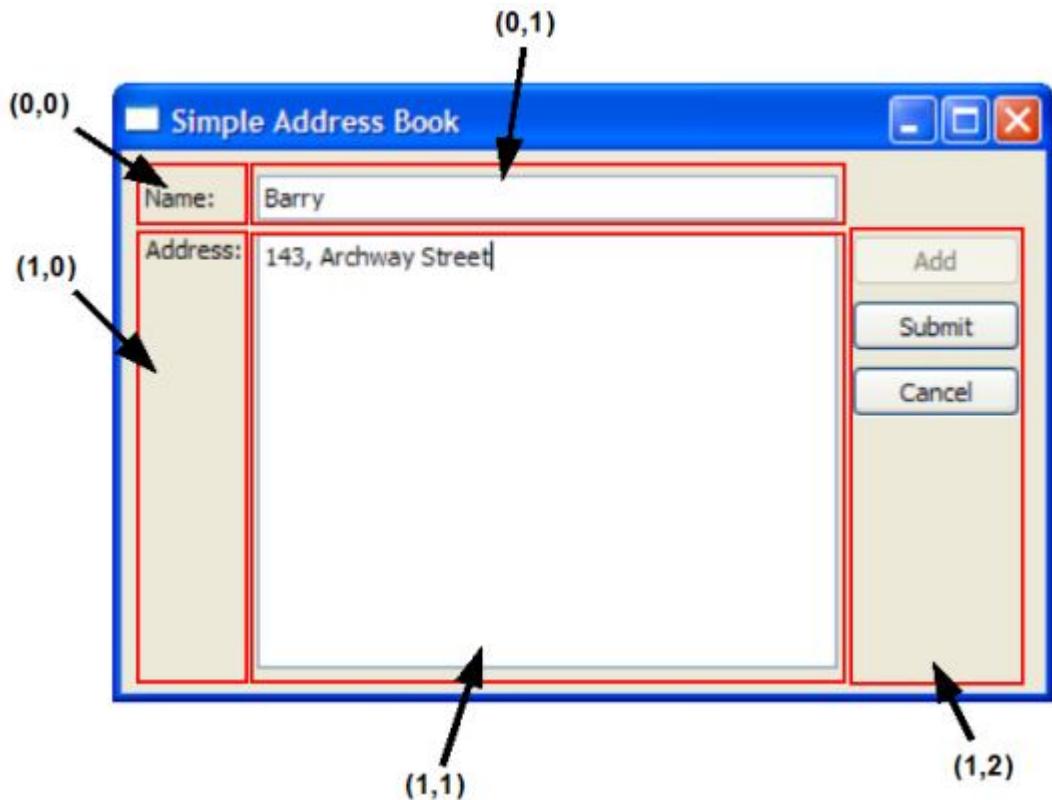
`addStretch()` 函数用来确保按钮并不是采用均匀间隔排列的, 而是更靠近 `widget` 的顶部。下图显示了是否使用 `addStretch()` 的差别。



然后，我们使用 `addLayout()` 将 `buttonLayout1` 增加至 `mainLayout`。这样我们就有了嵌套布局，因为 `buttonLayout1` 现在是 `mainLayout` 的子项。

```
QGridLayout *mainLayout = new QGridLayout;
mainLayout->addWidget(nameLabel, 0, 0);
mainLayout->addWidget(nameLine, 0, 1);
mainLayout->addWidget(addressLabel, 1, 0, Qt::AlignTop);
mainLayout->addWidget(addressText, 1, 1);
mainLayout->addLayout(buttonLayout1, 1, 2);
```

布局坐标显示如下：



在 `addContact()` 函数中，我们使用 `oldName` 和 `oldAddress` 储存最后显示的联系人详细信息。然后，我们清空这些输入字段并关闭只读模式。输入焦点设置在 `nameLine`，显示 `submitButton` 和 `cancelButton`。

```
void AddressBook::addContact()
{
    oldName = nameLine->text();
    oldAddress = addressText->toPlainText();

    nameLine->clear();
    addressText->clear();

    nameLine->setReadOnly(false);
    nameLine->setFocus(Qt::OtherFocusReason);
    addressText->setReadOnly(false);

    addButton->setEnabled(false);
    submitButton->show();
    cancelButton->show();
}
```

`submitContact()` 函数可分为三个部分：

1. 我们从 nameLine 和 addressText 提取联系人的详细信息，然后将其储存在 `QString` 对象中。我们还要验证确保用户没有在输入字段为空时点击提交，否则，会显示 `QMessageBox` 提示用户输入姓名和地址。

```
2.     void AddressBook::submitContact()
3.     {
4.         QString name = nameLine->text();
5.         QString address = addressText->toPlainText();
6.
7.         if (name == "" || address == "") {
8.             QMessageBox::information(this, tr("Empty Field"),
9.                                     tr("Please enter a name and address."));
10.            return;
11.
12.        }
13.    }
```

11. 我们接着继续检查是否联系人已存在。如果不存在，将联系人添加至 contacts，然后显示 `QMessageBox` 提示用户已添加联系人。

```
12.    if (!contacts.contains(name)) {
13.        contacts.insert(name, address);
14.        QMessageBox::information(this, tr("Add Successful"),
15.                                tr("\'%1\' has been added to your address book.").arg(name));
16.    } else {
17.        QMessageBox::information(this, tr("Add Unsuccessful"),
18.                                tr("Sorry, \'%1\' is already in your address book.").arg(name));
19.    }
20.
```

如果联系人已存在，还是会显示 `QMessageBox` 以提示用户，以免添加重复的联系人。由于 contacts 对象是基于姓名地址的键-值对，因此要确保键唯一。

20. 在处理了上述两种情况后，使用以下代码将按钮恢复为正常状态：

```
21.    if (contacts.isEmpty()) {
22.        nameLine->clear();
23.        addressText->clear();
24.    }
25.
26.    nameLine->setReadOnly(true);
27.    addressText->setReadOnly(true);
28.    addButton->setEnabled(true);
29.    submitButton->hide();
30.    cancelButton->hide();
31.
```

下面的屏幕截图显示了用于向用户显示提示信息的 `QMessageBox` 对象。



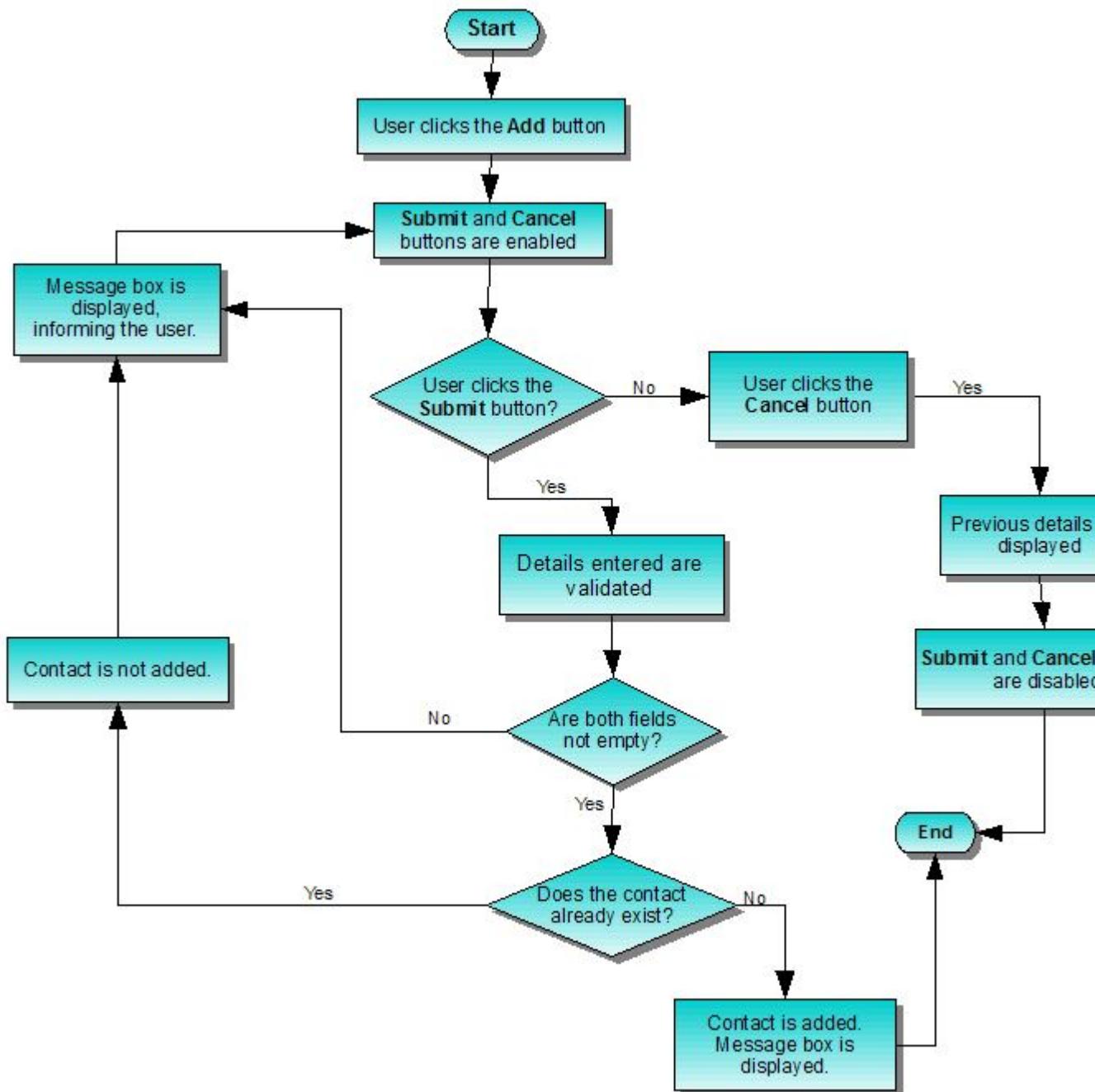
`cancel()` 函数恢复上次显示的联系人详细信息，并启用 `addButton`，还会隐藏 `submitButton` 和 `cancelButton`。

```
void AddressBook::cancel()
{
    nameLine->setText(oldName);
    nameLine->setReadOnly(true);

    addressText->setText(oldAddress);
    addressText->setReadOnly(true);

    addButton->setEnabled(true);
    submitButton->hide();
    cancelButton->hide();
}
```

添加联系人的总体思想就是提高用户操作的灵活性，可在任何时候点击提交或取消。下面的流程图详细说明了此概念：



地址簿 3 — 浏览地址簿条目

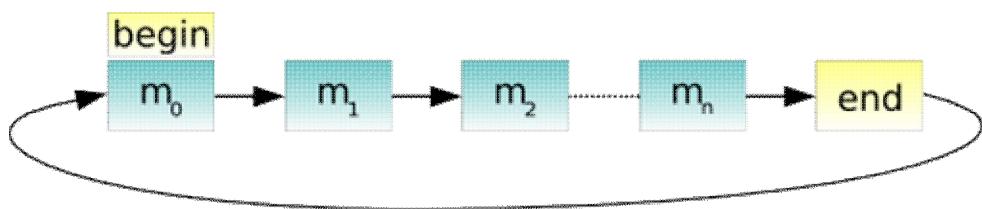
文件:

- tutorials/addressbook/part3/addressbook.cpp
- tutorials/addressbook/part3/addressbook.h
- tutorials/addressbook/part3/main.cpp
- tutorials/addressbook/part3/part3.pro

构建地址簿应用程序现已进展过半。我们需要增加一些函数，以便浏览联系人。但首先要决定采用何种数据结构方式来储存这些联系人。

在第二章中，我们使用了 [QMap](#) 键-值对，即联系人姓名作为键，而联系人地址作为值。这种方式很适合我们的实例。不过，要浏览和显示每个条目，还需要进行一些改进。

我们改进 [QMap](#) 的方式是，将数据结构替换为类似循环链接的列表，其中所有元素都是相互关联的，包括第一个元素和最后一个元素。下图图解说明了该数据结构。



定义 AddressBook 类

要给地址簿应用程序增加浏览功能，我们需要为 `AddressBook` 类再增加两个函数：`next()` 和 `previous()`。将这两个函数添加到 `addressbook.h` 文件中：

```
void next();
void previous();
```

我们还需要使用其他两个 [QPushButton](#) 对象，因此将 `nextButton` 和 `previousButton` 声明为私有变量：

```
QPushButton *nextButton;
QPushButton *previousButton;
```

应用 AddressBook 类

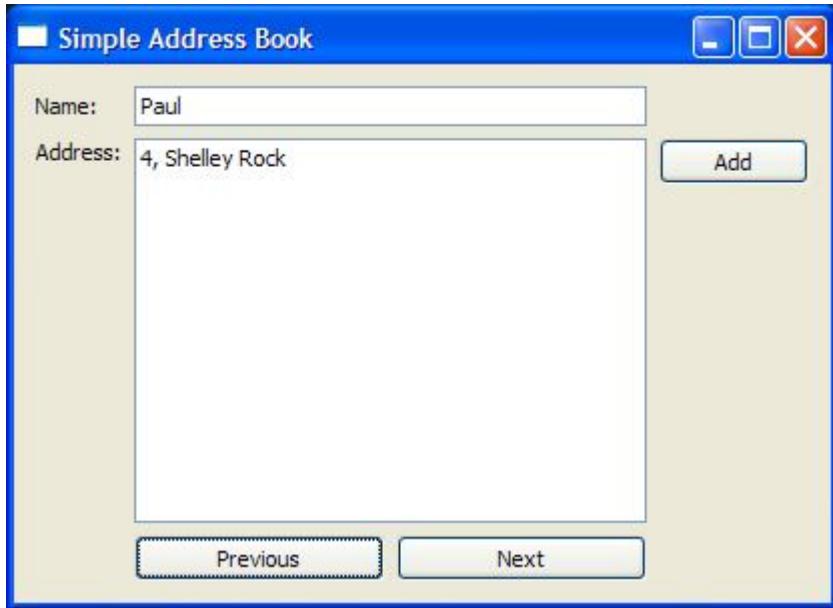
在 `addressbook.cpp` 的 `AddressBook` 构造器中，我们实例化 `nextButton` 和 `previousButton`，并且这两项默认为禁用。这是因为仅当地址簿中有多个联系人时才会启用浏览功能。

```
nextButton = new QPushButton(tr("&Next"));
nextButton->setEnabled(false);
previousButton = new QPushButton(tr("&Previous"));
previousButton->setEnabled(false);
```

然后，我们将这两个按钮与其相应的槽关联：

```
connect(nextButton, SIGNAL(clicked()), this, SLOT(next()));  
connect(previousButton, SIGNAL(clicked()), this, SLOT(previous()));
```

下图即为预期的图形用户界面。请注意，该用户界面已很接近应用程序最终的样子。



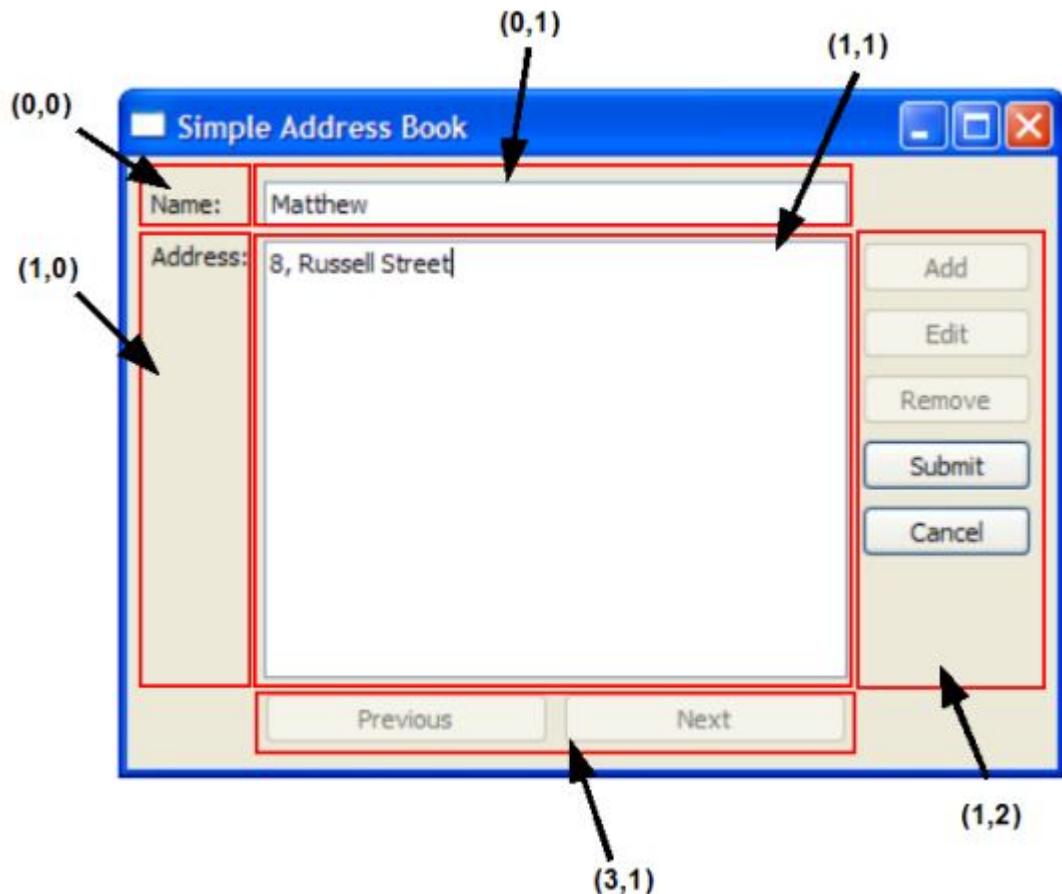
我们按照 `next()` 和 `previous()` 函数的基本规范，将 `nextButton` 放置在右侧，而 `previousButton` 放置在左侧。为了使布局更加直观，我们使用 [QHBoxLayout](#) 将 widget 并排放置：

```
QHBoxLayout *buttonLayout2 = new QHBoxLayout;  
buttonLayout2->addWidget(previousButton);  
buttonLayout2->addWidget(nextButton);
```

然后，将 [QHBoxLayout](#) 对象 `buttonLayout2` 增加至 `mainLayout`。

```
mainLayout->addLayout(buttonLayout2, 3, 1);
```

下图显示了 `widget` 在 `mainLayout` 中的坐标位置。



在 `addContact()` 函数中，我们必须禁用这几个按钮，这样用户就不会在增加联系人时尝试进行浏览。

```
nextButton->setEnabled(false);
previousButton->setEnabled(false);
```

此外，在 `submitContact()` 函数中，我们启用了浏览按钮 `nextButton` 和 `previousButton`，这取决于 `contacts` 的多少。如上文所述，浏览功能仅在地址簿中有多个联系人时才会启用。以下代码行说明了如何实现此功能：

```
int number = contacts.size();
nextButton->setEnabled(number > 1);
previousButton->setEnabled(number > 1);
```

我们还在 `cancel()` 函数中加入这几行代码。

记得我们曾使用 `QMap` 对象 `contacts` 模拟了一个循环链接的列表。因此，在 `next()` 函数中，我们获取 `contacts` 的迭代器，然后执行以下操作：

- 如果迭代器未达到 `contacts` 结尾，就会增加一。

- 如果迭代器已达到 contacts 的结尾，就移至 contacts 的起始位置。这给人感觉 QMap 就像是一个循环链接的列表。

```
void AddressBook::next()
{
    QString name = nameLine->text();
    QMap<QString, QString>::iterator i = contacts.find(name);

    if (i != contacts.end())
        i++;

    if (i == contacts.end())
        i = contacts.begin();

    nameLine->setText(i.key());
    addressText->setText(i.value());
}
```

一旦在 contacts 中循环至正确的对象，就会通过 nameLine 和 addressText 显示对象的内容。

同样，在 previous() 函数中，我们获取 contacts 的迭代器，然后执行以下操作：

- 如果迭代器达到 contacts 的结尾，就清除显示内容，然后返回。
- 如果迭代器在 contacts 的起始位置，就将其移至结尾。
- 然后，将迭代器减一。

```
void AddressBook::previous()
{
    QString name = nameLine->text();
    QMap<QString, QString>::iterator i = contacts.find(name);

    if (i == contacts.end()){
        nameLine->clear();
        addressText->clear();
        return;
    }

    if (i == contacts.begin())
        i = contacts.end();

    i--;
    nameLine->setText(i.key());
```

```
    addressText->setText(i.value());  
}
```

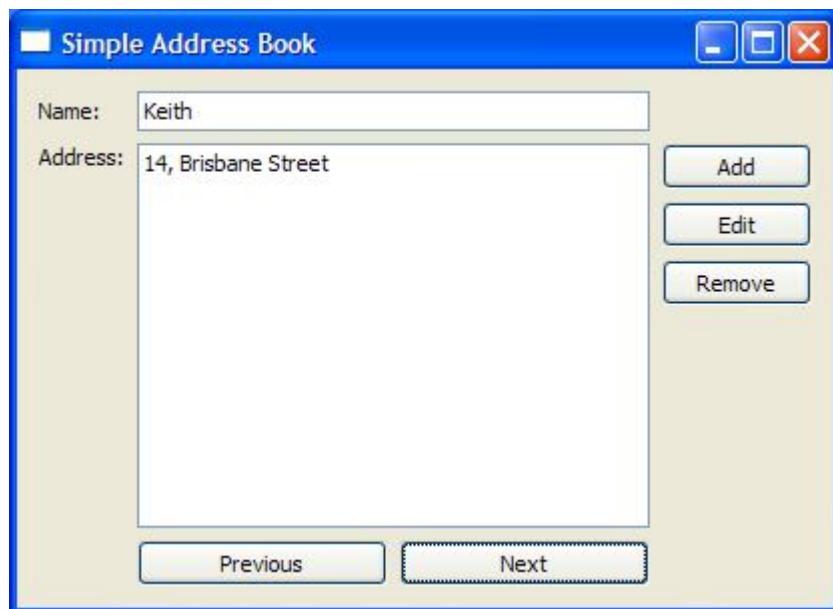
接着，重新显示 contacts 中当前对象的内容。

地址簿 4 — 编辑和删除地址

文件：

- [tutorials/addressbook/part4/addressbook.cpp](#)
- [tutorials/addressbook/part4/addressbook.h](#)
- [tutorials/addressbook/part4/main.cpp](#)
- [tutorials/addressbook/part4/part4.pro](#)

在本章中，我们将了解如何修改储存在地址簿应用程序中的联系人的内容。



现有的地址簿不仅可以井井有条地储存联系人，还可进行浏览。再添加上编辑和删除功能，以便在需要时更改联系人的详细信息，这样更易于使用。不过，还需使用 enum 类型进行一些改进。在前几章中，我们使用以下两种模式：AddingMode 和 NavigationMode。但是，他们并未定义为 enum。我们是采用手动方式启用和禁用相应的按钮，这就导致有多行重复的代码。

在本章中，我们定义带有以下三种不同值的 Mode enum 类型：

- NavigationMode、
- AddingMode 和
- EditingMode。

定义 AddressBook 类

addressbook.h 文件已更新为包含 Mode enum 类型：

```
enum Mode { NavigationMode, AddingMode, EditingMode };
```

我们还要向当前的公有槽列表增加两个新槽：editContact() 和 removeContact()。

```
void editContact();
void removeContact();
```

为了在模式间切换，我们引入了 updateInterface() 函数来控制所有 QPushbutton 对象的启用和禁用。要实现上文提及的编辑和删除功能，我们还要增加两个新按钮：editButton 和 removeButton。

```
void updateInterface(Mode mode);
...
QPushbutton *editButton;
QPushbutton *removeButton;
...
Mode currentMode;
```

最后，我们声明 currentMode 来跟踪 enum 的当前模式。

应用 AddressBook 类

我们现在必须应用地址簿应用程序的模式更改功能。editButton 和 removeButton 已实例化并默认为禁用，这是因为地址簿启动时在内存中没有联系人。

```
editButton = new QPushbutton(tr("&Edit"));
editButton->setEnabled(false);
removeButton = new QPushbutton(tr("&Remove"));
removeButton->setEnabled(false);
```

这些按钮会与其相应的槽 editContact() 和 removeContact() 关联，然后我们将其添加至 buttonLayout1。

```
connect(editButton, SIGNAL(clicked()), this, SLOT(editContact()));
connect(removeButton, SIGNAL(clicked()), this, SLOT(removeContact()));
...
```

```
buttonLayout1->addWidget(editButton);
buttonLayout1->addWidget(removeButton);
```

在将模式切换到 `EditingMode` 之前，`editContact()` 函数使用 `oldName` 和 `oldAddress` 储存联系人旧的详细信息。在该模式下，`submitButton` 和 `cancelButton` 均已启用，这样用户就可以更改联系人的详细信息并可点击任何一个按钮。

```
void AddressBook::editContact()
{
    oldName = nameLine->text();
    oldAddress = addressText->toPlainText();

    updateInterface(EdittingMode);
}
```

`submitContact()` 函数已被 `if-else` 语句分为两部分。我们查看 `currentMode` 是否在 `AddingMode` 模式下。如果是，我们继续添加操作。

```
void AddressBook::submitContact()
{
    ...
    if (currentMode == AddingMode) {

        if (!contacts.contains(name)) {
            contacts.insert(name, address);
            QMessageBox::information(this, tr("Add Successful"),
                tr("\"%1\" has been added to your address book.").arg(name));
        } else {
            QMessageBox::information(this, tr("Add Unsuccessful"),
                tr("Sorry, \"%1\" is already in your address book.").arg(name));
            return;
        }
    }
}
```

否则，我们查看 `currentMode` 是否在 `EditMode` 模式下。如果是，我们比较 `oldName` 和 `name`。如果姓名已更改，我们从 `contacts` 中删除旧的联系人并插入已更新的联系人。

```
} else if (currentMode == EdittingMode) {

    if (oldName != name) {
        if (!contacts.contains(name)) {
            QMessageBox::information(this, tr("Edit Successful"),
                tr("\"%1\" has been edited in your address book.").arg(oldName));
```

```

        contacts.remove(oldName);
        contacts.insert(name, address);
    } else {
        QMessageBox::information(this, tr("Edit Unsuccessful"),
                               tr("Sorry, \"%1\" is already in your address book.").arg(name));
        return;
    }
} else if (oldAddress != address) {
    QMessageBox::information(this, tr("Edit Successful"),
                           tr(" \"%1\" has been edited in your address book.").arg(name));
    contacts[name] = address;
}
}

updateInterface(NavigationMode);
}

```

如果仅更改了地址（例如 `oldAddress` 与 `address` 不同），我们就更新联系人的地址。最后，我们将 `currentMode` 设置为 `NavigationMode`。这一步至关重要，因为它会重新启用所有已禁用的按钮。

要从地址簿中删除联系人，我们采用 `removeContact()` 函数。该函数查看 `contacts` 中是否包含该联系人。

```

void AddressBook::removeContact()
{
    QString name = nameLine->text();
    QString address = addressText->toPlainText();

    if (contacts.contains(name)) {

        int button = QMessageBox::question(this,
                                           tr("Confirm Remove"),
                                           tr("Are you sure you want to remove \"%1\"?").arg(name),
                                           QMessageBox::Yes | QMessageBox::No);

        if (button == QMessageBox::Yes) {

            previous();
            contacts.remove(name);

            QMessageBox::information(this, tr("Remove Successful"),
                                   tr(" \"%1\" has been removed from your address book.").arg(name));
        }
    }
}

```

```
    }

    updateInterface(NavigationMode);
}
```

如果有，我们显示 [QMessageBox](#)，确认用户的删除操作。一旦用户确认操作，我们调用 `previous()` 确保用户界面显示其他联系人，然后我们使用 [QMap](#) 的 `remove()` 函数删除已确认的联系人。出于好意，我们会显示 [QMessageBox](#) 提示用户。在该函数中使用两种信息框显示如下：



更新用户界面

我们在上文提到 `updateInterface()` 函数，它可根据当前的模式启用和禁用按钮。该函数会根据传递给它的 `mode` 参数更新当前的模式，在校验值之前将参数分配给 `currentMode`。

这样，每个按钮就根据当前的模式进行启用或禁用。`AddingMode` 和 `EditingMode` 的代码显示如下：

```
void AddressBook::updateInterface(Mode mode)
{
    currentMode = mode;

    switch (currentMode) {

        case AddingMode:
        case EditingMode:

            nameLine->setReadOnly(false);
            nameLine->setFocus(Qt::OtherFocusReason);
            addressText->setReadOnly(false);

            addButton->setEnabled(false);
            editButton->setEnabled(false);
            removeButton->setEnabled(false);
    }
}
```

```
nextButton->setEnabled(false);
previousButton->setEnabled(false);

submitButton->show();
cancelButton->show();
break;
```

不过对于 `NavigationMode`, 我们在 `QPushButton::setEnabled()` 函数的参数中加入了条件。这样可确保 `editButton` 和 `removeButton` 在地址簿中至少有一个联系人的情况下启用, 而 `nextButton` 和 `previousButton` 仅在地址簿中有多个联系人时才启用。

```
case NavigationMode:

    if (contacts.isEmpty()) {
        nameLine->clear();
        addressText->clear();
    }

    nameLine->setReadOnly(true);
    addressText->setReadOnly(true);
    addButton->setEnabled(true);

    int number = contacts.size();
    editButton->setEnabled(number >= 1);
    removeButton->setEnabled(number >= 1);
    nextButton->setEnabled(number > 1);
    previousButton->setEnabled(number > 1);

    submitButton->hide();
    cancelButton->hide();
    break;
}
```

通过在同一函数中设置模式和更新用户界面, 我们可以避免用户界面与应用程序内部状态不同步的可能性。

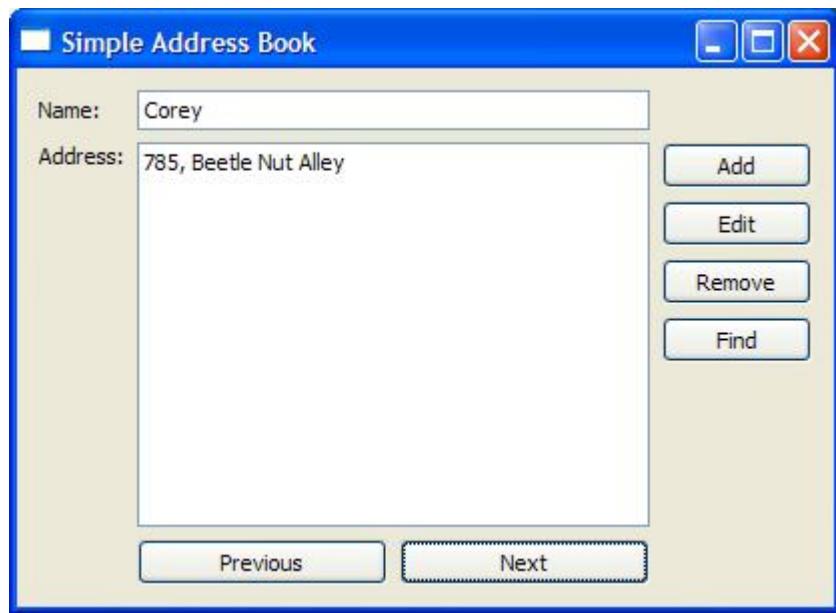
地址簿 5 — 添加查找功能

文件:

- [tutorials/addressbook/part5/addressbook.cpp](#)

- [tutorials/addressbook/part5/addressbook.h](#)
- [tutorials/addressbook/part5/finddialog.cpp](#)
- [tutorials/addressbook/part5/finddialog.h](#)
- [tutorials/addressbook/part5/main.cpp](#)
- [tutorials/addressbook/part5/part5.pro](#)

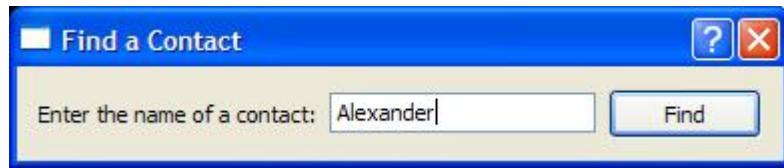
在本章中，我们将了解如何在地址簿应用程序中定位联系人和地址。



随着我们不断为地址簿应用程序添加联系人，使用下一个和上一个按钮浏览联系人就会变得很繁琐。在这种情况下，使用 [查找函数](#) 查找联系人就会更加有效。上面的屏幕截图显示了 [查找按钮](#) 及其在按钮面板上的位置。

当用户点击 [查找按钮](#) 时，有必要显示一个对话框，用户可在其中输入联系人的姓名。Qt 提供了 [QDialog](#)（我们会在本章中将其用作子类），可使用 [FindDialog](#) 类。

定义 [FindDialog](#) 类



要使用 [QDialog](#) 的子类，我们首先要在 [finddialog.h](#) 文件中声明 [QDialog](#) 的头信息。此外，我们还使用向前 (forward) 声明来声明 [QLineEdit](#) 和 [QPushButton](#)，这样我们就能在对话框类中使用这些 widget。

因为在 AddressBook 类中, FindDialog 类包含了 `Q_OBJECT` 宏, 并且其构造器已定义为接收父级 `QWidget`, 即使对话框以单独的窗口方式打开。

```
#include <QDialog>

class QLineEdit;
class QPushButton;

class FindDialog : public QDialog
{
    Q_OBJECT

public:
    FindDialog(QWidget *parent = 0);
    QString getFindText();

public slots:
    void findClicked();

private:
    QPushButton *findButton;
    QLineEdit *lineEdit;
    QString findText;
};
```

我们定义了公有函数 `getFindText()`, 供实例化 `FindDialog` 的类使用, 这样这些类可以获取用户输入的文本。公有槽 `findClicked()` 定义为在用户点击**查找**按钮时处理搜索字符串。

最后, 我们定义私有变量 `findButton`、`lineEdit` 和 `findText`, 分别对应**查找**按钮、用户输入搜索字符串的行编辑框和储存搜索字符串供稍后使用的内部字符串。

应用 `FindDialog` 类

在 `FindDialog` 的构造器中, 我们设置私有变量 `lineEdit`、`findButton` 和 `findText`。使用 `QHBoxLayout` 放置 widget。

```
FindDialog::FindDialog(QWidget *parent)
    : QDialog(parent)
{
    QLabel *findLabel = new QLabel(tr("Enter the name of a contact:"));
    lineEdit = new QLineEdit;
```

```

findButton = new QPushButton(tr("&Find"));
findText = "";

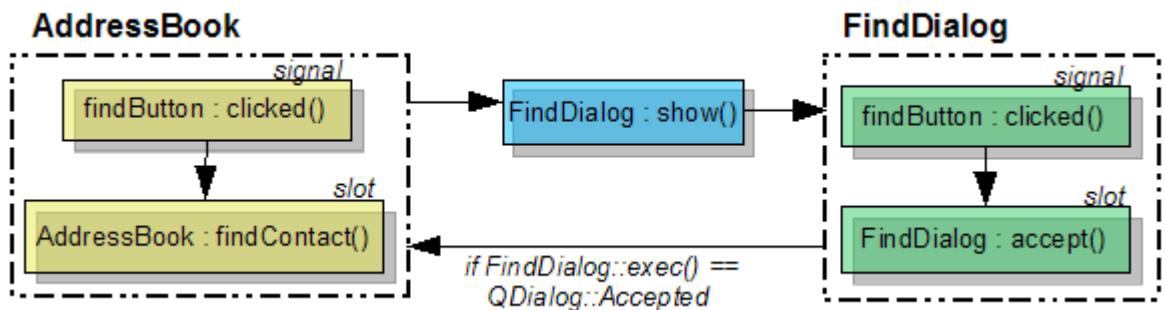
QHBoxLayout *layout = new QHBoxLayout;
layout->addWidget(findLabel);
layout->addWidget(lineEdit);
layout->addWidget(findButton);

setLayout(layout);
setWindowTitle(tr("Find a Contact"));
connect(findButton, SIGNAL(clicked()), this, SLOT(findClicked()));
connect(findButton, SIGNAL(clicked()), this, SLOT(accept()));

}

```

我们设定布局和窗口标题，并将信号与其各自的槽关联。请注意，`findButton` 的 `clicked()` 信号已与 `findClicked()` 和 `accept()` 关联。`QDialog` 提供的 `accept()` 槽会隐藏对话框并将结果代码设置为 `Accepted`。我们使用该函数有助于 `AddressBook` 的 `findContact()` 函数知晓 `FindDialog` 对象关闭的时间。我们在讨论 `findContact()` 函数时将对该函数做进一步说明。



在 `findClicked()` 中，我们验证 `lineEdit` 以确保用户没有在尚未输入联系人姓名时就点击**查找**按钮。然后，我们将 `findText` 设置为从 `lineEdit` 提取的搜索字符串。之后，我们清空 `lineEdit` 的内容并隐藏对话框。

```

void FindDialog::findClicked()
{
    QString text = lineEdit->text();

    if (text.isEmpty()) {
        QMessageBox::information(this, tr("Empty Field"),
                               tr("Please enter a name."));
        return;
    }
}

```

```
    } else {
        findText = text;
        lineEdit->clear();
        hide();
    }
}
```

findText 变量已有公有 getter 函数 `getFindText()` 与其相关联。既然我们仅在构造器和 `findClicked()` 函数中直接设定了 `findText`，我们就不再创建 `getFindText()` 的同时再创建 setter 函数。由于 `getFindText()` 是公有的，实例化和使用 `FindDialog` 的类可始终读取用户已输入并确认的搜索字符串。

```
QString FindDialog::getFindText()
{
    return findText;
}
```

定义 AddressBook 类

要确保我们可使用 `AddressBook` 类中的 `FindDialog`，我们要在 `addressbook.h` 文件中包含 `finddialog.h`。

```
#include "finddialog.h"
```

至此，所有地址簿功能都有了 `QPushButton` 和对应的槽。同样，`Find` 功能有 `findButton` 和 `findContact()`。

`findButton` 声明为私有变量，而 `findContact()` 函数声明为公有槽。

```
void findContact();
...
QPushButton *findButton;
```

最后，我们声明私有变量 `dialog`，用于引用 `FindDialog` 的实例。

```
FindDialog *dialog;
```

在实例化对话框后，我们可能会对其进行多次使用。使用私有变量可在类中不同位置对其进行多次引用。

应用 AddressBook 类

在 AddressBook 类的构造器中，实例化私有对象 findButton 和 findDialog：

```
findButton = new QPushButton(tr("&Find"));
findButton->setEnabled(false);
...
dialog = new FindDialog;
```

接下来，将 findButton 的 `clicked()` 信号与 `findContact()` 关联。

```
connect(findButton, SIGNAL(clicked()), this, SLOT(findContact()));
```

现在唯一要完成的就是 `findContact()` 函数的编码：

```
void AddressBook::findContact()
{
    dialog->show();

    if (dialog->exec() == QDialog::Accepted) {
        QString contactName = dialog->getFindText();

        if (contacts.contains(contactName)) {
            nameLine->setText(contactName);
            addressText->setText(contacts.value(contactName));
        } else {
            QMessageBox::information(this, tr("Contact Not Found"),
                                   tr("Sorry, \"%1\" is not in your address book.").arg(contactName));
            return;
        }
    }

    updateInterface(NavigationMode);
}
```

我们从显示 `FindDialog` 的实例 `dialog` 开始入手。这时用户开始输入联系人姓名进行查找。用户点击对话框的 `findButton` 后，对话框会隐藏，并且结果代码设置为 `QDialog::Accepted`。这样就确保了 `if` 语句始终为真。

然后，我们就开始使用 `FindDialog` 的 `getFindText()` 函数提取搜索字符串，这个字符串也就是本例中的 `contactName`。如果地址簿中有联系人，就立即显示该联系人。否则，显示如下所示的 `QMessageBox` 表明搜索失败。

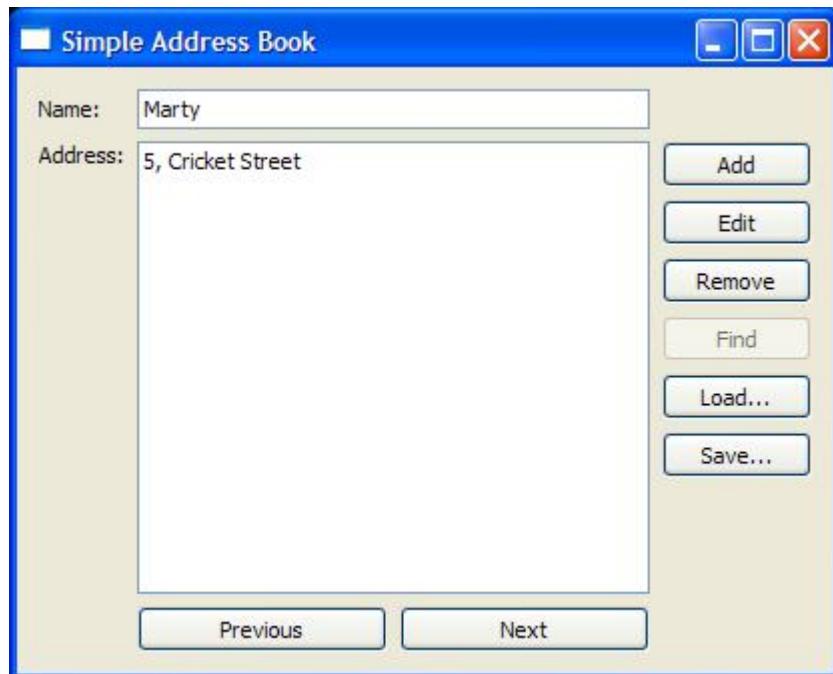


地址簿 6 — 加载和保存

文件:

- tutorials/addressbook/part6/addressbook.cpp
- tutorials/addressbook/part6/addressbook.h
- tutorials/addressbook/part6/finddialog.cpp
- tutorials/addressbook/part6/finddialog.h
- tutorials/addressbook/part6/main.cpp
- tutorials/addressbook/part6/part6.pro

本章描述了用于编写地址簿应用程序的加载和保存程序所使用的 Qt 文件处理功能。



虽然浏览和搜索联系人是非常实用的功能，但只有在可以保存现有联系人并可以在以后加载的前提下地址簿才真正完全可用。Qt 提供大量用于[输入和输出](#)的类，但我们只选择两个易于合并使用的类：[QFile](#) 和 [QDataStream](#)。

`QFile` 对象表示磁盘上可读取和写入的文件。`QFile` 是代表多种不同设备且应用更广的 `QIODevice` 类的子类。

`QDataStream` 对象用于按顺序排列二进制数据，以便储存在 `QIODevice` 中并供以后检索。读取或写入 `QIODevice` 就如同打开数据流，然后读取或写入一样简单，只是参数为不同的设备。

定义 AddressBook 类

我们声明两个公有槽 `saveToFile()` 和 `loadFromFile()`，以及两个 `QPushButton` 对象 `loadButton` 和 `saveButton`。

```
void saveToFile();
void loadFromFile();
...
QPushButton *loadButton;
QPushButton *saveButton;
```

应用 AddressBook 类

在构造器中，我们实例化 `loadButton` 和 `saveButton`。理想情况下，将按钮标签设置为“从文件加载联系人”和“将联系人保存至文件”会更方便用户使用。不过，由于其他按钮的大小限制，我们将标签设置为加载... 和 保存...。幸运的是，Qt 提供了使用 `setToolTip()` 来设置工具提示的简单方式，我们可通过如下方式将其用于按钮：

```
loadButton->setToolTip(tr("Load contacts from a file"));
...
saveButton->setToolTip(tr("Save contacts to a file"));
```

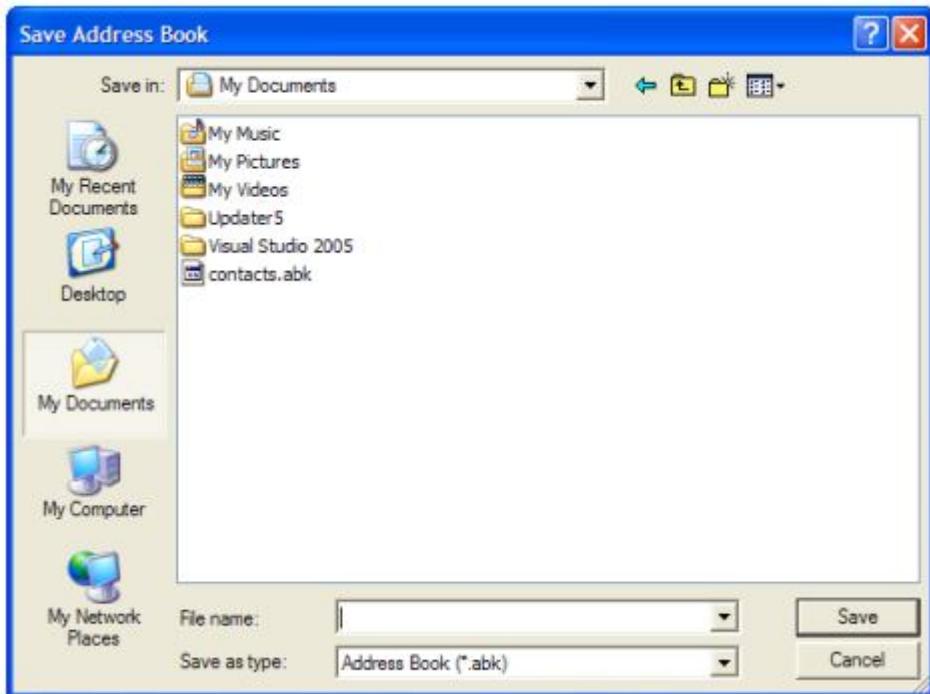
虽然此处没有显示，但与其他应用的功能一样，我们在右侧的布局面板 `button1Layout` 上添加按钮，然后将按钮的 `clicked()` 信号与其相应的槽关联。

至于保存功能，我们首先使用 `QFileDialog::getSaveFileName()` 获取 `fileName`。这是 `QFileDialog` 提供的一个便捷功能，可弹出样式文件对话框并允许用户输入文件名或选择现有的 `.abk` 文件。`.abk` 文件是保存联系人时创建的地址簿扩展名。

```
void AddressBook::saveToFile()
{
    QString fileName = QFileDialog::getSaveFileName(this,
        tr("Save Address Book"), "",
```

```
tr("Address Book (*.abk);;All Files (*)");
```

弹出的文件对话框屏幕截图显示如下：



如果 fileName 不为空，我们就使用 fileName 创建 QFile 对象 file。QFile 与 QDataStream 一同使用，这是因为 QFile 是 QIODevice。

接下来，我们尝试以 WriteOnly 模式打开文件。如果未能打开，会显示 QMessageBox 提示用户。

```
if (fileName.isEmpty())
    return;
else {
    QFile file(fileName);
    if (!file.open(QIODevice::WriteOnly)) {
        QMessageBox::information(this, tr("Unable to open file"),
                               file.errorString());
        return;
    }
}
```

否则，会用实例表示 QDataStream 对象 out，以写入打开的文件。QDataStream 要求读写操作需使用相同版本的数据流。在将数据按顺序写入 file 之前，将使用的版本设置为采用 Qt 4.5 的版本就可确保版本相同。

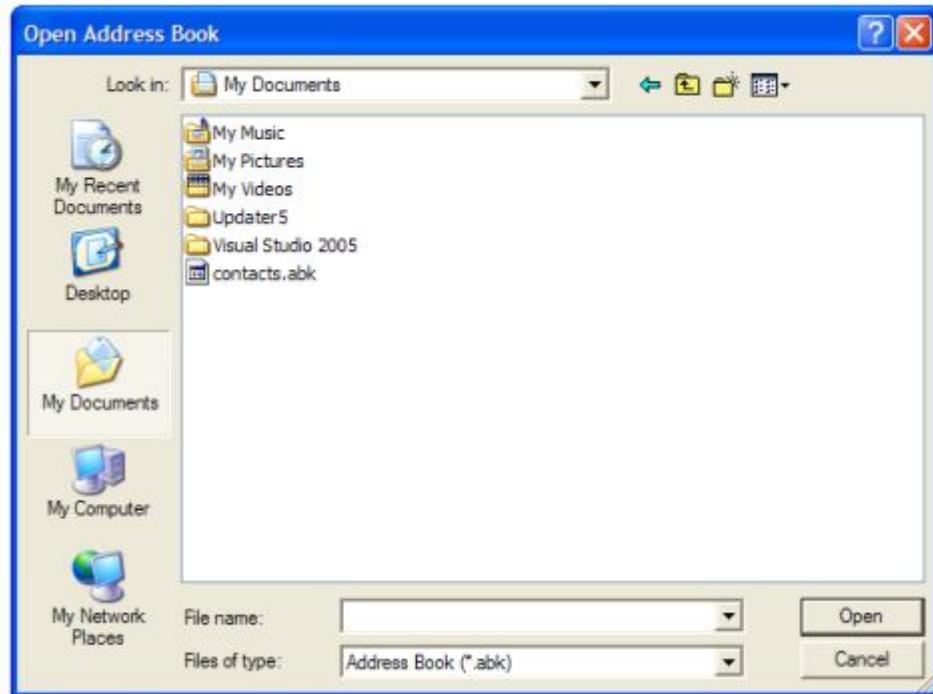
```
QDataStream out(&file);
```

```
    out.setVersion(QDataStream::Qt_4_5);
    out << contacts;
}
}
```

至于加载功能，我们也是使用 `QFileDialog::getOpenFileName()` 获取 `fileName`。该函数与 `QFileDialog::getSaveFileName()` 相对应，也是弹出样式文件对话框并允许用户输入文件名或选择现有的 `.abk` 文件加载到地址簿中。

```
void AddressBook::loadFromFile()
{
    QString fileName = QFileDialog::getOpenFileName(this,
        tr("Open Address Book"), "",
        tr("Address Book (*.abk);;All Files (*)"));
```

例如，在 Windows 上，该函数弹出本地文件对话框，如以下屏幕截图所示。



如果 `fileName` 不为空，还是使用 `QFile` 对象 `file`，然后尝试在 `ReadOnly` 模式下打开文件。与 `saveToFile()` 的应用方式类似，如果尝试失败，会显示 `QMessageBox` 提示用户。

```
if (fileName.isEmpty())
    return;
else {
```

```

QFile file(fileName);

if (!file.open(QIODevice::ReadOnly)) {
    QMessageBox::information(this, tr("Unable to open file"),
                           file.errorString());
    return;
}

QDataStream in(&file);
in.setVersion(QDataStream::Qt_4_5);
contacts.empty(); // empty existing contacts
in >> contacts;

```

否则，会用实例表示 [QDataStream](#) 对象 `in`，按上文所述设置其版本，然后将按顺序排列的数据读入 `contacts` 数据结构。请注意，在将数据读入之前清空 `contacts` 可简化文件读取过程。更高级的方法是将联系人读取至临时 [QMap](#) 对象，然后仅复制 `contacts` 中不存在的联系人。

```

if (contacts.isEmpty()) {
    QMessageBox::information(this, tr("No contacts in file"),
                           tr("The file you are attempting to open contains no contacts."));
} else {
    QMap<QString, QString>::iterator i = contacts.begin();
    nameLine->setText(i.key());
    addressText->setText(i.value());
}
}

updateInterface(NavigationMode);
}

```

要显示从文件中读取的联系人，必须要先验证获取的数据，以确保读取的文件实际包含地址簿联系人。如果为真，显示第一个联系人，否则显示 [QMessageBox](#) 提示出现问题。最后，我们更新界面以相应地启用和禁用按钮。

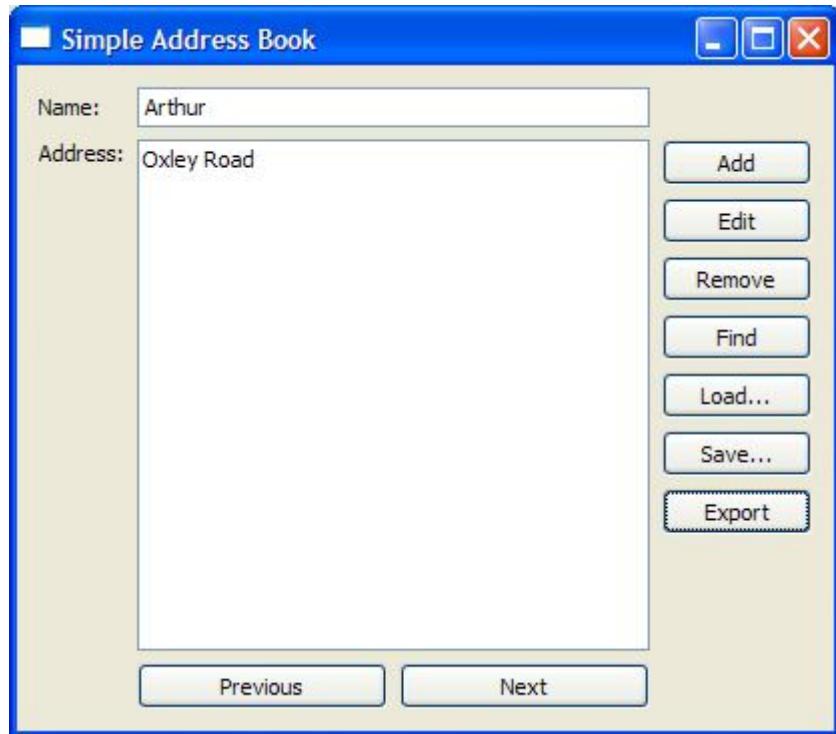
地址簿 7 — 附加功能

文件：

- [tutorials/addressbook/part7/addressbook.cpp](#)
- [tutorials/addressbook/part7/addressbook.h](#)
- [tutorials/addressbook/part7/finddialog.cpp](#)
- [tutorials/addressbook/part7/finddialog.h](#)

- [tutorials/addressbook/part7/main.cpp](#)
- [tutorials/addressbook/part7/part7.pro](#)

本章讲述了部分可使地址簿应用程序日常使用更加便捷的附加功能。



虽然地址簿应用程序其自身功能已经很实用，但是如果可和其他应用程序互换联系人数据就会更加有益。vCard 格式是一种流行的文件格式，就可用于此目的。在本章中，我们会扩展地址簿客户端，可将联系人导出到 vCard .vcf 文件中。

定义 AddressBook 类

我们在 addressbook.h 文件的 AddressBook 类中添加 [QPushButton](#) 对象 `exportButton` 以及对应的公有槽 `exportAsVCard()`。

```
void exportAsVCard();  
...  
QPushButton *exportButton;
```

应用 AddressBook 类

在 AddressBook 构造器中，我们将 `exportButton` 的 `clicked()` 信号连接至 `exportAsVCard()`。我们还会将该按钮添加至 `buttonLayout1`，它是负责右侧按钮面板的布局类。

在 `exportAsVCard()` 函数中，我们从提取 `name` 中联系人姓名开始入手。我们声明 `firstName`、`lastName` 和 `nameList`。接下来，我们查找 `name` 中第一处空白的索引。如果有空白，就将联系人的姓名分隔为 `firstName` 和 `lastName`。然后，将空白替换为下划线（“_”）。或者，如果没有空白，就认定联系人只有名字。

```
void AddressBook::exportAsVCard()
{
    QString name = nameLine->text();
    QString address = addressText->toPlainText();
    QString firstName;
    QString lastName;
    QStringList nameList;

    int index = name.indexOf(" ");

    if (index != -1) {
        nameList = name.split(QRegExp("\\s+"), QString::SkipEmptyParts);
        firstName = nameList.first();
        lastName = nameList.last();
    } else {
        firstName = name;
        lastName = "";
    }

    QString fileName = QFileDialog::getSaveFileName(this,
        tr("Export Contact"), "",
        tr("vCard Files (*.vcf);;All Files (*)"));

    if (fileName.isEmpty())
        return;

    QFile file(fileName);
```

至于 `saveToFile()` 函数，会打开文件对话框，让用户选择文件的位置。通过选择的文件名称，我们创建要写入的 `QFile` 实例。

我们尝试以 `WriteOnly` 模式打开文件。如果操作失败，会显示 `QMessageBox` 提示用户出现问题并返回。否则，将文件作为参数传递给 `QTextStream` 对象 `out`。与 `QDataStream` 类似，`QTextStream` 类提供了读取纯文本和将其写入到文件的功能。因此，所生成的 `.vcf` 文件可以在文本编辑器中打开进行编辑。

```
if (!file.open(QIODevice::WriteOnly)) {
    QMessageBox::information(this, tr("Unable to open file"),
        file.errorString());
```

```
    return;
}

QTextStream out(&file);
```

然后，我们写出依次带有 BEGIN:VCARD 和 VERSION:2.1 标记的 vCard 文件。联系人的姓名使用 N: 标记写入。至于写入 vCard “File as” 属性的 FN: 标记，我们必须要查看是否联系人带有姓。如果联系人有姓，就使用 nameList 中的详细信息填入该标记。否则，仅写入 firstName。

```
out << "BEGIN:VCARD" << "\n";
out << "VERSION:2.1" << "\n";
out << "N:" << lastName << ";" << firstName << "\n";

if (!nameList.isEmpty())
    out << "FN:" << nameList.join(" ") << "\n";
else
    out << "FN:" << firstName << "\n";
```

我们继续写入联系人的地址。地址中的分号使用 “\” 进行转义，新行使用分号进行替换，而逗号使用空白进行替换。最后，我们依次写入 ADR;HOME:;、address 和 END:VCARD 标记。

```
address.replace(";", "\\;", Qt::CaseInsensitive);
address.replace("\n", ";", Qt::CaseInsensitive);
address.replace(" ", " ", Qt::CaseInsensitive);

out << "ADR;HOME%;" << address << "\n";
out << "END:VCARD" << "\n";

QMessageBox::information(this, tr("Export Successful"),
    tr("\'%1\' has been exported as a vCard.").arg(name));
}
```

最后，会显示 QMessageBox 提示用户已成功导出 vCard。

vCard 是 *Internet Mail Consortium* 的商标。

Widgets 教程

Widget 是使用 Qt 编写的图形用户界面 (GUI) 应用程序的基本生成块。每个 GUI 组件，如按钮、标签或文本编辑器，都是一个 `widget`，并可以放置在现有的用户界面中或作为单独的窗口显示。每种类型的组件都是由 `QWidget` 的特殊子类提供的，而 `QWidget` 自身又是 `QObject` 的子类。

Widgets 教程

简介

`Widget` 是使用 Qt 编写的图形用户界面 (GUI) 应用程序的基本生成块。每个 GUI 组件，如按钮、标签或文本编辑器，都是一个 `widget`，并可以放置在现有的用户界面中或作为单独的窗口显示。每种类型的组件都是由 `QWidget` 的特殊子类提供的，而 `QWidget` 自身又是 `QObject` 的子类。

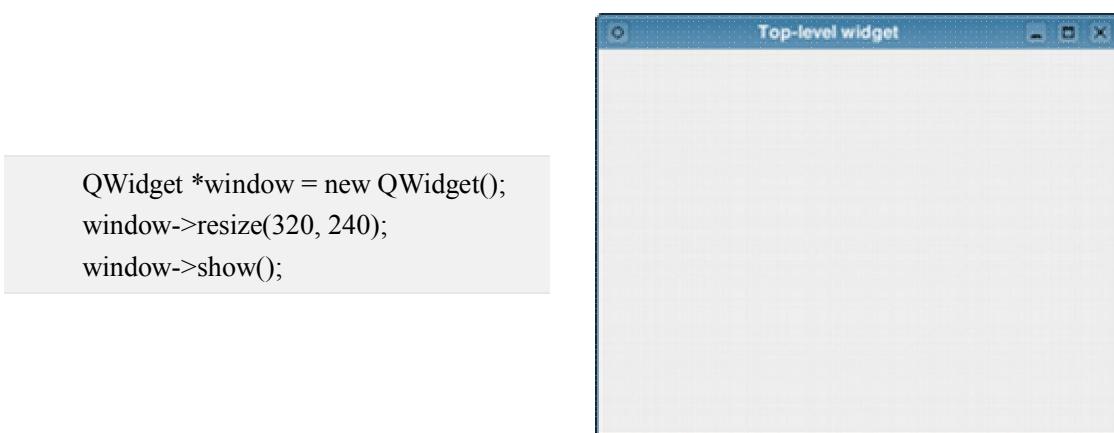
`QWidget` 不是一个抽象类；它可用作其他 `widget` 的容器，并很容易作为子类使用来创建定制 `widget`。它经常用来创建放置其他 `widget` 的窗口。

至于 `QObject`，可使用父对象创建 `widget` 以表明其所属关系，这可确保删除不再使用的对象。使用 `widget`，这些父子关系就有了更多的意义：每个子类都显示在其父级所拥有的屏幕区域内。也就是说，当删除窗口时，其包含的所有 `widget` 也都自动删除。

创建窗口

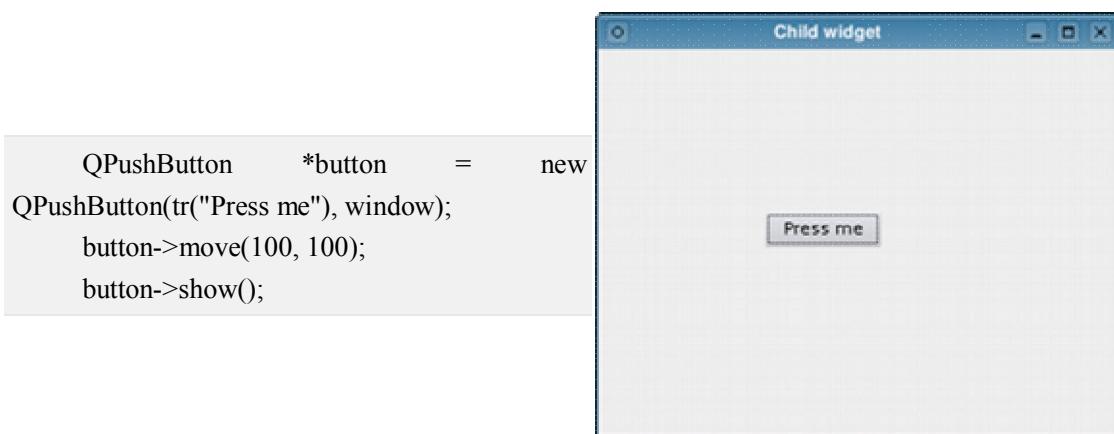
如果 `widget` 未使用父级进行创建，则在显示时视为窗口或顶层 `widget`。由于顶层 `widget` 没有父级对象类来确保在其不再使用时就删除，因此需要开发人员在应用程序中对其进行跟踪。

在下例中，我们使用 `QWidget` 创建和显示具有默认大小的窗口：



```
QWidget *window = new QWidget();
window->resize(320, 240);
window->show();
```

我们可以通过将 `window` 作为父级传递给其构造器来向窗口添加子 `widget`。在这种情况下，我们向窗口添加按钮并将其放置在特定位置：



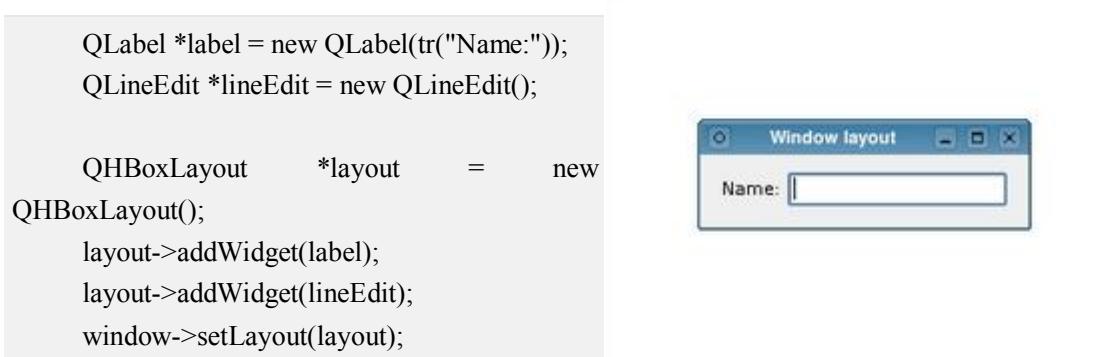
```
QPushButton *button = new QPushButton(tr("Press me"), window);
button->move(100, 100);
button->show();
```



该按钮现在为窗口的子项，并在删除窗口时一同删除。请注意，隐藏或关闭窗口不会自动删除该按钮。

使用布局

通常，子 `widget` 是通过使用布局对象在窗口中进行排列，而不是通过指定位置和大小进行排列。在此处，我们构造要并排排列的标签和行编辑框 `widget`。



```
QLabel *label = new QLabel(tr("Name:"));
QLineEdit *lineEdit = new QLineEdit();

QHBoxLayout *layout = new QHBoxLayout();
layout->addWidget(label);
layout->addWidget(lineEdit);
window->setLayout(layout);
```

我们构造的布局对象管理通过 `addWidget()` 函数提供的 `widget` 的位置和大小。布局本身是通过调用 `setLayout()` 提供给窗口的。布局仅可通过其对所管理的 `widget` (和其他布局) 的效果才可显示。

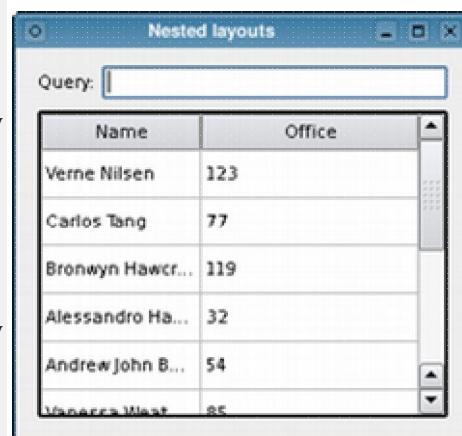
在上文示例中，每个 `widget` 的所属关系并不明显。由于我们未使用父级对象构造 `widget` 和布局，我们会看到一个空窗口和两个包含了标签与行编辑框的窗口。不过，如果我们告知布局来管理标签和行编辑框，并在窗口中设置布局，两个 `widget` 与布局本身就会成为窗口的子项。

由于 `widget` 可包含其他 `widget`，布局可用来提供按不同层次分组的 `widget`。这里，我们要在显示查询结果的表视图上方、窗口顶部的行编辑框旁，显示一个标签。

```
QLabel *queryLabel = new QLabel(tr("Query:"));
QLineEdit *queryEdit = new QLineEdit();
QTableView *resultView = new QTableView();

QHBoxLayout *queryLayout = new QHBoxLayout();
queryLayout->addWidget(queryLabel);
queryLayout->addWidget(queryEdit);

QVBoxLayout *mainLayout = new QVBoxLayout();
mainLayout->addLayout(queryLayout);
mainLayout->addWidget(resultView);
window->setLayout(mainLayout);
```



除了 `QHBoxLayout` 和 `QVBoxLayout`，Qt 还提供了 `QGridLayout` 和 `QFormLayout` 类来协助实现更复杂的用户界面。

Part II Qt 学习之路:

来自于 **FinderCheng** 的 Qt 学习之路。简介：在本系列文章中，**FinderCheng** 使用 Qt4 进行 C++ GUI 的开发。我是参照着《C++ GUI Programming with Qt4》一书进行学习的。其实，我也只是初学 Qt4，在这里将这个学习笔记记下来，希望能够方便更多的朋友学习 Qt4。我是一个 Java 程序员，感觉 Qt4 的一些命名规范以及约束同 Java 有异曲同工之妙，因而从 Java 迁移到 Qt4 似乎困难不大。不过，这也主要是因为 Qt4 良好的设计等等。

Qt 学习之路(1): 前言

Qt 是一个著名的 C++ 库——或许并不能说这只是一个 GUI 库，因为 Qt 十分庞大，并不仅仅是 GUI。使用 Qt，在一定程度上你获得的是一个“一站式”的服务：不再需要研究 STL，不再需要 C++ 的 `<string>`，因为 Qt 有它自己的 `QString` 等等。或许这样说很偏激，但 Qt 确实是一个“伟大的 C++ 库”。

我们所使用的 Qt，确切地说也就是它的 GUI 编程部分。C++ 的 GUI 编程同 Java 不同：GUI 并不是 C++ 标准的一部分。所以，如果使用 Java，那么你最好的选择就是 AWT/Swing，或者也可以使 SWT/JFace，但是，C++ 的 GUI 编程给了你更多的选择：wxWidget，gtk++ 以及 Qt。这几个库我都有接触，但是接触都不是很多，只能靠一些资料和自己的一点粗浅的认识说一下它们之间的区别(PS：更详尽的比较在[前面的文章](#)中有)。

首先说 wxWidget，这是一个标准的 C++ 库，和 Qt 一样庞大。它的语法看上去和 MFC 类似，有大量的宏。据说，一个 MFC 程序员可以很容易的转换到 wxWidget 上面来。wxWidget 有一个很大的优点，就是它的界面都是原生风格的。这是其他的库所不能做到的。wxWidget 的运行效率很高，据说在 Windows 平台上比起微软自家的 MFC 也不相上下。

gtk++ 其实是一个 C 库，不过由于 C++ 和 C 之间的关系，这点并没有很大的关系。但是，gtk++ 是一个使用 C 语言很优雅的实现了面向对象程序设计的范例。不过，这也同样带来了一个问题——它的里面带有大量的类型转换的宏来模拟多态，并且它的函数名“又臭又长(不过这点我倒是觉得无所谓，因为它的函数名虽然很长，但是同样很清晰)”，使用下划线分割单词，看上去和 Linux 如出一辙。由于它是 C 语言实现，因此它的运行效率当然不在话下。gtk++ 并不是模拟的原生界面，而有它自己的风格，所以有时候就会和操作系统的界面显得格格不入。

再来看 Qt，和 wxWidget 一样，它也是一个标准的 C++ 库。但是它的语法很类似于 Java 的 Swing，十分清晰，而且 SIGNAL/SLOT 机制使得程序看起来很明白——这也是我首先选择 Qt 的一个很重要的方面，因为我是学 Java 出身的 😊。不过，所谓“成也萧何，败也萧何”，这种机制虽然很清楚，但是它所带来的后果是你需要使用 Qt 的 qmake 对程序进行预处理，才能够再使用 make 或者 nmake 进行编译。并且它的界面也不是原生风格的，尽管 Qt 使用 style 机制十分巧妙的模拟了本地界面。另外值得一提的是，Qt 不仅仅运行在桌面环境中，Qt 已经被 Nokia 收购，它现在已经成为 Symbian 系列的主要界面技术——Qt 是能够运行于嵌入式平台的。

以往人们对 Qt 的授权多有诟病。因为 Qt 的商业版本价格不菲，开源版本使用的是 GPL 协议。但是现在 Qt 的开源协议已经变成 LGPL。这意味着，你可以将 Qt 作为一个库连接到一个闭源软件里面。可以说，现在的 Qt 协议的争议已经不存在了——因为 wxWidgets 或者 gtk+ 同样使用的是类似的协议发布的。

在本系列文章中，我们将使用 Qt4 进行 C++ GUI 的开发。我是参照着《C++ GUI Programming with Qt4》一书进行学习的。其实，我也只是初学 Qt4，在这里将这个学习笔记记下来，希望能够方便更多的朋友学习 Qt4。我是一个 Java 程序

员，感觉 Qt4 的一些命名规范以及约束同 Java 有异曲同工之妙，因而从 Java 迁移到 Qt4 似乎困难不大。不过，这也主要是因为 Qt4 良好的设计等等。

闲话少说，还是尽快开始下面的学习吧！

Qt 学习之路(2): Hello, world!

任何编程技术的学习第一课基本上都会是 Hello, world!，我也不想故意打破这个惯例——照理说，应该首先回顾一下 Qt 的历史，不过即使不说这些也并无大碍。

或许有人总想知道，Qt 这个单词是什么意思。其实，这并不是一个缩写词，仅仅是因为它的发明者，TrollTech 公司的 CEO，Haarard Nord 和 Trolltech 公司的总裁 Eirik Chambe-Eng 在联合发明 Qt 的时候并没有一个很好的名字。在这里，字母 Q 是 Qt 库中所有类的前缀——这仅仅是因为在 Haarard 的 emacs 的 字体中，这个字母看起来特别的漂亮；而字母 t 则代表“toolkit”，这是在 Xt(X toolkit) 中得到的灵感。

顺便说句，Qt 原始的公司就是上面提到的 Trolltech，貌似有一个中文名字是奇趣科技——不过现在已经被 Nokia 收购了。因此，一些比较旧的文章里面会提到 Trolltech 这个名字。

好了，闲话少说，先看看 Qt 的开发吧！事先说明一下，我是一个比较懒的人，不喜欢配置很多的东西，而 Qt 已经提供了一个轻量级的 IDE，并且它的网站上也有 for Eclipse 和 VS 的开发插件，不过在这里我并不想用这些大块头 😊

Qt 有两套协议——商业版本和开源的 LGPL 版本。不同的是前者要收费，而后者免费，当然，后者还要遵循 LGPL 协议的规定，这是题外话。

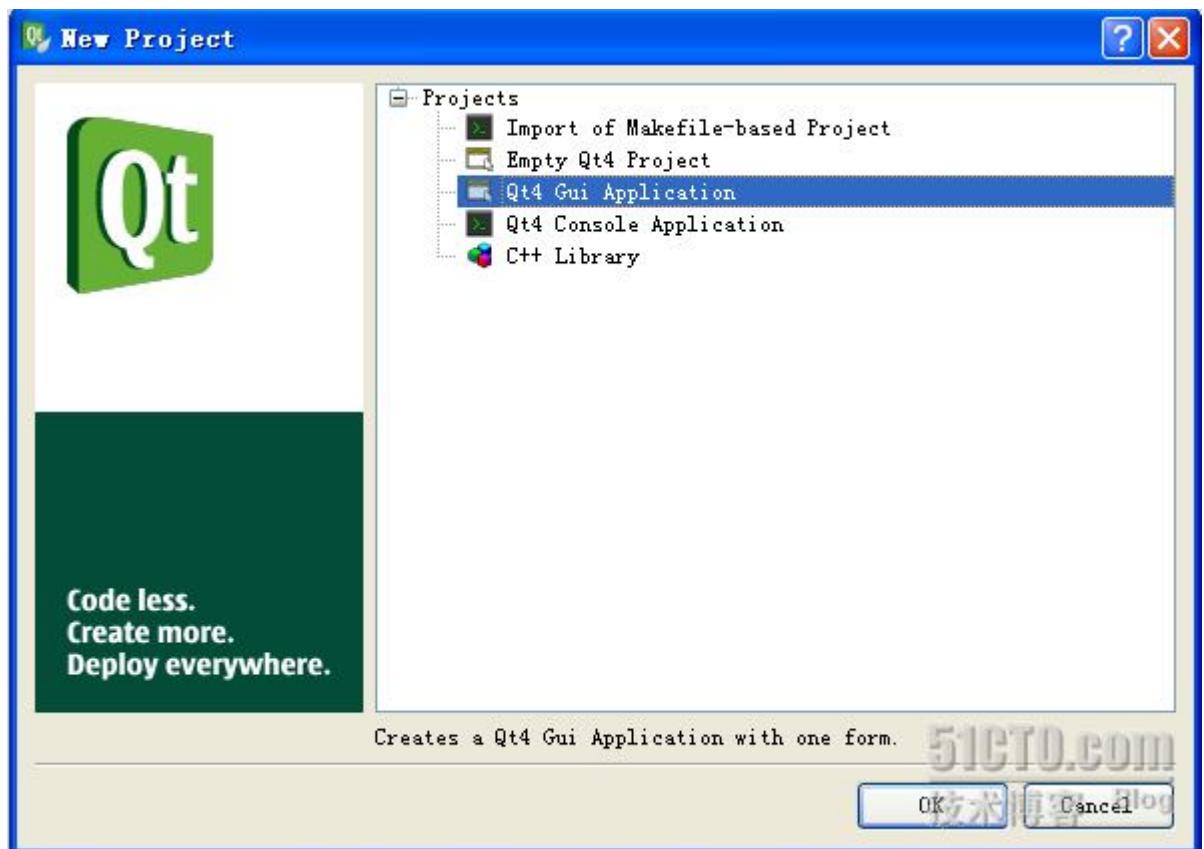
Qt 的网址是 <https://qt.nokia.com/downloads>，不过我打开这个站点总是很慢，不知道为什么。你可以找到大大的 LGPL/Free 和 Commercial，好了，我选的是 LGPL 版本的，下载包蛮大，但是下载并不会很慢。下载完成后安装就可以了，其它不用管了。这样，整个 Qt 的开发环境就装好了——如果你需要的话，也可以把 qmake 所在的目录添加进环境变量，不过我就不做了。

安装完成后会有个 Qt Creator 的东西，这就是官方提供的一个轻量级 IDE，不过它的功能还是蛮强大的。运行这个就会发现，其实 Qt 不仅仅是 Linux KDE 桌面的底层实现库。而且是这个 IDE 的实现 😊 这个 IDE 就是用 Qt 完成的。

Qt Creator 左面从上到下依次是 Welcome(欢迎页面，就是一开始出现的那个)；Edit(我们的代码编辑窗口)；Debug(调试窗口)；Projects(工程窗口)；Help(帮助，这个帮助完全整合的 Qt 的官方文档，相当有用)；Output(输出窗口)。

下面我们来试试我们的 Hello, world! 吧！

在 Edit 窗口空白处点右键，有 New project... 这里我们选第三项，Qt Gui Application。



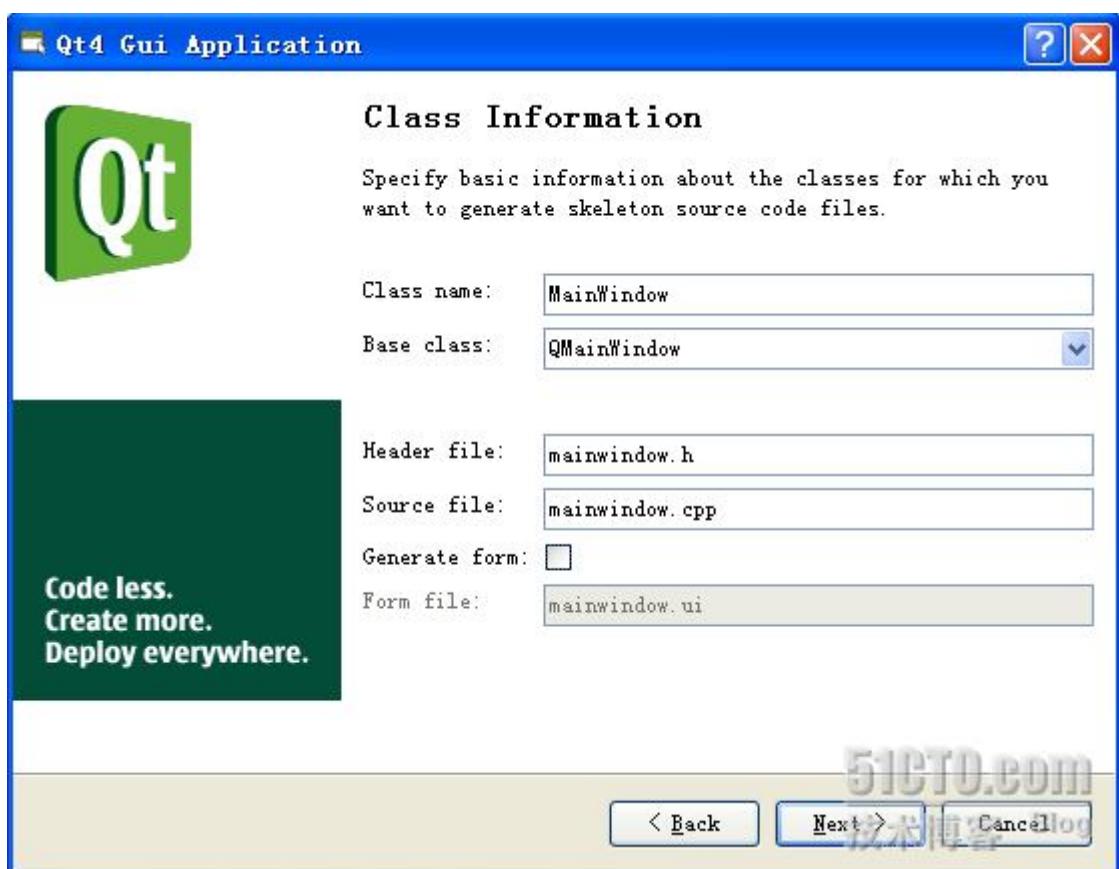
然后点击 OK，来到下一步，输入工程名字和保存的位置



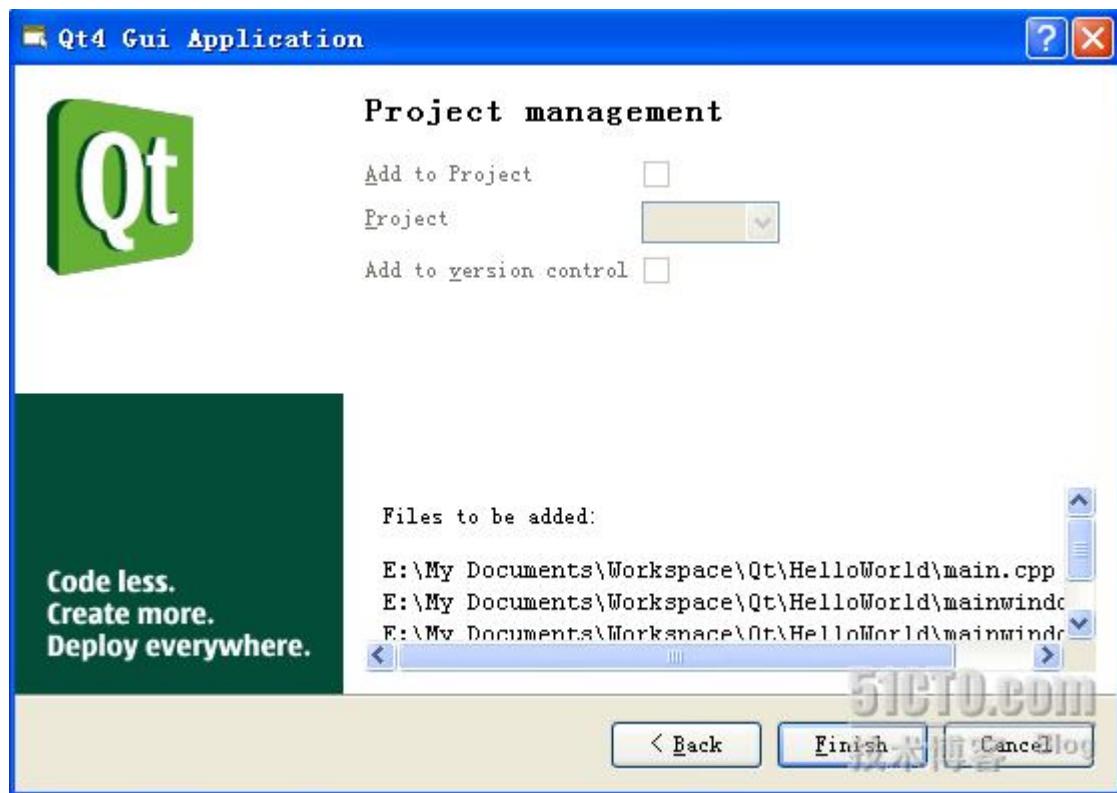
点击 Next, 来到选择库的界面。这里我们系统默认为我们选择了 Qt core 和 GUI, 还记得我们建的是 Gui Application 吗? 嗯, 就是这里啦, 它会自动为我们加上 gui 这个库。现在应该就能看出, Qt 是多么庞大的一个库, 它不仅仅有 Gui, 而且有 Network, OpenGL, XML 之类。不过, 现在在这里我们不作修改, 直接 Next。



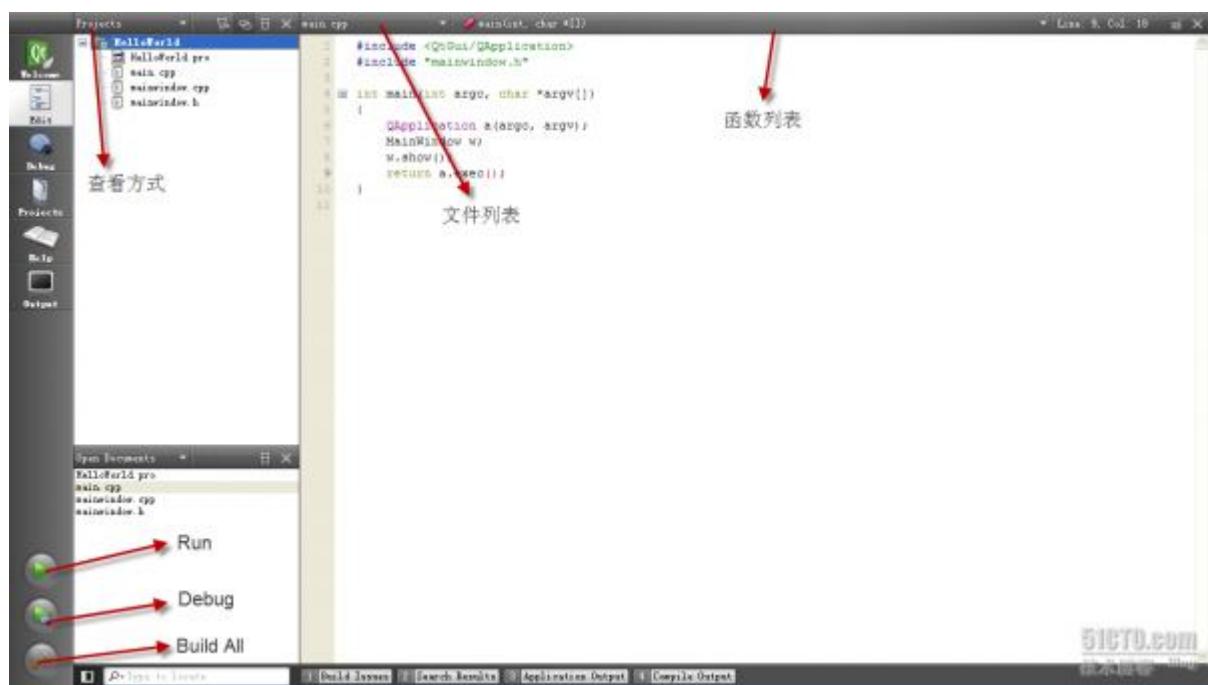
下一个界面需要我们定义文件名，我们不修改默认的名字，只是为了清除起见，把 generate form 的那个勾去掉即可。



Next 之后终于到了 Finish 了——漫长的一系列啊！检查无误后 Finish 就好啦！



之后可以看到，IDE 自动生成了四个文件，一个.pro 文件，两个.cpp 和一个.h。这里说明一下，.pro 就是工程文件(project)，它是 qmake 自动生成的用于生产 makefile 的配置文件。这里我们先不去管它。main.cpp 里面就是一个 main 函数，其他两个文件就是先前我们曾经指定的文件名的文件。



现在，我们把 main.cpp 中的代码修改一下：

```
#include "QtGui/QApplication"
#include "QLabel"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QLabel *label = new QLabel("Hello, world!");
    label->show();
    return a.exec();
}
```

好了！我们的第一个 Qt 程序已经完成了。

PS：截了很多图，说得详细些，以后可就没这么详细的步骤啦，嘿嘿…相信很多朋友应该一下子就能看明白这个 IDE 应该怎么使用了的，无需我多费口舌。呵呵。

下一篇中，将会对这个 Hello, world! 做一番逐行解释！

Qt 学习之路(3): Hello, world!(续)

下面来逐行解释一下前面的那个 Hello, world! 程序，尽管很简单，但却可以对 Qt 程序的结构有一个清楚的认识。现在再把代码贴过来：

```
#include "QApplication"
#include "QLabel"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel *label = new QLabel("Hello, world!");
    label->show();
    return app.exec();
}
```

第 1 行和第 2 行就是需要引入的头文件。和普通的 C++ 程序没有什么两样，如果要使用某个组件，就必须要引入相应的头文件，这类似于 Java 的 import 机制。值得说明的是，Qt 中头文件和类名是一致的。也就是说，如果你要使用某个类的话，它的类名就是它的头文件名。

第 3 行是空行 😊

第 4 行是 main 函数函数头。这与普通的 C++ 程序没有什么两样，学过 C++ 的都明白。因此你可以看到，实际上，Qt 完全通过普通的 main 函数进入，这不同于 wxWidgets，因为 wxWidgets 的 Hello, world 需要你继承它的一个 wxApp 类，并覆盖它的 wxApp::OnInit 方法，系统会自动将 OnInit 编译成入口函数。不过在 Qt 中，就不需要这些了。

第 5 行，噢噢，大括号…

第 6 行，创建一个 QApplication 对象。这个对象用于管理应用程序级别的资源。QApplication 的构造函数要求两个参数，分别来自 main 的那两个参数，因此，Qt 在一定程度上是支持命令行参数的。

第 7 行，创建一个 QLabel 对象，并且能够显示 Hello, world! 字符串。和其他库的 Label 控件一样，这是用来显示文本的。在 Qt 中，这被称为一个 widget（翻译出来是小东西，不过这个翻译并不好…），它等同于 Windows 技术里面的控件（controls）和容器（containers）。也就是说，widget 可以放置其他的 widget，就像 Swing 的组件。大多数 Qt 程序使用 QMainWindow 或者 QDialog 作为顶级组件，但 Qt 并不强制要求这点。在这个例子中，顶级组件就是一个 QLabel。

第 8 行，使这个 label 可见。组件创建出来之后通常是不可见的，要求我们手动的使它们可见。这样，在创建出组建之后我们就可以对它们进行各种定制，以避免出现之后在屏幕上面会有闪烁。

第 9 行，将应用程序的控制权移交给 Qt。这时，程序的事件循环就开始了，也就是说，这时可以相应你发出的各种事件了。这类似于 gtk+ 最后的一行 gtk_main()。

第 10 行，大括号……程序结束了。

注意，我们并没有使用 delete 去删除创建的 QLabel，因为在程序结束后操作系统会回收这个空间——这只是因为这个 QLabel 占用的内存比较小，但有时候这么做会引起麻烦的，特别是在大程序中，因此必须小心。

好了，程序解释完了。按照正常的流程，下面应该编译。前面也提过，Qt 的编译不能使用普通的 make，而必须先使用 qmake 进行预编译。所以，第一步应该是在工程目录下使用

```
qmake -project
```

命令创建 .pro 文件（比如说是叫 helloworld.pro）。然后再在 .pro 文件目录下使用

```
qmake helloworld.pro (make)
```

或者

```
qmake -tp vc helloworld.pro (nmake)
```

生成 makefile，然后才能调用 make 或者是 nmake 进行编译。不过因为我们使用的是 IDE，所以这些步骤就不需要我们手动完成了。

值得说明一点的是，这个 qmake 能够生成标准的 makefile 文件，因此完全可以利用 qmake 自动生成 makefile——这是题外话。

好了，下面修改一下源代码，把 QLabel 的创建一句改成

```
QLabel *label = new QLabel("Hello, world!");
```

运行一下：



同 Swing 的 JLabel 一样，Qt 也是支持 HTML 解析的。

好了，这个 Hello, world 就说到这里！明确一下 Qt 的程序结构，在一个 Qt 源代码中，一下两条语句是必不可少的：

```
QApplication app(argc, argv);
//...
return app.exec();
```

Qt 学习之路(4): 初探信号槽

看过了简单的 Hello, world! 之后，下面来看看 Qt 最引以为豪的信号槽机制！

所谓信号槽，简单来说，就像是插销一样：一个插头和一个插座。怎么说呢？当某种事件发生之后，比如，点击了一下鼠标，或者按了某个按键，这时，这个组件就会发出一个信号。就像是广播一样，如果有了事件，它就漫天发声。这时，如果有一个槽，正好对应上这个信号，那么，这个槽的函数就会执行，也就是回调。就像广播发出了，如果你感兴趣，那么你就会对这个广播有反应。干巴巴的解释很无力，还是看代码：

```
#include "QtGui/QApplication"
#include "QtGui/QPushButton"
```

```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QPushButton *button = new QPushButton("Quit");
    QObject::connect(button, SIGNAL(clicked()), &a,
SLOT(quit()));
    button->show();
    return a.exec();
}

```

这是在 Qt Creator 上面新建的文件，因为前面已经详细的说明怎么新建工程，所以这里就不再赘述了。这个程序很简单，只有一个按钮，点击之后程序退出。(顺便说一句，Qt 里面的 button 被叫做 QPushButton，真搞不明白为什么一个简单的 button 非得加上 push 呢？呵呵)

主要是看这一句：

```

QObject::connect(button, SIGNAL(clicked()), &a,
SLOT(quit()));

```

QObject 是所有类的根。Qt 使用这个 QObject 实现了一个单根继承的 C++。它里面有一个 connect 静态函数，用于连接信号槽。

当一个按钮被点击时，它会发出一个 clicked 信号，意思是，向周围的组件们声明：我被点击啦！当然，其它很多组件都懒得理他。如果对它感兴趣，就告诉 QObject 说，你帮我盯着点，只要 button 发出 clicked 信号，你就告诉我——想了想之后，说，算了，你也别告诉我了，直接去执行我的某某某函数吧！就这样，一个信号槽就形成了。具体来说呢，这个例子就是 QApplication 的实例 a 说，如果 button 发出了 clicked 信号，你就去执行我的 quit 函数。所以，当我们点击 button 的时候，a 的 quit 函数被调用，程序退出了。所以，在这里，clicked() 就是一个信号，而 quit() 就是槽，形象地说就是把这个信号插进这个槽里面去。

Qt 使用信号槽机制完成了事件监听操作。这类似与 Swing 里面的 listener 机制，只是要比这个 listener 简单得多。以后我们会看到，这种信号槽的定义也异常的简单。值得注意的是，这个信号槽机制仅仅是使用的 QObject 的 connect 函数，其他并没有什么耦合——也就是说，完全可以利用这种机制实现你自己的信号监听！不过，这就需要使用 qmake 预处理一下了！

细心的你或许发现，在 Qt Creator 里面，SIGNAL 和 SLOT 竟然变颜色了！没错，Qt 确实把它们当成了关键字！实际上，Qt 正是利用它们扩展了 C++ 语言，因此才需要使用 qmake 进行预处理，比便使普通的 C++ 编译器能够顺利编译。另外，这里的 signal 和 Unix 系统里面的 signal 没有任何的关系！哦哦，有一点关系，那就是名字是一样的！

信号槽机制是 Qt 关键部分之一，以后我们还会再仔细的探讨这个问题的。

Qt 学习之路(5): 组件布局

同 Swing 类似，Qt 也提供了几种组件定位的技术。其中就包括绝对定位和布局定位。

顾名思义，绝对定位就是使用最原始的定位方法，给出这个组件的坐标和长宽值。这样，Qt 就知道该把组件放在哪里，以及怎么设置组件的大小了。但是这样做的一个问题是，如果用户改变了窗口大小，比如点击了最大化或者拖动窗口边缘，这时，你就要自己编写相应的函数来响应这些变化，以避免那些组件还只是静静地呆在一个角落。或者，更简单的方法是直接禁止用户改变大小。

不过，Qt 提供了另外的一种机制，就是布局，来解决这个问题。你只要把组件放入某一种布局之中，当需要调整大小或者位置的时候，Qt 就知道该怎样进行调整。这类似于 Swing 的布局管理器，不过 Qt 的布局没有那么多，只有有限的几个。

来看一下下面的例子：

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget *window = new QWidget;
    window->setWindowTitle("Enter your age");

    QSpinBox *spinBox = new QSpinBox;
    QSlider *slider = new QSlider(Qt::Horizontal);
    spinBox->setRange(0, 130);
    slider->setRange(0, 130);

    QObject::connect(slider, SIGNAL(valueChanged(int)),
spinBox, SLOT(setValue(int)));
    QObject::connect(spinBox, SIGNAL(valueChanged(int)),
slider, SLOT(setValue(int)));
    spinBox->setValue(35);

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(spinBox);
    layout->addWidget(slider);
    window->setLayout(layout);

    window->show();
}
```

```
    return app.exec();  
}
```

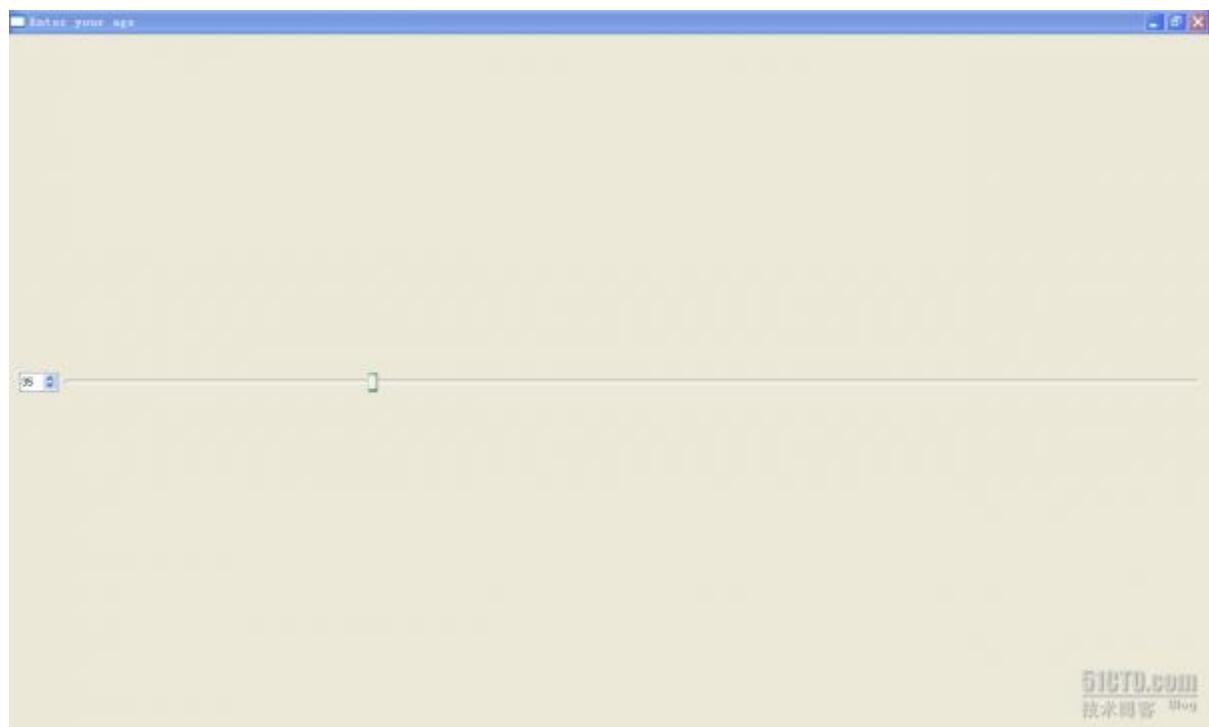
这里使用了两个新的组件：QSpinBox 和 QSlider，以及一个新的顶级窗口 QWidget。QSpinBox 是一个有上下箭头的微调器，QSlider 是一个滑动杆，只要运行一下就会明白到底是什么东西了。

代码并不是那么难懂，还是来简单的看一下。首先创建了一个 QWidget 的实例，调用 setWindowTitle 函数来设置窗口标题。然后创建了一个 QSpinBox 和 QSlider，分别设置了它们值的范围，使用的是 setRange 函数。然后进行信号槽的链接。这点后面再详细说明。然后是一个 QHBoxLayout，就是一个水平布局，按照从左到右的顺序进行添加，使用 addWidget 添加好组件后，调用 QWidget 的 setLayout 把 QWidget 的 layout 设置为我们定义的这个 Layout，这样，程序就完成了！

编译运行一下，可以看到效果：



如果最大化的话：



虽然我并没有添加任何代码，但是那个 layout 就已经明白该怎样进行布局。

或许你发现，那两个信号槽的链接操作会不会产生无限递归？因为 `steValue` 就会引发 `valueChanged` 信号！答案是不会。这两句语句实现了，当 `spinBox` 发出 `valueChanged` 信号的时候，会回调 `slider` 的 `setValue`，以更新 `slider` 的值；而 `slider` 发出 `valueChanged` 信号的时候，又会回调 `slider` 的 `setValue`。但是，如果新的 `value` 和旧的 `value` 是一样的话，是不会发出这个信号的，因此避免了无限递归。

迷糊了吧？举个例子看。比如下面的 `spinBox->setValue(35)` 执行的时候，首先，`spinBox` 会将自己的值设为 35，这样，它的值与原来的不一样了（在没有 `setValue` 之前的时候，默认值是 0），于是它发出了 `valueChanged` 信号。`slider` 接收到这个信号，于是回调自己的 `setValue` 函数，将它的值也设置成 35，它也发出了 `valueChanged` 信号。当然，此时 `spinBox` 又收到了，不过它发现，这个 35 和它本身值是一样的，于是它就不发出信号，所以信号传递就停止了。

那么，你会问，它们是怎么知道值的呢？答案很简单，因为你的信号和槽都接受了一个 `int` 参数！新的值就是通过这个进行传递的。实际上，我们利用 Qt 的信号槽机制完成了一个数据绑定，使两个组件或者更多组件的状态能够同步变化。

Qt 一共有三种主要的 layout，分别是：

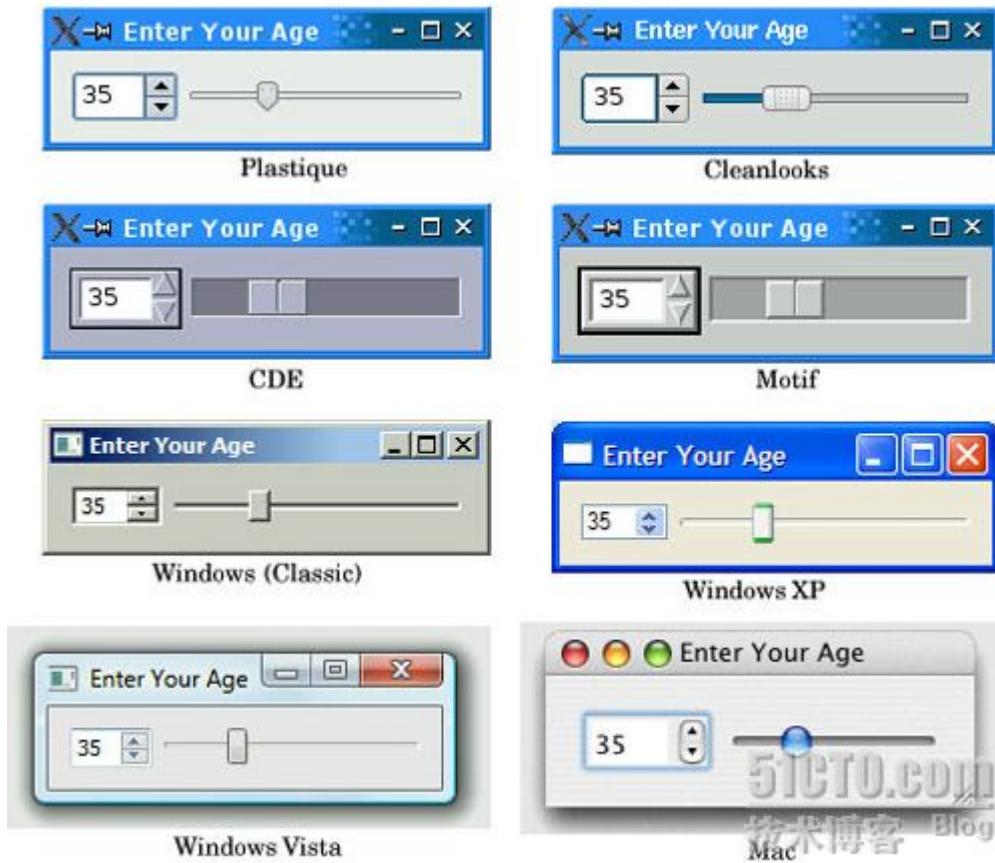
`QHBoxLayout`— 按照水平方向从左到右布局；

`VBoxLayout`— 按照竖直方向从上到下布局；

`QGridLayout`— 在一个网格中进行布局，类似于 HTML 的 `table`。

`layout` 使用 `addWidget` 添加组件，使用 `addLayout` 可以添加子布局，因此，这就有了无穷无尽的组合方式。

我是在 Windows 上面进行编译的，如果你要在其他平台上面，应用程序就会有不同的样子：



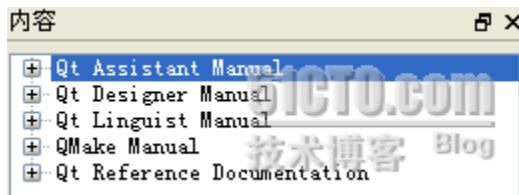
还记得前面曾经说过，Qt 不是使用的原生组件，而是自己绘制模拟的本地组件的样子，不过看看这个截图，它模拟的不能说百分百一致，也可说是惟妙惟肖了…
😊

Qt 学习之路(6): API 文档的使用

今天来说一下有关 Qt API 文档的使用。因为 Qt 有一个商业版本，因此它的文档十分健全，而且编写良好。对于开发者来说，查看文档时开发必修课之一——没有人能够记住那么多 API 的使用！

在 Qt 中查看文档是一件很简单的事情。如果你使用 QtCreator，那么左侧的 Help 按钮就是文档查看入口。否则的话，你可以在 Qt 的安装目录下的 bin 里面的 assistant.exe 中看到 Qt 的文档。在早期版本中，Qt 的文档曾以 HTML 格式发布，不过在 2009.03 版中我没有找到 HTML 格式的文档，可能 Qt 已经把它全部换成二进制格式的了吧？——当然，如果你全部安装了 Qt 的组件，是可以在开始菜单中找到 assistant 的！

assistant 里面的文档有很多项：



其中，第一个是帮助的帮助:-)；第二个是 Qt Designer 的帮助；第三个是 Qt Linguist 的帮助；第四个是 QMake 的帮助；最后一个是 Qt 的 API 文档，在 QtCreator 中默认打开的就是这部分。

不过，关于文档的内容这里实在不好阐述，因为整个文档太大了，我也并没有看过多少，很多时候都是随用随查，就好像是字典一样——谁也不会天天没事抱着本字典去看不是？还有就是这里的文档都是英文的，不过如果是做开发的话，了解一些英文还是很有帮助的，不是吗？

Qt 学习之路(7): 创建一个对话框(上)

首先说明一点，在 C++ GUI Programming with Qt4, 2nd 中，这一章连同以后的若干章一起，完成了一个比较完整的程序——一个模仿 Excel 的电子表格。不过这个程序挺大的，而且书中也没有给出完整的源代码，只是分段分段的——我不喜欢这个样子，我想要看到我写出来的是什么东西，这是最主要的，而不是慢慢的过上几章的内容才能看到自己的作品。所以，我打算换一种方式，每章只给出简单的知识，但是每章都能够运行出东西来。好了，扯完了，下面开始！

以前说的主要是一些基础知识，现在我们来真正做一个东西——一个查找对话框。什么？什么叫查找对话框？唉唉，先看看我们的最终作品吧！



好了，首先新建一个工程，就叫 FindDialog 吧！嗯，当然还是 Qt Gui Application，然后最后一步注意，Base Dialog 选择 QDialog，而不是默认的 QMainWindow，因为我们要学习建立对话框嘛！名字随便起，不过我就叫 finddialog 啦！Ganarate form 还是不要的。然后 Finish 就好了。

打开 finddialog.h，开始编写头文件。

```

class QCheckBox;
class QLabel;
class QLineEdit;
class QPushButton;

class FindDialog : public QDialog
{
    Q_OBJECT

public:
    FindDialog(QWidget *parent = 0);
    ~FindDialog();
signals:
    void findNext(const QString &str, Qt::CaseSensitivity cs);
    void findPrevious(const QString &str,
                      Qt::CaseSensitivity cs);
private slots:
    void findClicked();
    void enableFindButton(const QString &text);
private:
    QLabel *label;
    QLineEdit *lineEdit;
    QCheckBox *caseCheckBox;
    QCheckBox *backwardCheckBox;
    QPushButton *findButton;
    QPushButton *closeButton;
};


```

大家都是懂得 C++ 的啊，所以什么#ifndef, #define 和#endif 的含义和用途就不再赘述了。

首先，声明四个用到的类。这里做的是前向声明，否则的话是编译不过的，因为编译器不知道这些类是否存在。简单来说，所谓前向声明就是告诉编译器，我要用这几个类，而且这几个类存在，你就不要担心它们存不存在的问题啦！

然后是我们的 FindDialog，继承自 QDialog。

下面是一个重要的东西：Q_OBJECT。这是一个宏。凡是定义信号槽的类都必须声明这个宏。至于为什么，我们以后再说。

然后是 public 的构造函数和析构函数声明。

然后是一个 signal:, 这是 Qt 的关键字——还记得前面说过的嘛？Qt 扩展了 C++ 语言，因此它有自己的关键字——这是对信号的定义，也就是说，FindDialog

有两个 public 的信号，它可以在特定的时刻发出这两个信号，就这里来说，如果用户点击了 Find 按钮，并且选中了 Search backward，就会发出 findPrevious()，否则发出 findNext()。

紧接着是 private slots: 的定义，和前面的 signal 一样，这是私有的槽的定义。也就是说，FindDialog 具有两个槽，可以接收某些信号，不过这两个槽都是私有的。

为了 slots 的定义，我们需要访问 FindDialog 的组件，因此，我们把其中的组件定义为成员变量以便访问。正是因为需要定义这些组件，才需要对它们的类型进行前向声明。因为我们仅仅使用的是指针，并不涉及到这些类的函数，因此并不需要 include 它们的头文件——当然，你想直接引入头文件也可以，不过那样的话编译速度就会慢一些。

好了，头文件先说这些，下一篇再说源代码啦！休息，休息一下！

Qt 学习之路(8): 创建一个对话框(下)

接着前一篇，下面是源代码部分：

```
FindDialog::FindDialog(QWidget *parent)
    : QDialog(parent)
{
    label = new QLabel(tr("Find &what:"));
    lineEdit = new QLineEdit;
    label->setBuddy(lineEdit);

    caseCheckBox = new QCheckBox(tr("Match &case"));
    backwardCheckBox = new QCheckBox(tr("Search
&backford"));

    findButton = new QPushButton(tr("&Find"));
    findButton->setDefault(true);
    findButton->setEnabled(false);

    closeButton = new QPushButton(tr("Close"));

    connect(lineEdit, SIGNAL(textChanged(const
QString&)), this, SLOT(enableFindButton(const
QString&)));
    connect(findButton, SIGNAL(clicked()), this,
SLOT(findClicked()));
    connect(closeButton, SIGNAL(clicked()), this,
SLOT(close()));
}
```

```

QHBoxLayout *topLeftLayout = new QHBoxLayout;
topLeftLayout->addWidget(label);
topLeftLayout->addWidget(lineEdit);

QVBoxLayout *leftLayout = new QVBoxLayout;
leftLayout->addLayout(topLeftLayout);
leftLayout->addWidget(caseCheckBox);
leftLayout->addWidget(backwardCheckBox);

QVBoxLayout *rightLayout = new QVBoxLayout;
rightLayout->addWidget(findButton);
rightLayout->addWidget(closeButton);
rightLayout->addStretch();

QHBoxLayout *mainLayout = new QHBoxLayout;
mainLayout->addLayout(leftLayout);
mainLayout->addLayout(rightLayout);
setLayout(mainLayout);

setWindowTitle(tr("Find"));
setFixedHeight(sizeHint().height());
}

FindDialog::~FindDialog()
{

}

void FindDialog::findClicked()
{
    QString text = lineEdit->text();
    Qt::CaseSensitivity cs = caseCheckBox->isChecked() ?
    Qt::CaseInsensitive : Qt::CaseSensitive;
    if(backwardCheckBox->isChecked()) {
        emit findPrevious(text, cs);
    } else {
        emit findNext(text, cs);
    }
}

void FindDialog::enableFindButton(const QString & text)
{
    findButton->setEnabled(!text.isEmpty());
}

```

}

CPP 文件要长一些哦——不过，它们的价钱也会更高，嘿嘿——嗯，来看代码，第一行 include 的是 QtGui。Qt 是分模块的，记得我们建工程的时候就会问你，使用哪些模块？QtCore？QtGui？QtXml？等等。这里，我们引入 QtGui，它包括了 QtCore 和 QtGui 模块。不过，这并不是最好的做法，因为 QtGui 文件很大，包括了 GUI 的所有组件，但是很多组件我们根本是用不到的——就像 Swing 的 import，你可以 import 到类，也可以使用*，不过都不会建议使用*，这里也是一样的。我们最好只引入需要的组件。不过，那样会把文件变长，现在就先用 QtGui 啦，只要记得正式开发时不能这么用就好啦！

构造函数有参数初始化列表，用来调用父类的构造函数，相当于 Java 里面的 super() 函数。这是 C++ 的相关知识，不是 Qt 发明的，这里不再赘述。

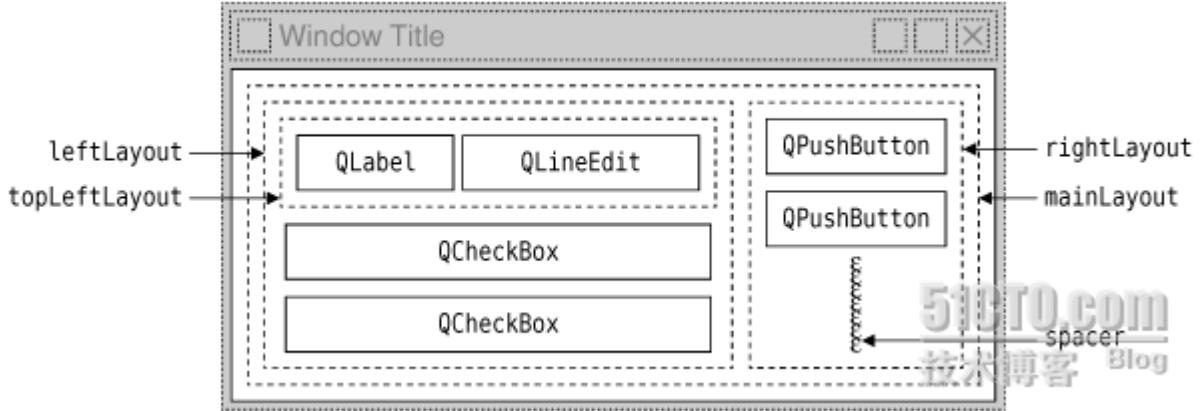
然后新建一个 QLabel。还记得前面的 Hello, world! 里面也使用过 QLabel 吗？那时候只是简单的传入一个字符串啊！这里怎么是一个函数 tr()？函数 tr() 全名是 QObject::tr()，被它处理的字符串可以使用工具提取出来翻译成其他语言，也就是做国际化使用。这以后还会仔细讲解，只要记住，Qt 的最佳实践：如果你想让你的程序国际化的话，那么，所有用户可见的字符串都要使用 QObject::tr()！但是，为什么我们没有写 QObject::tr()，而仅仅是 tr() 呢？原来，tr() 函数是定义在 Object 里面的，所有使用了 Q_OBJECT 宏的类都自动具有 tr() 函数。

字符串中的&代表快捷键。注意看下面的 findButton 的&Find，它会生成 Find 字符串，当你按下 Alt+F 的时候，这个按钮就相当于被点击——这么说很难受，相信大家都明白什么意思。同样，前面 label 里面也有一个&，因此它的快捷键就是 Alt+W。不过，这个 label 使用了 setBuddy 函数，它的意思是，当 label 获得焦点时，比如按下 Alt+W，它的焦点会自动传给它的 buddy，也就是 lineEdit。看，这就是伙伴的含义（buddy 英文就是伙伴的意思）。

后面几行就比较简单了：创建了两个 QCheckBox，把默认的按钮设为 findButton，把 findButton 设为不可用——也就是变成灰色的了。

再下面是三个 connect 语句，用来连接信号槽。可以看到，当 lineEdit 发出 textChanged(const QString&) 信号时，FindDialog 的 enableFindButton(const QString&) 函数会被调用——这就是回调，是有系统自动调用，而不是你去调用——当 findButton 发出 clicked() 信号时，FindDialog 的 findClicked() 函数会被调用；当 closeButton 发出 clicked() 信号时，FindDialog 的 close() 函数会被调用。注意，connect() 函数也是 QObject 的，因为我们继承了 QObject，所以能够直接使用。

后面的很多行语句都是 layout 的使用，虽然很复杂，但是很清晰——编写 layout 布局最重要一点就是思路清楚，想清楚哪个套哪个，就会很好编写。这里我们的对话框实际上是这个样子的：



注意那个 spacer 是由 rightLayout 的 addStretch() 添加的，就像弹簧一样，把上面的组件“顶起来”。

最后的 setWindowTitle() 就是设置对话框的标题，而 setFixedHeight() 是设置成固定的高度，其参数值 sizeHint() 返回“最理想”的大小，这里我们使用的是 height() 函数去到“最理想”的高度。

好了，下面该编写槽了——虽然说是 slot，但实际上它就是普通的函数，既可以和其他函数一样使用，又可以被系统回调。

先看 findClicked() 函数。首先取出lineEdit 的输入值；然后判断 caseCheckBox 是不是选中，如果选中就返回 Qt::CaseInsensitive，否则返回 Qt::CaseSensitive，用于判断是不是大小写敏感的查找；最后，如果 backwardCheckBox 被选中，就 emit(发出) 信号 findPrevious()，否则 emit 信号 findNext。

enableFindButton() 则根据lineEdit 的内容是不是变化——这是我们的 connect 连接的——来设置 findButton 是否可以使用，这个很简单，不再说了。

这样，FindDialog.cpp 也就完成了。下面编写 main.cpp——其实 QtCreator 已经替我们完成了——

```
#include "finddialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    FindDialog *dialog = new FindDialog;
    dialog->show();
    return app.exec();
}
```

运行一下看看我们的成果吧！

虽然很简单，也没有什么实质性的功能，但是我们已经能够制作对话框了——Qt 的组件成百上千，不可能全部介绍完，只能用到什么学什么，更重要的是，我们已经了解了其编写思路，否则的话，即便是你拿着全世界所有的砖瓦，没有设计图纸，你也不知道怎么把它们组合成高楼大厦啊！

嘿嘿，下回见！

Qt 学习之路(9): 深入了解信号槽

信号槽机制是 Qt 编程的基础。通过信号槽，能够使 Qt 各组件在不知道对方的情形下能够相互通讯。这就将类之间的关系做了最大程度的解耦。

槽函数和普通的 C++ 成员函数没有很大的区别。它们也可以使 `virtual` 的；可以被重写；可以使 `public`、`protected` 或者 `private` 的；可以由其它的 C++ 函数调用；参数可以是任何类型的。如果说区别，那就是，槽函数可以和一个信号相连接，当这个信号发生时，它可以被自动调用。

`connect()` 语句的原型类似于：

```
connect (sender, SIGNAL(signal), receiver, SLOT(slot));
```

这里，`sender` 和 `receiver` 都是 `QObject` 类型的，`signal` 和 `slot` 都是没有参数名称的函数签名。`SIGNAL()` 和 `SLOT()` 宏用于把参数转换成字符串。

深入的说，信号槽还有更多可能的用法，如下所示。

一个信号可以和多个槽相连：

```
connect (slider, SIGNAL(valueChanged(int)),
         spinBox, SLOT(setValue(int)));
connect (slider, SIGNAL(valueChanged(int)),
         this, SLOT(updateStatusBarIndicator(int)));
```

注意，如果是这种情况，这些槽会一个接一个的被调用，但是它们的调用顺序是不确定的。

多个信号可以连接到一个槽：

```
connect (lcd, SIGNAL(overflow()),
         this, SLOT(handleMathError()));
connect (calculator, SIGNAL(divisionByZero()),
         this, SLOT(handleMathError()));
```

这是说，只要任意一个信号发出，这个槽就会被调用。

一个信号可以连接到另外的一个信号：

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString
&)));
```

这是说，当第一个信号发出时，第二个信号被发出。除此之外，这种信号-信号的形式和信号-槽的形式没有什么区别。

槽可以被取消链接：

```
disconnect(lcd, SIGNAL(overflow()),
           this, SLOT(handleMathError()));
```

这种情况并不经常出现，因为当一个对象 delete 之后，Qt 自动取消所有连接到这个对象上面的槽。

为了正确的连接信号槽，信号和槽的参数个数、类型以及出现的顺序都必须相同，例如：

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString
&)),
        this, SLOT(processReply(int, const QString
&)));
```

这里有一种例外情况，如果信号的参数多于槽的参数，那么这个参数之后的那些参数都会被忽略掉，例如：

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString
&)),
        this, SLOT(checkErrorCode(int)));
```

这里，const QString &这个参数就会被槽忽略掉。

如果信号槽的参数不相容，或者是信号或槽有一个不存在，或者在信号槽的连接中出现了参数名字，在 Debug 模式下编译的时候，Qt 都会很智能的给出警告。

在这之前，我们仅仅在 widgets 中使用到了信号槽，但是，注意到 connect() 函数其实是在 QObject 中实现的，并不局限于 GUI，因此，只要我们继承 QObject 类，就可以使用信号槽机制了：

```
class Employee : public QObject
{
    Q_OBJECT
```

```

public:
    Employee() { mySalary = 0; }
    int salary() const { return mySalary; }

public slots:
    void setSalary(int newSalary);

signals:
    void salaryChanged(int newSalary);

private:
    int mySalary;
};

```

在使用时，我们给出下面的代码：

```

void Employee::setSalary(int newSalary)
{
    if (newSalary != mySalary) {
        mySalary = newSalary;
        emit salaryChanged(mySalary);
    }
}

```

这样，当 `setSalary()` 调用的时候，就会发出 `salaryChanged()` 信号。注意这里的 `if` 判断，这是避免递归的方式！还记得前面提到的循环连接吗？如果没有 `if`，当出现了循环连接的时候就会产生无限递归。

Qt 学习之路(10): Meta-Object 系统

前面说过，Qt 使用的是自己的预编译器，它提供了对 C++ 的一种扩展。利用 Qt 的信号槽机制，就可以把彼此独立的模块相互连接起来，不需要实现知道模块的任何细节。

为了达到这个目的，Qt 提出了一个 Meta-Object 系统。它提供了两个关键的作用：信号槽和内省。

面向对象程序设计里面会讲到 Smalltalk 语言有一个元类系统。所谓元类，就是这里所说的 Meta-Class。如果写过 HTML，会知道 HTML 标签里面也有一个，这是用于说明页面的某些属性的。同样，Qt 的 Meta-Object 系统也是类似的作用。内省又称为反射，允许程序在运行时获得类的相关信息，也就是 meta-information。什么是 meta-information 呢？举例来说，像这个类叫什么名字？它有什么属性？有什么方法？它的信号列表？它的槽列表？等等这些信

息，就是这个类的 meta-information，也就是“元信息”。这个机制还提供了对国际化的支持，是 QSA(Qt Script for Application)的基础。

标准 C++ 并没有 Qt 的 meta-information 所需要的动态 meta-information。所以，Qt 提供了一个独立的工具，moc，通过定义 Q_OBJECT 宏实现到标准 C++ 函数的转变。moc 使用纯 C++ 实现的，因此可以在任何编译器中使用。

这种机制工作过程是：

首先，Q_OBJECT 宏声明了一些 QObject 子类必须实现的内省的函数，如 metaObject(), tr(), qt_metacall() 等；

第二，Qt 的 moc 工具实现 Q_OBJECT 宏声明的函数和所有信号；

第三，QObject 成员函数 connect() 和 disconnect() 使用这些内省函数实现信号槽的连接。

以上这些过程是 qmake, moc 和 QObject 自动处理的，你不需要去考虑它们。如果实现好奇的话，可以通过查看 QMetaObject 的文档和 moc 的源代码来一睹芳容。

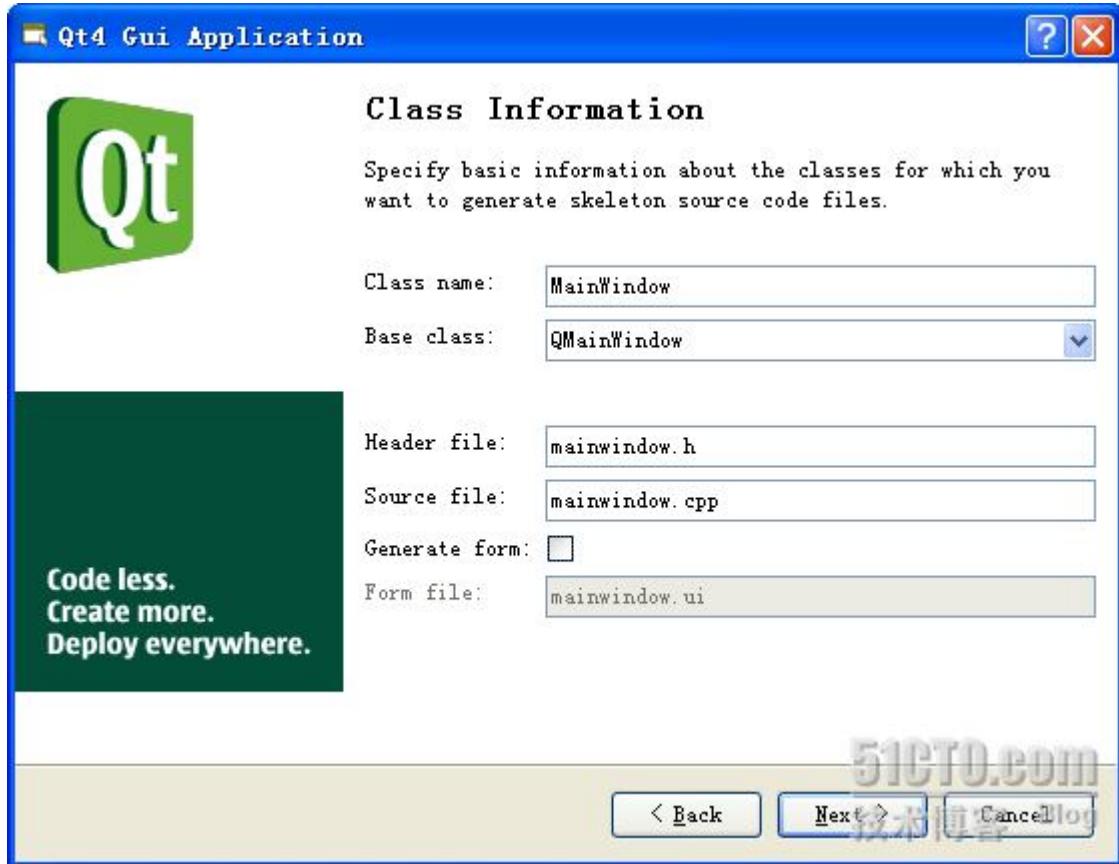
Qt 学习之路(11): MainWindow

尽管 Qt 提供了很方便的快速开发工具 QtDesigner 用来拖放界面元素，但是现在我并不打算去介绍这个工具，原因之一在于我们的学习大体上是依靠手工编写代码，过早的接触设计工具并不能让我们对 Qt 的概念突飞猛进……

前面说过，本教程很大程度上依照的是《C++ GUI Programming with Qt4, 2nd Edition》这本书。但是，这本书中接下来的部分用了很大的篇幅完成了一个简单的类似 Excel 的程序。虽然最终效果看起来很不错，但我并不打算完全依照这个程序来，因为这个程序太大，以至于我们在开始之后会有很大的篇幅接触不到能够运行的东西，这无疑会严重打击学习的积极性——至少我是如此，看不到做的东西很难受——所以，我打算重新组织一下这个程序，请大家按照我的思路试试看吧！

闲话少说，下面开始新的篇章！

就像 Swing 的顶层窗口一般都是 JFrame 一样，Qt 的 GUI 程序也有一个常用的顶层窗口，叫做 MainWindow。好了，现在我们新建一个 Gui Application 项目 MyApp，注意在后面选择的时候选择 Base Class 是 QMainWindow。



然后确定即可。此时，QtCreator 已经为我们生成了必要的代码，我们只需点击一下 Run，看看运行出来的结果。



一个很简单的窗口，什么都没有，这就是我们的主窗口了。

MainWindow 继承自 QMainWindow。QMainWindow 窗口分成几个主要的区域：



最上面是 Window Title，用于显示标题和控制按钮，比如最大化、最小化和关闭等；下面一些是 Menu Bar，用于显示菜单；再下面一点事 Toolbar areas，用于显示工具条，注意，Qt 的主窗口支持多个工具条显示，因此这里是 ares，你可以把几个工具条并排显示在这里，就像 Word2003 一样；工具条下面是 Dock window areas，这是停靠窗口的显示区域，所谓停靠窗口就是像 Photoshop 的工具箱一样，可以在主窗口的四周显示；再向下是 Status Bar，就是状态栏；中间最大的 Central widget 就是主要的工作区了。

好了，今天的内容不多，我们以后的工作就是要对这个 MainWindow 进行修改，以满足我们的各种需要。

Qt 学习之路(12): 菜单和工具条

在前面的 QMainWindow 的基础之上，我们开始着手建造我们的应用程序。虽然现在已经有一个框架，但是，确切地说我们还一行代码没有写呢！下面的工作就不那么简单了！在这一节里面，我们要为我们的框架添加菜单和工具条。

就像 Swing 里面的 Action 一样，Qt 里面也有一个类似的类，叫做 QAction。顾名思义，QAction 类保存有关于这个动作，也就是 action 的信息，比如它的文本描述、图标、快捷键、回调函数(也就是信号槽)，等等。神奇的是，QAction 能够根据添加的位置来改变自己的样子 —— 如果添加到菜单中，就会显示成一个菜单项；如果添加到工具条，就会显示成一个按钮。这也是为什么要把菜单和按钮放在一节里面。下面开始学习！

首先，我想添加一个打开命令。那么，就在头文件里面添加一个私有的 QAction 变量：

```
class QAction;
```

```
//...  
private:  
QAction *openAction;  
//...
```

注意，不要忘记 QAction 类的前向声明哦！要不就会报错的！

然后我们要在 cpp 文件中添加 QAction 的定义。为了简单起见，我们直接把它定义在构造函数里面：

```
openAction = new QAction(tr("&Open"), this);  
openAction->setShortcut(QKeySequence::Open);  
openAction->setStatusTip(tr("Open a file."));
```

第一行代码创建一个 QAction 对象。QAction 有几个重载的构造函数，我们使用的是

```
QAction(const QString &text, QObject* parent)
```

这一个。它有两个参数，第一个 text 是这个动作的文本描述，用来显示文本信息，比如在菜单中的文本；第二个是 parent，一般而言，我们通常传入 this 指针就可以了。我们不需要去关心这个 parent 参数具体是什么，它的作用是指明这个 QAction 的父组件，当这个父组件被销毁时，比如 delete 或者由系统自动销毁，与其相关联的这个 QAction 也会自动被销毁。

如果你还是不明白构造函数的参数是什么意思，或者说想要更加详细的了解 QAction 这个类，那么就需要自己翻阅一下它的 API 文档。前面说过有关 API 的使用方法，这里不再赘述。这也是学习 Qt 的一种方法，因为 Qt 是一个很大的库，我们不可能面面俱到，因此只为说道用到的东西，至于你自己想要实现的功能，就需要自己去查文档了。

第二句，我们使用了 setShortcut 函数。shortcut 是这个动作的快捷键。Qt 的 QKeySequence 已经为我们定义了很多内置的快捷键，比如我们使用的 Open。你可以通过查阅 API 文档获得所有的快捷键列表，或者是在 QtCreator 中输入::后会有系统的自动补全功能显示出来。这个与我们自己定义的有什么区别呢？简单来说，我们完全可以自己定义一个 tr(“Ctrl+O”) 来实现快捷键。原因在于，这是 Qt 跨平台性的体现。比如 PC 键盘和 Mac 键盘是不一样的，一些键在 PC 键盘上有，而 Mac 键盘上可能并不存在，或者反之，所以，推荐使用 QKeySequence 类来添加快捷键，这样，它会根据平台的不同来定义不同的快捷键。

第三句是 setStatusTip 函数。这是添加状态栏的提示语句。状态栏就是主窗口最下面的一条。现在我们的程序还没有添加状态栏，因此你是看不到有什么作用的。

下面要做的是把这个 QAction 添加到菜单和工具条：

```

QMenu *file = menuBar()->addMenu(tr("&File"));
file->addAction(openAction);

QToolBar *toolBar = addToolBar(tr("&File"));
toolBar->addAction(openAction);

```

QMainWindow 有一个 menuBar() 函数，会返回菜单栏，也就是最上面的那一条。如果不存在会自动创建，如果已经存在就返回那个菜单栏的指针。直接使用返回值添加一个菜单，也就是 addMenu，参数是一个 QString，也就是显示的菜单名字。然后使用这个 QMenu 指针添加这个 QAction。类似的，使用 addToolBar 函数的返回值添加了一个工具条，并且把这个 QAction 添加到了上面。

好了，主要的代码已经写完了。不过，如果你只修改这些的话，是编译不过的哦！因为像 menuBar() 函数返回一个 QMenuBar 指针，但是你并没有 include 它的头文件哦！虽然没有明着写出 QMenuBar 这个类，但是实际上你已经用到了它的 addMenu 函数了，所以还是要注意的！

下面给出来全部的代码：

```

1.mainwindow.h
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QtGui/QMainWindow>

class QAction;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    QAction *openAction;
};

#endif // MAINWINDOW_H

2.mainwindow.cpp

#include <QtGui/QAction>;
#include <QtGui/QMenu>;

```

```

#include <QtGui/QMenuBar>;
#include <QtGui/QKeySequence>;
#include <QtGui/QToolBar>;
#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent)
{
    openAction = new QAction(tr("&Open"), this);
    openAction->setShortcut(QKeySequence::Open);
    openAction->setStatusTip(tr("Open a file."));

    QMenu *file = menuBar()->addMenu(tr("&File"));
    file->addAction(openAction);

    QToolBar *toolBar = addToolBar(tr("&File"));
    toolBar->addAction(openAction);
}

MainWindow::~MainWindow()
{
}

```

main.cpp 没有修改，这里就不给出了。下面是运行结果：



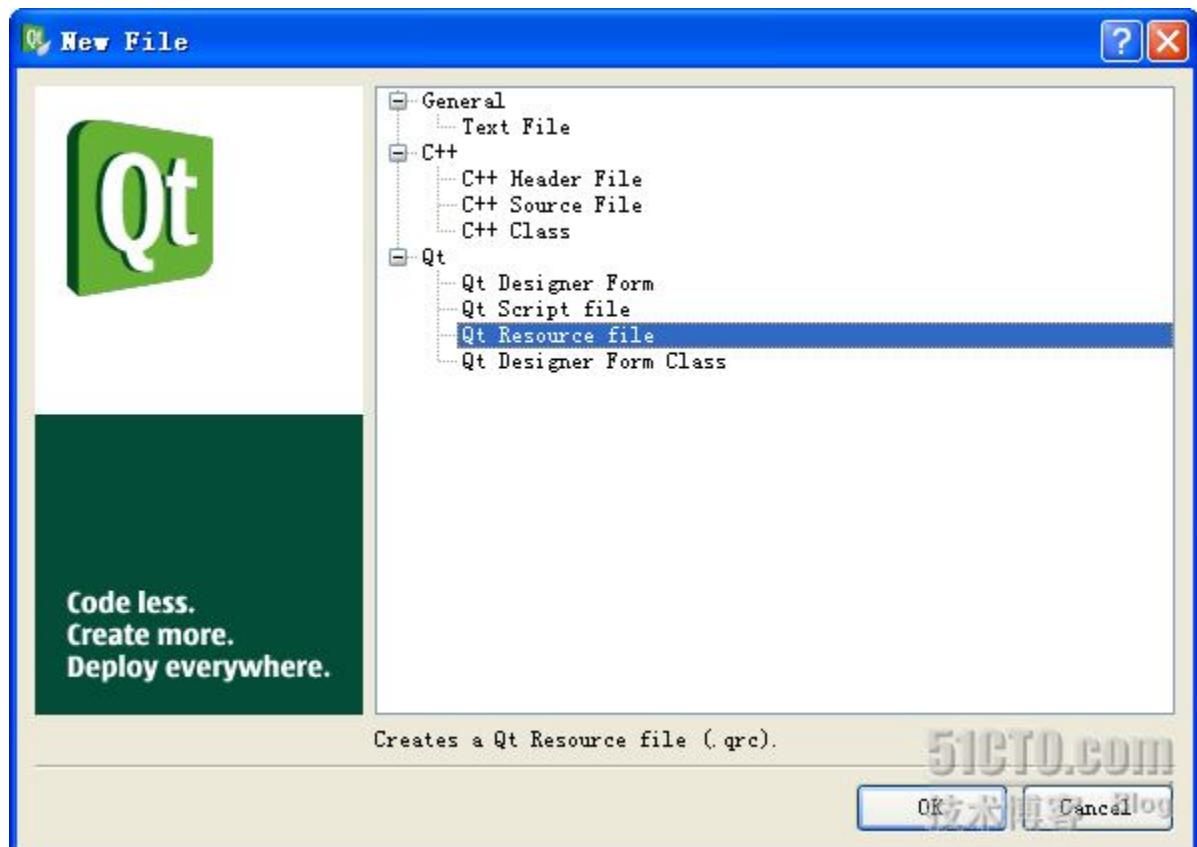
很丑，是吧？不过我们已经添加上了菜单和工具条了哦！按一下键盘上的 Alt+F，因为这是我们给它定义的快捷键。虽然目前挺难看，不过以后就会变得漂亮的！想想看，Linux 的 KDE 桌面可是 Qt 实现的呢！

Qt 学习之路(13): 菜单和工具条(续)

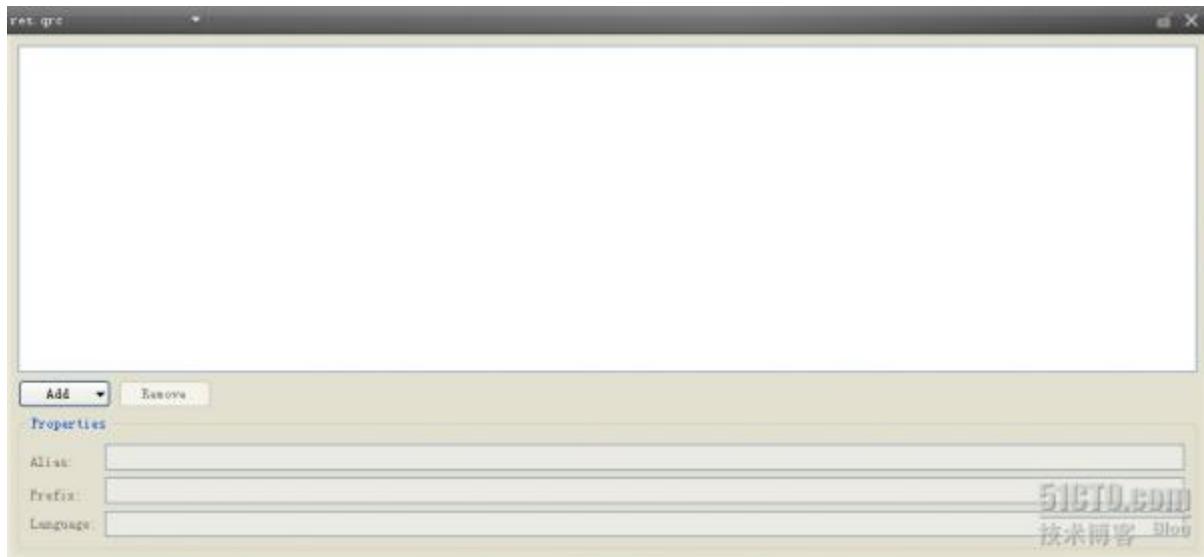
前面一节我们已经把 QAction 添加到菜单和工具条上面。现在我们要添加一些图片美化一下，然后把信号槽加上，这样，我们的 action 就可以相应啦！

首先来添加图标。QAction 的图标会显示在菜单项的前面以及工具条按钮上面显示。

为了添加图标，我们首先要使用 Qt 的资源文件。在 QtCreator 的项目上右击，选择 New File…，然后选择 resource file。

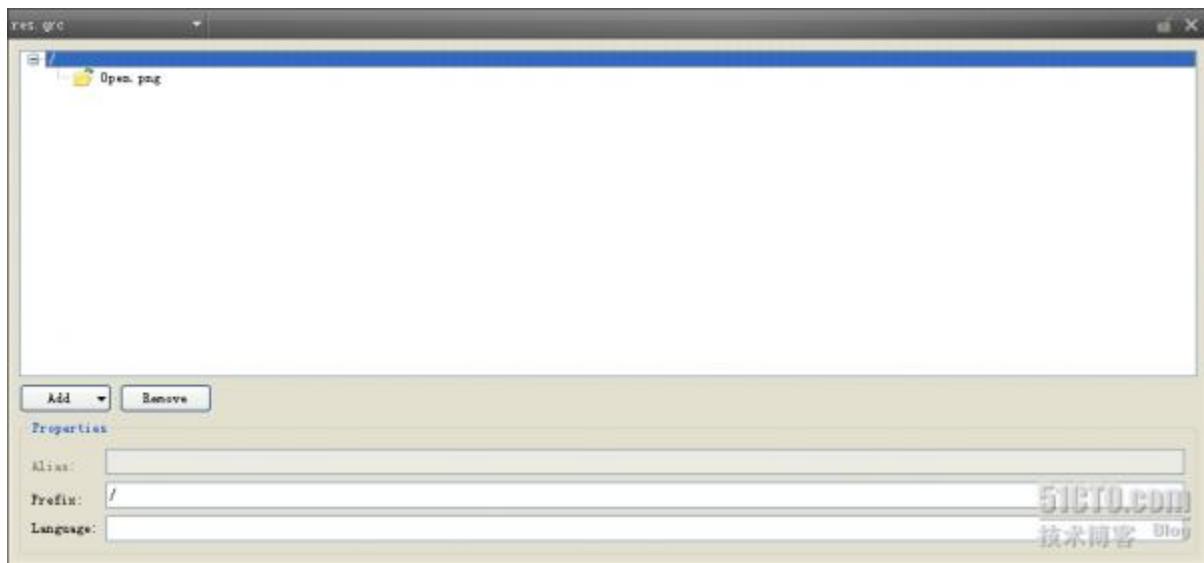


然后点击 next，选择好位置，Finish 即可。为了使用方便，我就把这个文件建在根目录下，建议应该在仔细规划好文件之后，建在专门的 resources 文件夹下。完成之后，生成的是一个.qrc 文件，qrc 其实是 Qt Recource Collection 的缩写。它只是一个普通的 XML 文件，可以用记事本等打开。不过，这里我们不去深究它的结构，完全利用 QtCreator 操作这个文件，



点击 Add 按钮，首先选择 Add prefix，然后把生成的/new/prefix 改成/。这是 prefix 就是以后使用图标时需要提供的前缀，以/开头。添加过 prefix 之后，然后在工程文件中添加一个图标，再选择 Add file，选择那个图标。这样完成之后保存 qrc 文件即可。

说明一下，QToolBar 的图标大小默认是 32*32，菜单默认是 16*16。如果提供的图标小于要求的尺寸，则不做操作，Qt 不会为你放大图片；反之，如果提供的图标文件大于相应的尺寸要求，比如是 64*64，Qt 会自动缩小尺寸。



图片的路径怎么看呢？可以看出，Qt 的资源文件视图使用树状结构，根是/，叶子节点就是图片位置，连接在一起就是路径。比如这张图片的路径就是 /Open.png。

注意，为了简单起见，我们没有把图标放在专门的文件夹中。正式的项目中应该单独有一个 resources 文件夹放资源文件的。

然后回到前面的 mainwindow.cpp，在构造函数中修改代码：

```
openAction = new QAction(tr("&Open"), this);
openAction->setShortcut(QKeySequence::Open);
openAction->setStatusTip(tr("Open a file."));
openAction->setIcon(QIcon(":/Open.png")); // Add code.
```

我们使用 setIcon 添加图标。添加的类是 QIcon，构造函数需要一个参数，是一个字符串。由于我们要使用 qrc 中定义的图片，所以字符串以 : 开始，后面跟着 prefix，因为我们先前定义的 prefix 是 /，所以就需要一个 /，然后后面是 file 的路径。这是在前面的 qrc 中定义的，打开 qrc 看看那张图片的路径即可。

好了，图片添加完成，然后点击运行，看看效果吧！



瞧！我们只需要修改 QAction，菜单和工具条就已经为我们做好了相应的处理，还是很方便的！

下一步，为 QAction 添加事件响应。还记得 Qt 的事件响应机制是基于信号槽吗？点击 QAction 会发出 triggered() 信号，所以，我们要做的是声明一个 slot，然后 connect 这个信号。

mainwindow.h

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

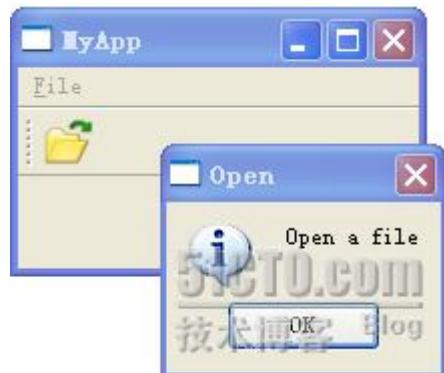
private slots:
    void open();
```

```
private:  
    QAction *openAction;  
};
```

因为我们的 open() 目前只要在类的内部使用，因此定义成 private slots 即可。然后修改 cpp 文件：

```
MainWindow::MainWindow(QWidget *parent)  
: QMainWindow(parent)  
{  
    openAction = new QAction(tr("&Open"), this);  
    openAction->setShortcut(QKeySequence::Open);  
    openAction->setStatusTip(tr("Open a file."));  
    openAction->setIcon(QIcon(":/Open.png"));  
    connect(openAction, SIGNAL(triggered()), this,  
            SLOT(open()));  
  
    QMenu *file = menuBar()->addMenu(tr("&File"));  
    file->addAction(openAction);  
  
    QToolBar *toolBar = addToolBar(tr("&File"));  
    toolBar->addAction(openAction);  
}  
  
void MainWindow::open()  
{  
    QMessageBox::information(NULL, tr("Open"), tr("Open a  
file"));  
}
```

注意，我们在 open() 函数中简单的弹出一个标准对话框，并没有其他的操作。编译后运行，看看效果：



好了，关于 QAction 的动作也已经添加完毕了！

至此， QAction 有关的问题先告一段落。最后说一下，如果你还不知道怎么添加子菜单的话，看一下 QMenu 的 API，里面会有一个 addMenu 函数。也就是说，创建一个 QMenu 然后添加就可以的啦！

Qt 学习之路(14): 状态栏

今天的内容主要还是继续完善前面的那个程序。我们要为我们的程序加上一个状态栏。

状态栏位于主窗口的最下方，提供一个显示工具提示等信息的地方。一般地，当窗口不是最大化的时候，状态栏的右下角会有一个可以调节大小的控制点；当窗口最大化的时候，这个控制点会自动消失。Qt 提供了一个 QStatusBar 类来实现状态栏。

Qt 具有一个相当成熟的 GUI 框架的实现——这一点感觉比 Swing 要强一些——Qt 似乎对 GUI 的开发做了很多设计，比如 QMainWindow 类里面就有一个 statusBar() 函数，用于实现状态栏的调用。类似 menuBar() 函数，如果不存在状态栏，该函数会自动创建一个，如果已经创建则会返回这个状态栏的指针。如果你要替换掉已经存在的状态栏，需要使用 QMainWindow 的 setStatusBar() 函数。

在 Qt 里面，状态栏显示的信息有三种类型：临时信息、一般信息和永久信息。其中，临时信息指临时显示的信息，比如 QAction 的提示等，也可以设置自己的临时信息，比如程序启动之后显示 Ready，一段时间后自动消失——这个功能可以使用 QStatusBar 的 showMessage() 函数来实现；一般信息可以用来显示页码之类的；永久信息是不会消失的信息，比如可以在状态栏提示用户 Caps Lock 键被按下之类。

QStatusBar 继承自 QWidget，因此它可以添加其他的 QWidget。下面我们在 QStatusBar 上添加一个 QLabel。

首先在 class 的声明中添加一个私有的 QLabel 属性：

```
private:  
QAction *openAction;  
QLabel *msgLabel;
```

然后在其构造函数中添加：

```
msgLabel = new QLabel;  
msgLabel->setMinimumSize(msgLabel->sizeHint());  
msgLabel->setAlignment(Qt::AlignHCenter);
```

```
statusBar() ->addWidget(msgLabel);
```

这里，第一行创建一个 QLabel 的对象，然后设置最小大小为其本身的建议大小——注意，这样设置之后，这个最小大小可能是变化的——最后设置显示规则是水平居中(HCenter)。最后一行使用 statusBar() 函数将这个 label 添加到状态栏。编译运行，将鼠标移动到工具条或者菜单的 QAction 上，状态栏就会有相应的提示：



看起来是不是很方便？只是，我们很快发现一个问题：当没有任何提示时，状态栏会有一个短短的竖线：



这是什么呢？其实，这是 QLabel 的边框。当没有内容显示时，QLabel 只显示出自己的一个边框。但是，很多情况下我们并不希望有这条竖线，于是，我们对 statusBar() 进行如下设置：

```
statusBar() ->setStyleSheet (QString ("QStatusBar::item{border: 0px}"));
```

这里先不去深究这句代码是什么意思，简单来说，就是把 QStatusBar 的子组件的 border 设置为 0，也就是没有边框。现在再编译试试吧！那个短线消失了！

QStatusBar 右下角的大小控制点可以通过 setSizeGripEnabled() 函数来设置是否存在，详情参见 API 文档。

好了，现在，我们的状态栏已经初步完成了。由于 QStatusBar 可以添加多个 QWidget，因此，我们可以构建出很复杂的状态栏。

Qt 学习之路(15): Qt 标准对话框之 QFileDialog

《Qt 学习之路》已经写到了第 15 篇，然而现在再写下去却有点困难，原因是当初并没有想到会连续的写下去，因此并没有很好的计划这些内容究竟该怎样去写。虽然前面说过，本教程主要线路参考《C++ Gui Programming with Qt 4, 2nd Edition》，然而最近的章节由于原文是一个比较完整的项目而有所改变，因此现在不知道该从何写起。

我并不打算介绍很多组件的使用，因为 Qt 有很多组件，各种组件用法众多，根本不可能介绍完，只能把 API 放在手边，边用边查。所以，对于很多组件我只是简单的介绍一下，具体用法还请自行查找(确切地说，我知道的也并不多，很多时候还是要到 API 里面去找)。

下面还是按照我们的进度，从 Qt 的标准对话框开始说起。所谓标准对话框，其实就是 Qt 内置的一些对话框，比如文件选择、颜色选择等等。今天首先介绍一下 QFileDialog。

QFileDialog 是 Qt 中用于文件打开和保存的对话框，相当于 Swing 里面的 JFileChooser。下面打开我们前面使用的工程。我们已经很有先见之明的写好了一个打开的 action，还记得前面的代码吗？当时，我们只是弹出了一个消息对话框(这也是一种标准对话框哦~)用于告知这个信号槽已经联通，现在我们要写真正的打开代码了！

修改 MainWindow 的 open 函数：

```
void MainWindow::open()
{
    QString path = QFileDialog::getOpenFileName(this, tr("Open Image"), ".", tr("Image Files (*.jpg *.png)"));
    if(path.length() == 0) {
        QMessageBox::information(NULL, tr("Path"), tr("You didn't select any files."));
    } else {
        QMessageBox::information(NULL, tr("Path"), tr("You selected ") + path);
    }
}
```

编译之前别忘记 include QFileDialog 哦！然后运行一下吧！点击打开按钮，就会弹出打开对话框，然后选择文件或者直接点击取消，会有相应的消息提示。

QFileDialog 提供了很多静态函数，用于获取用户选择的文件。这里我们使用的是

Qt 学习之路(16): Qt 标准对话框之 QColorDialog

继续来说 Qt 的标准对话框，这次说说 QColorDialog。这是 Qt 提供的颜色选择对话框。

使用 QColorDialog 也很简单，Qt 提供了 getColor() 函数，类似于 QFileDialog 的 getOpenFileName()，可以直接获得选择的颜色。我们还是使用前面的 QAction 来测试下这个函数：

```
QColor color = QColorDialog::getColor(Qt::white, this);
QString msg = QString("r: %1, g: %2, b:
%3").arg(QString::number(color.red())),
QString::number(color.green()),
QString::number(color.blue()));
QMessageBox::information(NULL, "Selected color", msg);
```

不要忘记 include QColorDialog 哟！这段代码虽然很少，但是内容并不少。

第一行 QColorDialog::getColor() 调用了 QColorDialog 的 static 函数 getColor()。这个函数有两个参数，第一个是 QColor 类型，是对话框打开时默认选择的颜色，第二个是它的 parent。

第二行比较长，涉及到 QString 的用法。如果我没记错的话，这些用法还没有提到过，本着“有用就说”的原则，尽管这些和 QColorDialog 毫不相干，这里还是解释一下。QString(“r: %1, g: %2, b: %3”) 创建了一个 QString 对象。我们使用了参数化字符串，也就是那些%1 之类。在 Java 的 properties 文件中，字符参数是用 {0}, {1} 之类实现的。其实这都是一些占位符，也就是，后面会用别的字符串替换掉这些值。占位符的替换需要使用 QString 的 arg() 函数。这个函数会返回它的调用者，因此可以使用链式调用写法。它会按照顺序替换掉占位符。然后是 QString::number() 函数，这也是 QString 的一个 static 函数，作用就是把 int、double 等值换成 QString 类型。这里是把 QColor 的 R、G、B 三个值输出了出来。关于 QString 类，我们会在以后详细说明。

第三行就比较简单了，使用一个消息对话框把刚刚拼接的字符串输出。

现在就可以运行这个测试程序了。看上去很简单，不是吗？

QColorDialog 还有一些其他的函数可以使用。

QColorDialog::setCustomColor() 可以设置用户自定义颜色。这个函数有两个值，第一个是自定义颜色的索引，第二个是自定义颜色的 RGB 值，类型是 QRgb，大家可以查阅 API 文档来看看这个类的使用，下面只给出一个简单的用法：

```
QColorDialog::setCustomColor(0, QRgb(0x0000FF));
```

getColor() 还有一个重载的函数，签名如下：

```
QColorDialog::( const QColor & initial, QWidget * parent,  
const QString & title, ColorDialogOptions options = 0 )
```

第一个参数 initial 和前面一样，是对话框打开时的默认选中的颜色；

第二个参数 parent，设置对话框的父组件；

第三个参数 title，设置对话框的 title；

第四个参数 options，是 QColorDialog::ColorDialogOptions 类型的，可以设置对话框的一些属性，如是否显示 Alpha 值等，具体属性请查阅 API 文档。特别的，这些值是可以使用 OR 操作的。

QColorDialog 相对简单一些，API 文档也很详细，大家遇到问题可以查阅文档的哦！

Qt 学习之路(tip): parent 参数

这是一篇很简单的文章，仅仅是用来说明一下一个参数的作用，因此我把它写成了 tip，而不是接下来的 17.

程序写的多了，你会发现几乎所有的 Qt 类的构造函数都会有一个 parent 参数。这个参数通常是 QObject* 或者是 QWidget* 类型的。很多情况下它都会有一个初始值 0，因此，即便你不去给它复制也没有丝毫的问题。于是，稍微偷懒一下，就会不自觉的忽略了这个参数。那么，这个参数到底是干什么用的呢？

其实，这个参数有很多用处。就像它的名字一样，这个参数指定了组件的父组件。对于一个对话框来说，对话框一般是不作为顶层容器出现的，因此在任务栏上一般是没有对话框的位置的。怎么指定这个对话框不是顶层容器呢？有父组件的组件不就不是顶层容器了吗？因此，只要你指定对话框的 parent 属性，任务栏就不会出现它的身影。当然，如果你不指定，这个对话框就成为顶层容器了，任务栏会给它留个位置的——利用这个特性，就可以实现特殊对话框可以在任务栏出现的效果，比如“关于”对话框的出现。

另外比较通用，也是很重要的作用是，parent 参数指明了组件的父组件，这样，当父组件 delete 时，Qt 可以保证所有子组件——也就是 parent 指针指向这个组件的所有组件——都会被正确的 delete 掉。这是 Qt 能够帮助我们管理一部分

内存的原因所在。Qt 是通过遍历 parent 属性来防止了这一部分内存泄漏的。因此，必要情况下还是不要忘记设置这个 parent 属性。当然，如果你不声明这个属性，当整个程序关闭时，操作系统会回收内存——因此我们所说的内存泄漏一般是指我们自己写的应用程序的内部，而不会影响到整个操作系统——当然，如果你实现太可恶，操作系统也会受不了自动关掉你的程序的:-)

Qt 学习之路(17): Qt 标准对话框之 QMessageBox

这次来说一下 QMessageBox 以及类似的几种对话框。其实，我们已经用过 QMessageBox 了，就在之前的几个程序中。不过，当时是大略的说了一下，现在专门来说说这几种对话框。

先来看一下最熟悉的 QMessageBox::information。我们在以前的代码中这样使用过：

```
QMessageBox::information(NULL, "Title", "Content",
QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);
```

下面是一个简单的例子：



现在我们从 API 中看看它的函数签名：

```
static StandardButton QMessageBox::information ( QWidget * parent, const QString & title, const QString & text,
StandardButtons buttons = Ok, StandardButton defaultButton
= NoButton );
```

首先，它是 static 的，所以我们能够使用类名直接访问到(怎么看都像废话…);然后看它那一堆参数，第一个参数 parent，说明它的父组件；第二个参数 title，也就是对话框的标题；第三个参数 text，是对话框显示的内容；第四个参数 buttons，声明对话框放置的按钮，默认是只放置一个 OK 按钮，这个参数可以使用或运算，例如我们希望有一个 Yes 和一个 No 的按钮，可以使用 QMessageBox::Yes | QMessageBox::No，所有的按钮类型可以在 QMessageBox 声明的 StandarButton 枚举中找到；第五个参数 defaultButton 就是默认选中的按钮，默认值是 NoButton，也就是哪个按钮都不选中。这么多参数，豆子也是记不住的啊！所以，我们在用 QtCreator 写的时候，可以在输入

QMessageBox::information 之后输入(, 稍等一下, QtCreator 就会帮我们把函数签名显示在右上方了, 还是挺方便的一个功能!

Qt 提供了五个类似的接口, 用于显示类似的窗口。具体代码这里就不做介绍, 只是来看一下样子吧!

```
QMessageBox::critical(NULL, "critical", "Content",
QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);
```



```
QMessageBox::warning(NULL, "warning", "Content",
QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);
```



```
QMessageBox::question(NULL, "question", "Content",
QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);
```



```
QMessageBox::about(NULL, "About", "About this application");
```



请注意, 最后一个 about() 函数是没有后两个关于 button 设置的按钮的!

QMessageBox 对话框的文本信息时可以支持 HTML 标签的。例如：

```
QMessageBox::about(NULL, "About", "About this <font  
color='red'>application</font>");
```

运行效果如下：



如果我们想自定义图片的话，也是很简单的。这时候就不能使用这几个 static 的函数了，而是要我们自己定义一个 QMessageBox 来使用：

```
QMessageBox message(QMessageBox::NoIcon, "Title", "Content  
with icon.");  
message.setIconPixmap(QPixmap("icon.png"));  
message.show();
```

需要注意的是，同其他的程序类似，我们在程序中定义的相对路径都是要相对于运行时的 .exe 文件的地址的。比如我们写” icon.png ”，意思是是在 .exe 的当前目录下寻找一个” icon.png ”的文件。这个程序的运行效果如下：



还有一点要注意，我们使用的是 png 格式的图片。因为 Qt 内置的处理图片格式是 png，所以这不会引起很大的麻烦，如果你要使用 jpeg 格式的图片的话，Qt 是以插件的形式支持的。在开发时没有什么问题，不过如果要部署的话，需要注意这一点。

最后再来说一下怎么处理对话框的交互。我们使用 QMessageBox 类的时候有两种方式，一是使用 static 函数，另外是使用构造函数。

首先来说一下 static 函数的方式。注意，static 函数都是要返回一个 StandardButton，我们就可以通过判断这个返回值来对用户的操作做出相应。

```
QMessageBox::StandardButton rb = QMessageBox::question(NULL,
"Show Qt", "Do you want to show Qt dialog?", QMessageBox::Yes
| QMessageBox::No, QMessageBox::Yes);
if (rb == QMessageBox::Yes)
{
    QMessageBox::aboutQt(NULL, "About Qt");
}
```

如果要使用构造函数的方式，那么我们就要自己运行判断一下啦：

```
QMessageBox message(QMessageBox::NoIcon, "Show Qt", "Do you
want to show Qt dialog?", QMessageBox::Yes | QMessageBox::No,
NULL);
if (message.exec() == QMessageBox::Yes)
{
    QMessageBox::aboutQt(NULL, "About Qt");
}
```

其实道理上也是差不多的。

Qt 学习之路(18): Qt 标准对话框之 QInputDialog

这是 Qt 标准对话框的最后一部分。正如同其名字显示的一样，QInputDialog 用于接收用户的输入。QInputDialog 提供了一些简单的 static 函数，用于快速的建立一个对话框，正像 QColorDialog 提供了 getColor 函数一样。

首先来看看 getText 函数：

```
bool isOK;
QString text = QInputDialog::getText(NULL, "Input Dialog",
                                      "Please input
your comment",
QLineEdit::Normal,
                                      "your comment",
&isOK);
if (isOK) {
    QMessageBox::information(NULL, "Information",
```

```

        "Your comment is:  

<strong>" + text + "</strong>",  

QMMessageBox::Yes |  

QMMessageBox::No,  

QMMessageBox::Yes);  

}

```

代码比较简单，使用 getText 函数就可以弹出一个可供用户输入的对话框：



下面来看一下这个函数的签名：

```

static QString QInputDialog::getText ( QWidget * parent,
                                      const QString
& title,
                                      const QString
& label,
                                      QLineEdit::EchoMode mode = QLineEdit::Normal,
                                      const QString
& text = QString(),
                                      bool * ok = 0,
                                      Qt::WindowFlags flags = 0 )

```

第一个参数 parent，也就是那个熟悉的父组件的指针；第二个参数 title 就是对话框的标题；第三个参数 label 是在输入框上面的提示语句；第四个参数 mode 用于指明这个 QLineEdit 的输入模式，取值范围是 QLineEdit::EchoMode，默认是 Normal，也就是正常显示，你也可以声明为 password，这样就是密码的输入显示了，具体请查阅 API；第五个参数 text 是 QLineEdit 的默认字符串；第六个参数 ok 是可选的，如果非 NULL，则当用户按下对话框的 OK 按钮时，这个 bool 变量会被置为 true，可以由这个去判断用户是按下的 OK 还是 Cancel，从而获知这个 text 是不是有意义；第七个参数 flags 用于指定对话框的样式。

虽然参数很多，但是每个参数的含义都比较明显，大家只要参照 API 就可以知道了。

函数的返回值是 `QString`, 也就是用户在 `QLineEdit` 里面输入的内容。至于这个内容有没有意义, 那就要看那个 `ok` 参数是不是 `true` 了。

`QInputDialog` 不仅提供了获取字符串的函数, 还有 `getInteger`, `getDouble`, `getItem` 三个类似的函数, 这里就不一一介绍。

Part 2: 进阶学习

Qt4 学习笔记

 Qt 可以运行在不同的平台, 像是 Unix/X11、Windows、Mac OS 与支援 framebuffer 的嵌入式 Linux 平台 (Embedded Linux Platform), 所使用的版本为 **Qt OpenSource 4.3.3** 版, 在 Vista 作业系统下撰写。

入门

先来尝试一下几个简单的 Qt 程式, 并初步了解一些核心特性。

- Hello!World! Orz..

第一步, 当然是先克服 Qt 的安装问题, 然后来个简单的 Hello!World! Orz...

- Windows XP/Vista 下安装 Qt4
- 第一个 Qt 程式
- 简单的显示中文 (使用 Unicode 转换)
- Signal 与 Slot

Signal 与 Slot 是 Qt 的特性, 让物件之间可以同步的 (Synchronous) 通知讯息, 但又不必知道彼此, Signal 是由物件发出。

- 使用 Signal 与 Slot (使用按钮关闭视窗)
- 使用 Signal 与 Slot (使用拉杆改变 LCD 数字)
- 自订 Signal 与 Slot
- 事件处理

事件基本上是非同步的 (Asynchronous), 通常由视窗或系统发出, 让应用程序可以回应使用者动作或系统讯息。

- 事件类型与处理器
- 事件接受与否、`event()`方法
- 事件过滤器
- 自订与传送事件
- 基本版面配置

Qt 的三种基本版面配置类型为水平、垂直及格状 (Grid) 配置。

- QHBoxLayout 与 QVBoxLayout 版面配置
- QGridLayout 版面配置
- 较复杂的版面配置
- 自订版面配置管理员 (Layout Manager)
- 其它

Qt 在发行时，本身即带有丰富的参考文件与范例，为入门时的必看资料。

- Qt 参考文件与范例
- 简介 Qt Designer

常用图型元件

元件讲是讲不完的，这边主要在简介几个元件作用，想了解每个元件详细使用方式，[Qt 参考文件与范例](#)才是王道。

- 按钮选项

按钮与选项是视窗程式中最基本的元件。

- QPushButton
- QCheckBox 与 QRadioButton
- QComboBox
- 对话方块

对话方块用来与使用者作简单的讯息交换与沟通。

- QInputDialog 与 QMessageBox
- QColorDialog 与 QFontDialog
- QFileDialog
- 自订对话方块 (Dialog)
- 文字栏位

文字栏位是使用者输入资讯的基本元件。

- QLineEdit
- QTextEdit
- 清单元件

这类元件通常用于显示项目清单，可以简单的列示，或者是使用树状或表格等。

- QListWidget 与 QListWidgetItem
- QTreeWidget 与 QTreeWidgetItem

- [QTableWidget 与 QTableWidgetItem](#)
- [Model 与 View 类别](#)
- 版面元件

除了上面介绍的三种基本版面配置之外，还可以使用一些版面元件来协助元件的群组与版面的切割。

- [QTabWidget](#)
- [QSplitter](#)
- [QStackedLayout](#)
- [QScrollArea](#)
- 其它元件

繁不不及备载...Orz...

- [QScrollBar](#)
- [QTimer 与 QLCDNumber](#)
- [QProgressBar](#)
- [QWizard](#)
- [QMainWindow](#)
- [QMdiArea](#)
- [QSplashScreen](#)

常用 API

Qt 已经不只是个 GUI 框架，它提供丰富的 API，可作为撰写应用程式的基础。

- [QString、容器元件](#)

[QString](#) 是常用的类别之一，拥有一些与容器类似的特性，而 Qt 提供了一系列的通用容器类别（Container class），使用上更轻量级且安全。

- [QString](#)
- [循序容器（ QVector、QLinkedList、 QList...）](#)
- [关联容器（ QMap、QHash...）](#)
- [泛型演算（ Generic Algorithms ）](#)
- 档案处理

档案输入输出是一个应用程式所必备的，藉由档案的输入输出，也可以一同了解 Qt 的 I/O 处理。

- [QFile](#)
- [QTextStream](#)
- [QDataStream](#)

- [QFileInfo 与 QDir](#)
- [Qt 资源系统](#)
- 资料库

QtSql 模组提供与平台、资料库无关的资料库存取需求，在这边使用 MySQL 作为示范。

- [Qt 的 MySQL 驱动程式](#)
- [QSqlQuery](#)
- [QSqlQueryModel 与 QSqlTableModel](#)

- 绘图

Qt 绘图基于 QPainter，可于 QPaintDevice 的子类别上，进行几何图案、图像、文字等绘制。

- [QPainter](#)
- [QMatrix](#)
- [QPixmap、 QBitmap、 QImage 与 QPicture](#)
- [QPrinter](#)
- 拖放（Drag & Drop）与剪贴

拖放动作对于使用者是很直觉的操作，而剪贴簿可以方便使用者于不同的应用程式间分享资料。

- [拖放事件](#)
- [拖放的执行与接受](#)
- [剪贴簿（QClipboard）](#)
- 网路

Qt 的网路模组提供网路存取时的高阶与低阶 API，基于非同步（Asynchronous）及 Signal 的行为。

- [QHttp](#)
- [QFtp](#)
- [QTcpSocket](#)
- [QTcpServer](#)

进阶议题

进阶议题就是比较进阶的议题。 。 XD

- 多执行绪（Multithreading）

想要实作多执行绪功能，只要继承 QThread 类别，单就表面上来看，执行绪并

不困难。

- [QThread](#)
- [执行绪的停止](#)
- [QMutex 与 QMutexLocker](#)
- [QWaitCondition](#)
- [QReadWriteLock 与 QSemaphore](#)
- [QThreadStorage](#)
- 国际化 (Internationalization)

让您的应用程式可以因地制宜，显示不同的语系文字。

- [使用 Unicode](#)
- [翻译应用程式](#)
- [多国语系选择与切换](#)

Qt 开发实例

Code Project: 创建一个媒体播放器

无论从功能还是大小来讲，Amarok 都是一款优秀的 KDE 音乐播放器。但它很难称作是一款快速点选式的音乐播放器，因为它要通过好几次点击和一些仔细的 GUI 导航才能听到音乐收藏中的音乐，这将给我们的 CPU 和大脑带来一定负担。这里我们将会构建所能想到的最简单和最直观的音乐播放器，给用户提供另一个选择。

从苹果的新款 iPod Shuffle 吸取一些灵感，只提供最基本的控制功能。一个按钮用于选择音乐，另一个按钮用于播放和暂停，还有一个按钮用于跳到下一段音乐。对于大多数用户而言，这些控制功能已经足够。与像 Amarok 这样臃肿而琐碎的播放器相比，它简直令人耳目一新。

编译环境

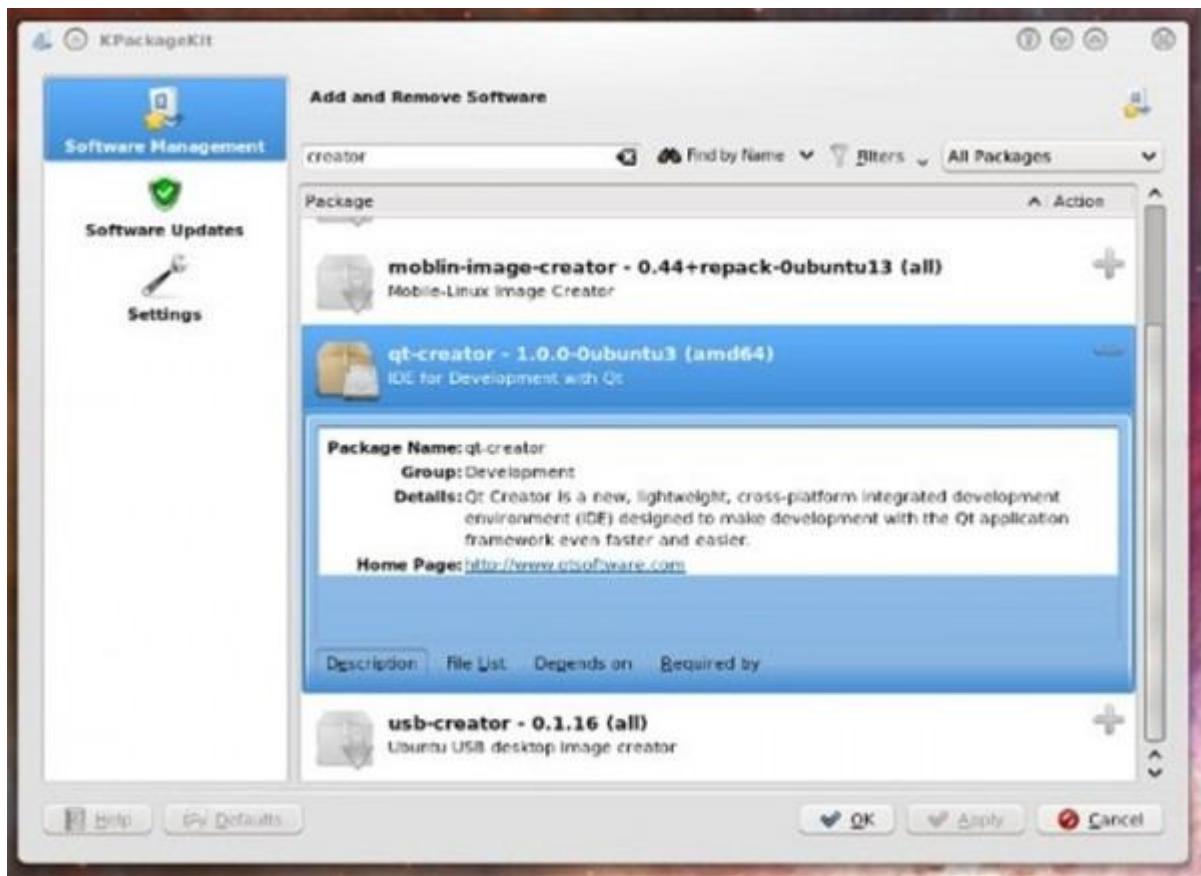
在进行任何编程工作之前，必须满足大家的播放器的几个要求。第一个要求是 Qt 4.5，需要安装它和与它相关的开发库。开发库包含了导入到我们自己项目中的头文件，这样我们就能在自己的应用程序之外使用 Qt 功能。

如果你的版本足够新，比如 Ubuntu Jaunty，那么会发现 Qt 4.5 已经包含在内了。如果版本较旧，必须自己添加它。Nokia 提供了易于安装的包，而且这些包可以与当前版本的 Qt 共存，或者也可以更新版本。

新手还需要一个工作开发环境。测试是否安装了工作开发环境的方法是，在命令行中输入“make”。如果提示无法找到该命令，则需要通过所使用版本的包管理器来安装 GNU 系列的编译工具。这包括一些基本的编程工具，比如 GCC 编译器套件、“make”编译系统和 GDB 调试器。大多数其他的版本都应该包含此编译环境的默认安装，如果没有也应该能够找到一个名称相似的元包。

Qt 4.5 包括 Creator IDE，但 Kubuntu 用户需要把它当作一个单独的包进行安装。因此在 Linux 和大多数其他操作系统上，当程序员试着猜测所使用的硬件和软件层时，音频播放是一个问题多发区。幸运的是，Qt 给出了我们所见过的最好的解决方案之一，即集成来自 KDE 的 Phonon 框架。Phonon 位于我们所使用的任何音频驱动器和路由层之上，并在编程和猜测配置复杂部分之间提供了一个接口。它让播放音频文件变得相对容易，而且是完全跨平台的。

你需要安装 Phonon 及其开发库。连同其他几个依赖项一起，它很可能还需要 GStreamer，这是大多数 Linux 版本的后端选择。Phonon 将直接与 GStreamer 对话，而我们只要与 Phonon 对话即可。



* Qt Creator 通常是标准 Qt 4.5 安装的一部分，但 Ubuntu 用户将需要手动抓取这个包。

起步

现在，如你已经安装了所有软件，接下来就可以正式开始了。运行 Creator，方法是使用启动菜单或者在命令行上输入“qtcreator”。此应用程序一开始会显示一个帮助性质的向导画面，在上面可以加载以前的项目或了解关于 Qt 的更多信息。要开始一个新项目，点击菜单 File>New。

这将运行新建项目向导，而我们的首个任务就是选择要创建项目的种类。在“Projects”标题下找到“Qt4 GUI Application”，然后点击 OK。现在，需要为项目输入一个名称，并选择创建项目目录的文件夹。

下一个窗口是模块选择页面，在这里可以选择要包含在项目内的其他组件。每个 Qt 项目都会使用的两个核心模块 QtCore 和 QtGui 已经被选中，只要选择要使用的任意非核心组件即可。例如，如果要开发浏览器，就需要包含 WebKit 模块。因为我们只需要与操作系统的音频层进行交互，所以只需要选择 Phonon。

选择 Phonon 后，最后一个页面将列出我们的项目中文件和类的默认名称，只要保持它们的默认值不变即可。点击“Finish”结束向导，创建文件夹和文件模板，所有这些都将自动被加载到 Creator 中。

如果你之前使用过任何形式的 IDE，就不会对 Qt Creator 感到陌生。它有好几种不同的操作模式。一开始要花最多时间熟悉的是 GUI 设计器，点击以“ui”结尾的文件（User Interface 设计器文件）时会自动运行它。这些文件之一已经添加到我们的项目中，在主显示区域左边的项目窗格中可以看到这个文件和四个其他文件。

其中两个文件的名称以“.cpp”结尾，用于保存我们的项目的源代码，但“main.cpp”实际上只能用于运行应用程序，因此我们能进行编辑的只有“mainwindow.cpp”文件。这个文件有一个对应的头文件，用于描述我们在mainwindow.cpp中使用的对象和开发文件。最后还有一个“.pro”文件，实际上这个文件是禁止可见的，因为它是 Qt 和 Creator 用于编译项目的项目文件。



* 如果你忘记在 Creator 的启动向导中包含所需模块, 可以自己把它们添加到项目的“.pro”文件中。

GUI 设计

在项目窗格中双击“ui”文件, 嵌入的设计器视图就会出现。如果你过去曾经尝试过一些 Qt 编程, 就会察觉布局与过去的 Qt Designer 应用程序有所不同, 它已经更加无缝地嵌入到了 Creator 中。没有人曾经说过 Designer 使用方便, 但它功能强大, 而且在创建动态和易扩展的 Qt GUI 应用程序方面比使用纯编程方法更加简便。

由于 Creator 中包含了 Designer, 现在我们可以在 GUI 修改和编程之间轻松来回切换, 这可以显著提高开发速度。

这个应用程序只有一个十分简单的 GUI。向下滚动可用小部件的列表, 从 Buttons 区将三个“Push Buttons”拖动到灰色的应用程序画布上。或者, 也可以先拖一个按钮, 然后再复制/粘贴两个。这些按钮是应用程序所拥有的全部控件。一个

按钮将用于选择要播放的文件，另一个按钮将用于播放和暂停音乐，而最后一个按钮将用于跳到下一首音乐。这与 Amarok 的 20 多个按钮形成了鲜明对比。

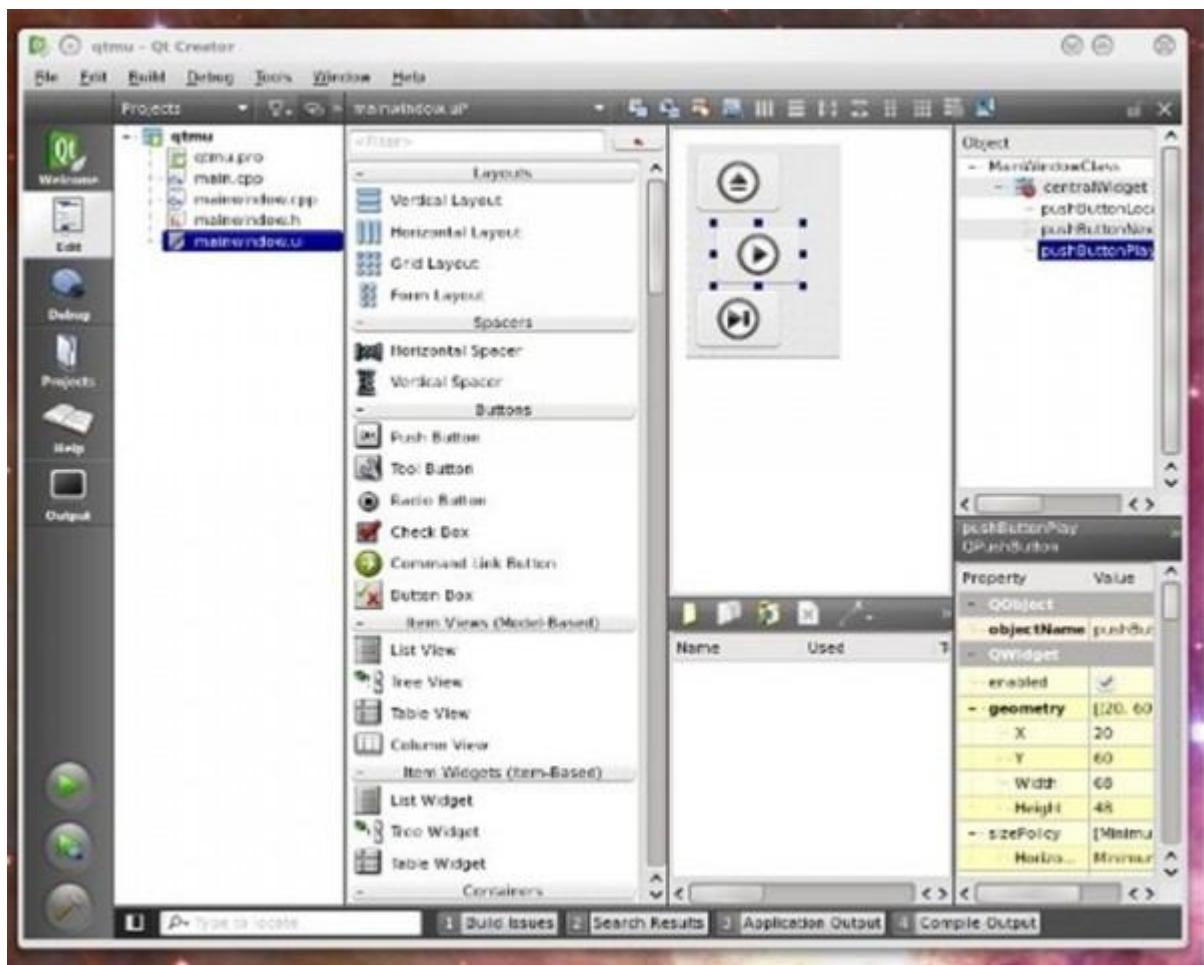
现在，来注释用户每个按钮的功能。为此，可以双击每个按钮并使用一个词来描述其功能，或者在按钮所在位置放置一个图标。如果要使用图标，选中按钮，然后在 Creator 画面的左下区中找到值面板。此面板中保存了 GUI 中当前选中小部件的大多数属性。向下滚动，直到在列表中看到“icon”为止。

选择该项，然后点击右侧的“...”按钮，以打开一个文件请求器。现在可以为按钮选择并使用任意图像文件，但如果安装了标准的 KDE，可以看一看

“/usr/share/icons/oxygen/32×32/actions”目录。此目录包含大多数常用功能的图标，“media-skip-forward”、“media-skip-backward”和“media-playback-start”特别适合我们要实现的功能。

现在，我们需要垂直排列这三个按钮，同时限制窗口的大小。按下左 Ctrl 键，然后分别选中每个按钮。它们现在应该都有突出显示的缩放边界。现在点击工具栏中的“Lay Out Vertically”按钮（或者右键单击菜单）。这将把三个按钮绑定在一起，并由上至下依次排列。

现在点击应用程序背景，并选择“Layout in a Grid”。这将把三个按钮拉伸至充满所有可用空白，无论窗口大小如何。但我们还想让窗口变小。拖动背景左下方的锚点，直到创建出一个小矩形，或者使用属性面板的“Geometry”区手动输入大小。最后我们获得了一个 115 像素宽和 162 像素高的窗口。也可以选择固定窗口的大小，方法是将 Horizontal 和 Vertical 参数的最大值设为和最小尺寸相同的数字。



* 使用 **Creator** 构建 GUI 需要一个熟悉的过程，但是基本上只要从左侧拖动组件，并将它们固定在布局引擎的合适位置上即可。

编写

Creator 中的 Designer 视图还有一个好处，即不用编写一行代码便可创建大量编程逻辑。这要感谢 Qt 的 Signals and Slots 机制。它们基本上是一种远程过程调用，但由 Qt 自动处理。在 web 浏览器中，例如当用户点击刷新按钮时，浏览器接口可能“发出”一个信号以刷新显示器。

在这个应用程序中，我们要发出三种信号，分别对应于我们的三个按钮——播放/暂停、快进和添加文件。我们需要编写函数来处理每种信号，但在 Designer 中，我以把应用程序配置为当特定操作发生时调用特定函数。为此，需要切换到“Signals/Slots”编辑模式，方法是使用工具栏中的图标或者按下 F4 键。

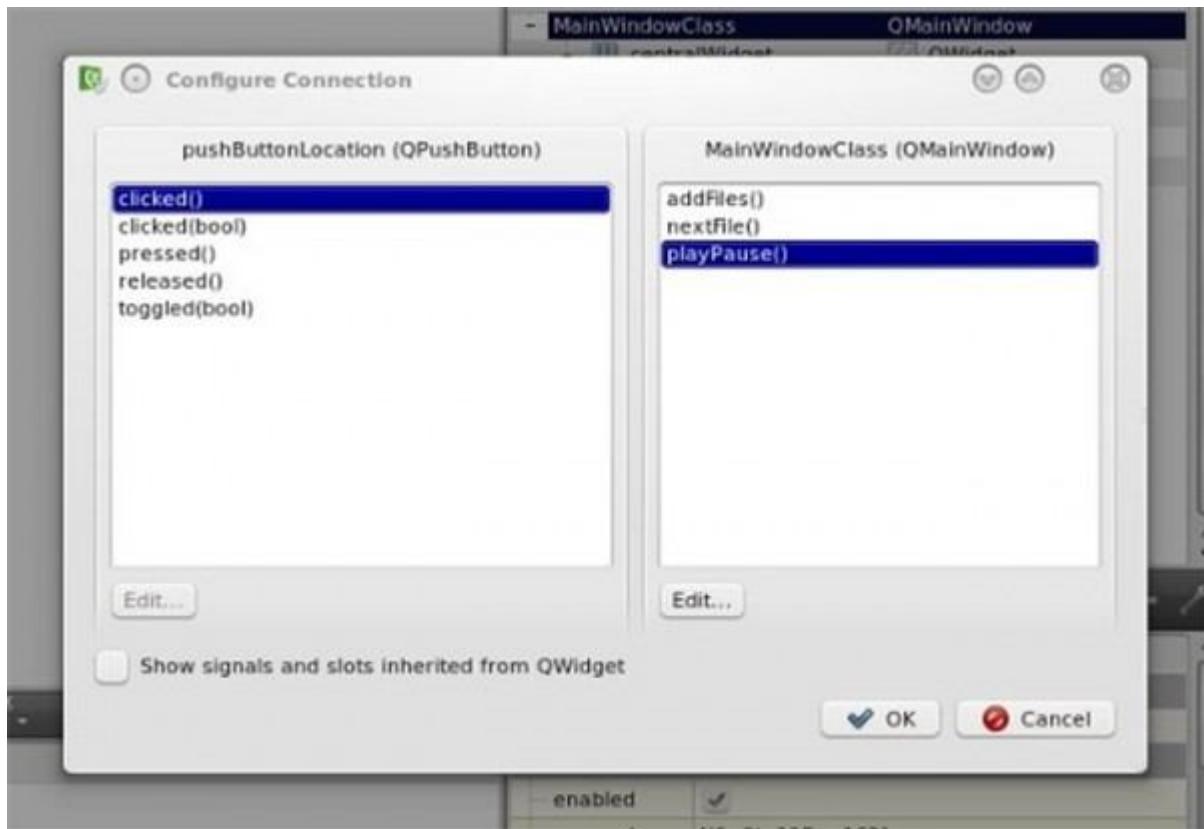
这种模式的原理是，首先选择要发出信号的小部件，然后将光标拖动到其槽将对信号做出反应的小部件。例如，通过 web 浏览器可以在刷新按钮和 WebKit 浏览

器小部件之间建立连接。然后就可以刷新显示器，同时无需编写一行代码。如果没有特定小部件接收信号，就像在我们的例子中一样，那么就将连接拖动到窗口背景上。我们需要为每个按钮做这项工作。

首次将连接放到背景上时，会出现一个“Configure Connection”窗口。在右侧面板上点击“Edit”按钮。这将打开另一个窗口，其中列出了作为应用程序背景的 MainWindowClass 的信号和槽。我们需要添加三个槽，以便接受三种不同的信号，而且这些槽将直接对应于我们将要在主应用程序中编写的函数。

按下“+”号，然后添加“playPause()”、“addFiles()”和“nextFile()”函数。点击 OK，使用 Edit Signals/Slots 模式，将来自每个按钮的“clicked()”信号连接到我们刚刚创建的正确槽。例如对于“Add Files”按钮，需将

“clicked()”信号连接到“addFiles()”函数。在编辑器窗口中可以看到所有连接，其中蓝色的直线和正方形用于指示来自某些小部件的信号与各个槽的对应关系。这是我们所需要进行的 GUI 编辑的最后一部分，保存工作进展，让我们开始下一个步骤。



* 了解信号和槽是 Qt 编程最重要的方面。

编码：mainwindow.h

我们已经为我们的应用程序创建了框架，现在只要添加功能即可。点击“mainwindow.h”，并在顶部添加如下代码行：

```
#include <QList>
#include <QFileDialog>
#include <QDesktopServices>
#include <Phonon>
```

以上代码的作用是，通过头文件导入我们要在代码中使用的 Qt 函数。现在我们需要添加我们的槽，它们在我们前面编辑过的 ui 文件中已经定义好了。在“public:”部分中的“~MainWindow();”行正下方，添加如下代码：

```
private slots:
    void playPause();
    void addFiles();
    void nextFile();
    void aboutToFinish();
    void finished();
```

可以看到，这些槽对应于我们在用户界面中创建的名称，而且我们还添加了更多的槽来处理内部通信。最后，在“private”部分中添加如下变量，我们将在应用程序的主要逻辑中用到这些变量：

```
QList<Phonon::MediaSource> sources;
Phonon::MediaObject *mediaObject;
Phonon::AudioOutput *audioOutput;
Phonon::MediaObject *metaInformationResolver;
```

编码：mainwindow.cpp

现在，我们需要做的第一件事情是初始化 Phonon，并建立内部的信号和槽。这可以通过标准的初始化方法 MainWindow::MainWindow 来完成。如果你看一看这个方法的内容，就会发现应用程序的 GUI 是由“ui->setupUi(this);”行来运行的。这意味着，我们需要在这之前加入我们的预运行代码。我们将从设置 Phonon 开始：

```
audioOutput = new
Phonon::AudioOutput(Phonon::MusicCategory, this);
mediaObject = new Phonon::MediaObject(this);
```

```
metaInformationResolver = new
Phonon::MediaObject(this);
Phonon::createPath(mediaObject, audioOutput);
```

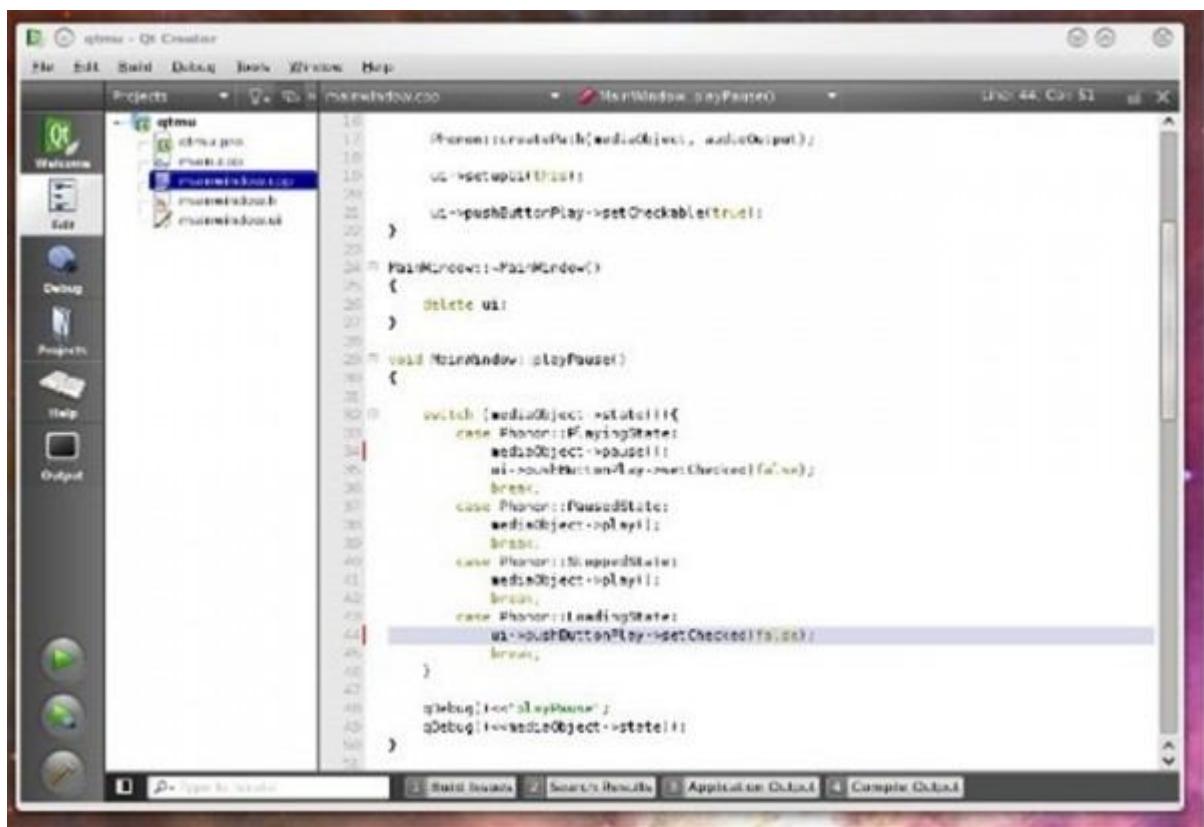
在 Phonon 术语中，我们要创建的 audioOutput 对象叫做音频接收槽。它是直接与音频驱动器通信的层的组成部分，并充当 MediaObject 的虚拟音频设备。MediaObject 位于这一层的上层，增加了诸如暂停、播放和倒带之类的功能。顺便提一句，MusicCategory 不一定是必需的，但它可以对未来发展起到作用，比如可以根据正在收听的内容自动变化的 KDE 均衡器。

我们将使用“metaInformationResolver”来指向当前音频文件，而最后一行在接收槽和媒体对象之间建立了连接。Phonon 使用了一种叫做“graph”的框架，这意味着对象就像是一幅图上的节点，需要连接起来才能创建流向。这正是以上代码的最后一行的作用。

现在添加如下用于处理 playPause() 槽的新函数：

```
void MainWindow::playPause()
{
    switch (mediaObject->state()) {
        case Phonon::PlayingState:
            mediaObject->pause();
            ui->pushButtonPlay->setChecked(false);
            break;
        case Phonon::PausedState:
            mediaObject->play();
            break;
        case Phonon::StoppedState:
            mediaObject->play();
            break;
        case Phonon::LoadingState:
            ui->pushButtonPlay->setChecked(false);
            break;
    }
}
```

以上代码应该不难理解。我们使用一条 Case 语句检查播放的当前状态。Phonon 通过 MediaObject 为我们提供了这种功能。它实际上就是一个整数，但是每个值代表着特定的状态。例如，如果应用程序是首次启动，播放列表中尚未加入任何文件，就会返回 LoadingState 值。播放和暂停由媒体对象进行处理，在“mediaObject->pause();”命令之后将自动从当前位置继续。



* 在 Creator 中输入代码时，可使用自动完成功能找到所需函数。

我们还尝试给用户提供一些关于当前播放状态的反馈。如果音乐正在播放，我们会让“Play”按钮凹下去，使其看起来好像是被按下一样。如果音乐处于任意其他状态，我们会让“Play”按钮恢复原样，使其看起来处于停止状态。为了实现这一点，需要将播放按钮取名为“pushButtonPlay”。可以在 Designer 视图中修改它的名称，方法是选择该按钮并修改“objectName”值。

```
void MainWindow::addFiles()
{
    QStringList files = QFileDialog::getOpenFileNames(this,
tr("Select Music Files"),

QDesktopServices::storageLocation(QDesktopServices::Music
Location));

    ui->pushButtonPlay->setChecked(false);
    if (files.isEmpty())
        return;
    int index = sources.size();
    foreach (QString string, files) {
        Phonon::MediaSource source(string);
```

```

        sources.append(source);
    }
    if (!sources.isEmpty()) {

metaInformationResolver->setCurrentSource(sources.at(index));
mediaObject->setCurrentSource(metaInformationResolver->currentSource());
}

}

```

现在，让我们开始实现添加文件的部分。创建文件请求器几乎是自动的，我们可以把结果放到一个字符串列表中。“QdesktopServices::MusicLocation”返回一个通常用于保存音乐文件的特定操作系统位置，而我们使用它作为请求器的起始位置。接下来，我们将把已经选择的每个音乐文件添加到我们早先创建的“metaInformationResolver”中，并使用它告诉 mediaObject 接下来播放哪个文件。

在所有这些工作间隙，我们进行一点清理工作，以确保队列中有文件，而且当处于播放状态时出现播放按钮。添加文件将自动停止播放。

```

void MainWindow::nextFile()
{
    int index = sources.indexOf(mediaObject->currentSource())
+ 1;

    if (sources.size() > index) {
        mediaObject->stop();
        mediaObject->setCurrentSource(sources.at(index));
        mediaObject->play();
    }
}

void MainWindow::aboutToFinish()
{
    int index = sources.indexOf(mediaObject->currentSource())
+ 1;

    if (sources.size() > index) {
        mediaObject->enqueue(sources.at(index));
    } else {

```

```
    ui->pushButtonPlay->setChecked(false);  
}  
  
}  
  
void MainWindow::finished()  
{  
  
    ui->pushButtonPlay->setChecked(false);  
  
}
```

[下载此项目的代码](#)



它看起来可能不起眼，但我们的音乐播放应用程序具备了一款音乐播放器所需要的大部分功能。

Code Project: 创建一个 FFMPEG 前端

命令行没有什么不好。对于我们很多人来说，这是使用 Linux 的最佳理由之一。可以通过输入内容实现几乎所有功能，而且命令行工具对于它们的运行方式通常能够提供极好的控制。但是命令行并不适合所有人，觉得命令行难以理解和令人生畏的 Linux 用户数量多得令人吃惊，这或许是完全避免使用 Linux 的原因之一。

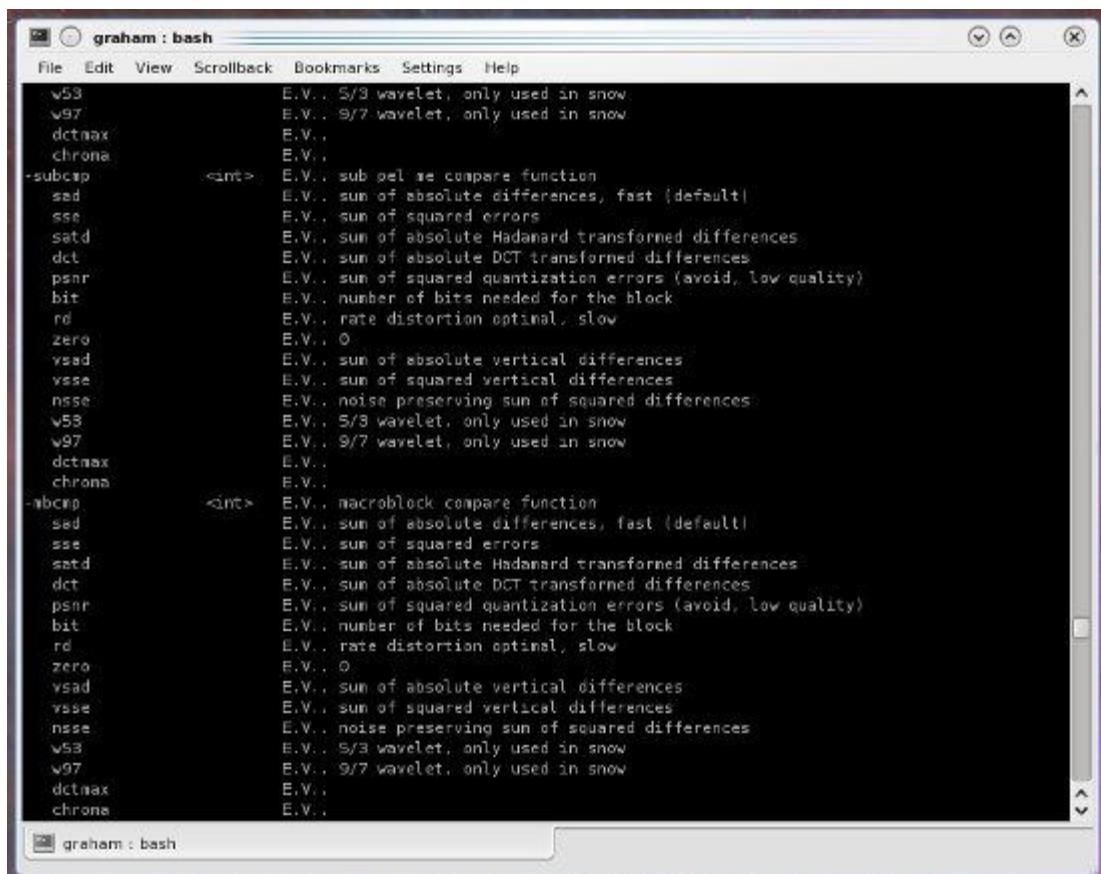
尽管如今不愿意使用命令行的用户可以不必再使用它，但这仍然意味着他们将遗漏一些很优秀的实用工具。

Qt 正好可以扭转这种局面。它是为你最喜爱的命令行工具创建友好 GUI 的理想工具。它不需要任何顶级的编程技巧，而且工作量也不大，但在此过程中你可以为讨厌命令行的朋友提供帮助，并且对开源应用程序开发做出自己的贡献。为命令行工具创建 GUI 是最佳起点之一！

如果你已经完成了我们以前的 Qt Creator 代码项目，[如何创建自己的媒体播放器](#)，就能够更加自如地完成这个任务……

如果说有一种工具非常需要 GUI，那就是 FFMPEG。FFMPEG 是一个十分优秀的命令行应用程序，它可以将视频和电影文件从一种格式转换为另一种格式。同时它的复杂程度也令人吃惊。有好几位开发人员都曾经试过为这个工具创建一个 GUI，但是 FFMPEG 的发展速度让人很难驾驭所有额外的组件和用于各个选项的不断改变的语法。

我们将会构建我们自己的 FFMPEG GUI，但本指南的意义并不在于帮助人们了解 FFMPEG 的非凡之处。本指南将展示为几乎任意命令行工具创建 GUI 是多么轻松的事情——只要替换几行代码即可，而且结果可能对我们大有好处。



```
graham : bash
File Edit View Scrollback Bookmarks Settings Help
v53          E.V., 5/3 wavelet, only used in snow
v97          E.V., 9/7 wavelet, only used in snow
dctmax       E.V.,
chroma      E.V.,
-subpel     <int>   E.V., sub pel se compare function
sad          E.V., sum of absolute differences, fast (default)
sse          E.V., sum of squared errors
satd         E.V., sum of absolute Hadamard transformed differences
dct          E.V., sum of absolute DCT transformed differences
psnr         E.V., sum of squared quantization errors (avoid, low quality)
bit          E.V., number of bits needed for the block
rd           E.V., rate distortion optimal, slow
zero         E.V., 0
vsad         E.V., sum of absolute vertical differences
vsse         E.V., sum of squared vertical differences
nsse         E.V., noise preserving sum of squared differences
v53          E.V., 5/3 wavelet, only used in snow
v97          E.V., 9/7 wavelet, only used in snow
dctmax       E.V.,
chroma      E.V.,
-mbcmp      <int>   E.V., macroblock compare function
sad          E.V., sum of absolute differences, fast (default)
sse          E.V., sum of squared errors
satd         E.V., sum of absolute Hadamard transformed differences
dct          E.V., sum of absolute DCT transformed differences
psnr         E.V., sum of squared quantization errors (avoid, low quality)
bit          E.V., number of bits needed for the block
rd           E.V., rate distortion optimal, slow
zero         E.V., 0
vsad         E.V., sum of absolute vertical differences
vsse         E.V., sum of squared vertical differences
nsse         E.V., noise preserving sum of squared differences
v53          E.V., 5/3 wavelet, only used in snow
v97          E.V., 9/7 wavelet, only used in snow
dctmax       E.V.,
chroma      E.V.,
```

* 应该只有意志坚定和不怕阅读符号文字的人才会输入“ffmpeg -h”

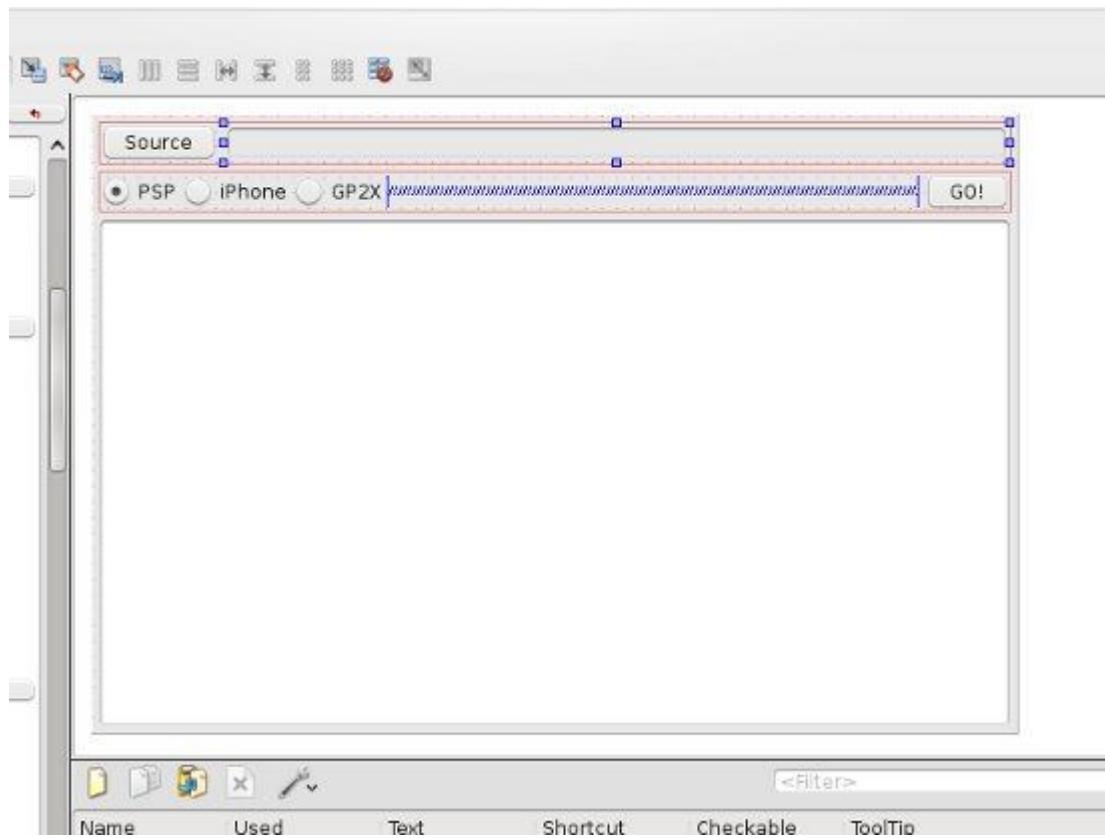
应用程序设计

我们将假定已经启动并正在运行 Creator，而且对于使用开发环境相对比较满意。如果不满足上述条件，请复习我们以前的指南。运行应用程序之后，创建一个新的项目并使用“Qt Gui Application”模板，同时保留其他选项的默认值。

和以前的指南一样，我们的开发工作仍然从 GUI 开始。点击“.ui”文件，Designer 视图随之出现。如果想从空白画布开始，可在视图中删除菜单栏、工具栏和状态栏，方法是在窗口右上方的对象视图中选中它们，右键单击每个组件，然后从出现的菜单中选择删除。

我们应用程序的 GUI 布局显然依赖于要运行的命令行工具。但对于我们的 FFMPEG 例子，我们要尽可能尝试保持内容的开放性和可修改性。我们需要一个按钮来添加要转换的源文件，以及一个用于显示此文件位置的文本字段。我们需要一些途径来选择转换过程的最终格式，并显示 FFMPEG 命令的结果。我们还需要另一个按钮，点击它便可启动整个转换过程。

但是用户界面的主要部分将会被我们选出让用户编辑的 FFMPEG 选项所占据。我们没有理由向普通用户公开 FFMPEG 中数以百计的选项，因此这里只会出现最常用和最简单的使用选项。



* 我们的目标是构建出的布局能够最好地利用空间。这也正是我们选择在另一个页面上隐藏命令行输出的原因。

GUI 的构建

从左边的 Button 小部件列表中拖两个按钮到空白画布上，然后双击每个按钮，修改它们的名称。一个按钮的名称定为“Source”，而另一个用于触发转换过程的按钮则需要取类似于“Go!”的名称。另外还可以使用图标代替文本，或者二者同时出现亦无不可（如果这是一种改进的话）。现在从 Inputs 列表中添加一个 Line Edit 小部件，然后点击左下方面板中的“enabled”属性以禁用它。

由于 Line Edit 小部件是交互式的一一即默认情况下，用户可以在里面输入内容——我们想禁用这项功能，而只使用它来显示我们要转换文件的源位置。如果想让用户手动输入或修改文件的名称和位置，可以保留此字段的值为 enabled。

我们还为用户提供了一条选择输出格式的方便途径。我们选择使用一列单选按钮，每个单选按钮代表一种格式。从调色板中拖动 Radio Button 小部件即可添加这些按钮。这种方法的好处在于，每次只能选择一种格式，而这正是我们需要的行为模式。对于每个单选按钮，还需要将其标签的文本修改为能够反映适合 FFMPEG 输出的设备。例如，如果选择 PSP 作为输出设备，将禁用 iPhone 输出和 GP2X 输出。

为了最大程度利用可用空间，我们准备使用选项卡小部件来保存余下的参数。就像浏览器中的选项卡式 web 页面一样，允许用户在一快 GUI 区域的两种不同视图之间进行切换。我们将使用一个选项卡来保存要公开的主要参数，而另一个选项卡可用于保存来自 FFMPEG 的原始命令输出。这应该意味着，除非用户切换选项卡，否则不会看到乱七八糟的 FFMPEG 输出。

从 Creator 中的 Containers 列表把选项卡小部件拖到画布上，选中之后，点击属性列表中的“currentTabText”属性，以便修改每个选项卡上显示的文本。我们选择了“Options”和“Output”。

被拖到选项卡小部件中的任意小部件都只显示在当前选项卡上，因此我们给 Options 选项卡添加了 5 个组合框和 5 个标签。对于像 FFMEG 这样的工具而言，组合框是对用户最友好的选项，因为它们将可用选项限制为只出现有效的选项。它对于程序员意味着额外的工作，但这种额外工作可以为可怜的老用户节省很多时间。借助这 5 个组合框，我们选择支持如下配置：视频的分辨率、帧率和比特率，以及视频的采样率和压缩比特率。

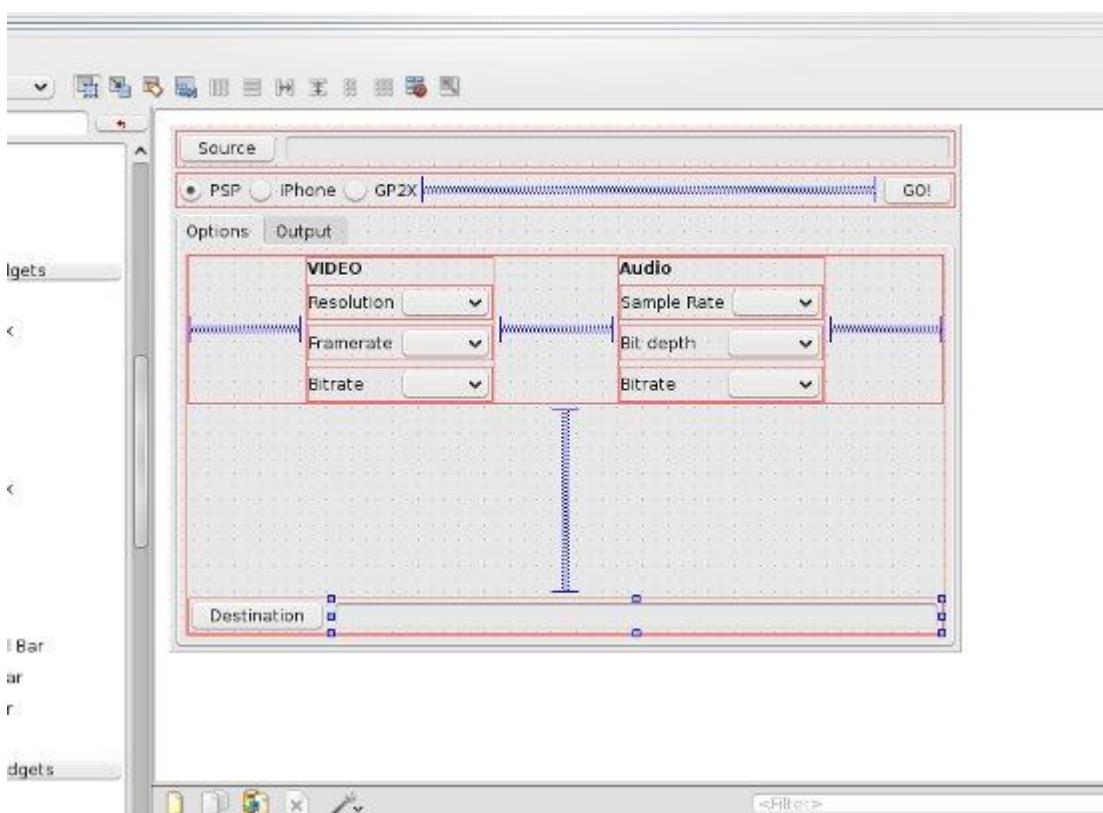
我们重新命名了每个组合框小部件，以便在源代码中能够更好地区分它们，同时在每个组合框旁边放置一个标签，并使用标签文本对每个选项进行了说明。在组合框下面，我们添加了更多小部件，用于保存目的文件的名称——一个按钮用于选择文件，以及一个禁用的线编辑字段用于显示位置。

最后，切换到选项卡小部件的下一个页面，并添加一个 TextBrowser 小部件。我们将使用这个小部件来显示 FFMPEG 命令的输出。

间距器

现在，我们的应用程序看起来应该相当复杂了。我们需要对各个小部件进行排列，让它们能够很好地缩放并且看起来间隔均匀。Qt 使用一个分组和间距器的系统来创建布局，但了解这个系统的使用原理需要花点功夫。它和 DTP 应用程序处理对象的方式不同。例如，要在选项页面中创建良好的布局，首先 shift 选择一个标签及其相关组合框，然后从工具栏选择“Lay Out Horizontally”。这将会把两个小部件锁定在并排的位置。

对其他小部件做同样的事情，然后 shift 选择用于修改视频设置的分组对，并选择“Lay Out Vertically”。这将把每对小部件都对齐到一列中，对音频设置也要做同样的事情。我们还为音频和视频列添加了标签，并在垂直布局集合中包含了它们。要想正确实现分组选择，过程更加麻烦，但如果犯错我们始终可以进行“Undo”和“Redo”操作。



最好把间距器看作弹簧，并重新复习一些旧的物理课本。

为了保证我们的 GUI 不会明显延伸为一个大窗口，我们需要使用“间距器”。它们看起来有点像弹簧，可以从小部件调色板添加垂直或水平的间距器。当在水平

或垂直布局中包含一个正确的间距器时，小部件将会与一块可扩展空间组合在一起，这块空间就是弹簧所在的位置。如果应用程序窗口扩展或收缩，这块空间也会根据弹簧长度成比例地伸缩。

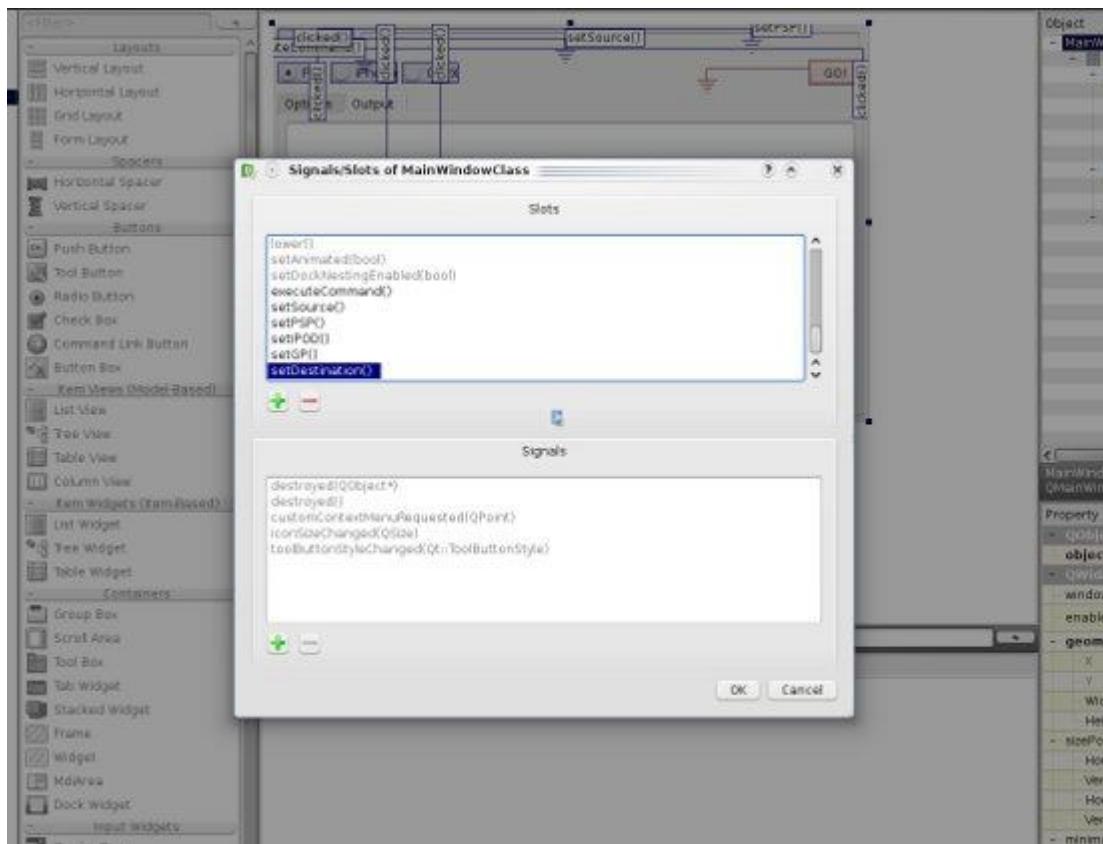
这用语言也许很难描绘，但只要动手实践便不难理解。最后的结果是布局系统十分灵活，但学习起来却比较困难。我们使用三个水平弹簧来分开视频和音频选项，以及它们与窗口边界的空间，同时使用一个垂直弹簧来将视频与音频编码选项和目的文件位置分开。间距器还可用于在单选按钮之间安插一点距离。

当我们大体上安排好所需要的一切小部件后，选择“Lay Out in a Grid”将所有小部件绑定到主窗口上。此时仍然能够拖动和添加小部件到布局上，但如果需要做较大的改动，则需要首先打破这种绑定，即选择“Break Layout”。

信号与槽

现在，可以把 GUI 设计与我们将要在源代码中添加的程序功能联系起来了。这要借助于我们前面提过的 Qt 的信号与槽机制。一开始，切换到 Signals/Slots 编辑模式 (F4)，并且从“Go”按钮拖出一个连接到窗口的背景画布上，用于有效地发送信号给 MainWindow 类。在出现的窗口中，点击左边的 Edit 按钮，从而打开 Signals/Slots 编辑窗口。

我们需要添加 6 个槽：executeCommand()、setSource()、setDestination()、setPSP()、setIPOD() 和 setGP()。点击 OK，选择新创建的“executeCommand()”槽，然后连接到我们当前正在编辑的 Go 按钮的 clicked 信号。需要对我们刚刚为其创建了槽的其他每个按钮做同样的事情，方法是把它们连接到它们对应的槽。例如，应该把来自 PSP 单选按钮的“clicked()”信号连接到“setPSP()”。



我们从 GUI 创建我们自己的槽，而且后面需要在源代码编辑器中为它们添加代码。

完成这项任务之后，Designer 编辑就结束了，保存项目并切换为编辑 `MainWindow.h`。我们需要在文件顶部添加几个头：

```
#include <QProcess>
#include <QByteArray>
#include <QTextBrowser>
#include <QFileDialog>
#include <QDesktopServices>
#include <QComboBox>
```

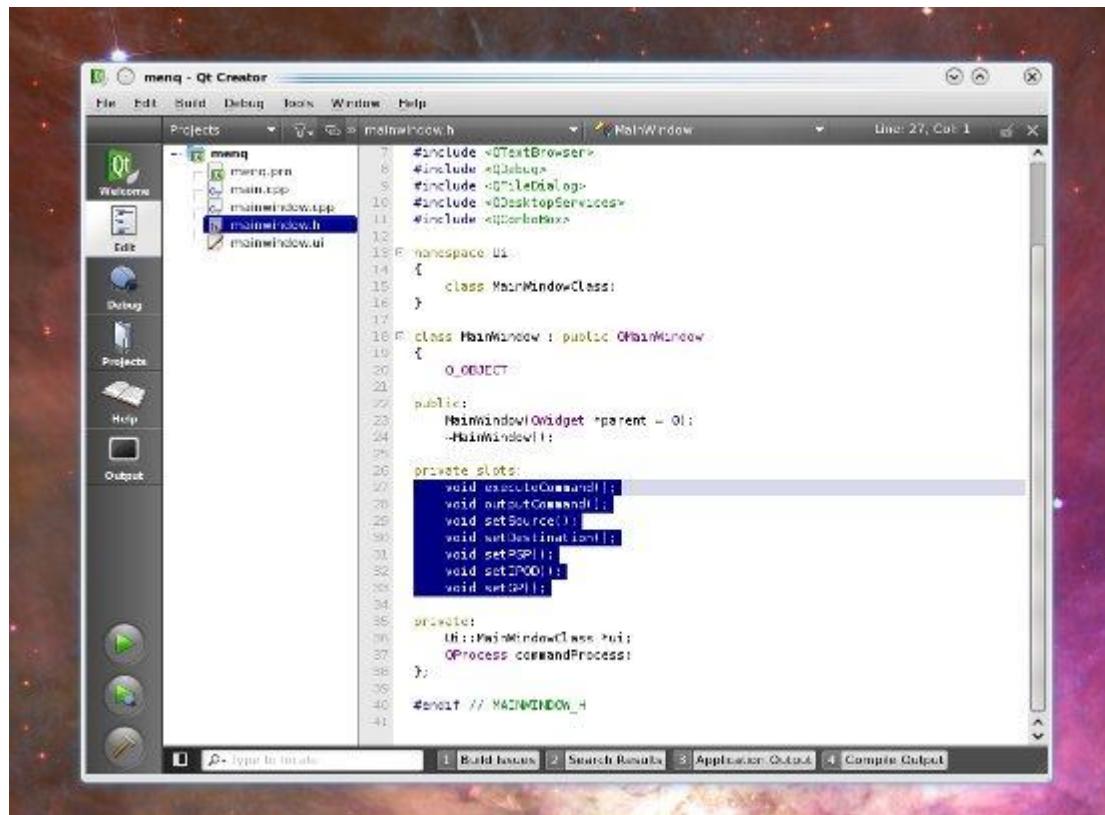
因为我们要对已经在 Designer 中添加给应用程序的小部件进行操作，所以这些头文件中的大部分都是必不可少的。`Qprocess` 和 `QbyteArray` 用于在 Qt 中运行外部的可执行文件，并抓取命令的输出，但我们很快就会讲到这个函数。现在，我们需要为要编写的槽添加定义。在“`public:`”部分下面添加如下代码：

```
private slots:
    void executeCommand();
    void outputCommand();
    void setSource();
    void setDestination();
```

```
void setPSP();
void setIPOD();
void setGP();
```

最后，我们需要在头文件的“private:”部分中添加一个变量。此变量将用于在 Qt 中处理外部可执行文件（FFMPEG）：

```
QProcess commandProcess;
```



* 我们从 GUI 创建我们自己的槽，而且后面需要在源代码编辑器中为它们添加代码。

编程

现在我们到了最富于技巧性的部分，即给应用程序添加功能。首先，我们要为三种要处理的不同输出格式编写“set”函数。这些函数都将使用与设备相关的参数填充组合框，而我们最终将采用这些参数来编译将创建正确输出的 FFMPEG 命令行。

当然，在能够给 GUI 添加值之前，首先针对每种格式要有一个有效的 FFMPEG 命令。例如，我们已经找到用于转换运行在 PSP 上的视频的最佳 FFMEPG 命令是：

```
ffmpeg -i space.mpg '-vcodec' 'libxvid' -s 320x240 -r 29.97  
-b 1500 -acodec libfaac  
-ac 2 -ar 24000 -ab 65535 -f psp M4V80113.mp4 -y
```

我们准备采用这些参数中的一部分，并使它们能够在我们的 GUI 中进行编辑。根据我们在其他项目中的经验，可以通过“ui”对象运行属于 GUI 中对象的方法，此对象默认是使用标准 Creator 模板创建的。例如，

“ui->comboResolution->clear()” 将在 comboResolution 组合框上执行清除工作。

Creator 集成环境的好处在于，可以使用自动完成功能列出每个对象的可用选项，而不用全靠记忆。下面是我们在 setPSP 函数中换到 GUI 中的 FFMPEG 选项。需要把它添加到 MainWindow.cpp 文件的底部：

```
void MainWindow::setPSP()  
{  
    ui->comboResolution->clear();  
    ui->comboFramerate->clear();  
    ui->comboBitrate->clear();  
    ui->comboSamplerate->clear();  
    ui->comboAbitrate->clear();  
    ui->comboResolution->addItem("240x320");  
    ui->comboResolution->addItem("160x120");  
    ui->comboFramerate->addItem("29.97");  
    ui->comboBitrate->addItem("1500");  
    ui->comboSamplerate->addItem("2400");  
    ui->comboAbitrate->addItem("65535");  
    ui->lineEdit_2->setText("M4V80113.mp4");  
  
}
```

这段代码的自解释程度相当高。为了节省空间，我们将会给其添加多个选项的唯一组合框是分辨率框，但可以很容易地看到如何添加其他选项。我们还需要为其他两种预设置创建相同模板，并把它们放在“setIPOD”和“setGP”函数槽中。由于我们只选择了一个参数集合让用户编辑，因此需要将这些参数与 FFMPEG 命令中的其他参数结合起来，而我们准备在处理运行外部 FFMPEG 命令的函数中做这件事情，这个函数叫做“executeCommand()”。

执行一个命令

```
void MainWindow::executeCommand()  
{  
    QStringList args;
```

```

    args << "-i";
    args << ui->lineEdit->text();
    args << "-y";
    args << "-s"; args << ui->comboResolution->currentText();
    args << "-r"; args << ui->comboFramerate->currentText();
    args << "-b"; args << ui->comboBitrate->currentText();
    args << "-ar"; args <<
ui->comboSamplerate->currentText();
    args << "-ab"; args << ui->comboBitrate->currentText();
    args << ui->lineEdit_2->text();

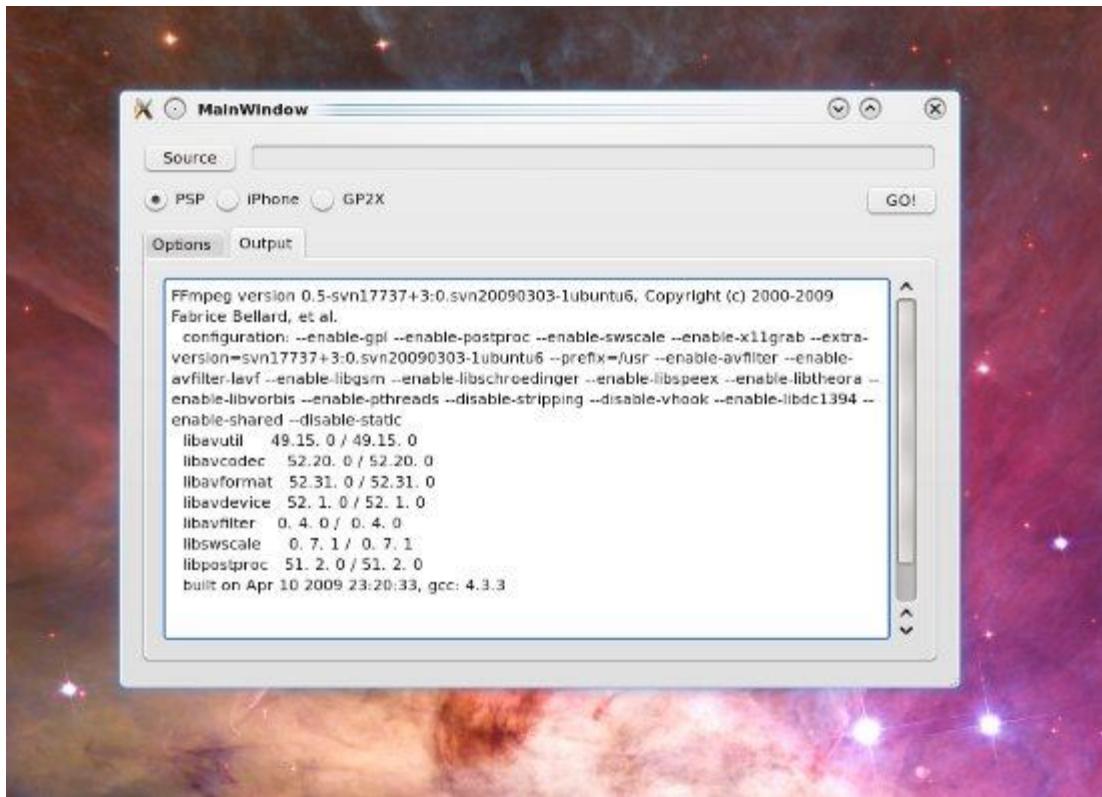
    if (ui->radioButton->isChecked()) {
        args << "-vcodec"; args << "libxvid";
        args << "-acodec"; args << "libfaac";
        args << "-ac"; args << "2";
        args << "-f"; args << "psp";
    }
    commandProcess.start("ffmpeg", args);
}

}

```

下面是对于以上代码块作用的解释。在 Qt 中运行外部命令的关键是一个叫做“Qprocess”的类。在步骤三结束时，我们在头文件中使用这个类创建了我们自己的对象，现在正是时候使用它来执行 FFMPEG。使用“commandProcess”变量时，我们只要使用两个变量运行“start”即可——命令本身和我们的参数列表。

我们使用“QStringList”来快速构造这个参数列表，首先从我们 GUI 中的小部件，然后是包含在有条件的“if”语句中的 PSP 特定参数（ui->radioPSP->isChecked）。需要为其他目的设备添加更多参数，才能让转换过程开始工作。



* 来自 **FFPMPEG** 命令的输出将显示在我们的主应用程序的输出选项卡中。

在用户点击我们在 GUI 中创建的“Go”按钮时，将执行这个函数，而且由于我们早先配置好的信号和槽，这个过程也是自动的。但是我们还需要捕捉该过程的输出，以便在文本视图中显示出来，同时给用户提供一些可视的反馈。令人高兴的是，由于信号与槽的神奇魔力，Qprocess 可以不太费力地完全执行这种操作。

在 MainWindow::MainWindow 初始化例行程序中，我们需要在 Qprocess 输出信号和我们将用于把输出转换为文本以便于显示的“outputCommand”槽之间手动创建连接。在“ui->setupUi”行前面添加如下两行：

```
connect (&commandProcess,
SIGNAL (readyReadStandardOutput()), this,
SLOT (outputCommand()));
connect (&commandProcess,
SIGNAL (readyReadStandardError()), this,
SLOT (outputCommand()));
```

正如我们看到的那样，从 Qprocess 发出的有两种类型的输出信号，而且我们将来自这两种信号的输出都发送给同一个函数“outputCommand”，现在需要把这个函数添加到源代码中：

```
void MainWindow::outputCommand()
{
```

```

    QByteArray cmdoutput =
commandProcess.readAllStandardOutput();
    QString txtoutput = cmdoutput;
    ui->textBrowser->append(txtoutput);
    cmdoutput = commandProcess.readAllStandardError();
    txtoutput = cmdoutput;
    ui->textBrowser->append(txtoutput);

}

```

这是当 Qt 从运行“FFMPEG”的Qprocess 检测输出时执行的函数。这有点繁复，因为我们不能假定命令的输出是文本，而且输出数据有两种流形式——一种用于来自命令的标准输出，而另一种用于错误输出。我们的安全做法是在使用 Qt 的优秀转换例行程序将这些数据转换为一个文本字符串之前，将流中的二进制数据复制到一个原始字节数组中。接着把这些数据添加到文本视图中，而且我们对于命令的错误输出重复这个过程。没有什么捷径可以同时抓取到这两种流。

最后，在完成我们的应用程序之前，最后一个步骤是添加两个槽，用于处理源和目的文件位置。这两个槽几乎是完全相同的。下面给出了处理目的文件位置的槽函数：

```

void MainWindow::setDestination()
{
    QString file = QFileDialog::getSaveFileName(this,
tr("Select Destination"),
QDesktopServices::storageLocation(QDesktopServices::Movie
sLocation));
    ui->lineEdit_2->setText(file);
}

```

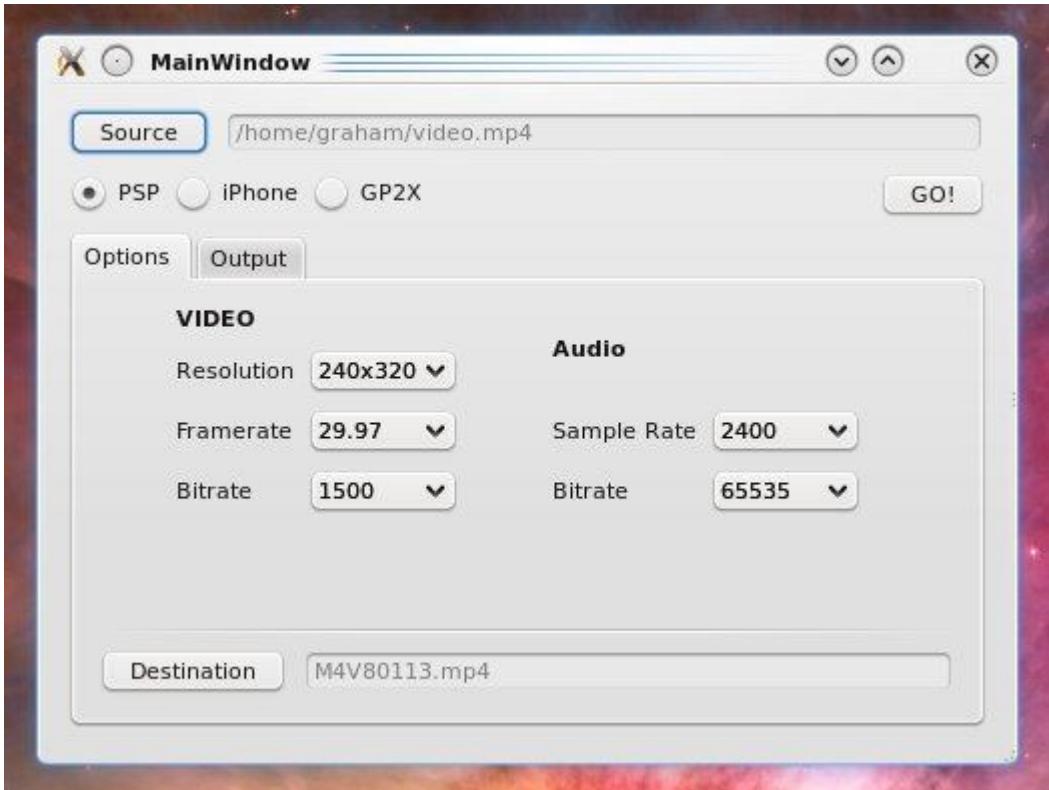
第一行创建了一个 Qt 文件请求器，自动指向系统默认的电影位置，而因为我们已经使用了“getSaveFileName”，用户将被询问一个不一定存在的文件的名称。这与“setSource”完全相反：

```

void MainWindow::setSource()
{
    QString file = QFileDialog::getOpenFileName(this,
tr("Select Source File"),
QDesktopServices::storageLocation(QDesktopServices::Movie
sLocation));
    ui->lineEdit->setText(file);
}

```

输入这最后两个函数后，我们应用程序的源代码就已经完成了，应该有一个可用的 FFmpeg GUI，可以自定义它以使用所需的任意参数。编译并运行就可以了。还应该看到，修改这些代码以使用其他命令行工具是多么轻松的事情。



下载此项目的代码：[qt_menq.tar](#)

借助基于 Qt 的前端，用户就不必深入研究 FFmpeg 的命令行用法了。

原文：

<http://www.tuxradar.com/content/code-project-create-ffmpeg-front-end>

[Code Project: 创建一个 Qt RSS 阅读器](#)

我们将构建一个完整的应用程序，使其不必太费事便可重新发布为一个真正的开源应用程序。这个应用程序就是一个 RSS 阅读器，它允许用户添加自己的种子，列出该种子上的内容，然后让用户在主应用程序自带的一个浏览器窗口中阅读这些内容。

如果你已经尝试过了我们前两个 Qt 代码项目——[创建一个 ffmpeg 前端](#)和[创建一个媒体播放器](#)，而且正在寻求更多 Qt 方面的乐趣，那么请读下去…

RSS 是一个以特定方式进行格式化的 XML 文本文件。它包含对网站上每段内容的简短描述。它最大的优点就是，始终随着新内容的发布而更新。使用 RSS 阅读器或像 Firefox 这样与 RSS 兼容的浏览器时，用户可以从网站订阅 RSS 种子，而且

阅读器将定期检查更新，并列出所有新的内容供用户浏览。而这也正是我们的应用程序所要实现的功能。

它还将引入一些主要的 Qt 技术，包括处理 XML 数据流的手段，如何动态填充树视图小部件，以及使用 WebKit 小部件并将所有小部件组合为一个可动态扩展的、将自动更新为显示 web 页面的应用程序窗口。这使得 RSS 阅读器成为启动更多目标远大的项目的最佳起点，即使你马上弃用 RSS 处理的代码，我们为这个应用程序所构建的可扩展 GUI 仍然可以发挥作用。

这正是首次运行 Qt Creator 并创建一个新项目时，需要选择三个单独的模块在应用程序中使用的原因。在向导中点击 Qt4 GUI Application 模块，给它取一个名字，然后启用如下三个模块：QtNetwork, QtWebkit 和 QtXML。这些模块将紧密联系我们将在本指南中讲到的三个新领域，而且从向导添加它们后，便不用再手动把它们添加到项目的 “.pro” 文件中。

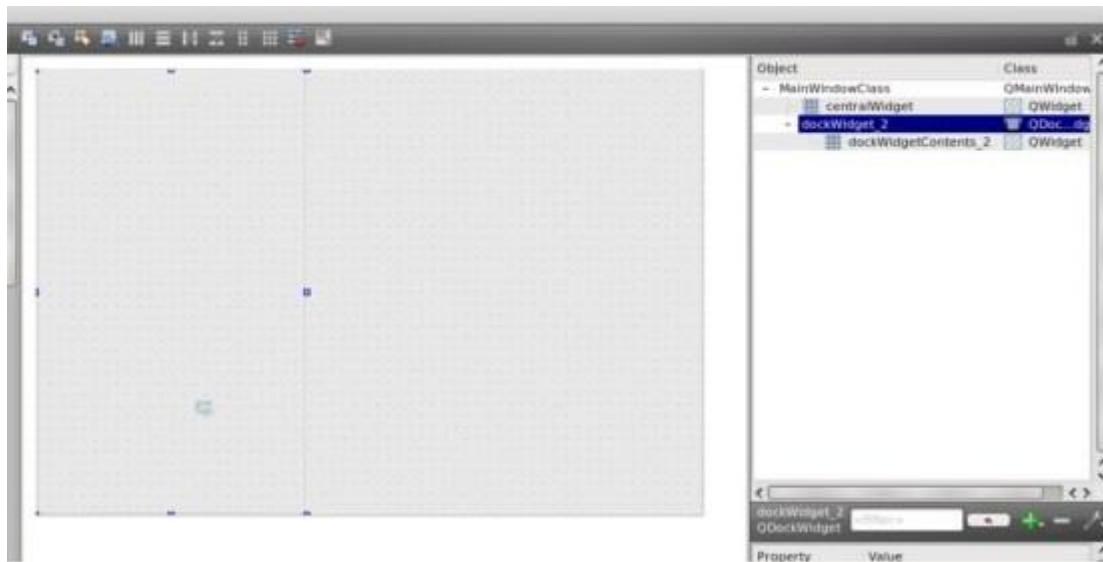
设计 GUI

和我们其他的 Qt 编程指南一样，在运行 Creator 创建一个新项目后，接下来要做的工作就是 GUI 设计。点击 “ui” 文件打开 Designer 视图。这一次，我们将采用稍微开放一点的方法进行设计。主窗口将被划分为两个面板。在左侧，我们将添加 RSS 消息列表，并让用户能够添加他们自己的种子。而窗口的右半部分将是 web 浏览器，我们将对这部分使用 WebKit 小部件。

但是 Qt 的聪明之处在于，我们可以根据用户是否想使用内部浏览器来使每个面板变得可扩展或可隐藏，或者干脆使用他们最惯于使用的浏览器。例如，如果用户不想看到 web 视图，只需要把中间的分离线拖到右边，它就会消失。这给予了我们的应用程序很大的灵活性，并不强迫想使用自己的浏览器阅读新闻的用户使用 web 视图。

这项特别的功能是通过 Qt 中的 Dock Widget 小部件实现的。当应用程序分为几个部分时，它提供了很强的灵活性，允许用户在四周拖放窗口的不同部分。从 Creator 页面的 Containers 列表中拖出两个 Dock Widget 小部件到空白的应用程序画布上。如果在应用程序中用不着，还可以从 Object 视图删除多余的菜单、工具栏和状态面板小部件。我们已经添加了两个可停靠小部件，因为我们要在应用程序的两侧使用它们来保存小部件，而且它们是可停靠小部件，用户能够拖动它们之间的分离线，从而改变应用程序每个半区的尺寸。

但在添加更多小部件之前，我们需要确保只启用了每个可停靠小部件的一组有限功能。我们不想让用户完全访问 Qt 可停靠小部件更多难以驾驭的功能，KDevelop 已经很好地证明了这一点。在 Object 列表中选择每个可停靠小部件，然后在下面的属性窗口中，确保将选中的 “Features” 字段设置为 “NoDockWidgetFeatures”。这将阻止用户将小部件拖动至窗口外部或者完全关闭它们。你可能想对浏览器面板启用这项功能，这由你自己决定。



* 可停靠小部件的优点是，用户可在应用程序运行时改变两侧的比例差异。

小部件面板

在添加其他小部件之前，选择可停靠小部件并点击“Lay Out Horizontally”按钮。接着点击“Lay Out in a Grid”按钮。这样做的效果是同时拉伸跨应用程序窗口的、中间具有一条分离线的两个可停靠小部件。当用户改变主窗口的大小时，这两个小部件将保持它们的相对位置。

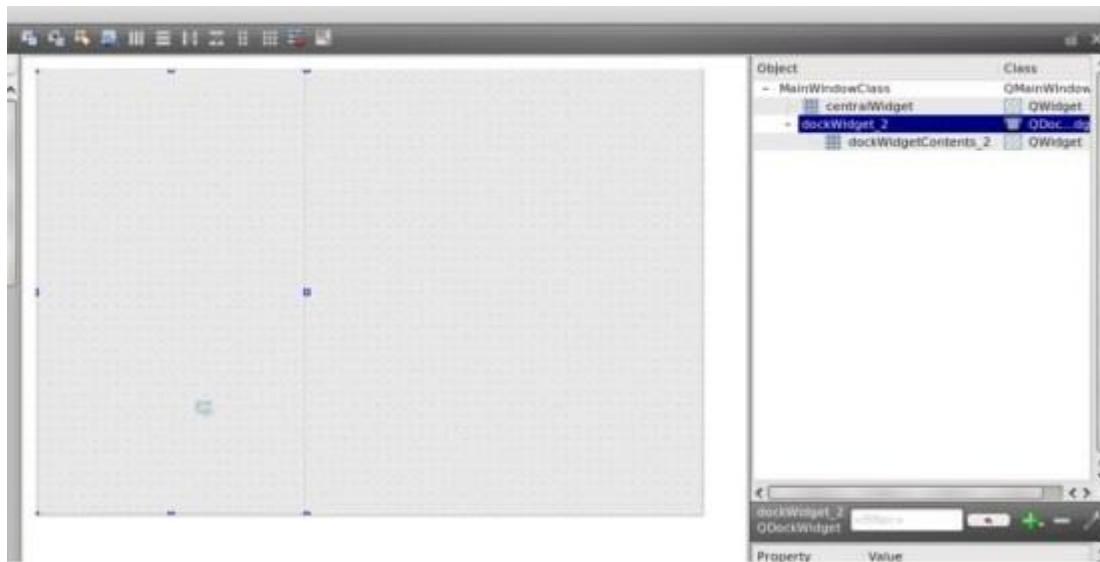
尽管网格被锁定，我们仍然能够以常规方式给可停靠小部件添加小部件，而且我们准备从左侧开始。如果在网格被锁定的情况下编辑 GUI，Designer 将使用蓝色光标突出显示每个小部件要插入的位置，这一点十分类似于字处理器。需要将三个小部件拖动到左边面板中——一个行编辑小部件和一个按钮，它们已经在窗口顶部水平对齐了，还有一个位于下方的树视图。行编辑小部件用于给用户输入 RSS 种子的 URL，按钮用于提交种子给我们的解析器，而树视图用于列出 RSS 种子的每个入口。

我们给行编辑小部件添加了一个默认的 URL。只要双击该小部件，然后输入类似于“<http://www.qteverywhere.com/rss>”的内容，再将按钮文本改为“Add Feed”。双击树视图，再添加两列，将它们分别取名为“Feed”、“Date”和“URL”。这些列将包含每个新闻内容的信息，但只有“Feed”和“Date”两列可见。这是因为我们要内部使用 URL 列，不显示给用户看。它将保存内容的 URL，这样当用户点击它时，我们可以把 URL 发送给 WebKit。

如果我们不使用这种方法，我们就不得不为应用程序实现一个成熟的 MVC 解决方案，而这描述起来都超过 4 页纸了。MVC（模型/视图/控制器）是一种将数据（在这个例子中是指 URL）与显示数据的视图分离，同时保持二者联系的方法。后一部分由控制器来处理。当我们使用它的任意容器类时，Qt 在后台使用的是 MVC，

而它用于添加和删除内容项的方法实际上是用于在后台处理 MVC 的便利函数。我们将在树视图中利用这一点，隐藏 URL 列并在应用程序中使用数据，但我们只能在源代码中做到这一点。

最后，将 WebKit 小部件拖动到右侧面板中。这是一个自包含的浏览器窗口，我们不需要添加任何别的内容就可以让它工作。只要保证所有小部件都经过了正确排列，以及你已经在两个面板上使用了一些间距器和“Lay Out in the Grid”模式，从而锁定可缩放窗口的布局。



* 在可停靠小部件的左侧，我们添加了树视图，XML 提要的 URL，以及用于从 Internet 抓取数据的按钮。

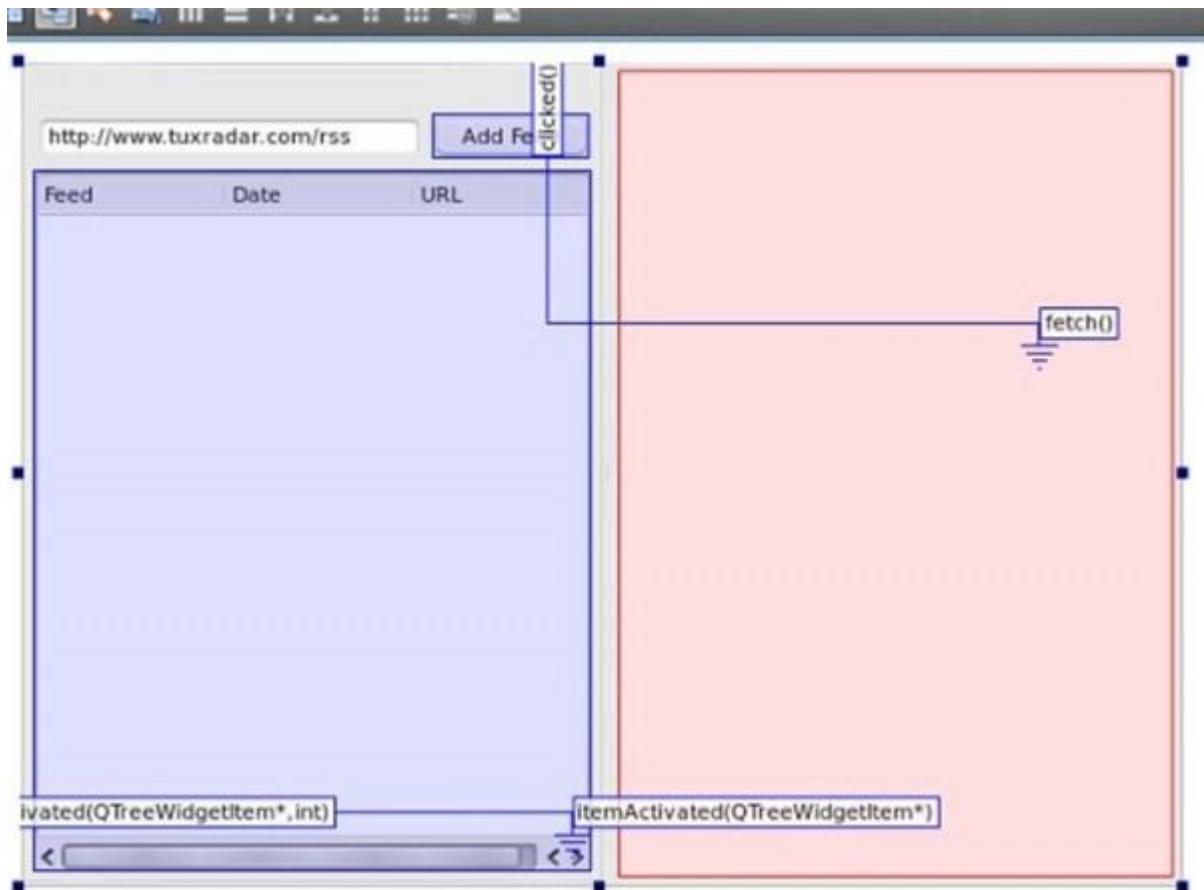
连接

既然我们的布局已经最终确定，下一步就要添加槽/信号连接，用于补充我们应用程序的功能。切换到 Signals/Slots 编辑器，方法是按下 F4 键或者在工具栏中点击相应按钮。从“Add Feed”按钮拖动一个信号到应用程序窗口的轮廓处，当“Configure Connection”窗口出现时，点击右侧面板上的“Edit”按钮。

我们需要添加两个槽。第一个用于给树视图添加种子，而另一个用于当用户在种子列表中选择一个新闻内容时更新 web 视图。我们将第一个槽称为“fetch()”，而将第二个槽称为“itemActivated(QTreeWidgetItem*)”。这是我们首次遇到通过信号/槽机制传递的参数，要在设计器中使用它们，必须满足一些严格的规定。其中最重要的一条是，对于一个在传递这类参数时要连接到槽的信号，二者都必须完全支持同一类型。在这个例子中是 QTreeWidgetItem 类型。

创建这两个槽并将“clicked”连接到“fetch()”之后，从树视图拖一个新连接到窗口背景。我们将看到，很多函数将 QItemTreeTree 参数作为一个参数包含在

内。这是树视图中每一项的类型，以这种方式传递它使我们能够轻松抓取到当前选中的新闻内容的 URL，并使用它来更新 web 浏览器。只要将位于左边的“`itemActivated(QTreeWidgetItem*)`”与我们刚刚为自己的应用程序创建的名称相同的新槽连接起来即可。



* 这是我们在 **Creator** 中建立并用于应用程序中各个函数的 **SIGNAL** 和 **SLOT** 连接的一个视图。

代码

现在我们已经建立了框架，是时候添加代码了。和我们其他的项目一样，我们从“mainwindow.h”开始，把它作为需要添加我们刚刚在 GUI 中创建的新槽的地方。我们还准备添加要在程序逻辑中使用的新槽，用于告诉我们的应用程序，从 Internet 读取 web 数据的过程已经结束。

```
void fetch();  
void itemActivated(QTreeWidgetItem * item);  
void readData(const QHttpResponseHeader &);
```

现在，我们需要给项目添加一些私有成员。我们将使用这些私有成员管理数据流，并且为解析从站点的 RSS 种子抓取到的 XML 数据和 HTML 数据而创建数据结构。

```
void parseXml();  
  
QString currentTag;  
QString linkString;  
QString titleString;  
QString dateString;  
  
QTreeWidgetItem *feed;  
  
int connectionId;  
QHttp http;  
QXmlStreamReader xml;
```

这是我们需要给头文件添加的内容。我们余下的编码将限制在“mainwindow.cpp”文件中，从位于该文件顶部的初始化函数开始。首先，我们需要在“setupUi”前面添加一个连接行，用于当我们知道 Qt 的 HTTP 抓取器已经正确解析 HTTP 时，自动运行我们的“readyRead”方法。其次，我们想隐藏 treeWidget 的两列，因为我们只使用这些列来保存数据，而不想让用户看到它们。一旦“setupUi”创建了 GUI，我们就可以这样修改它。下面给出相应的代码：

```
connect(&http, SIGNAL(readyRead(const  
QHttpResponseHeader &)), this,  
        SLOT(readData(const QHttpResponseHeader &)));  
  
ui->setupUi(this);  
  
ui->treeWidget->setColumnHidden(1, true);  
ui->treeWidget->setColumnHidden(2, true);
```

现在，我们准备编写 fetch() 函数。当我们在应用程序中输入 RSS 种子的 URL，然后点击“Add Feed”按钮时，将触发这个函数的功能。

```
void MainWindow::fetch()  
{  
    xml.clear();  
  
    QUrl url(ui->lineEdit->text());  
  
    http.setHost(url.host());  
    connectionId = http.get(url.path());  
}
```

这段代码相对较为直观。首先，我们清理了保存 XML 日期的流读取对象，然后将我们用于保存 URL 的行编辑组件中的文本转换为一个 QUrl，这是 Qt 中访问在线资源的首选方法。接下来，我们使用这个资源设定 QHttp 的位置，这个类是用于实现 HTTP 协议的。我们需要 使用这个类来抓取 XML 数据，而下一行中使用“get”函数和经过转换的 URL 调用了这个函数。当“http”成功打开 HTTP 位置时，它将发出我们前面连接到我们自己的“readData”函数的“readyRead”信号。现在我们需要添加这个函数：

```
void MainWindow::readData(const QHttpResponseHeader &resp)
{
    if (resp.statusCode() != 200)
        http.abort();
    else {
        xml.addData(http.readAll());
        parseXml();
    }
}
```

这个函数的全部功能就是检查是否找到了 URL，如果没有找到，它会中断，而且我们的应用程序也不会再往下执行。但如果远程位置是合法的，在把数据发送给“parseXML”函数之前，我们首先会使用数据填满我们的 XML 容器——xml.addData(http.readAll())。这是应用程序的一个难点，因为需要遍历从 internet 抓取的 XML 树，并把我们需要的数据块放到 treeView 中。因此，对应代码的篇幅要长很多。

```
void MainWindow::parseXml()
{
    while (!xml.atEnd()) {
        xml.readNext();
        if (xml.isStartElement()) {

            if (xml.name() == "item") {

                if (titleString!="") {
                    feed = new QTreeWidgetItem;
                    feed->setText(0, titleString);
                    feed->setText(2, linkString);
                    ui->treeWidget->addTopLevelItem(feed);

                }

                linkString.clear();
                titleString.clear();
            }
        }
    }
}
```

```

        dateString.clear();
    }

    currentTag = xml.name().toString();
} else if (xml.isEndElement()) {
    if (xml.name() == "item") {

        QTreeWidgetItem *item = new
QTreeWidgetItem(feed);
        item->setText(0, titleString);
        item->setText(1, dateString);
        item->setText(2, linkString);
        ui->treeWidget->addTopLevelItem(item);

        titleString.clear();
        linkString.clear();
        dateString.clear();
    }
}

} else if (xml.isCharacters() && !xml.isWhitespace())
{
    if (currentTag == "title")
        titleString += xml.text().toString();
    else if (currentTag == "link")
        linkString += xml.text().toString();
    else if (currentTag == "pubDate")
        dateString += xml.text().toString();
}
}

if (xml.error() && xml.error() !=
QXmlStreamReader::PrematureEndOfDocumentError) {
    qWarning() << "XML ERROR:" << xml.lineNumber() << ":" << xml.errorString();
    http.abort();
}
}

```

这段代码看起来有点吓人，但这主要是因为它包含了多条嵌套的“if”语句，用于处理我们在 RSS 种子中将会遇到的不同类型的 XML 元素。

我们从依次读取每个元素开始，然后检查该元素在树上是属于新的内容项，一个元素的末端，还是包含真正的元素数据。

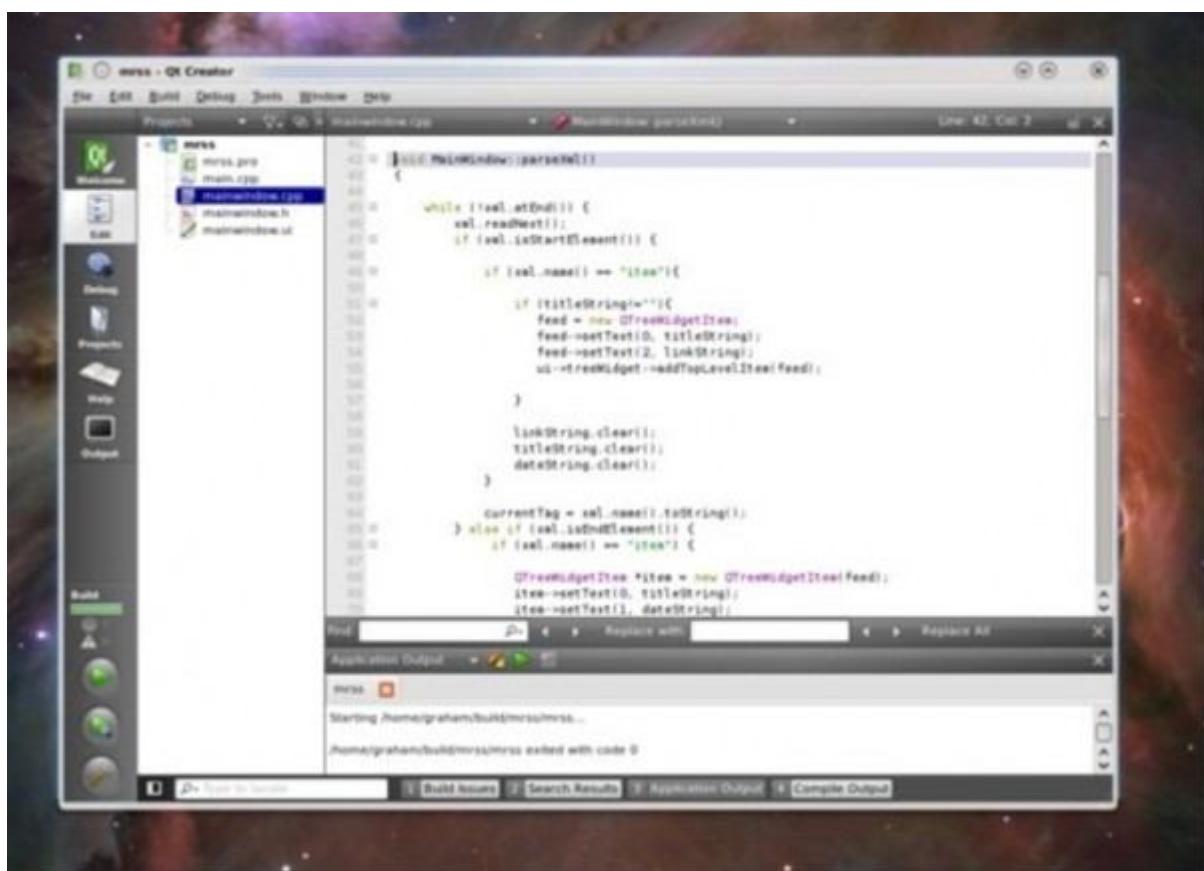
当代码检测到 XML 种子中一个新元素的起点时，它会将“currentTag”设为该元素中所包含数据的类型。我们只对“title”、“link”和“pubDate”字段感兴趣。

趣，而且当 XML 流移动到文件的字符部分时，每个字段的文本就会根据 currentTag 类型转移到“titleString”、“linkString”和“dateString”中。当检测到一个元素的终点时，我们已经知道这些字符串是否已经被填充，以及是否可以把数据复制到我们 GUI 中的“treeView”对象中。这就是“item->setText”行的作用，而这些字符串被添加到一个 treeView 顶部的内容项中，该 treeView 创建来保存“StartElement”部分中的每个 RSS 种子的树视图。

如果我们在理解这个函数时有困难，使用 Creator 的优秀调试器把代码完整地运行一遍，会对我们有所帮助。如果在这个函数中设置一个断点，当应用程序的执行到达代码的这个部分时，我们就能够使用 Debug 菜单单步调试每一行，并监视感兴趣参数的值。

最后，我们需要添加的最后一个函数是用于在用户点击 RSS 新闻项之一时，将该新闻项指向的 web 页面装载到我们的 WebKit 小部件中。这个函数是使用我们在 GUI 设计器中建立的“itemActivated” SIGNAL/SLOT 连接来执行的。我们从树小部件内容项中找到了由信号传递的 URL，并把这些数据发送给 webView 小部件，然后把它转换为一个 QUrl，而我们也正是这样做的。幸运的是，此功能一共只需要两行代码：

```
void MainWindow::itemActivated(QTreeWidgetItem * item)
{
    ui->webView->load(QUrl(item->text(2)));
    ui->webView->show();
}
```



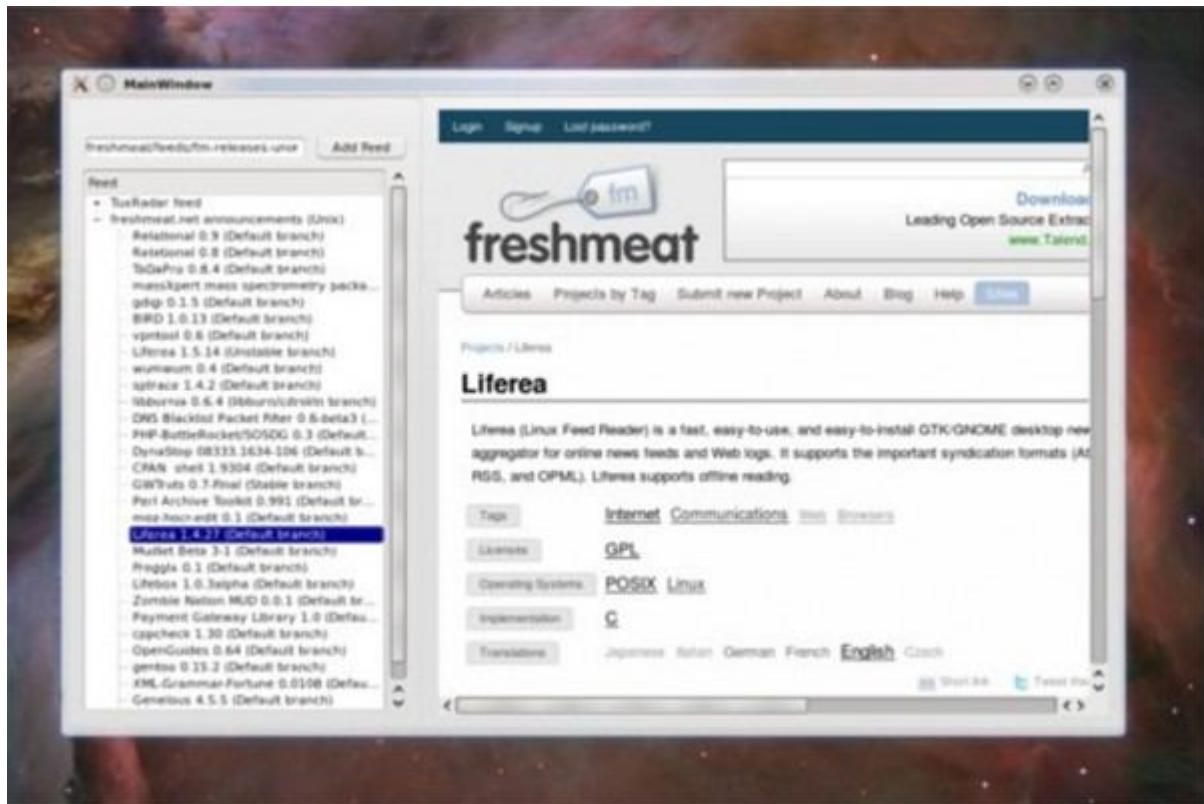
* 尝试使用调试器来理解我们项目中一些更为复杂的函数。

运行应用程序

以上就是需要完成的全部工作。剩下的就是保存项目，编译和运行。点击“Add Feed”按钮可以添加我们在GUI中创建的默认RSS种子，而且我们应该看到，位于左侧的树视图使用了来自TuxRadar.com的所有最新内容进行填充。点击其中任意的内容，右侧的web查看器就会加载相应的页面。

但这个应用程序的最大优点是缩放两个面板的方式。在树视图和web页面之间，应该能够找到三个很小的垂直点。可以把这些点拖到左边或右边，从而改变RSS种子或正在显示的web页面的比例。如果将这一条完全移到右边，web视图就会完全关闭。如果我们只想看到种子列表，这是个不错的主意。

可以轻松给这个应用程序添加所需要的内容。我们从一个具有自动功能的刷新按钮开始。这个按钮功能是大约每小时增加一次新种子，或者在点击它时手动刷新。应用程序还迫切需要保存其设置的功能。这是一项十分艰巨的任务，几乎可以作为另一份指南的主题——为什么不亲自写写试试看呢？



完成之后的应用程序：密切注意 Akregator，我们的 RSS 应用程序是跨平台的，而且使用 WebKit 来呈现 web 页面。

下载源代码：[qt_mrss.tar](#)

原文链接：

<http://www.tuxradar.com/content/code-project-create-qt-rss-reader>

翻译 CSDN：<http://qt.csdn.net/articles.aspx?pointid=178>

Qt Graphics View 详解

Qt 的 Graphics View 框架（一）

Graphics View 提供了一个界面，它既可以管理大数量的定制 2D graphical items，又可与它们交互，有一个 view widget 可以把这些项绘制出来，并支持旋转与缩放。这个框架也包含一个事件传播结构，对于在 scene 中的这些 items，它具有双精度的交互能力。Items 能处理键盘事件，鼠标的按，移动、释放、双击事件，也可以跟踪鼠标移动。Graphics View 使用 BSP 树来提供对 item 的快速查找，使用这种技术，它可以实时地绘制大规模场景，甚至以百万 items 计。Graphics View 在 Qt 4.2 中被引用，它替代了它的前辈 QCanvas。

Graphics View 的体系结构

Graphics View 提供的是一种类似于 Qt model-view 的编程。多个 views 可以监视同一个场景，而场景包含多个具有多种几何外形的 items。

场景

`QGraphicsScene` 表示 Graphics View 中的场景，它有以下职责：

为管理大量的 items 提供一个快速的接口。

传播事件到每个 item。

管理 item 的状态，例如选择，焦点处理。

提供未经变换的渲染功能，主要用于打印。

场景作为 `QGraphicsItem` 对象的容器。通过调用 `QgraphicsScene::addItem()` 把这些 Items 加入到场景中。可以使用众多的查找函数来获取特定的 items。

`QGraphicsScene::items()` 与它的许多重载函数可获取那些与点、矩形，多边形，向量路径等相交或是有包含有关系的 items。`QGraphicsScene::itemAt()` 返回特定上最顶端的 item。所有的 item 查找函数都以出栈序列返回（也就是说，第一个返回的是最顶端的，最后一个返回的是最底端的）。

```
QGraphicsScene scene;
QGraphicsRectItem
*rect=scene.addItem(QRectF(0,0,100,100));
QGraphicsItem *item=scene.itemAt(50,50);
//item==rect;
```

`QGraphicsScene` 的事件传播结构会把场景事件投递到 items，也管理多个 items 之间的传递。假如场景收到了鼠标在某个位置 press 事件，场景会把这个事件投递给处在那个位置的 item。QGraphicsScene 也管理某种 item 状态，像选择与焦点。你可以通过调用 `QGraphicsScene::setSelectionArea()` 来选择 items，它需要提供一个任意的形状为参数。这个函数也作为在 QGraphicsView 实现橡皮筋选择功能的一个基础。为得到这些已经被选择的 items，调用

`QGraphicsScene::selectedItem()`。另一个状态处理是否一个 item 拥有键盘输入焦点。你可以调用 `QGraphicsScene::setFocusItem()` 或

`QGraphics::setFocus()` 来设定焦点，也可用 `QGraphicsScene::focusItem()` 来得到当前拥有焦点的那个 item。最后，`QGraphicsScene` 允许你通过调用 `QGraphicsScene::render()` 函数把部分场景送到绘图设备进行渲染。

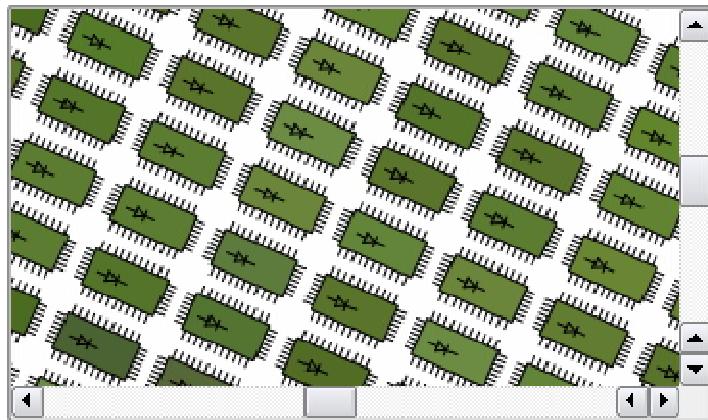
视图

`QGraphicsView` 提供了视图部件，它可视化场景中的内容。你可以联结多个视图到同一个场景，对这个相同的数据集提供几个视口。视口部件是一个滚动区域，

它提供了滚动条以对大场景进行浏览。为了使用 OpenGL, 你应该调用 `QGraphicsView::setViewport()` 来把一个 `QGLWidget` 设为视口。视图从键盘, 鼠标接收输入事件, 在发送这些事件到场景之前, 会对这些事件进行适当的翻译(把事件坐标转换成对应的场景坐标)。

利用转换矩阵, `QGraphicsView::matrix()`, 视图可变换场景的坐标系统。这允许高级的导航特性, 如缩放, 旋转。为了方便, `QGraphicsView` 也提供了在视图与场景之间进行坐标转换的函数:

`QGraphicsView::mapToScene()`, `QGraphicsView::mapForScene()`。



The Item

`QGraphicsItem` 是场景中图形 items 的基类。Graphics View 提供了一些标准的、用于典型形状的 items。像矩形 (`QGraphicsRectItem`), 椭圆

(`QGraphicsEllipseItem`), 文本 (`QGraphicsTextItem`), 当你写定制的 item 时, 那些最有用的一些 `QGraphicsItem` 特性也是有效的。除此这外, `QGraphicsItem` 支持以下特性:

*鼠标按、移动、释放、双击事件, 鼠标悬停事件, 滚轮事件, 弹出菜单事件。

*键盘输入焦点, 键盘事件。

*拖拽

*组, 包括父子关系, 使用 `QGraphicsItemGroup`

*碰撞检测

Items 如同 `QGraphicsView` 一样, 位于本地坐标系, 它也为 item 与场景之间, item 与 item 之间的坐标转换提供许多工具函数。而且, 也像 `QGraphicsView` 一样, 它使用矩阵来变换它的坐标系统: `QGraphicsItem::matrix()`。它对旋转与缩放单个的 Item 比较有用。

Items 可以包含别的 items(孩子)。父 items 的转换被它的子孙所继承。然而, 它的所有函数(也就是,

`QGraphicsItem::contains()`, `QGraphicsItem::boundingRect()`, `QGraphicsItem::collidesWith()`), 不会积累这些转换, 依然在本地坐标下工作。

`QGraphicsItem` 通过 `QGraphicsItem::shape()`, `QGraphicsItem::collideWith()` 来支持碰撞检测。这两个都是虚函数。从 `shape()` 返回你的 item 的形状(以本地坐标 `QPainterPath` 表示), `QGraphicsItem` 会为你处理所有的碰撞检测。假如你想提供自己的碰撞检测, 你应该重新实现 `QGraphicsItem::collideWith()`。

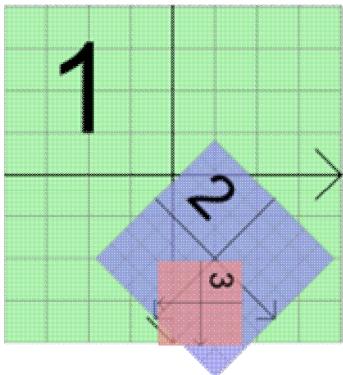
转载自：<http://www.cppblog.com/yuanya/jie/archive/2007/09/26/32960.html>

Qt 的 Graphics View 框架（二）

Graphics View 坐标系统

Graphics View 基于笛卡尔坐标系。item 在场景中的位置与几何形状通过 x, y 坐标表示。当使用未经变形的视图来观察场景时，场景中的一个单位等于屏幕上的一 个 像素。在 Graphics View 中有三个有效的坐标系统：Item 坐标系，场景坐标系，视图坐标系。为了简化你的实现，Graphics View 提供了方便的函数，允许三个坐标系之间相互映射。

当渲染时，Graphics View 的场景坐标对应于 QPainter 的逻辑坐标，视图坐标与设备坐标相同。



Item 坐标

Items 位于它们自己的坐标系中。它的坐标都以点(0, 0)为 中心点，这也是所有变换的中心点。在 item 坐标系中的几何图元，经常被称为 item 点，item 线，item 矩形。当创建一个定制的 item，item 坐标是所需要考虑的。[QGraphicsScene](#) 与 [QGraphicsView](#) 可以为你执行所有转换，这使得实现定制的 item 变得容易。举例来说，假如你收到鼠标按或是拖进入事件，事件的位置以 item 坐标的形式给出。[QGraphicsItem::contains\(\)](#) 虚函数，当某个点的位置在你的 item 范围内时，返回 true，否则返回 false。这个点参数使用 item 坐标，相似地，item 的包围矩形与形状也使用 item 坐标。

Item 位置指的是 item 的中心点在它父亲的坐标系中的坐标。以这种思想来看，场景指的就是那些祖先最少的 item 的“父亲”。最上级的 Item 位置就是在场景中的位置。

子 坐标与父坐标之间是相关的，假如孩子未经变换，子坐标与父坐标之间的差值等于在父坐标系下，父 item 与子 item 之间的距离。例如，假如一个未经变换的 子 item 位置与其父 item 的中心重合，那么这两个 item 的坐标系统完全相同。如果孩子的位置是 (10, 0)，那么孩子坐标系中的(0, 10)点，对 应于父坐标系

中的 (10, 10) 点。

因为 item 的位置与变换是相对于父 item 的，子 item 的坐标不会被父亲的变换影响，尽管父 item 的变换隐含地对子 item 做了变换。在上面的例子中，即使父 item 旋转，缩放，子 item 的 (0, 10) 点依然对应于父 item 的 (10, 10) 点。然而，相对于场景来讲，子 item 会遵循父 item 的变换。假如父 item 被缩放(2X, 2X)，子 item 的位置在场景中的坐标是 (20, 0)，它的 (10, 0) 点则与场景中的 (40, 0) 对应。除了 `QGraphicsItem::pos()`，`QGraphicsItem` 的函数以 Item 坐标工作，如一个 item's 包围矩形总是以 item 坐标的形式给出。

场景坐标

场景坐标系统描述了每个最顶级 item 的位置，也是从视图向场景投递场景事件的基础。场景中的每个 item 有场景位置与包围矩形

(`QGraphicsItem::scenePos()`, `QGraphicsItem::sceneBoundingRect()`)，另外，它有自己本地 item 位置与包围矩形。场景位置描述了 item 在场景坐标下的位置，它的场景包围矩形则用于 `QGraphicsScene` 决定场景中哪块区域发生了变化。场景中的变化通过 `QGraphicsScene::changed()` 信号来通知，它的参数是场景矩形列表。

视图坐标

视图坐标是 widget 的坐标，视图坐标中每个单位对应一个像素。这种坐标的特殊之处在于它是相对于 widget 或是视口的，不会被所观察的场景所影响。

`QGraphicsView` 的视口的左上角总是 (0, 0)，右下角总是(视口宽，视口高)。所有的鼠标事件与拖拽事件，最初以视图坐标表示，就应该把这些坐标映射到场景坐标以便与 item 交互。

坐标映射

经常，处理场景中 item 时，在场景与 item 之间，item 与 item 之间，视图与场景之间进行坐标映射，形状映射是非常有用的。举例来讲，当你在 `QGraphicsView` 的视口中点击鼠标时，你应该通过调用 `QGraphicsView::mapToScene()` 与 `QGraphicsScene::itemAt()` 来获知光标下是场景中的哪个 item。假如你想获知一个 item 位于视口中的什么位置，你应该先在 item 上调用

`QGraphicsItem::mapToScene()`，然后调用 `QGraphicsView::mapFromScene()`。最后，假如你想在一个视图椭圆中有哪些 items，你应该把 `QPainterPath` 传递到 `mapToScene()`，然后再把映射后的路径传递到 `QGraphicsScene::items()`。

你可以调用 `QGraphicsItem::mapToScene()` 与 `QGraphicsItem::mapFromScene()` 在 item 与场景之间进行坐标与形状的映射。也可以在 item 与其父 item 之间通过 `QGraphicsItem::mapToParent()` 与 `QGraphicsItem::mapFromItem()` 进行映射。所有映射函数可以包括点，矩形，多边形，路径。视图与场景之间的映射也

与此类似。对于从视图与 item 之间的映射，你应该首先映射到场景，然后再从场景向 item 进行映射。

转载自：<http://www.cppblog.com/yuanyajie/archive/2007/09/26/32961.html>

Qt 的 Graphics View 框架（三）

关键特性

缩放与旋转

QGraphicsView 通过 QGraphicsView::setMatrix() 支持同 QPainter 一样的仿射变换，通过对一个视图应用变换，你可以很容易地支持普通的导航特性如缩放与旋转。下面是一个例子：

```
class View : public QGraphicsView
{
    Q_OBJECT
// .....
public slots:
void zoomIn() { scale(1.2, 1.2); }
void zoomOut() { scale(1/1.2, 1/1.2); }
void rotateLeft() { rotate(-10); }
void rotateRight() { rotate(10); }
};
```

这些槽应与 QToolButtons 联接，并使 autoRepeat 有效。当对视图变换时，QGraphicsView 会对视图中心进行校正。

拖拽

因为 QGraphicsView 继承自 QWidget，它也提供了像 QWidget 那样的拖拽功能，另处，为了方便，Graphics View 框架也为场景，每个 item 提供拖拽支持。当视图接收到拖拽事件，它可翻译为 QGraphicsSceneDragDropEvent，再发送到场景。场景接管这个事件，把它发送到光标下接受拖拽的第一个 item。

从一个 item 开始拖拽时，创建一个 QDrag 对象，传递开始拖拽的那个 widget 的指针。Items 可以同时被多个视图观察，但只有一个视图可以开始拖拽。拖拽在多数情况下是从按下鼠标或是移动鼠标开始的，因此，在 mousePressEvent() 或 mouseMoveEvent() 中，你可以从事件中得到那个原始的 widget 指针，例如：

```
void CustomItem::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    QMimeData *data=new QMimeData;
    data->setColor(Qt::green);
    QDrag *drag=new QDrag(event->widget());
    drag->setMimeData(data);
    drag->start();
}
```

为了在场景中截取拖拽事件，你应重新实现 `QGraphicsScene::dragEnterEvent()` 和在 `QGraphicsItem` 的子类里任何与你特定场景需要的事件处理器。`items` 也可以通过调用 `QGraphicsItem::setAcceptDrops()` 获得拖拽支持，为了处理将要进行的拖拽，你需要重新实现 `QGraphicsItem::dragEnterEvent()`, `QGraphicsItem::dragMoveEvent()`, `QGraphicsItem::dragLeaveEvent()` 和 `QGraphicsItem::dropEvent()`。

光标与工具提示

像 `QWidget` 一样，`QGraphicsItem` 也 支持光标 (`QgraphicsItem::setCursor()`) 与 工具提示 (`QGraphicsItem::setToolTip()`)。当光标进入到 `item` 的区域，光标与工具提示被 `QGraphicsView` 激活（通过调用 `QGraphicsItem::contains()` 检测）。你也可以直接在 视图上设置一个缺省光标 (`QGraphicsView::setCursor()`)。

动画

Graphics View 支持几种级别的动画。你可以很容易地通过把 `QGraphicsItemAnimation` 与你的 `item` 联结来 装配出动画路径，这允许以时间线来控制动画，在所有平台上以稳定的速率运作。`QGraphicsItemAnimation` 允许你为 `item` 的位置，旋转，缩放，剪切，变换等产生一条路径，动画可以用 `QSlider` 来控制，或更为普遍使用的 `QTimeLine`。另一种是从 `QObject` 和 `QGraphicsItem` 继承，`item` 可以设置自己的定时器，以在 `QObject::timeEvent()` 中增加步进的方式来控制动画。第三种，是通过调用 `QGraphicsScene::advance()` 来推进场景，它又依次调用 `QGraphicsItem::advance()`。

OpenGL 渲染

为了使用 OpenGL 渲染，你要设置一个新的 `QGLWidget` 作为 `QGraphicsView` 的视口：`QGraphicsView::setViewPort()`。假如你让 OpenGL 提供反锯齿功能，你需要 OpenGL 采样缓冲支持。

```
QGraphicsView view(&scene);
view.setViewport(new QGLWidget(QGLFormat(QGL::SampleBuffers)));
```

Item 组

通过把一个 item 做为另一个 item 的孩子，你可以得到 item 组的大多数本质特性：这些 items 会一起移动，所有变换会从父到子传递。QGraphicsItem 也可以为它的孩子处理所有的事件，这样就允许以父亲代表它所有的孩子，可以有效地把所有的 items 看作一个整体。另外，[QGraphicsItemGroup](#) 是一个特殊的 item，它既对孩子事件进行处理又有一个接口把 items 从一个组中增加和删除。把一个 item 加到 QGraphicsItemGroup 仍会保留 item 的原始位置与变换，而给一个 item 重新指定父 item 则会让 item 根据其新的父亲重新定位。可以用 QGraphicsScene::createItemGroup() 建组。

转自自：<http://www.cppblog.com/yuanyajie/archive/2007/09/27/32962.html>

Part3：深入理解 Qt

Inside Qt 系列

QKevin 所著，通过剖析 Qt 源代码，深入浅出的解释了 Qt 中的许多机制，了解 Qt 内部是如何 work 的。着实为想深入了解 Qt 的开发者提供了很好的学习机会。如果你已经学习了上面的内容并且熟练掌握 Qt 编程，那么大力推荐这一系列文章！

Inside QT Series：序

写了这么多年的程序，除了留下很多 code (其中有很多是 garbage) 之外，再没有其它东西，或许我该写点儿什么了，写一些关于我的工作的东西，自己所了解的技术，也把自己在工作过程中新学习的一些东西放在这儿，就算是为了以后做一个参考。

第一个”大项目”，就是准备写一个系列文章，专门介绍 Qt Framework 的，这个系列文章不是为 Qt 新手所写的，而是写一个相对来说深入一些的话题，基本思路是，分析 Qt 的 source code，让我们透过 Qt 编程技术来了解 Qt 内部是如何 work 的。希望这个系列文章的读者具备 C/C++ 的基础和基本的 QT 知识，写过一些 QT code，如果能有人从我的文章中获得些什么，那我就十分欣慰了。

等等，你说什么，你还不知道什么是 Qt？那么就请到 Qt 的 homepage 看看吧：qt.nokia.com

或者看看本站整理的 Qt 简介：《Qt Framework 简介》

好了，让我们开始吧，请看第一篇：

Inside QT Series (一): Let's go, Starting From the QObject
<http://www.insideqt.com/bbs/viewthread.php?tid=4>

=====

==

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。
作者保留版权，未经作者同意，不得用于任何商业用途

《Inside Qt Series》专栏文章总索引：

<http://www.insideqt.com/bbs/viewthread.php?tid=9>

本文原始地址：

<http://www.insideqt.com/bbs/viewthread.php?tid=3>

Inside Qt Series (一): Let's go, Starting From the QObject

QObject 这个 class 是 QT 对象模型的核心，绝大部分的 QT 类都是从这个类继承而来。这个模型的中心特征就是一个叫做信号和槽 (signal and slot) 的机制来实现对象间的通讯，你可以把一个信号和另一个槽通过 connect(...) 方法连接起来，并可以使用 disconnect(...) 方法来断开这种连接，你还可以通过调用 blockSignal(...) 这个方法来临时的阻塞信号，

QObject 把它们自己组织在对象树中。当你创建一个 QObject 并使用其它对象作为父对象时，这个对象会自动添加到父对象的 children() list 中。父对象拥有这个对象，比如，它将在它的析构函数中自动删除它所有的 child 对象。你可以通过 findChild() 或者 findChildren() 函数来查找一个对象。

每个对象都有一个对象名称 (objectName()) 和类名称 (class name)，他们都可以通过相应的 metaObject 对象来获得。你还可以通过 inherits() 方法来判断一个对象的类是不是从另一个类继承而来。

当对象被删除时，它发出 destroyed() 信号。你可以捕获这个信号来避免对 QObject 的无效引用。

QObject 可以通过 event() 接收事件并且过滤其它对象的事件。详细情况请参考 installEventFilter() 和 eventFilter()。

对于每一个实现了信号、槽和属性的对象来说，`Q_OBJECT` 宏都是必须要加上的。`QObject` 实现了这么多功能，那么，它是如何做到的呢？让我们通过它的 Source Code 来解开这个秘密吧。

`QObject` 类的实现文件一共有四个：

- * `qobject.h`, `QObject` class 的基本定义，也是我们一般定义一个类的头文件
- * `qobject.cpp`, `QObject` class 的实现代码基本上都在这个文件
- * `qobjectdefs.h`, 这个文件中最重要的东西就是定义了 `QMetaObject` class，这个 class 是为了实现 `signal`、`slot`、`properties`，的核心部分。
- * `qobject_p.h`, 这个文件中的 code 是辅助实现 `QObject` class 的，这里面最重要的东西是定义了一个 `QObjectPrivate` 类来存储 `QObject` 对象的成员数据。

理解这个 `QObjectPrivate` class 又是我们理解 QT kernel source code 的基础，这个对象包含了每一个 QT 对象中的数据成员，好了，让我们首先从理解 `QObject` 的数据存储代码开始我的 QT Kernel Source Code 之旅。

敬请关注下一节：`QObject` 对象数据存储

=====

==

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (二): 对象数据存储(A)

前言，为什么先说这个？

我们知道，在 C++ 中，几乎每一个类(class)中都需要有一些类的成员变量(class member variable)，在通常情况下的做法如下：

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

就是在类定义的时候，直接把类成员变量定义在这里，甚至于，把这些成员变量的存取范围直接定义成是 public 的，您是不是这样做的呢？

在 QT 中，却几乎都不是这样做的，那么，QT 是怎么做的呢？

几乎每一个 C++ 的类中都会保存许多的数据，要想读懂别人写的 C++ 代码，就一定需要知道每一个类的数据是如何存储的，是什么含义，否则，我们不可能读懂别人的 C++ 代码。在这里也就是说，要想读懂 QT 的代码，第一步就必须先搞清楚 QT 的类成员数据是如何保存的。

为了更容易理解 QT 是如何定义类成员变量的，我们先说一下 QT 2.x 版本中的类成员变量定义方法，因为在 2.x 中的方法非常容易理解。然后在介绍 QT 4.4 中的类成员变量定义方法。

QT 2.x 中的方法

在定义 class 的时候(在.h 文件中)，只包含有一个类成员变量，只是定义一个成员数据指针，然后由这个指针指向一个数据成员对象，这个数据成员对象包含所有这个 class 的成员数据，然后在 class 的实现文件(.cpp 文件)中，定义这个私有数据成员对象。示例代码如下：

```
//-----
-----
// File name: person.h

struct PersonalDataPrivate; // 声明私有数据成员类型

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;
}

```

在最初学习 QT 的时候，我也觉得这种方法很麻烦，但是随着使用的增多，我开始很喜欢这个方法了，而且，现在我写的代码，基本上都会用这种方法。具体说来，它有如下优点：

* 减少头文件的依赖性

把具体的数据成员都放到 cpp 文件中去，这样，在需要修改数据成员的时候，只需要改 cpp 文件而不需要头文件，这样就可以避免一次因为头文件的修改而导致所有包含了这个文件的文件全部重新编译一次，尤其是当这个头文件是非常底层的头文件和项目非常庞大的时候，优势明显。

同时，也减少了这个头文件对其他头文件的依赖性。可以把只在数据成员中需要

用到的在 cpp 文件中 include 一次就可以，在头文件中就可以尽可能的减少 include 语句

* 增强类的封装性

这种方法增强了类的封装性，无法再直接存取类成员变量，而必须写相应的 get/set 成员函数来做这些事情。

关于这个问题，仁者见仁，智者见智，每个人都有不同的观点。有些人就是喜欢把类成员变量都定义成 public 的，在使用的时候方便。只是我个人不喜欢这种方法，当项目变得很大的时候，有非常多的人一起在做这个项目的时候，自己所写的代码处于底层有非常多的人需要使用 (#include) 的时候，这个方法的弊端就充分的体现出来了。

还有，我不喜欢 QT 2.x 中把数据成员的变量名都定义成只有一个字母，d，看起来很不直观，尤其是在 search 的时候，很不方便。但是，QT kernel 中的確就是这么干的。

那么，在最新的 QT4 里面是如何实现的呢？请关注下一节。

Inside Qt Series (三): 对象数据存储(B)

QT 4.4.x 中的方法

在 QT 4.4 中，类成员变量定义方法的出发点没有变化，只是在具体的实现手段上发生了非常大的变化，下面具体来看。

在 QT 4.4 中，使用了非常多的宏来做事，这凭空的增加了理解 QT source code 的难度，不知道他们是不是从 MFC 学来的。就连在定义类成员数据变量这件事情上，也大量的使用了宏。

在这个版本中，类成员变量不再是给每一个 class 都定义一个私有的成员，而是把这一项 common 的工作放到了最基础的基类 QObject 中，然后定义了一些相关的方法来存取，好了，让我们进入具体的代码吧。

```
//-----
// file name: qobject.h

class QObjectData
{
public:
virtual ~QObjectData() = 0;

// 省略
};
```

```

class QObject
{
Q_DECLARE_PRIVATE(QObject)

public:

QObject(QObject *parent=0);

protected:

QObject(QObjectPrivate &dd, QObject *parent = 0);
QObjectData *d_ptr;
}

```

这些代码就是在 `qobject.h` 这个头文件中的。在 `QObject` class 的定义中，我们看到，数据员的定义为：`QObjectData *d_ptr;` 定义成 `protected` 类型的就是要让所有的派生类都可以存取这个变量，而在外部却不可以直接存取这个变量。而 `QObjectData` 的定义却放在了这个头文件中，其目的就是为了要所有从 `QObject` 继承出来的类的成员变量也都相应的要在 `QObjectData` 这个 class 继承出来。而纯虚的析构函数又决定了两件事：

- * 这个 class 不能直接被实例化。换句话说就是，如果你写了这么一行代码，`new QObjectData`，这行代码一定会出错，`compile` 的时候是无法过关的。
- * 当 `delete` 这个指针变量的时候，这个指针变量是指向的任意从 `QObjectData` 继承出来的对象的时候，这个对象都能被正确 `delete`，而不会产生错误，诸如，内存泄漏之类的。

我们再来看看这个宏做了什么，`Q_DECLARE_PRIVATE(QObject)`

```

#define Q_DECLARE_PRIVATE(Class) \
inline Class##Private* d_func() { return \
reinterpret_cast<Class##Private *>(d_ptr); } \
inline const Class##Private* d_func() const { return \
reinterpret_cast<const Class##Private *>(d_ptr); } \
friend class Class##Private;

```

这个宏主要是定义了两个重载的函数，`d_func()`，作用就是把在 `QObject` 这个 class 中定义的数据成员变量 `d_ptr` 安全的转换成为每一个具体的 class 的数据成员类型指针。我们看一下在 `QObject` 这个 class 中，这个宏展开之后的情况，就一目了然了。

`Q_DECLARE_PRIVATE(QObject)` 展开后，就是下面的代码：

```

inline QObjectPrivate* d_func() { return \
reinterpret_cast<QObjectPrivate *>(d_ptr); }

```

```
inline const QObjectPrivate* d_func() const
{ return reinterpret_cast<const QObjectPrivate
*>(&d_ptr); } \
friend class QObjectPrivate;
```

宏展开之后，新的问题又来了，这个 `QObjectPrivate` 是从哪里来的？在 `QObject` 这个 class 中，为什么不直接使用 `QObjectData` 来数据成员变量的类型？

还记得我们刚才说过吗，`QObjectData` 这个 class 的析构函数的纯虚函数，这就说明这个 class 是不能实例化的，所以，`QObject` 这个 class 的成员变量的实际类型，这是从 `QObjectData` 继承出来的，它就是 `QObjectPrivate`！

这个 class 中保存了许多非常重要而且有趣的东西，其中包括 QT 最核心的 signal 和 slot 的数据，属性数据，等等，我们将会在后面详细讲解，现在我们来看一下它的定义：

下面就是这个 class 的定义：

```
class QObjectPrivate : public QObjectData
{
Q_DECLARE_PUBLIC(QObject)

public:

QObjectPrivate(int version = QObjectPrivateVersion);
virtual ~QObjectPrivate();
```

// 省略

```
}
```

那么，这个 `QObjectPrivate` 和 `QObject` 是什么关系呢？他们是如何关联在一起的呢？

=====

==

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。
作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (四): 对象数据存储(C)

接上节，让我们来看看这个 `QObjectPrivate` 和 `QObject` 是如何关联在一起的。

```

//-----
// file name: qobject.cpp

QObject::QObject(QObject *parent)
: d_ptr(new QObjectPrivate)
{
// .....
}

QObject::QObject(QObjectPrivate &dd, QObject *parent)
: d_ptr(&dd)
{
// .....
}

```

怎么样，是不是一目了然呀？

从第一个构造函数可以很清楚的看出来，`QObject` class 中的 `d_ptr` 指针将指向一个 `QObjectPrivate` 的对象，而 `QObjectPrivate` 这个 class 是从 `QObjectData` 继承出来的。

这第二个构造函数干什么用的呢？从 `QObject` class 的定义中，我们可以看到，这第二个构造函数是被定义为 `protected` 类型的，这说明，这个构造函数只能被继承的 class 使用，而不能使用这个构造函数来直接构造一个 `QObject` 对象，也就是说，如果写一条下面的语句， 编译的时候是会失败的，

```
new QObject(*new QObjectPrivate, NULL)
```

为了看的更清楚，我们以 `QWidget` 这个 class 为例说明。

`QWidget` 是 QT 中所有 UI 控件的基类，它直接从 `QObject` 继承而来，

```

class QWidget : public QObject, public QPaintDevice
{
Q_OBJECT
Q_DECLARE_PRIVATE(QWidget)
// .....
}

```

我们看一个这个 class 的构造函数的代码：

```

QWidget::QWidget(QWidget *parent, Qt::WindowFlags f)
: QObject(*new QWidgetPrivate, 0), QPaintDevice()
{
d_func()->init(parent, f);

```

```
}
```

非常清楚，它调用了基类 QObject 的保护类型的构造函数，并且以 *new QWidgetPrivate 作为第一个参数传递进去。也就是说，基类(QObject)中的 d_ptr 指针将会指向一个 QWidgetPrivate 类型的对象。

再看 QWidgetPrivate 这个 class 的定义：

```
class QWidgetPrivate : public QObjectPrivate
{
Q_DECLARE_PUBLIC(QWidget)
// .....
}
```

好了，这就把所有的事情都串联起来了。

关于 QWidget 构造函数中的唯一的语句 d_func() -> init(parent, f) 我们注意到在 class 的定义中有这么一句话： Q_DECLARE_PRIVATE(QWidget)

我们前面讲过这个宏，当把这个宏展开之后，就是这样的：

```
inline QWidgetPrivate* d_func() { return
reinterpret_cast<QWidgetPrivate*>(&d_ptr); }
inline const QWidgetPrivate* d_func() const
{ return reinterpret_cast<const QWidgetPrivate*>(&d_ptr); }
friend class QWidgetPrivate;
```

很清楚，它就是把 QObject 中定义的 d_ptr 指针转换为 QWidgetPrivate 类型的指针。

小结：

要理解 QT Kernel 的 code，就必须要知道 QT 中每一个 Object 内部的数据是如何保存的，而 QT 没有象我们平时写 code 一样，把所有的变量直接定义在类中，所以，不搞清楚这个问题，我们就无法理解一个相应的 class。其实，在 QT4.4 中的类成员数据的保存方法在本质是与 QT2.x 中的是一样的，就是在 class 中定义一个成员数据的指针，指向成员数据集合对象(这里是一个 QObjectData 或者是其派生类)。初始化这个成员变量的办法是定义一个 保护类型的构造函数，然后在派生类的构造函数 new 一个派生类的数据成员，并将这个新对象赋值给 QObject 的数据指针。在使用的时候，通过预先定义个宏里面的一个 inline 函数来把数据指针在安全类型转换，就可以使用了。

=====

==

声明:

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。
作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (五): 元对象系统(Meta-Object System)

从本节开始，我们讲解 QT Meta-Object System 的功能，以及实现。

在使用 Qt 开发的过程中，大量的使用了 signal 和 slot。比如，响应一个 button 的 click 事件，我们一般都写如下的代码：

```
class MyWindow : public QWidget
{
Q_OBJECT
public:
MyWindow(QWidget* parent) : QWidget(parent)
{
QPushButton* btnStart = new QPushButton("start", this);
connect(btnStart, SIGNAL(clicked()), 
SLOT(slotStartClicked()));
}

private slots:
void slotStartClicked();

void MyWindow:: slotStartClicked()
{
// 省略
}
```

在这段代码中，我们把 btnStart 这个 button 的 clicked() 信号和 MyWindow 的 slotStartClicked() 这个槽相连接，当 btnStart 这个 button 被用户按下(click)的时候，就会发出一个 clicked() 的信号，然后，MyWindow:: slotStartClicked() 这个 slot 函数就会被调用用来响应 button 的 click 事件。

这段代码是最为典型的 signal/slot 的应用实例，在实际的工作过程中，signal/slot 还有更为广泛的应用。准确的说，signal/slot 是 QT 提供的一种在对象间进行通讯的技术，那么，这个技术在 QT 中是如何实现的呢？

这就是 QT 中的元对象系统(Meta Object System)的作用，为了更好的理解它，让我先来对它的功能做一个回顾，让我们一起来揭开它神秘的面纱。

Meta-Object System 的基本功能

Meta Object System 的设计基于以下几个基础设施：

- * QObject 类

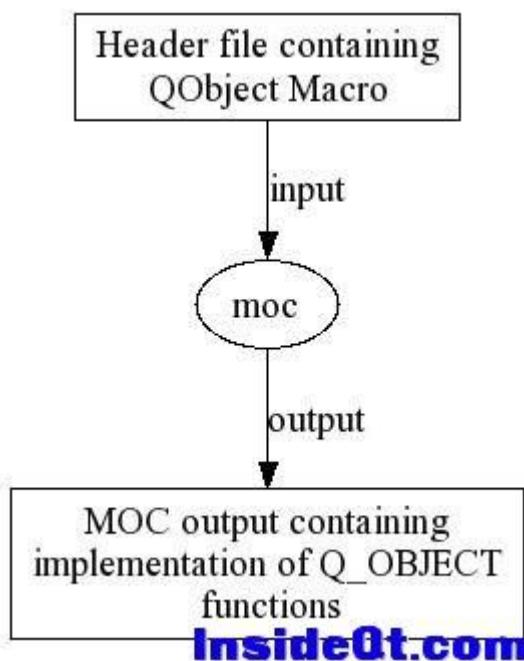
作为每一个需要利用元对象系统的类的基类

- * 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 Compiler)，

moc 分析 C++ 源文件，如果它发现在一个头文件(header file)中包含 Q_OBJECT 宏定义，然后动态的生成另外一个 C++ 源文件，这个新的源文件包含 Q_OBJECT 的实现代码，这个新的 C++ 源文件也会被编译、链接到这个类的二进制代码中去，因为它也是这个类的完整的一部分。通常，这个新的 C++ 源文件会在以前的 C++ 源文件名前面加上 moc_ 作为新文件的文件名。其具体过程如下图所示：



除了提供在对象间进行通讯的机制外，元对象系统还包含以下几种功能：

- * QObject::metaObject() 方法

它获得与一个类相关联的 meta-object

* QMetaObject::className() 方法

在运行期间返回一个对象的类名，它不需要本地 C++ 编译器的 RTTI (run-time type information) 支持

* QObject::inherits() 方法

它用来判断生成一个对象类是不是从一个特定的类继承出来，当然，这必须是在 QObject 类的直接或者间接派生类当中

* QObject::tr() and QObject::trUtf8()

这两个方法为软件的国际化翻译字符串

* QObject::setProperty() and QObject::property()

这两个方法根据属性名动态的设置和获取属性值

除了以上这些功能外，它还使用 qobject_cast() 方法在 QObject 类之间提供动态转换，qobject_cast() 方法的功能类似于标准 C++ 的 dynamic_cast()，但是 qobject_cast() 不需要 RTTI 的支持，在一个 QObject 类或者它的派生类中，我们可以不定义 Q_OBJECT 宏。如果我们在一个类中没有定义 Q_OBJECT 宏，那么在这里所提到的相应的功能在这个类中也不能使用，从 meta-object 的观点来说，一个没有定义 Q_OBJECT 宏的类与它最接近的那个祖先类是相同的，那就是所，QMetaObject::className() 方法所返回的名字并不是这个类的名字，而是与它最接近的那个祖先类的名字。所以，我们强烈建议，任何从 QObject 继承出来的类都定义 Q_OBJECT 宏。

下一节，我们来了解另一个重要的工具：Meta-Object Compiler

=====

==

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛 (InsideQt.com) 原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。

作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (六)：元对象编译器 – Meta Object Compiler (moc)

元对象编译器用来处理 QT 的 C++ 扩展，moc 分析 C++ 源文件，如果它发现在一个头文件 (header file) 中包含 Q_OBJECT 宏定义，然后动态的生成另外一个 C++ 源文件，这个新的源文件包含 Q_OBJECT 的实现代码，这个新的 C++ 源文件也会被编译、链接到这个类的二进制代码中去，因为它也是这个类的完整的一部分。通常，这个新的 C++ 源文件会在以前的 C++ 源文件名前面加上 moc_ 作为新文件的文件名。

如果使用 qmake 工具来生成 Makefile 文件，所有需要使用 moc 的编译规则都会给自动的包含到 Makefile 文件中，所以对程序员来说不需要直接的使用 moc

除了处理信号和槽之外，moc 还处理属性信息，`Q_PROPERTY()`宏定义类的属性信息，而`Q_ENUMS()`宏则定义在一个类中的枚举类型列表。`Q_FLAGS()`宏定义在一个类中的 flag 枚举类型列表，`Q_CLASSINFO()`宏则允许你在一个类的 meta 信息中插入 name/value 对。

由 moc 所生成的文件必须被编译和链接，就象你自己写的另外一个 C++ 文件一样，否则，在链接的过程中就会失败。

Code example:

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_PROPERTY(Priority priority READ priority WRITE
               setPriority)
    Q_ENUMS(Priority)
    Q_CLASSINFO("Author", "Oscar Peterson")
    Q_CLASSINFO("Status", "Active")

public:
    enum Priority { High, Low, VeryHigh, VeryLow };

    MyClass(QObject *parent = 0);
    virtual ~MyClass();

    void setPriority(Priority priority);
    Priority priority() const;
};

=====
```

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛 (InsideQt.com) 原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。
作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (七): Signal & Slot

本节介绍 Signal 和 slot 的基本知识。

信号和槽是用来在对象间通讯的方法，当一个特定事件发生的时候，signal 会被 emit 出来，slot 调用是用来响应相应的 signal 的。QT 对象已经包含了许多预定义的 signal，但我们总是可以在派生类中添加新的 signal。QT 对象中

也已经包含了许多预定义的 signal，但我们在派生类中添加新的 slot 来处理我们感兴趣的 signal.

signal 和 slot 机制是类型安全的，signal 和 slot 必须互相匹配(实际上，一个 slot 的参数可以比对应的 signal 的参数少，因为它可以忽略多余的参数)。signal 和 slot 是松散的配对关系，发出 signal 的对象不关心是那个对象链接了这个 signal，也不关心是那个或者有多少 slot 链接到了这个 signal。QT 的 signal 和 slot 机制保证了，如果一个 signal 和 slot 相链接，slot 会在正确的时机被调用，并且是使用正确的参数。Signal 和 slot 都可以携带任何数量和类型的参数，他们都是类型安全的。

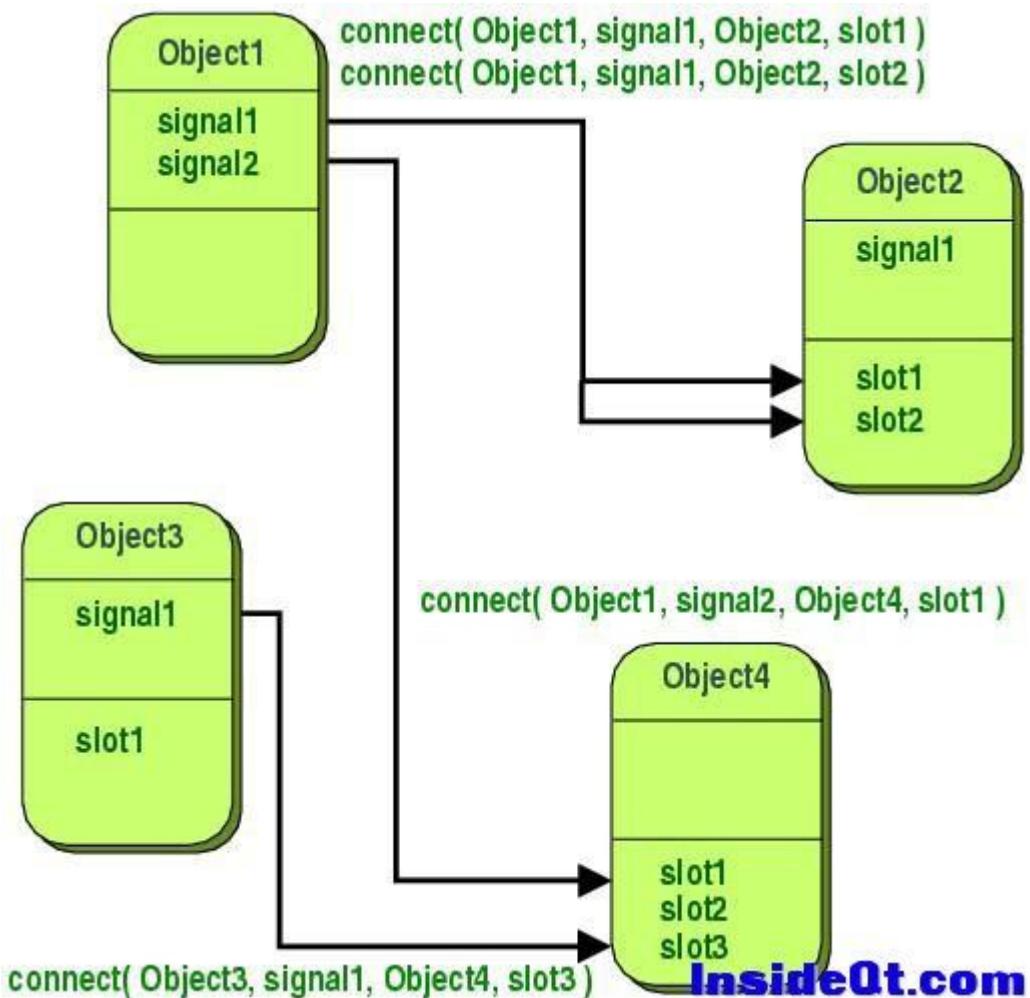
所有从 QObject 直接或者间接继承出来的类都能包含信号和槽，当一个对象的状态发生变化的时候，信号就可以被 emit 出来，这可能是某个其它的对象所关心的。这个对象并不关心有那个对象或者多少个对象链接到这个信号了，这是真实的信息封装，它保证了这个对象可以作为一个软件组件来被使用。

槽(slot)是用来接收信号的，但同时他们也是一个普通的类成员函数，就象一个对象不关心有多少个槽链接到了它的某个信号，一个对象也不关心一个槽链接了多少个信号。这保证了用 QT 创建的对象是一个真实的独立的软件组件。

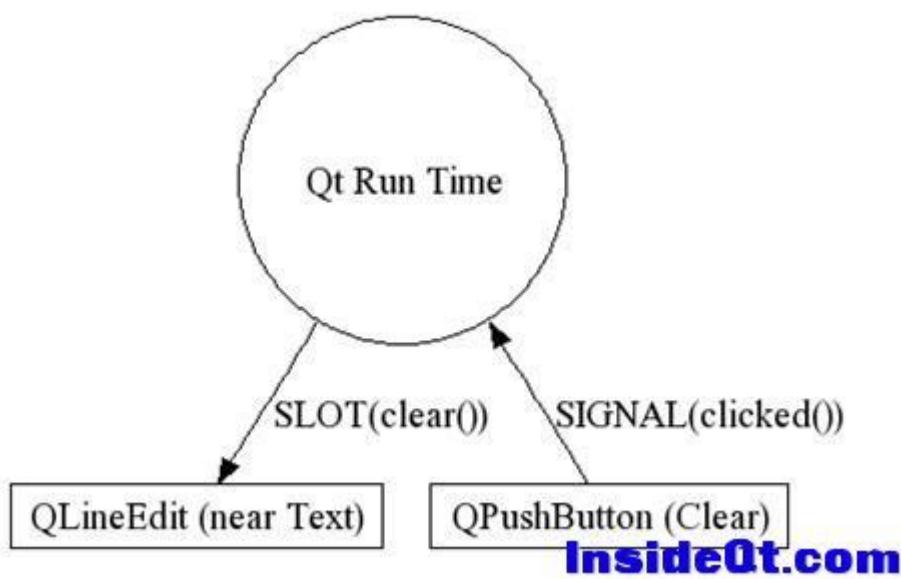
一个信号可以链接到多个槽，一个槽也可以链接多个信号。同时，一个信号也可以链接到另外一个信号。所有使用了信号和槽的类都必须包含 Q_OBJECT 宏，而且这个类必须从 QObject 类派生(直接或者间接派生)出来，

当一个 signal 被 emit 出来的时候，链接到这个 signal 的 slot 会立刻被调用，就好像是一个函数调用一样。当这件事情发生的时候，signal 和 slot 机制与 GUI 的事件循环完全没有关系，当所有链接到这个 signal 的 slot 执行完成之后，在 emit 代码行之后的代码会立刻被执行。当有多个 slot 链接到一个 signal 的时候，这些 slot 会一个接着一个的、以随机的顺序被执行。

Signal 代码会由 moc 自动生成，开发人员一定不能在自己的 C++ 代码中实现它，并且，它永远都不能有返回值。Slot 其实就是一个普通的类函数，并且可以被直接调用，唯一特殊的地方是它可以与 signal 相链接。C++ 的预处理器更改或者删除 signal, slot, emit 关键字，所以，对于 C++ 编译器来说，它处理的是标准的 C++ 源文件。



如下图所示：假定 QPushButton 的 signal clicked() 已经和 QLineEdit 的 signal clear() 连接成功，那么当 QPushButton 的 clicked() signal 被 emit 出来的时候，QLineEdit 的 clear() slot 就会被调用。



=====

==

声明:

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。

作者保留版权，未经作者同意，不得用于任何商业用途

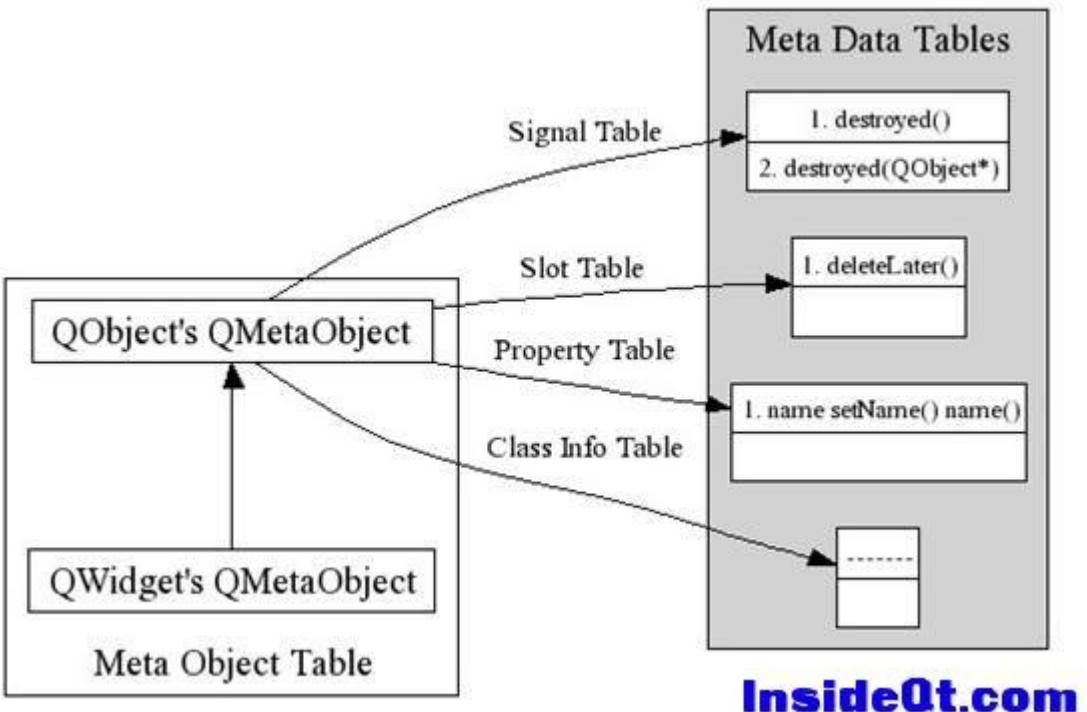
Inside Qt Series (八): Meta Object Class overview

前面我们介绍了 Meta Object 的基本功能，和它支持的最重要的特性之一：Signal & Slot 的基本功能。现在让我们来进入 Meta Object 的内部，看看它是如何支持这些能力的。

Meta Object 的所有数据和方法都封装在一个叫 QMetaObject 的类中。它包含并且可以查询一个 Qt 类的 meta 信息，meta 信息包含以下几种：

- * 信号表(signal table)，其中有这个对应的 QT 类的所有 Signal 的名字
- * 槽表(slot table)，其中有这个对应的 QT 类中的所有 Slot 的名字。
- * 类信息表(class info table)，包含这个 QT 类的类型信息
- * 属性表(property table)，其中有这个对应的 QT 类中的所有属性的名字。
- * 指向 parent meta object 的指针(pointers to parent meta object)

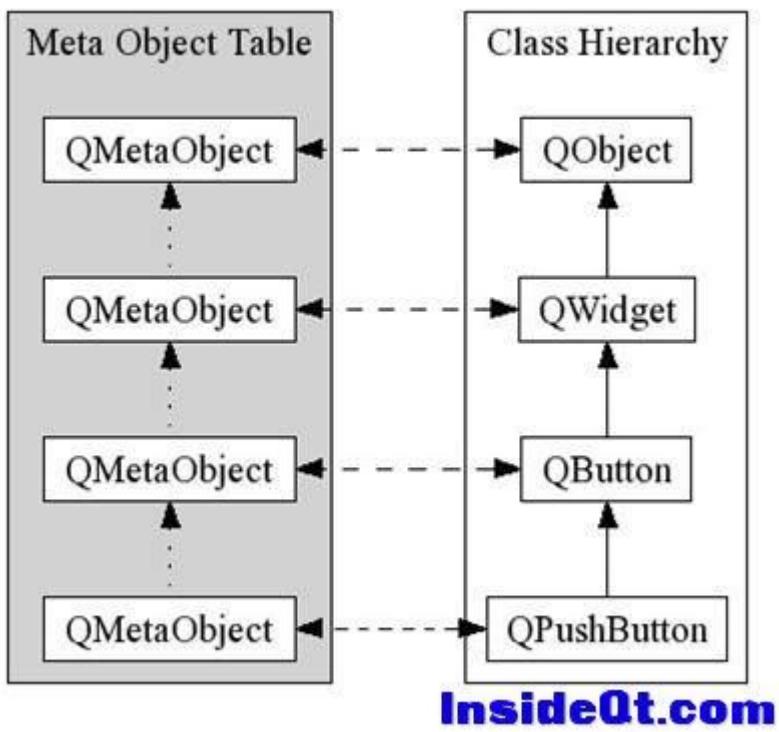
请参考下图，Qt Meta Data Tables:



QMetaObject 对象与 QT 类之间的关系:

- * 每一个 QMetaObject 对象包含了与之相对应的一个 QT 类的元信息
- * 每一个 QT 类(QObject 以及它的派生类) 都有一个与之相关联的静态的 (static) QMetaObject 对象 (注: class 的定义中必须有 Q_OBJECT 宏, 否则就没有这个 Meta Object)
- * 每一个 QMetaObject 对象保存了与它相对应的 QT 类的父类的 QMetaObject 对象的指针。 或者, 我们可以这样说: “每一个 QMetaObject 对象都保存了一个其父亲(parent)的指针”. 注意: 严格来说, 这种说法是不正确的, 最起码 是不严谨的。

请参考下图, Qt Meta Class 与 Qt class 之间的对应关系:



InsideQt.com

Q_OBJECT 宏

Meta Object 的功能实现，这个宏立下了汗马功劳。首先，让我们来看看这个宏是如何定义的：

```
#define Q_OBJECT \
public: \
Q_OBJECT_CHECK \
static const QMetaObject staticMetaObject; \
virtual const QMetaObject *metaObject() const; \
virtual void *qt_metacast(const char *); \
QT_TR_FUNCTIONS \
virtual int qt_metacall(QMetaObject::Call, int, void **); \
private:
```

这里，我们先忽略 Q_OBJECT_CHECK 和 QT_TR_FUNCTIONS 这两个宏。

我们看到，首先定义了一个静态类型的类变量 staticMetaObject，然后有一个获取这个对象指针的方法 metaObject()。这里最重要的就是类变量 staticMetaObject 的定义。这说明所有的 QObject 的对象都会共享这一个 staticMetaObject 类变量，靠它来完成所有信号和槽的功能，所以我们就有必要来仔细的看看它是怎么回事了。

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。
作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (九): QMetaObject class data members

我们来看一下 QMetaObject 的定义，我们先看一下 QMetaObject 对象中包含的成员数据。

```
struct Q_CORE_EXPORT QMetaObject
{
// .....
struct { // private data
const QMetaObject *superdata;
const char *stringdata;
const uint *data;
const void *extradata;
} d;
};
```

上面的代码就是 QMetaObject 类所定义的全部数据成员。就是这些成员记录了所有 signal, slot, property, class information 这么多的信息。下面让我们来逐一解释这些成员变量：

const QMetaObject *superdata:

这个变量指向与之对应的 QObject 类的父类，或者是祖先类的 QMetaObject 对象。

如何理解这一句话呢？我们知道，每一个 QMetaObject 对象，一定有一个与之相对应的 QObject 类（或者由其直接或间接派生出的子类），注意：这里是类，不是对象。

那么每一个 QObject 类（或其派生类）可能有一个父类，或者父类的父类，或者很多的继承层次之前的祖先类。或者没有父类（QObject）。那么 superdata 这个变量就是指向与其最接近的祖先类中的 QMetaObject 对象。对于 QObject 类 QMetaObject 对象来说，这是一个 NULL 指针，因为 QObject 没有父类。

下面，让我们来举例说明：

```
class Animal : public QObject
{
Q_OBJECT
//.....
```

```
};

class Cat : public Animal
{
Q_OBJECT
// .....
}
```

那么，Cat::staticMetaObject.d.superdata 这个指针变量指向的对象是 Animal::staticMetaObject
而 Animal::staticMetaObject.d.superdata 这个指针变量指向的对象是 QObject::staticMetaObject.
而 QObject::staticMetaObject.d.superdat 这个指针变量的值为 NULL。

但如果我们把上面 class 的定义修改为下面的定义，就不一样了：

```
class Animal : public QObject
{

// Q_OBJECT, 这个 class 不定义这个
// .....
};

class Cat : public Animal
{
Q_OBJECT
// .....
}
```

那么，Cat::staticMetaObject.d.superdata 这个指针变量指向的对象是 QObject::staticMetaObject
因为 Animal::staticMetaObject 这个对象是不存在的。

```
const char *stringdata:
```

顾名思义，这是一个指向 string data 的指针。但它和我们平时所使用的一般的字符串指针却很不一样，我们平时使用的字符串指针只是指向一个字符串的指针，而这个指针却指向的是很多个字符串。那么它不就是字符串数组吗？哈哈，也不是。因为 C++ 的字符串数组要求数组中的每一个字符串拥有相同的长度，这样才能组成一个数组。那它是不是一个字符串指针数组呢？也不是，那它到底是什么呢？让我们来看一看它的具体值，还是让我们以 QObject 这个 class 的 QMetaObject 为例来说明吧。

下面是 `QObject::staticMetaObject.d.stringdata` 指针所指向的多个字符串数组，其实它就是指向一个连续的内存区，而这个连续的内存区中保存了若干个字符串。

```
static const char qt_meta_stringdata_QObject[] =
{
    "QObject\0\0destroyed(QObject*)\0destroyed()\0"
    "deleteLater()\0_q_reregisterTimers(void*)\0"
    "QString\0objectName\0parent\0QObject(QObject*)\0"
    "QObject()\0"
};
```

这个字符串都是些什么内容呀？有，Class Name, Signal Name, Slot Name, Property Name。看到这些大家是不是觉得很熟悉呀，对啦，他们就是 Meta System 所支持的最核心的功能属性了。

既然他们都是不等长的字符串，那么 Qt 是如何来索引这些字符串，以便于在需要的时候能正确的找到他们呢？第三个成员正式登场了。

Inside Qt Series (十): connect, 幕后的故事

我们都知道，把一个 signal 和 slot 连接起来，需要使用 `QObject` 类的 `connect` 方法，它的作用就是把一个 object 的 signal 和另外一个 object 的 slot 连接起来，以达到对象间通讯的目的。

`connect` 在幕后到底都做了些什么事情？为什么 `emit` 一个 signal 后，相应的 slot 都会被调用？好了，让我们来逐一解开其中的谜团。

SIGNAL 和 SLOT 宏定义

我们在调用 `connect` 方法的时候，一般都会这样写：

```
obj.connect(&obj, SIGNAL(destroyed()), &app, SLOT(aboutQt()));
```

我们看到，在这里 signal 和 slot 的名字都被包含在了两个大写的 SIGNAL 和 SLOT 中，这两个是什么呢？原来 SIGNAL 和 SLOT 是 Qt 定义的两个宏。好了，让我们先来看看这两个宏都做了写什么事情：

这里是这两个宏的定义：

```
# define SLOT(a)          " 1" #a
# define SIGNAL(a)        " 2" #a
```

原来 Qt 把 signal 和 slot 都转化成了字符串，并且还在这个字符串的前面加上了附加的符号，signal 前面加了' 2'，slot 前面加了' 1'。也就是说，我们前面写了下面的 `connect` 调用，在经过 `moc` 编译器转换之后，就便成了：

```
obj.connect(&obj, "2destroyed()", &app, "1aboutQt()");
```

当 connect 函数被调用了之后，都会去检查这两个参数是否是使用这两个宏正确的转换而来的，它检查的根据就是这两个前置数字，是否等于 1 或者是 2，如果不是， connect 函数当然就会失败啦！

然后，会去检查发送 signal 的对象是否有这个 signal，方法就是查找这个对象的 class 所对应的 staticMetaObject 对象中所包含 的 d.stringdata 所指向的字符串中是否包含这个 signal 的名字，在这个检查过程中，就会用到 d.data 所指向的那一串整数，通过这些整数 值来计算每一个具体字符串的起始地址。同理，还会使用同样的方法去检查 slot，看响应这个 signal 的对象是否包含有相应的 slot。这两个检查的任 何一个如果失败的话，connect 函数就失败了，返回 false.

前面的步骤都是在做一些必要的检查工作，下一步，就是要把发送 signal 的对象和响应 signal 的对象关联起来。在 QObject 的私有数据类 QObjectPrivate 中，有下面这些数据结构来保存这些信息：

```
class QObjectPrivate : public QObjectData
{
    struct Connection
    {
        QObject *receiver;
        int method;
        uint connectionType : 3; // 0 == auto, 1 == direct, 2 == queued,
        4 == blocking
        QBasicAtomicPointer<int> argumentTypes;
    };

    typedef QList<Connection> ConnectionList;

    QObjectConnectionListVector *connectionLists;

    struct Sender
    {
        QObject *sender;
        int signal;
        int ref;
    };

    QList<Sender> senders;
}
```

在发送 signal 的对象中，每一个 signal 和 slot 的 connection，都会创建一个 QObjectPrivate::Connection 对象，并且把这个对象保存到 connectionList 这个 Vector 里面去。

在响应 signal 的对象中，同样，也是每一个 signal 和 slot 的 connection，都会一个创建一个 Sender 对象，并且把这个对象附加在 Senders 这个列表中。

以上就是 connect 的过程，其中，创建 QObjectPrivate::Connection 对象和 Sender 对象的过程有一点点复杂，需要仔细思考才可以，有兴趣的朋友可以去读一下源代码。

=====

==

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。

作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (十一): emit, 幕后的故事

当我们写下一下 emit signal 代码的时候，与这个 signal 相连接的 slot 就会被调用，那么这个调用是如何发生的呢？让我们来逐一解开其中的谜团。

让我们来看一段例子代码：

```
class ZMytestObj : public QObject
{
Q_OBJECT
signals:
void sigMenuClicked();
void sigBtnClicked();
};
```

MOC 编译器在做完预处理之后的代码如下：

```
// SIGNAL 0
void ZMytestObj::sigMenuClicked()
{
QMetoObject::activate(this, &staticMetaObject, 0, 0);
}

// SIGNAL 1
void ZMytestObj::sigBtnClicked()
{
QMetoObject::activate(this, &staticMetaObject, 1, 0);
}
```

哈哈，看到了吧，每一个 signal 都会被转换为一个与之相对应的成员函数。也就是说，当我们写下这样一行代码：

```
emit sigBtnClicked();
```

当程序运行到这里的时候，实际上就是调用了 void ZMytestObj::sigBtnClicked() 这个函数。

大家注意比较这两个函数的函数体，

void ZMytestObj::sigMenuClicked() void ZMytestObj::sigBtnClicked(),
它们唯一的区别就是调用 QMetaObject::activate 函数时给出的参数不同，一个
个是 0，一个是 1，它们的含义是什么呢？它们表示是这个类中的第几个 signal
被发送出来了，回头再去看头文件就会发现它们就 是在这个类定义中，signal
定义出现的顺序，这个参数可是非常重要的，它直接决定了进入这个函数体之后
所发生的事情。

当执行流程进入到 QMetaObject::activate 函数中后，会先从 connectionLists
这个变量中取出与这个 signal 相对应的 connection list，它根据的就是刚才
所传入进来的 signal index。这个 connection list 中保存了所有和这个 signal
相链接的 slot 的信息，每一对 connection(即： signal 和 slot 的连接)是这
个 list 中的一项。

在每个具体的链接记录中，还保存了这个链接的类型，是自动链接类型，还是
队列链接类型，或者是阻塞链接类型，不同的类型处理方法还不一样的。这里，
我们就只说一下直接调用的类型。

对于直接链接的类型，先找到接收这个 signal 的对象的指针，然后是处理这个
signal 的 slot 的 index，已经是否有需要处理的参数，然后就使用这些信息去
调用 receiver 的 qt_metcall 方法。

在 qt_metcall 方法中就简单了，根据 slot 的 index，一个大 switch 语句，调
用相应的 slot 函数就 OK 了。

=====

==

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文
章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。
作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (十二): Qt 对象之间的父子关系

很多 C/C++初学者常犯的一个错误就是，使用 malloc、new 分配了一块内存却忘
记释放，导致内存泄漏。Qt 的对象模型提供了一种 Qt 对象之间的父 子关系，
当很多个对象都按一定次序建立起来这种父子关系的时候，就组织成了一棵树。

当 delete 一个父对象的时候，Qt 的对象模型机制保证了会自动的把 它的所有子对象，以及孙对象，等等，全部 delete，从而保证不会有内存泄漏的情况发生。

任何事情都有正反两面作用，这种机制看上去挺好，但是却会对很多 Qt 的初学者造成困扰，我经常给别人回答的问题是：1，new 了一个 Qt 对象之后，在什么情况下应该 delete 它？2，Qt 的析构函数是不是有 bug？3，为什么正常 delete 一个 Qt 对象却会产生 segment fault？等等诸如此类的问题，这篇文章就是针对这个问题的详细解释。

在每一个 Qt 对象中，都有一个链表，这个链表保存有它所有子对象的指针。当创建一个新的 Qt 对象的时候，如果把另外一个 Qt 对象指定为这个对象的父对象，那么父对象就会在它的子对象链表中加入这个子对象的指针。另外，对于任意一个 Qt 对象而言，在其生命周期的任何时候，都还可以通过 setParent 函数 重新设置它的父对象。当一个父对象在被 delete 的时候，它会自动的把它所有的子对象全部 delete。当一个子对象在 delete 的时候，会把它自己 从它的父对象的子对象链表中删除。

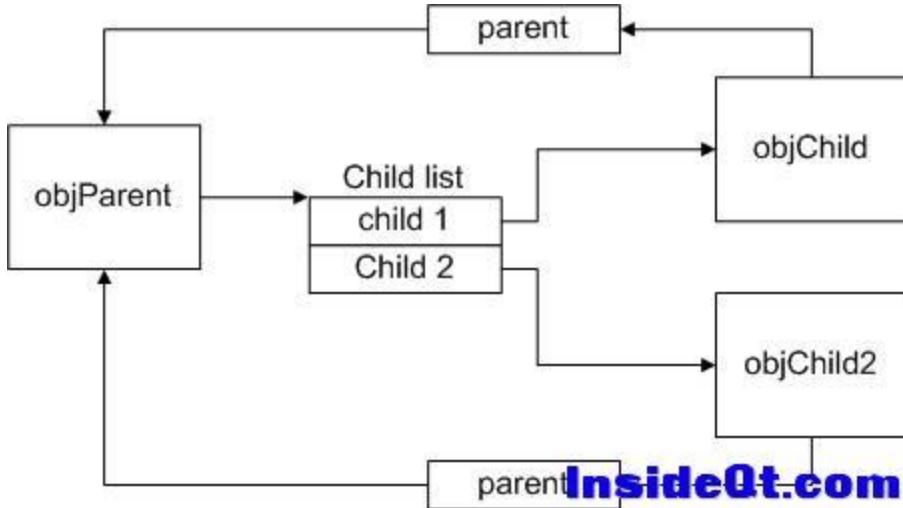
QWidget 是所有在屏幕上显示出来的界面对象的基类，它扩展了 Qt 对象的父子关系。一个 Widget 对象也就自然的成为其父 Widget 对象的子 Widget，并且显示在它的父 Widget 的坐标系统中。例如，一个对话框(dialog)上的按钮(button)应该是这个对话框的子 Widget。

关于 Qt 对象的 new 和 delete，下面我们举例说明。

例如，下面这一段代码是正确的：

```
int main()
{
    QObject* objParent = new QObject(NULL);
    QObject* objChild = new QObject(objParent);
    QObject* objChild2 = new QObject(objParent);
    delete objParent;
}
```

我们用一张图来描述这三个对象之间的关系：



在上述代码片段中，`objParent` 是 `objChild` 的父对象，在 `objParent` 对象中有一个子对象链表，这个链表中保存它所有子对象的指针，在这里，就是保存了 `objChild` 和 `objChild2` 的指针。在代码的结束部分，就只有 `delete` 了一个对象 `objParent`，在 `objParent` 对象的析构函数会遍历它的子对象链表，并且把它所有的子对象 (`objChild` 和 `objChild2`) 一一删除。所以上面这段代码是安全的，不会造成内存泄漏。

如果我们把上面这段代码改成这样，也是正确的：

```
int main()
{
    QObject* objParent = new QObject(NULL);
    QObject* objChild = new QObject(objParent);
    QObject* objChild2 = new QObject(objParent);
    delete objChild;
    delete objParent;
}
```

在这段代码中，我们就只看一下和上一段代码不一样的地方，就是在 `delete objParent` 对象之前，先 `delete objChild` 对象。在 `delete objChild` 对象的时候，`objChild` 对象会自动的把自己从 `objParent` 对象的子对象链表中删除，也就是说，在 `objChild` 对象被 `delete` 完成之后，`objParent` 对象就只有一个子对象 (`objChild2`) 了。然后在 `delete objParent` 对象的时候，会自动把 `objChild2` 对象也 `delete`。所以，这段代码也是安全的。

Qt 的这种设计对某些调试工具来说却是不友好的，比如 valgrind。比如上面这段代码，valgrind 工具在分析代码的时候，就会认为 `objChild2` 对象没有被正确的 `delete`，从而会报告说，这段代码存在内存泄漏。哈哈，我们知道，这个报告是不对的。

我们在看一看这一段代码：

```

int main()
{
    QWidget window;
    QPushButton quit("Exit", &window);
}

```

在这段代码中，我们创建了两个 widget 对象，第一个是 window，第二个是 quit，他们都是 Qt 对象，因为 QPushButton 是从 QWidget 派生出来的，而 QWidget 是从 QObject 派生出来的。这两个对象之间的关系是，window 对象是 quit 对象的父对象，由于他们 都会被分配在栈(stack)上面，那么 quit 对象是不是会被析构两次呢？我们知道，在一个函数体内部声明的变量，在这个函数退出的时候就会被析构，那 么在这段代码中，window 和 quit 两个对象在函数退出的时候析构函数都会被调用。那么，假设，如果是 window 的析构函数先被调用的话，它就会去 delete quit 对象；然后 quit 的析构函数再次被调用，程序就出错了。事实情况不是这样的，C++标准规定，本地对象的析构函数的调用顺序与他们的构造顺序相 反。那么在这段代码中，这就是 quit 对象的析构函数一定会比 window 对象的析构函数先被调用，所以，在 window 对象析构的时候，quit 对象已 经不存在了，不会被析构两次。

如果我们把代码改成这个样子，就会出错了，对照前面的解释，请你自己来分析一下吧。

```

int main()
{
    QPushButton quit("Exit");
    QWidget window;
    quit.setParent(&window);
}

```

但是我们自己在写程序的时候，也必须重点注意一项，千万不要 delete 子对象两次，就像前面这段代码那样，程序肯定就 crash 了。

最后，让我们来结合 Qt source code，来看看这 parent/child 关系是如何实现的。

在本专栏文章的第一部分“对象数据存储”，我们说到过，所有 Qt 对象的私有数据成员的基类是 QObjectData 类，这个类的定义如下：

```

typedef QList<QObject*> QObjectList;
class QObjectData
{
public:
    QObject *parent;
    QObjectList children;
}

```

```
// 忽略其它成员定义  
};
```

我们可以看到，在这里定义了指向 parent 的指针，和保存子对象的列表。其实，把一个对象设置成另一个对象的父对象，无非就是在操作这两个数据。把子对象中的这个 parent 变量设置为指向其父对象；而在父对象的 children 列表中加入子对象的指针。当然，我这里说的非常简单，在实际的代码中复杂的多，包含有很多条件判断，有兴趣的朋友可以自己去读一下 Qt 的源代码。

=====

==

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。

作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (十三): Qt/e 体系结构概述

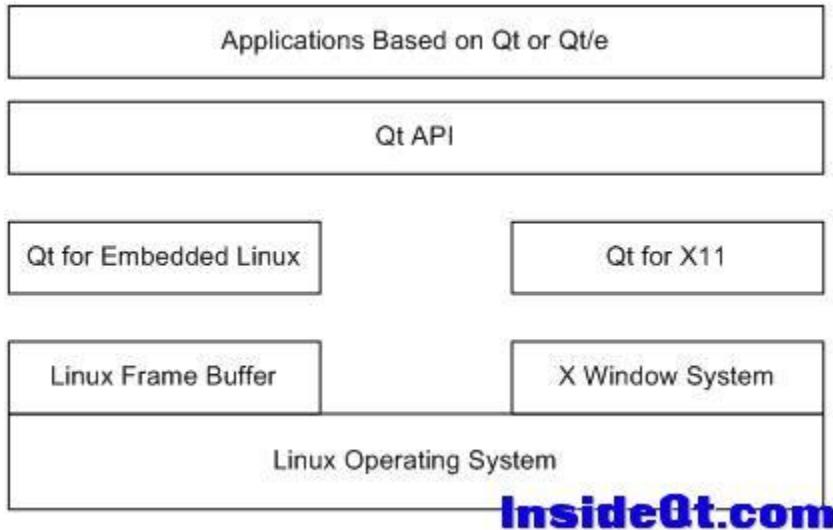
各位朋友好，从本节(Inside Qt Series 十三)开始，本专栏文章的内容开始转向 Qt for Embedded Linux 的技术内幕介绍。在后续内容中，我对 Qt for Embedded Linux 一律简称为 Qt/e，不再对这个 term 做更多的解释。需要您注意的一点是，在本系列文章中的任何部分，这个 term 都是指 Qt for Embedded Linux，而不是指 Qt for Windows CE。

这些内容所适用的软件版本是：Qt for Embedded Linux 4.5.1, Open Source edition

首先，让我们来看看 Qt/e 的系统结构介绍：

Qt for destop Linux 和 Qt for Embedded Linux 最大的区别就在于他们所依赖的底层显示基础的不同，这也导致了他们在体系结构上的差异。对于 Qt for desktop Linux 来说，底层的显示技术构建在 X Window System 之上，完全依赖于 X System，他们在下层完全是调用了 X Lib 的系统方法来把界面上的东西显示出来。

Qt for embedd linux 在这方面则完全不同，它并没有构建在 X Window 之上，而是构建在 Linux 的 Framebuffer 之上，把在界面上需要显示的内容直接写入了 framebuffer。因为在嵌入式系统上 把 X System 给省略了，这样会节省许多的系统开销。而直接写 framebuffer，又会加快显示速度。这种区别如图所示：



但就是这一个改变，导致了在 Qt/E 凭空多出了一个 Server 这么一层，这一层负责监听系统事件，尤其是键盘和鼠标事件，屏幕输出，管理 region，管理顶层窗口，管理光标和屏幕保护程序等等诸多功能。系统产生的键盘鼠标事件，首先就传给了这个 server application，然后 server 在根据具体的情况把这些事件分发给相应的应用程序。

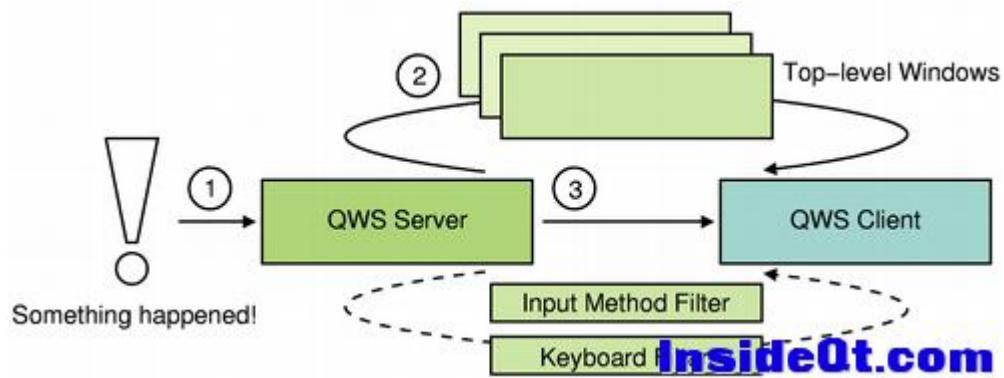
每一个 Qt/e 应用程序，都需要这样一个 server 存在。一个程序运行起来后，如是自己成为 Server 进程，就是连接到一个已经存在的 Server 进程。所以，第一个运行起来的 Qt/E 应用程序就会启动这个 server 让自己成为这个 Server 进程，后续运行的程序就会连接到这个 Server 来管理自己。

在 Server 端，每一个连接到 QWServer 的 client 都有一个 QWSClient 对象与之对应，这个对象主要记录了 client ID。在应用程序中每创建一个顶层窗口，那么在 server 端就会有创建一个 QWSWindow 实例来与之对应。

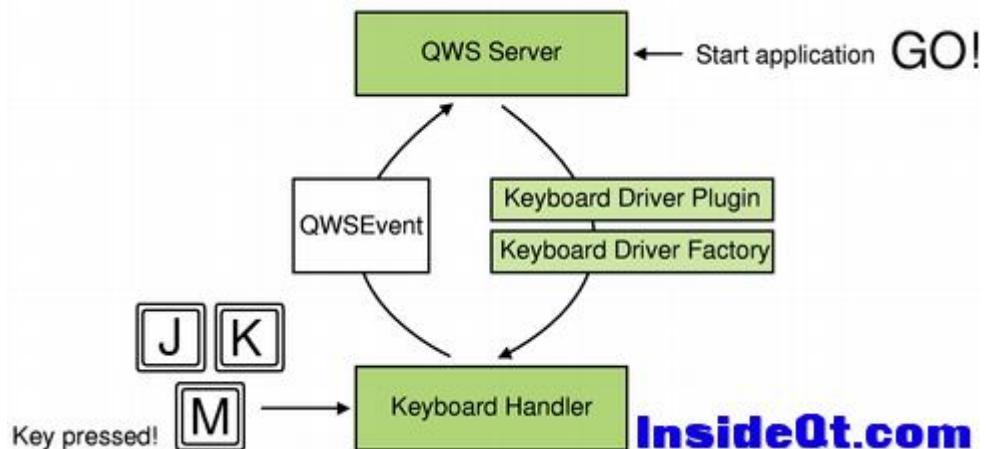
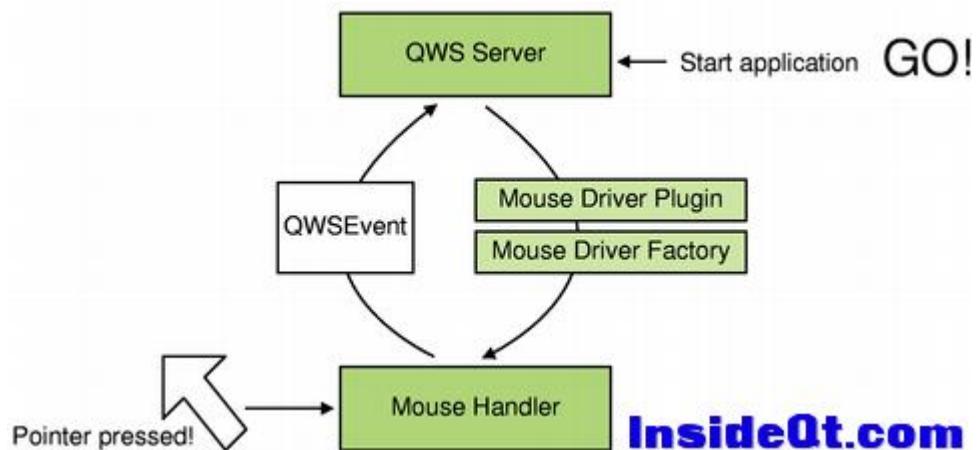
每一个 Server 实例都是由一个 QWSserver 类来实现的。

每当 Server 收到一个 event 的时候，它需要判断应该发送给那一个窗口，这时候，它就会从 QWSWindow 列表中去找，然后根据这个窗口去找对应的 client application，然后用一个 QSEvent 对象来包装这个 event，通过 socket 机制发送给具体的 client application。如果当前系统安装了一个输入法，那么每一次键盘事件产生的时候，都会去调用输入法的相应方法。

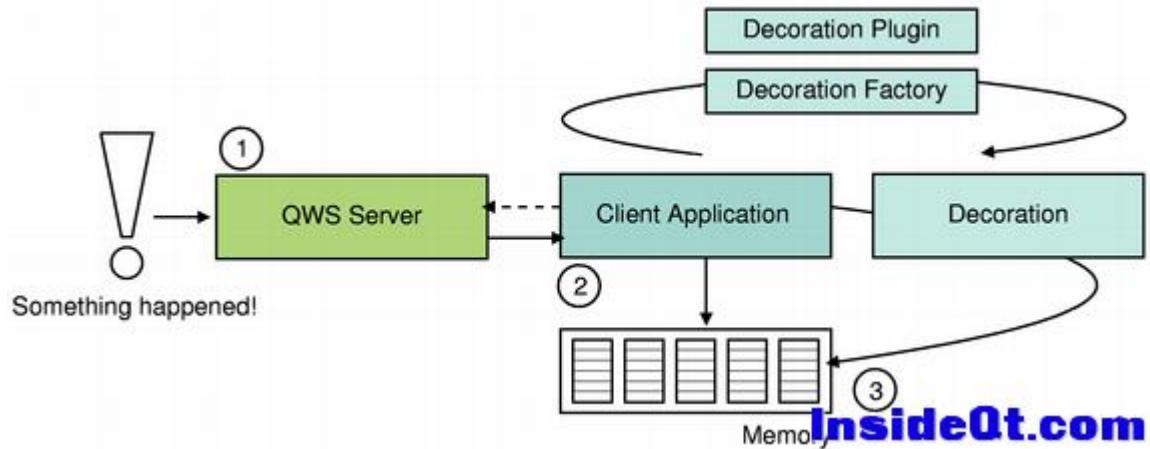
如图所示(取自 Qte 文档)：



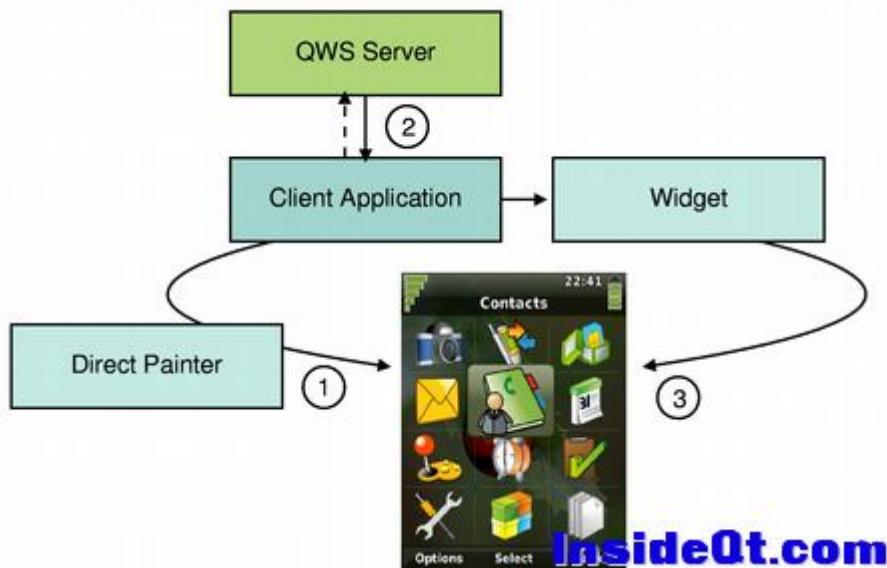
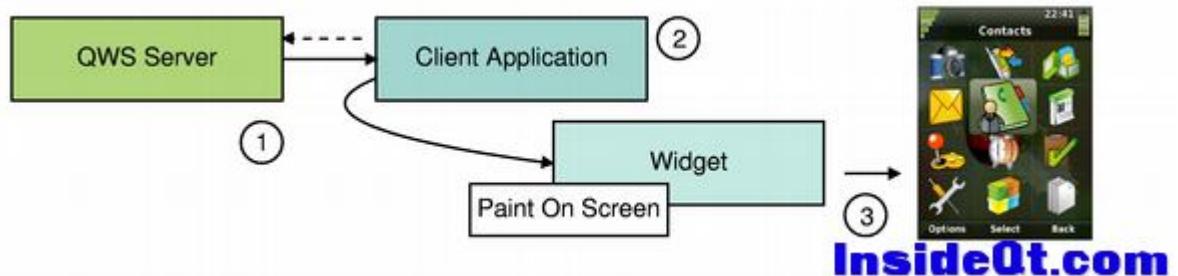
鼠标事件的处理和键盘事件的处理也符合上面的流程。鼠标驱动由一个 `QWSMouseHandler` 对象封装，键盘驱动由一个 `QWSKeyboardHandler` 封装。这两个驱动程序对象都会通过 Qt 的 plugin 机制加载。具体的鼠标和键盘事件发生之后，都会封装成为一个 `QWSEvent` 对象并发送给具体的 client。如图所示(取自 Qte 文档)：



图形输出，QWS 的缺省行为是每一个 widget 会把自己画在一块内存中，然后由 Server 负责把这块内存 copy 到 Linux 的 Framebuffer 上去，如图所示(取自 Qt 文档)：



但是对于大多数嵌入式系统来说，其中的显示子系统都是确定的，这样对于 client 应用程序来说，就可以直接输出到 Framebuffer 上面去。有两种 方法可以实现这一点，第一种是为每一个 Widget 都设置 Qt::WA_PaintOnScreen 属性，另一种是 QDPainter 来在 Framebuffer 中保留一块区域，如图所示(取自 Qt 文档)：



=====

==

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛(InsideQt.com)原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。
作者保留版权，未经作者同意，不得用于任何商业用途

Inside Qt Series (十四): Qt/e 输入法程序设计指南

注，本输入法设计指南针对 Qt for Embedded Linux 4.5.1，并且以中文输入法为例做说明，并且本文只是侧重于说明 Qt/Embedded 对输入法的支持接口，并不涉及到如何把键盘输入转换为中文所对应的编码方法。对其它 Qt 版本的适用性未曾验证。

大家都知道，Qt for Embedded Linux 是 Client/Server 结构，在 Server 端负责监听系统消息，尤其是键盘和鼠标消息，而输入法又是一个全局性的模块，所以在 Qt /Embedded 中，就把输入法的设计放在了 Server 这一层上。具体来说，就是，输入法是属于 Server 层的一部分。

Qt/e 输入法基类，QWSInputMethod，在这个基类中定义了一些接口用以支持输入法程序设计，我们需要做的就是从 QWSInputMethod 这个类 继承出一个输入法类，在这个类中处理键盘和鼠标事件，把接收到的键盘事件按照输入法的编码规则转换为对应的中文，一个汉字，或者是一个中文短语，我们可以 把这个正在输入过程中的汉字或者短语发送给当前的编辑窗口，或者把最终用户的选择发送到当前编辑窗口。我们需要自己定义一个输入法窗口来显示用户当前的输入，我们可以称之为 IME Window。

文字的输入一般分为三个步骤：

1，开始输入

当用户在键盘上按下第一个按键的时候，输入法上下文就被创建出来，这个输入法上下文包含键盘输入字符

2，编辑

当有任何一个新的按键被按下的时候，输入法就会尝试着去创建与键盘输入相对应的中文字符，这个时候，输入法上下文处于激活状态，用户可以在这个输入法上下文中前后移动光标。

3，完成输入

在用户认为输入已经完成的时候，用户会选择以某种方式来选择最终的字符串，通常是使用键盘按键；或者鼠标点击；用户所选择的字符串最终应该被发送到当前的编辑窗口。

QWSInputMethod 类是 Qt 提供的、专门为输入法程序设计的基类，这个类定义了一系列的通用接口来对输入法提供支持，现在，让我们来看看这个类所定义的几个主要的接口：

```
virtual bool filter(int unicode, int keycode, int modifiers, bool isPress,  
bool autoRepeat );
```

这个接口的作用就是过滤键盘事件，详细一点儿说，就是我们可以在这个函数中处理键盘输入，并且根据相应的输入法规则把键盘输入转换为相应的中文。这个函数的参数含义如下：

unicode: Qte 统一使用的键盘按键编码，本文中，我们不使用这个参数

keycode: 键值，Qt 定义了一系列的键值与键盘一一对应，具体定义在 Qt namespace 中，比如说，Qt::Key_Left, Qt::Key_Up, Qt::Key_Right, Qt::Key_Down, 这四个定义对应到四个方向键，Qt::Key_0 则对应数字键 0, Qt::Key_A 则对应大写字母 A，等等。详细列表请参考 Qt 在线文档

modifiers: 这个参数表示是否有其它的辅助按键同时被按下，比如，Alt, Ctrl, Shift, 等，其预定义值如下：

Qt::NoModifier, 没有辅助键被按下

Qt::ShiftModifier, Shift 键被按下

Qt::ControlModifier, Ctrl 键被按下

Qt::AltModifier, Alt 键被按下

Qt::MetaModifier, Meta 键被按下

Qt::KeypadModifier, keypad 的按键被按下

Qt::GroupSwitchModifier, 仅用于 X11, Mode_switch 键被按下

更多解释请参考 Qt 在线文档

这些定义相互之间并不冲突，它们是按照“与”的关系组合在一起，在我们的使用中，我们可以用 C++ 的&操作符来判断某一个键是否被按下，比如，如果我们需要判断 Alt 键是否被按下，就应该这样做：

```
if (Qt::AltModifier & modifiers)  
{  
    //Alt 键被按下  
}
```

isPress: 这个参数表示键是被按下(press)，还是被释放(release)

autoRepeat: 这个参数表示这个按键事件是否是自动重复产生的

返回值：返回 true 表示这个按键事件已经被处理了，不需要继续分发；返回 false 表示这个按键没有被处理，Qt 会继续分发这个事件

```
void sendCommitString(const QString & commitString, int  
replaceFromPosition = 0, int replaceLength = 0);
```

这个接口函数表示把相应的字符串发送到当前编辑窗口，一般用于在用户作出最终的选择之后，把相应的字符串发送出去。

```
void sendPreeditString(const QString & preeditString, int cursorPosition, int selectionLength = 0);
```

把目前正在编辑的字符串发送给当前编辑窗口

下面我们写一个最简单的例子，

首先我们从 QWSInputMethod 派生出一个类来，在这个类中，我们集成了安装/卸载输入法，响应键盘事件，响应用户选择，上翻页，下翻页的功能。

```
// file: xinputmethod.h

struct XInputMethodPrivate;

class XInputMethod : public QWSInputMethod
{
    Q_OBJECT

public:

    static void installInputMethod();
    static void releaseInputMethod();
    static XInputMethod* instance();

    virtual bool filter(int unicode, int keycode, int modifiers,
        bool isPress, bool autoRepeat);
    virtual ~XInputMethod();

private:
    XInputMethod();

    void toggleIME();

    void newCharacter(char);
    bool makeSelection(int);
    void showNextPage();
    void showPreviousPage();

    XInputMethodPrivate* mpdata;
};
```

首先，我们定义了一个类的私有数据成员结构体，这种方法也是从 Qt 学来的。关于这个方法的详细解释，请看本系列文章的 2, 3, 4 篇，《对象数据存储》。这里，我们定义了一个 XWindow 类型的 pframe 指针变量，注意，这个 XWindow

和 Linux 系统的 XWindow 不是一回事，这个 XWindow 是本文中的输入法用户界面窗口类。

```
struct XIInputMethodPrivate
{
    static XIInputMethod* pInputMethod;

    XWindow* pframe;

    XData imedata;

    XIInputMethodPrivate(): pframe(NULL) { }
};

XIInputMethod* XIInputMethodPrivate::pInputMethod = NULL;
```

我们开发了一个输入法，最重要的就是需要 install，这样系统中才会有输入法模块，输入法才能工作。我们来看一下最重要的 install 和 release 输入法的代码。这里就是调用 QWServer 类中的成员函数来实现的。

QWServer::setCurrentInputMethod 这个函数为当前的 Qt/Embedded 安装一个输入法，如果把参数设置为 NULL，就是卸载输入法。

```
void XIInputMethod::installInputMethod()
{
    XIInputMethod* pim = instance();

    if (pim)
    {
        QWServer::setCurrentInputMethod(pim);
    }
}

void XIInputMethod::releaseInputMethod()
{
    if (XIInputMethodPrivate::pInputMethod)
    {
        QWServer::setCurrentInputMethod(NULL);
        delete XIInputMethodPrivate::pInputMethod;
        XIInputMethodPrivate::pInputMethod = NULL;
    }
}

XIInputMethod* XIInputMethod::instance()
{
    if (NULL == XIInputMethodPrivate::pInputMethod)
```

```

{
XInputMethodPrivate::pInputMethod = new XInputMethod();
}

return XInputMethodPrivate::pInputMethod;
}

```

输入法安装完成之后，在我们的输入法类中就可以接收到键盘事件了，这是在 QWSInputMethod 类中定义的虚函数 filter 完成的；我们重新实现这个函数，

- 在这里，我们用 ALT+Z 按键来显示/隐藏输入法用户界面。
- 当用户界面显示出来之后，就处理键盘点击事件，当用户输入'a' – 'z'，或者 'A' – 'Z' 的时候，就启动输入法引擎，把用户输入安装编码规则转换为相应的汉字，或者短语；紧接着，就在用户界面窗口上显示出来用户的输入和转换后的中文字符。
- 当用户输入数字 0 – 9 的时候，用户处理用户选择候选字。
- 当用户输入 PageDown 的时候，用来处理下翻页
- 当用户输入 PageUp 的时候，用来处理上翻页

```

bool XInputMethod::filter(int /*unicode*/, int keycode, int
modifiers, bool isPress, bool /*autoRepeat*/)
{
if (isPress && (& Qt::AltModifier && modifiers)
&& (& Qt::Key_Z == keycode))
{
toggleIME();

return true;
}

if (mpdata && mpdata->pframe && mpdata->pframe->isVisible() && isPress)
{
if ((Qt::Key_A <= keycode) && (Qt::Key_Z >= keycode))
{
char ch = (char) ((Qt::ShiftModifier & modifiers) ?
keycode : (keycode - Qt::Key_A + 'a'));

newCharacter(ch);

return true;
}

if ((Qt::Key_0 <= keycode) && (Qt::Key_9 >= keycode))

```

```

{
return makeSelection(keycode - Qt::Key_0);
}

if (Qt::Key_PageDown == keycode)
{
showNextPage();
return true;
}

if (Qt::Key_PageUp == keycode)
{
showPreviousPage();
return true;
}

return false;
}

void XInputMethod::toggleIME()
{
if (mpdata->pframe->isVisible())
{
mpdata->pframe->hide();
mpdata->imedata.reset();
}
else
{
mpdata->pframe->show();
}
}

```

在这个函数中把通过编码转换后的中文字符加入到
`mpdata->imedata.listHanzi` 这个变量中，就可以在界面上显示出来了。
由于本文仅仅只是为了讲解 Qt/Embedded 的输入法设计接口，没有编码方面的内
容，所以这里就加入了两个字符做为示例。

```

void XInputMethod::newCharacter(char ch)
{
mpdata->imedata.strPinyin += ch;

mpdata->imedata.listHanzi << "a";
mpdata->imedata.listHanzi << "b";

```

```
    mpdata->pframe->update();
}
```

用户按下数字键，选择当前显示的字符，注意，这里有一个很重要的地方，就是使用 QWSInputMethod 类的方法 sendCommitString，把用户选择的字符发送给当前的应用程序编辑窗口。

```
bool XInputMethod::makeSelection(int number)
{
    number--;
    if ((mpdata->imedata.first_visible + number) <
        mpdata->imedata.listHanzi.count())
    {
        QString result =
            mpdata->imedata.listHanzi[mpdata->imedata.first_visible +
            number];
        if (!result.isEmpty())
        {
            sendCommitString(result);
            mpdata->imedata.reset();
            mpdata->pframe->update();
        }
        return true;
    }
}

return false;
}
```

显示下一页

```
void XInputMethod::showNextPage()
{
    if ((mpdata->imedata.first_visible +
        mpdata->imedata.counts_per_page) <
        mpdata->imedata.listHanzi.count())
    {
        mpdata->imedata.first_visible +=
            mpdata->imedata.counts_per_page;
        mpdata->pframe->update();
    }
}
```

<pre>显示上一页

```
void XInputMethod::showPreviousPage()
{
    if ((mpdata->imedata.first_visible -
        mpdata->imedata.counts_per_page) &gt;= 0)
    {
        mpdata->imedata.first_visible -=
        mpdata->imedata.counts_per_page;
        mpdata->pframe->update();
    }
}
```

另外，我们还需要一个窗口来显示用户的输入字符，和经过中文编码转换后的中文，我们称之为 XIMWindow。这个用户界面窗口的代码，就不做详细解释了，它是很简单的，附件文件包含了完整的代码，有兴趣的朋友可以下载下来读一下。关于这个窗口，有一点需要注意的就是，由于输入法需要在最顶层显示出来，免得被其它窗口给覆盖了，所以在创建窗口的时候，需要设置好相应的 Widget Flag 才行。

```
#define IME_WND_FLAG (Qt::WindowStaysOnTopHint |  
Qt::FramelessWindowHint | Qt::Tool)
```

下面这张图片是这个程序的截图：



本文只是一个最简单的 Qte 输入法指南，只演示了最重要的输入法接口，仅仅只能起到一个入门的作用，一个实用的输入法还要包括字符转换，用户界面，根据需要，可能还需要鼠标事件处理，等等，有兴趣的朋友请参考 Qt 的在线文档和

源代码。

这里是本设计指南的源代码：[xinputmethod.tar](#)

Inside Qt Series (十五): Qt/e 输入法，How it works?

前面我们介绍了 Qte 输入法的基本设计思路，以及一个最简单的例子，那么，Qte 的输入法是如何工作的呢？本节我们就来看一下 Qte 的源代码，一起来解开这个谜团。

在 Qte 的 Client/Server 体系结构中，QWServer 类负责管理 Qte 的 Server，监听系统事件，尤其是键盘和鼠标事件。当这些监听的事件发生的时候，server 会做出判断，这些事件应该发送给那一个客户端。

如果当前系统安装了输入法，那么键盘和鼠标事件在派发之前，就会先送给输入法，让输入法来做一下判断，看输入法是否会处理这个键盘按键，如果输入法已经处理，就不在继续分发这个事件，否则就会按照原先的事件分发机制继续分发这个事件。也就是说，输入法会在应用程序之前接收到键盘事件。

Qte 已经定义了一个输入法基类 QWSInputMethod，在这个类中封装了一些基本的输入法函数。我们一起来看看 QWSInputMethod 类的定义：

```
class QWSInputMethod : public QObject
{
    Q_OBJECT
public:
    QWSInputMethod();
    virtual ~QWSInputMethod();

    enum UpdateType {Update, FocusIn, FocusOut, Reset,
Destroyed};

    virtual bool filter(int unicode, int keycode, int modifiers,
bool isPress, bool autoRepeat);

    virtual bool filter(const QPoint &, int state, int wheel);

    virtual void reset();
    virtual void updateHandler(int type);
    virtual void mouseHandler(int pos, int state);
    virtual void queryResponse(int property, const
QVariant&);

protected:
    uint setInputResolution(bool isHigh);
```

```

    uint inputResolutionShift() const;
    void sendMouseEvent(const QPoint &pos, int state, int
wheel);

    void sendEvent(const QInputMethodEvent*);
    void sendPreeditString(const QString &preeditString, int
cursorPosition, int selectionLength = 0);
    void sendCommitString(const QString &commitString, int
replaceFrom = 0, int replaceLength = 0);
    void sendQuery(int property);

private:
    bool mIResolution;
};

```

这个类从 QObject 类继承而来，定义了 Q_OBJECT 宏，说明这个类支持 Qt 对象模型的操作，signal/slot，property，都没有问题，这里最关键的几个函数有，两个重载的 filter 函数，一个用来过滤键盘事件，另一个用来过滤鼠标事件，sendEvent 函数用来发送输入法事件，在这个事件中可以打包 preedit string，commit string，它还有一个 list，可以添加任意多的其它数据。sendPreeditString 函数用来把正在输入过程中的字符串发送到当前编辑窗口，而 sendCommitString 则用来把最终的用户选择的字符串发送到当前编辑窗口。

QWServer 类提供了一个函数来安装输入法，void setCurrentInputMethod (QWSInputMethod * method)，这个函数的参数就是一个 QWSInputMethod 类的指针。QWServer 是如何管理 QWSInputMethod 的呢？在 Server 端，定义了这么几个变量，

```

static QWSInputMethod *current_IM = 0;
static QWSWindow *current_IM_composing_win = 0;
static int current_IM_winId = -1;
static bool force_reject_strokeIM = false;

```

其中，最重要的就是 current_IM 了，这个指针指向当前安装的输入法对象，它就是在 QWServer::setCurrentInputMethod 函数中赋值的。

这里是 QWServer::setCurrentInputMethod 这个函数的源代码：

```

void QWServer::setCurrentInputMethod(QWSInputMethod *im)
{
    if (current_IM)
        current_IM->reset();
    current_IM = im;
}

```

再看看这个键盘事件处理函数：

```
void QWSWidget::sendKeyEvent(int unicode, int keycode,
Qt::KeyboardModifiers modifiers,
bool isPress, bool autoRepeat)
{
//.....
#ifndef QT_NO_QWS_INPUTMETHODS

if (!current_IM || !current_IM->filter(unicode, keycode,
modifiers, isPress, autoRepeat))
QWSWidgetPrivate::sendKeyEventUnfiltered(unicode, keycode,
modifiers, isPress, autoRepeat);
#else
QWSWidgetPrivate::sendKeyEventUnfiltered(unicode, keycode,
modifiers, isPress, autoRepeat);
#endif
}
```

在 QWSWidget::sendKeyEvent 函数中，会去检查当前是否安装了输入法，如果是，就会去调用这个输入法的 filter 函数来过滤键盘事件，如果这个函数返回值为 true，就不在继续分发这个 key 事件。

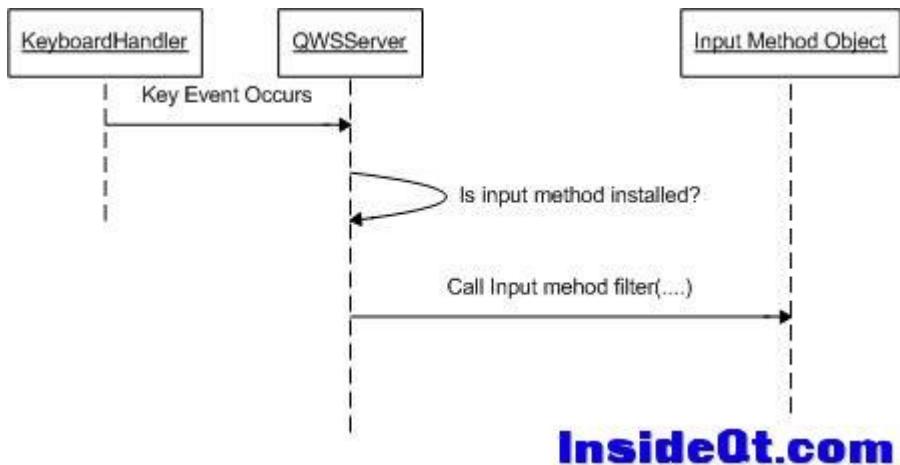
再看看这个鼠标事件处理函数：

```
void QWSWidget::sendMouseEvent(const QPoint& pos, int
state, int wheel)
{
// -----
const int btnMask = Qt::LeftButton | Qt::RightButton |
Qt::MidButton;
int stroke_count; // number of strokes to keep shown.
if (force_reject_strokeIM || !current_IM)
{
stroke_count = 0;
} else {
stroke_count = current_IM->filter(tpos, state, wheel);
}
}
```

在 QWSWidget::sendMouseEvent 函数里面，同样会去检查当前是否安装了输入法，如果是，就会去调用输入法的 filter 函数来过滤鼠标事件，如果这个函数返回值为 true，就不在继续分发这个 key 事件。

看，Qt/Embedded 输入法的工作原理其实就这么简单！

下面以一张简单的 UML sequence 图来说明一下：



InsideQt.com

=====

==

声明：

《Inside Qt Series》专栏文章是 Qt 核心技术论坛 (InsideQt. com) 原创技术文章。

本系列专栏文章可随意转载，但必须保留本段声明和每一篇文章的原始地址。