Qt GUI Programming Tutorial

文档编写规范：

1．文档中的中文字符全部采用宋体，英文全部采用 Courier New。

2．标题宋体二号居中加粗，正文用宋体五号。

3．程序代码必须插入到一个表格中，字体 Courier New 五号加粗。

4．要求文档中若使用一级标题、二级标题和三级标题请参照文档中规范。

5．推荐使用 Office 2010

文件名称： Qt GUI Programming Tutorial
文档简介： 本文档主要介绍如何开发基于 Qt 的 GUI 程序，侧重介绍 Qt 中提供的图形类。
编写作者： **唐华明**
责任校订： **无**
当前版本： v0.2 (2016-01-21)
替代版本： v0.1 (2016-01-20)
补充说明： 完成文档第 1 章内容。

目录

1 Qt 基础入门

主要内容：

本章主要引导初学者快速熟悉 Qt 应用程序开发流程。首先介绍编写第一个 Qt GUI 应用程序，简单的输出一行“Hello Qt”按钮，并详细介绍这个程序的结构。

接着，介绍信号与槽机制，这是 Qt 特有的对象间通信机制。

1.1 Qt 开发环境的搭建

注：此部分工作已经有很多文档介绍了，也不是本文档的重点，略。

1.2 第一个 Qt 程序

没有什么比“让一个程序跑起来”更让人激动人心了。因此，在本文档中我们将一致贯彻这样的思想，首先想办法让程序跑起来，然后再来慢慢研究程序的运行原理。

我们接下来就首先运行对于初学者而言的第一个 Qt 应用程序。新建文本文件，命名为 helloqt.cpp，然后编辑该文件内容如下：

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QPushButton btn("Hello Qt");
    btn.show();
    QObject::connect(&btn, SIGNAL(clicked()), &a, SLOT(quit()));

    return a.exec();
}
```

程序运行效果如下，点击其中的按钮，程序退出。

仅仅有这个代码文件就想得到上图所示的运行效果还是不够的，我们还需要对这段代码进行编译。目前有两种方式来编译，分别是命令行方式和使用 Qt Creator 集成开发环境编译。

1.2.1 命令行方式编译

利用 Qt 提供的 qmake 工具可以很方便的进行编译，编译流程如下：

```
# qmake -project
# qmake
# make
```

```
# ./helloqt
```

首先，调用“qmake -project”命令用于生成程序的项目文件(*.pro)；接着，调用“qmake”命令生成程序的 Makefile 文件；然后，调用“make”命令根据 Makefile 编译程序得到可执行文件；最后，执行生成的可执行文件，程序运行就可以看到应用程序界面。

1.2.2 Qt Creator 集成开发环境编译

建立一个 GUI 工程，过程略。

使用命令行的方式需要进行繁琐的环境变量配置，对于初学者而言难度较大，因此建议使用 Qt Creator 集成开发环境进行编译。

1.2.3 代码分析

第 1 行包含头文件<QApplication>，所有的 Qt 图形化应用程序都必须包含此文件，它包含了 Qt 图形化应用程序的各种资源、基本设置、控制流以及事件处理等。如果开发非图形化的应用程序，则需要包含<QCoreApplication>头文件。(更多内容请在帮助手册中搜索“qapplication”主题)

第 2 行为包含程序中要使用的按钮控件的头文件。在 Qt 中，包含头文件既可以采用类似<QPushButton>的形式，也可以写成<qpushbutton.h>的形式。

第 3 行为应用程序入口，所有的 Qt 程序都必须有一个 main()函数，以 argc 和 argv 作为入口参数。

第 4 行新建一个 QApplication 对象，每个 Qt 应用程序都必须有且仅有一个 QApplication 对象，采用 argc 和 argv 作为参数，便于程序处理命令行参数。(在 QApplication 的帮助主题中提供了命令行参数的使用样例)

第 5 行创建了一个 QPushButton 对象，并且设置它的显示文本为“Hello Qt”，由于此处没有指定按钮控件的父窗体，因此它将以自己作为主窗口。

第 6 行调用按钮控件的 show()方法，显示此按钮。按钮被创建时默认是不显示的，必须调用 show()函数来显示它。

第 7 行建立按钮控件 clicked()信号与 QApplication 对象的 quit()槽函数之间的连接，执行退出应用程序的操作。信号与槽机制作为 Qt 最重要的特性，提供任意两个 Qt 对象之间的通信机制。其中，信号会在某个特定的情况或者动作下被触发，槽是用于接收并处理信号的函数。

1.3 信号与槽机制

原文链接：http://www.ibm.com/developerworks/cn/linux/guitoolkit/qt/signal-slot/

信号与槽作为 QT 的核心机制在 QT 编程中有着广泛的应用，本文介绍了信号与槽的一些基本概念、元对象工具以及在实际使用过程中应注意的一些问题。

QT 是一个跨平台的 C++ GUI 应用构架，它提供了丰富的窗口部件集，具有面向对象、易于扩展、真正的组件编程等特点，更为引人注目的是目前 Linux 上最为流行的 KDE 桌面环境就是建立在 QT 库的基础之上(比如，openSUSE 就默认使用 KDE 桌面)。QT 支持下列平台：MS/WINDOWS-95、98、NT 和 2000；UNIX/X11-Linux、Sun Solaris、HP-UX、Digital Unix、IBM AIX、SGI IRIX；EMBEDDED- 支持 framebuffer 的 Linux 平台。伴随着 KDE 的快速发展和普及，QT 很可能成为 Linux 窗口平台上进行软件开发时的 GUI 首选。

1.3.1 概述

信号和槽机制是 QT 的核心机制，要精通 QT 编程就必须对信号和槽有所了解。信号和槽是一种高级接口，应用于对象之间的通信，它是 QT 的核心特性，也是 QT 区别于其它工具包的重要地方。信号和槽是 QT 自行定义的一种通信机制，它独立于标准的 C/C++ 语言，因此**要正确的处理信号和槽，必须借助一个称为 moc（Meta Object Compiler）的 QT 工具，该工具是一个 C++ 预处理程序，它为高层次的事件处理自动生成所需要的附加代码**。

在我们所熟知的很多 GUI 工具包中，窗口小部件（widget）都有一个回调函数用于响应它们能触发的每个动作，这个回调函数通常是一个指向某个函数的指针。但是，在 QT 中信号和槽取代了这些凌乱的函数指针，使得我们编写这些通信程序更为简洁明了。 **信号和槽能携带任意数量和任意类型的参数，他们是类型完全安全的，不会像回调函数那样产生 core dumps**。

所有从 QObject 或其子类（例如 Qwidget）派生的类都能够包含信号和槽。当对象改变其状态时，信号就由该对象发射（emit）出去，这就是对象所要做的全部事情，它不知道另一端是谁在接收这个信号。这就是真正的信息封装，它确保对象被当作一个真正的软件组件来使用。槽用于接收信号，但它们是普通的对象成员函数。一个槽并不知道是否有任何信号与自己相连接。而且，对象并不了解具体的通信机制。你可以将很多信号与单个的槽进行连接，也可以将单个的信号与很多的槽进行连接，甚至于将一个信号与另外一个信号相连接也是可能的，这时无论第一个信号什么时候发射，系统都将立刻发射第二个信号。总之，信号与槽构造了一个强大的部件编程机制。

1.3.2 信号

当某个信号对其客户或所有者发生的内部状态发生改变，信号被一个对象发射。只有定义过这个信号的类及其派生类能够发射这个信号。当一个信号被发射时，与其相关联的槽将被立刻执行，就像一个正常的函数调用一样。**信号 - 槽机制完全独立于任何 GUI 事件循环。**只有当所有的槽返回以后发射函数（emit）才返回。 如果存在多个槽与某个信号相关联，那么，当这个信号被发射时，这些槽将会一个接一个地执行，但是它们**执行的顺序将会是随机的、不确定的**，我们不能人为地指定哪个先执行、哪个后执行。

信号的声明是在头文件中进行的，QT 的 signals 关键字指出进入了信号声明区，随后即可声明自己的信号。例如，下面定义了三个信号：

```
signals:
    void mySignal();
    void mySignal(int x);
    void mySignal(int x, int y);
```

在上面的定义中，signals 是 QT 的关键字，而非 C/C++ 的。接下来的一行 void mySignal() 定义了信号 mySignal，这个信号没有携带参数；接下来的一行 void mySignal(int x) 定义了重载信号 mySignal，但是它携带一个整形参数，这有点类似于 C++ 中的虚函数。从形式上讲信号的声明与普

通的 C++ 函数是一样的，但是**信号却没有函数体定义**，另外，信号的返回类型都是 void，不要指望能从信号返回什么有用信息。

信号由 moc 自动产生，它们不应该在.cpp 文件中实现。也就是说，我们只需要在声明类的.h 文件中声明信号的形式和参数列表即可，不需要在对应的类的.cpp 文件中来定义其行为。

小结：一个类的信号应该满足如下语法约束

(1) 函数的返回值是 void 类型，因为触发信号函数的目的是执行与其绑定的槽函数，无需信号函数返回任何值。

(2) 用户只能声明、不能实现信号函数，因为 Qt 的 moc 会实现它，无需开发人员关心。

(3) 只能使用 **emit** 关键字“调用”信号函数，不能使用普通的调用方式。

(4) 信号函数被 moc 自动设置为 **protected**，因而只有包含这个信号函数的类及其派生类能够触发该信号函数。

(5) 信号函数的参数个数、类型由程序员自由设定，只要是元对象系统支持的类型，这些参数的职责是封装类的状态信息，并将这些信息传递给槽函数。

(6) 只有 QObject 及其派生类才可以定义信号函数。

1.3.3 槽

槽是普通的 C++ 成员函数，可以被正常调用，它们唯一的特殊性就是很多信号可以与其相关联。当与其关联的信号被发射时，这个槽就会被调用。槽可以有参数，但槽的参数不能有缺省值（不能初始化）。

既然槽是普通的成员函数，因此与其它的函数一样，它们也有存取权限。槽的存取权限决定了谁能够与其相关联。同普通的 C++ 成员函数一样，槽函数也分为三种类型，即 public slots、private slots 和 protected slots。

- **public slots：**在这个区内声明的槽意味着任何对象都可将信号与之相连接。这对于组件编程非常有用，你可以创建彼此互不了解的对象，将它们的信号与槽进行连接以便信息能够正确的传递。
- **protected slots：**在这个区内声明的槽意味着当前类及其子类可以将信号与之相连接。这适用于那些槽，它们是类实现的一部分，但是其界面接口却面向外部。
- **private slots：**在这个区内声明的槽意味着只有类自己可以将信号与之相连接。这适用于联系非常紧密的类。

槽也能够声明为虚函数，这也是非常有用的。槽的声明也是在头文件中进行的。例如，下面声明了三个槽：

```
public slots:
    void mySlot();
    void mySlot(int x);
    void mySlot(int x, int y);
```

小结：一个类的槽函数应该满足以下语法约束

(1) 函数返回值是 void 类型，因为信号与槽机制是单向的：信号函数被触发后，与其绑定的槽函数会被执行，但不要求槽函数返回任何执行结果。

(2) 一个类的槽函数可以如同其他成员函数一样被正常调用。

(3) 一个类的槽函数可以是 public，protected 以及 private，这些关键词的含义依旧，也就是说，它们能够控制其他类是否能够以正常的方式调用一个槽函数。但是，**这些关键字对 connect 函数不起作用**，也就是说，我们可以将 protected 甚至 private 的槽函数和一个信号函数绑定。当该信号函数被触发时，甚至 private 的槽函数也会被执行。**从某种意义讲，Qt 的信号与槽机制破坏了 C++的存取控制规则，但是这种机制带来的灵活性远胜于可能导致的问题**。

(4) 只有 QObject 及其派生类才可以定义槽函数。

1.3.4 信号与槽的关联

通过调用 QObject 对象的 connect 函数来将某个对象的信号与另外一个对象的槽函数相关联，这样当发射者发射信号时，接收者的槽函数将被调用。该函数的定义如下：

```
bool QObject::connect(const QObject *sender, const char *signal,
                      const QObject *receiver, const char *member) [static]
```

这个函数的作用就是将发射者 sender 对象中的信号 signal 与接收者 receiver 中的 member 槽函数联系起来。**当指定信号 signal 时必须使用 QT 的宏 SIGNAL()，当指定槽函数时必须使用宏 SLOT()。**如果发射者与接收者属于同一个对象的话，那么在 connect 调用中接收者参数可以省略。

例如，下面定义了两个对象：标签对象 label 和滚动条对象 scroll，并将 valueChanged() 信号与标签对象的 setNum() 相关联，另外信号还携带了一个整形参数，这样标签总是显示滚动条所处位置的值。

```
QLabel *label = new QLabel;
QScrollBar *scroll = new QScrollBar;
QObject::connect(scroll,SIGNAL(valueChanged(int)),label,SLOT(setNum(int)));
```

一个信号甚至能够与另一个信号关联，看下面的例子：

```
class MyWidget : public QWidget
{
public:
    MyWidget();
    ...
signals:
    void aSignal();
    ...
private:
    QPushButton *aButton;
    ...
};
```

```
MyWidget::MyWidget()
{
    aButton = new QPushButton(this);
    connect(aButton,SIGNAL(clicked()),SIGNAL(aSignal()));
}
```

在上面的构造函数中，MyWidget 创建了一个私有的按钮 aButton，按钮的单击事件产生的信号 clicked() 与另外一个信号 aSignal() 进行了关联。这样一来，当信号 clicked() 被发射时，信号 aSignal() 也接着被发射。当然，你也可以直接将单击事件与某个私有的槽函数相关联，然后在槽中发射 aSignal() 信号，这样的话似乎有点多余。

当信号与槽没有必要继续保持关联时，我们可以使用 disconnect 函数来断开连接。其定义如下：

```
bool QObject::disconnect(const QObject *sender, const char *signal,
                         const QObject *receiver, const char *member) [static]
```

这个函数断开发射者中的信号与接收者中的槽函数之间的关联。

有三种情况必须使用 disconnect() 函数：

- **断开与某个对象相关联的任何对象。**这似乎有点不可理解，事实上，当我们在某个对象中定义了一个或者多个信号，这些信号与另外若干个对象中的槽相关联，如果我们要切断这些关联的话，就可以利用这个方法，非常之简洁。

```
disconnect(myObject, 0, 0, 0);
或者
myObject->disconnect();
```

- **断开与某个特定信号的任何关联。**

```
disconnect(myObject,SIGNAL(mySignal()), 0, 0);
或者
myObject->disconnect(mySignal());
```

- **断开两个对象之间的关联。**

```
disconnect(myObject, 0, myReceiver, 0);
或者
myObject->disconnect(myReceiver);
```

在 disconnect 函数中 0 可以用作一个通配符，分别表示任何信号、任何接收对象、接收对象中的任何槽函数。但是发射者 sender 不能为 0，其它三个参数的值可以等于 0。

1.3.5 元对象工具

元对象编译器 moc（meta object compiler）对 C++ 文件中的类声明进行分析并产生用于初始化元对象的 C++ 代码，元对象包含全部信号和槽的名字以及指向这些函数的指针。

moc 读 C++ 源文件，如果发现有 Q_OBJECT 宏声明的类，它就会生成另外一个 C++ 源文件，这个新生成的文件中包含有该类的元对象代码。例如，假设我们有一个头文件 mysignal.h，在这个文件中包含有信号或槽的声明，那么在编译之前 moc 工具就会根据该文件自动生成一个名为 mysignal.moc.h 的 C++ 源文件并将其提交给编译器；类似地，对应于 mysignal.cpp 文件 moc 工具将自动生成一个名为 mysignal.moc.cpp 文件提交给编译器。

元对象代码是 signal/slot 机制所必须的。用 moc 产生的 C++ 源文件必须与类实现一起进行编译和连接，或者用 #include 语句将其包含到类的源文件中。moc 并不扩展 #include 或者 #define 宏定义，它只是简单的跳过所遇到的任何预处理指令。

1.3.6 应注意的问题

信号与槽机制是比较灵活的，但有些局限性我们必须了解，这样在实际的使用过程中做到有的放矢，避免产生一些错误。下面就介绍一下这方面的情况。

1．信号与槽的效率是非常高的，但是同真正的回调函数比较起来，由于增加了灵活性，因此在**速度上还是有所损失**，当然这种损失相对来说是比较小的，通过在一台 i586-133 的机器上测试是 10 微秒（运行 Linux），可见这种机制所提供的简洁性、灵活性还是值得的。但如果我们要追求高效率的话，比如在实时系统中就要尽可能的少用这种机制。

注：由于函数调用本来就很占用时间，而使用信号与槽机制将更加耗时；虽然从单次调用来说，时间上的差别不是太明显，但是在频繁使用信号与槽的时候其影响就相当可观了。在这时，我们可以使用其他的传递消息机制或者使用多线程技术，还有一种更加常用的方法是将多个分开传递的信息合并到一个信号与槽传递中来完成。**最后还需要注意的是，虽然将多个短数据合并到一起传递可以在某种程度上提升程序运行效率，但是如果数据量太大（比如一幅图像 400x600 个点对应的 RGB 值数组）反而会导致程序运行时卡住，所以使用的时候还是要根据实际情况而定。**

2．信号与槽机制与普通函数的调用一样，如果使用不当的话，在程序执行时也有可能产生死循环。因此，**在定义槽函数时一定要注意避免间接形成无限循环，即在槽中再次发射所接收到的同样信号。**例如，在前面给出的例子中如果在 mySlot() 槽函数中加上语句 emit mySignal() 即可形成死循环。

3．如果一个信号与多个槽相联系的话，那么，当这个信号被发射时，与之相关的槽被激活的**顺序将是随机的**。

4．**宏定义不能用在 signal 和 slot 的参数中**。

既然 moc 工具不扩展 #define，因此，在 signals 和 slots 中携带参数的宏就不能正确地工作，如果不带参数是可以的。例如，下面的例子中将带有参数的宏 SIGNEDNESS(a) 作为信号的参数是**不合语法**的：

```
#ifdef Ultrix
#define SIGNEDNESS(a) unsigned a
#else
```

```
#define SIGNEDNESS(a) a
#endif
class Whatever : public QObject
{
    [...]
signals:
    void someSignal(SIGNEDNESS(a));
    [...]
};
```

5．**构造函数不能用在 signals 或者 slots 声明区域内。**

的确，将一个构造函数放在 signals 或者 slots 区内有点不可理解，无论如何，不能将它们放在 private slots、protected slots 或者 public slots 区内。下面的用法是**不合语法要求**的：

```
class SomeClass : public QObject
{
    Q_OBJECT
public slots:
    SomeClass(QObject *parent, const char *name)
        : QObject(parent, name){}//在槽声明区内声明构造函数不合语法
};
```

6．**函数指针不能作为信号或槽的参数**。

例如，下面的例子中将 void (*applyFunction)(QList*, void*) 作为参数是不合语法的：

```
class someClass : public QObject
{
    Q_OBJECT
    [...]
public slots:
    void apply(void (*applyFunction)(QList*,void*),char*);//不合语法
};
```

你可以采用下面的方法绕过这个限制：

```
typedef void (*ApplyFunctionType)(QList*, void*);
class someClass : public QObject
{
    Q_OBJECT
    [...]
public slots:
    void apply(ApplyFunctionType,char*);
```

```
};
```

7．**信号与槽不能有缺省参数。**

既然 signal->slot 绑定是发生在运行时刻，那么，从概念上讲使用缺省参数是困难的。下面的用法是不合理的：

```
class someClass : public QObject
{
    Q_OBJECT
public slots:
    void someSlot(int x = 100);//在槽函数声明中将 x 的缺省值定义为 100 是错误的
};
```

8．**信号与槽也不能携带模板类参数。**

如果将信号、槽声明为模板类参数的话，即使 moc 工具不报告错误，也不可能得到预期的结果。 例如，下面的例子中当信号发射时，槽函数不会被正确调用：

```
[...]
public slots:
    void MyWidget::setLocation(pair<int,int> location);
    [...]
public signals:
    void MyWidget::moved(pair<int,int> location);
```

但是你可以使用 typedef 语句绕过这个限制，如下所示：

```
typedef pair<int, int> InPair;
[...]
public slots:
    void MyWidget::setLocation(InPair location);
    [...]
public signals:
    void MyWidget::moved(InPair location);
```

这样使用的话，你就可以得到正确的结果。

说明：使用以上方法，实际上也只是能保证编译不报错而已，在执行 connect 连接对应的信号与槽的时候还是会失败，原因是使用 typedef 定义的类型并不是元类型(MetaType)。**Qt 的信号与槽机制只支持元类型数据的传递，你可以通过查询 enum QMetaType::Type 枚举类型的值来确定你想要传递的数据类型能不能够得到信号与槽机制的支持。**此外，如果你想要使你自己定义的数据类型支持信号与槽机制，那么可以将这个自定义类型注册到元类型当中，即使用 Q_DECLARE_METATYPE 宏和 qRegisterMetaType 方法，详细内容请参考以下链接：

http://www.360doc.com/content/11/0513/17/2775766_116494672.shtml

http://blog.163.com/jx_yp/blog/static/119704459201111910343279 5/

9．**嵌套的类不能位于信号或槽区域内，也不能有信号或者槽。**

例如，下面的例子中，在 class B 中声明槽 b() 是不合语法的，在信号区内声明槽 b() 也是不合语法的。

```
class A
{
    Q_OBJECT
public:
    class B
    {
    public slots://在嵌套类中声明槽，不合语法
        void b();
        [...]
    };
};
```

10．友元声明不能位于信号或者槽声明区内。

相反，它们应该在普通 C++ 的 private、protected 或者 public 区内进行声明。下面的例子是不合语法规范的：

```
class someClass : public QObject
{
    Q_OBJECT
public signals://信号定义区
    friend class ClassTemplate;//此处定义不合语法
};
```

2 布局管理

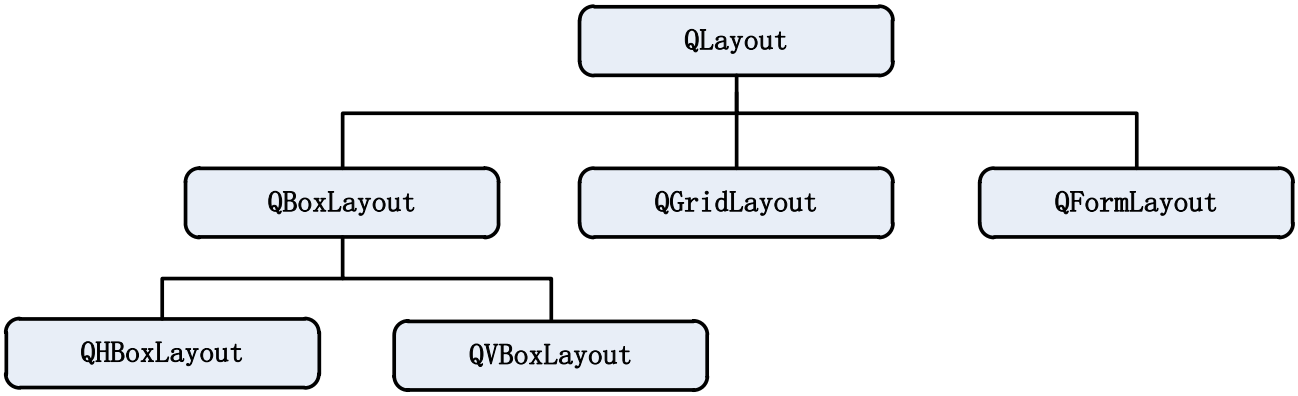
主要内容：
在学习了第一章如何在窗口中添加单个部件元素之后，要想在窗口中指定位置添加多个部件，就需要知道 Qt 中是如何实现布局管理的，也就是这一章将要介绍的内容。

Qt 为程序开发提供了灵活、强大的布局管理方法，其中包括基本的布局类 QLayout、多文档窗体、分割窗体、停靠窗体、堆栈窗体等。

2.1 基本布局管理

注：请参看 Qt 帮助手册中“Layout Management”主题了解更多。

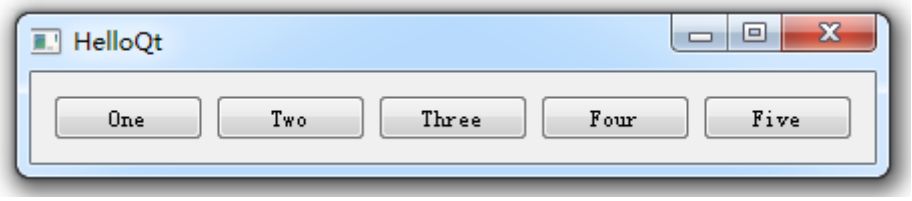
给你的部件提供一个漂亮的布局，最简单的方法就是使用内建的布局管理器 QHBoxLayout、QVBoxLayout、QGridLayout 和 QFormLayout，这些类都继承自 QLayout。QLayout 是 QObject 的派生类，而不是 QWidget。这些类负责一系列部件的几何管理，为了创建更加复杂的布局，你可以将这些布局管理器类组合使用。各布局类之间的结构关系如下图所示：



各布局类之间的结构关系图

2.1.1 水平布局

水平布局可以由 QHBoxLayout 类来实现将所有部件放置在一个水平行排列，默认从左到右的顺序排列。



水平布局效果图

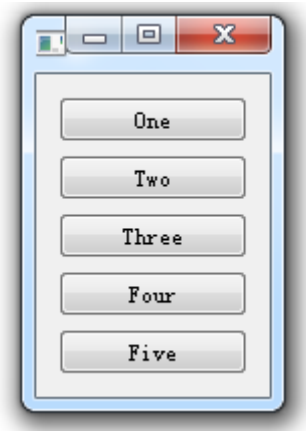
下面给出实现水平布局效果的代码：

```
#include <QApplication>
#include <QPushButton>
#include <QWidget>
#include <QHBoxLayout>
```

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget *window = new QWidget;
    QPushButton *btn1 = new QPushButton("One");
    QPushButton *btn2 = new QPushButton("Two");
    QPushButton *btn3 = new QPushButton("Three");
    QPushButton *btn4 = new QPushButton("Four");
    QPushButton *btn5 = new QPushButton("Five");

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(btn1);
    layout->addWidget(btn2);
    layout->addWidget(btn3);
    layout->addWidget(btn4);
    layout->addWidget(btn5);

    window->setLayout(layout);
    window->show();

    return a.exec();
}
```

2.1.2 垂直布局

垂直布局可以通过 QVBoxLayout 类实现，所有的部件在一个垂直列中，按照从上到下的顺序排列。



垂直布局效果图

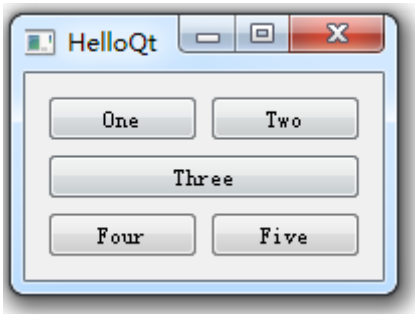
下面给出实现垂直布局效果的代码：

```
#include <QApplication>
#include <QPushButton>
#include <QWidget>
#include <QVBoxLayout>
```

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget *window = new QWidget;
    QPushButton *btn1 = new QPushButton("One");
    QPushButton *btn2 = new QPushButton("Two");
    QPushButton *btn3 = new QPushButton("Three");
    QPushButton *btn4 = new QPushButton("Four");
    QPushButton *btn5 = new QPushButton("Five");

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(btn1);
    layout->addWidget(btn2);
    layout->addWidget(btn3);
    layout->addWidget(btn4);
    layout->addWidget(btn5);

    window->setLayout(layout);
    window->show();

    return a.exec();
}
```

2.1.3 网格布局

可以用 QGridLayout 类实现网格布局，所有的部件在一张二维的网格中，单个部件可以占据多个网格单元。



网格布局效果图

下面给出网格布局效果的实现代码：

```
#include <QApplication>
#include <QPushButton>
#include <QWidget>
#include <QGridLayout>

int main(int argc, char *argv[])
```

```
{
    QApplication a(argc, argv);
    QWidget *window = new QWidget;
    QPushButton *btn1 = new QPushButton("One");
    QPushButton *btn2 = new QPushButton("Two");
    QPushButton *btn3 = new QPushButton("Three");
    QPushButton *btn4 = new QPushButton("Four");
    QPushButton *btn5 = new QPushButton("Five");

    QGridLayout *layout = new QGridLayout;
    layout->addWidget(btn1, 0, 0);
    layout->addWidget(btn2, 0, 1);
    layout->addWidget(btn3, 1, 0, 1, 2);
    layout->addWidget(btn4, 2, 0);
    layout->addWidget(btn5, 2, 1, 1, 1);

    window->setLayout(layout);
    window->show();

    return a.exec();
}
```

在前面水平布局和垂直布局的例子中我们可以看到调用 addWidget()方法时只需要传递部件的指针作为参数即可，然而对于网格布局而言，还需要通过额外的参数来指定部件占据的网格数量以及起始位置。QGridLayout 类的 addWidget()方法有两种重载形式：

```
void QGridLayout::addWidget(QWidget * widget, int row, int column, Qt::Alignment alignment = 0)
```

这个函数用于部件只占据一个网格单元的情况。第一个参数和其他布局类一样，都是传递部件的指针。第二和第三个参数用于指定该部件在网格单元中的位置（第 row 行，第 column 列），默认情况下，最左上角的位置坐标为(0,0)，也就是说网格编号是从 0 开始的。

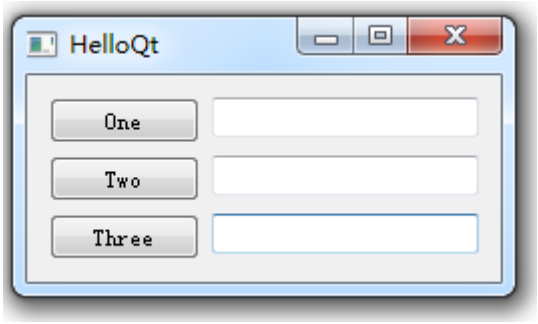
最后一个 alignment 参数用于指定部件的对齐方式，默认值为 0，表示部件填充整个网格单元。

```
void QGridLayout::addWidget(QWidget * widget, int fromRow, int fromColumn, int rowSpan, int columnSpan, Qt::Alignment alignment = 0)
```

这个函数用于部件需要占据多个网格单元的情况。第一个参数和其他布局类一样，都是传递部件的指针。第二和第三个参数用于指定该部件左上角在网格中的起始位置，也就是说从(fromRow,fromColumn)位置处的网格单元开始填充该部件。第四和第五个参数用于指定部件总共占据的网格数，即整个部件需要占据 rowSpan 行网格和 columnSpan 列网格。那么，代码中 layout->addWidget(btn5, 2, 1, 1, 1)的运行效果实际上与 layout->addWidget(btn5, 2, 1)的效果是相同的。

2.1.4 表格布局

表格布局由 QFormLayout 类来实现，用于提供一个 2 列“标签-域”样式的布局方式。QFormLayout 将会把两个部件添加到一行，通常是采用一个 QLabel 和一个 QLineEdit 来创建一个表格项。将一个 QLabel 和一个 QLineEdit 添加到同一行时将会自动把 QLineEdit 设置为 QLabel 的伙伴(buddy)。



表格布局效果图

下面给出实现表格布局效果的实现代码：

```
#include <QApplication>
#include <QPushButton>
#include <QLineEdit>
#include <QWidget>
#include <QFormLayout>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget *window = new QWidget;
    QPushButton *btn1 = new QPushButton("One");
    QPushButton *btn2 = new QPushButton("Two");
    QPushButton *btn3 = new QPushButton("Three");
    QLineEdit   *edt1 = new QLineEdit;
    QLineEdit   *edt2 = new QLineEdit;
    QLineEdit   *edt3 = new QLineEdit;

    QFormLayout *layout = new QFormLayout;
    layout->addRow(btn1, edt1);
    layout->addRow(btn2, edt2);
    layout->addRow(btn3, edt3);

    window->setLayout(layout);
    window->show();

    return a.exec();
}
```

2.1.5 使用布局的注意事项

当你在使用一个布局类的时候，你不需要在创建子部件的时候指定它的父对象，布局管理器将会自动重新设置子部件的父亲（使用 `QWidget::setParent()`），这样，这些子部件就变成了安装布局的部件的孩子，即调用 `setLayout()` 方法的 `Widget` 就是所有这些子部件的父亲。

注意：布局管理器中的部件是安装布局的 Widget 的孩子，而不是布局管理器它自身的孩子。部件只能选择其他部件作为父对象，而不能是布局（布局类继承自 QObject，而不是 QWidget）。你也可以在一个布局中调用 addLayout()方法来嵌套另一个布局，这个内嵌的布局就变成了它嵌入的布局的一个孩子。

当你将一个部件添加到一个布局中，布局管理器将会处理以下工作：
（1）所有部件将会根据它的 QWidget::sizePolicy()和 QWidget::sizeHint()首先被分配一定大小的空间（不是内存空间，而是位置空间）。
（2）如果某个部件设置了伸展因子(stretch factor)，并且取值大于 0，那么将会根据它的伸展因子设置的比例来分配空间（将会在后面解释）。
（3）If any of the widgets have stretch factors set to zero they will only get more space if no other widgets want the space. Of these, space is allocated to widgets with an Expanding size policy (QSizePolicy::Expanding) first.
（4）Any widgets that are allocated less space than their minimum size (or minimum size hint if no minimum size is specified) are allocated this minimum size they require. (Widgets don't have to have a minimum size or minimum size hint in which case the stretch factor is their determining factor.)
（5）Any widgets that are allocated more space than their maximum size are allocated the maximum size space they require. (Widgets do not have to have a maximum size in which case the stretch factor is their determining factor.)

2.1.6 伸展因子

Widgets are normally created without any stretch factor set. When they are laid out in a layout the widgets are given a share of space in accordance with their QWidget::sizePolicy() or their minimum size hint whichever is the greater. Stretch factors are used to change how much space widgets are given in proportion to one another.

If we have three widgets laid out using a QHBoxLayout with no stretch factors set we will get a layout like this:

我们可以看到这 3 个部件的大小是一样的。

If we apply stretch factors to each widget, they will be laid out in proportion (but never less than their minimum size hint), e.g.

我们可以看到这 3 个部件各自占用的空间大小就不一样了。

2.2 多文档窗体

2.3 分割窗体

2.4 停靠窗体

2.5 堆栈窗体

3 标准对话框

主要内容：

操作系统提供了一系列标准的对话框，这些对话框在应用中会频繁使用，因此我们在学习了如何布局之后就首先来学习标准对话框的使用，主要包括 QMessageBox、QErrorMessage、QInputDialog、QPrintDialog、QProgressDialog、QFileDialog、QColorDialog 和 QFontDialog。

3.1 文件对话框

3.2 颜色对话框

3.3 字体对话框

3.4 进度对话框

3.5 消息对话框

QMessageBox 类提供一个模态对话框(modal dialog)，用于告知用户信息或者询问用户并等待用户回答。

一个消息框显示一段主文本(text)向用户报警，一段信息文本(informative text)来进一步解释此报警或者询问用户，还有一个可选的详细信息(detailed text)来提供更加详细的数据（如果用于请求的话)。一个消息框还可以显示一个图标和标准按钮来接收用户响应。

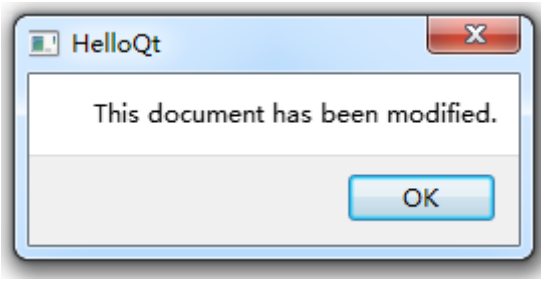
提供了两种 API 来使用 QMessageBox，基于属性(property-based)的 API 和静态函数(static functions)。调用一个静态函数是一种更简单的方法，但是它相比于基于属性的 API 而言缺乏灵活性，结果提供的信息也更少。因此，推荐使用基于属性的 API。

3.5.1 基于属性的 API

要使用基于属性的 API，需要构造一个 QMessageBox 实例，设置需要的属性，然后调用 exec()方法来显示消息。最简单的配置就是仅仅设置消息文本 text 属性。

```
QMessageBox msgBox;
msgBox.setText("This document has been modified.");
msgBox.exec();
```

用户必须点击 OK 按钮才能够消除消息对话框。除对话框以外的 GUI 界面全部处于阻塞状态，直到消息对话框消除以后才能被激活。

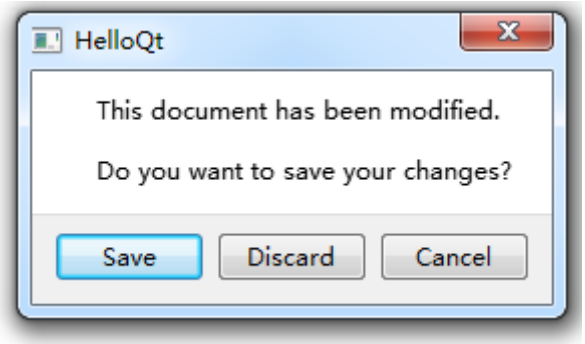


一种更好的方法是在提醒用户发生的事件的同时询问用户该如何处理当前事件。将这个问题存储在 informative text 信息文本属性中，并设置 standard buttons 属性来设置你想要用户响应的按钮集合。这些按钮可以通过位运算“或”操作符选取枚举变量 StandardButtons 中的值联合使用。按钮的显示顺序与平台相关，例如，在 Windows 平台上，Save 显示在 Cancel 的左边；而在 Mac OS 平台上，却恰好相反。

有多个按钮时，需要从这些按钮中标记一个默认按钮获取焦点。

```
QMessageBox msgBox;
msgBox.setText("This document has been modified.");
msgBox.setInformativeText("Do you want to save your changes?");
msgBox.setStandardButtons(QMessageBox::Save | QMessageBox::Discard |
QMessageBox::Cancel);
msgBox.setDefaultButton(QMessageBox::Save);
int ret = msgBox.exec();
```

在 Windows 平台下的运行效果如下图所示：



exec()槽函数将会返回被点击的按钮在枚举变量 StandardButtons 中的值。

```
switch(ret){
    case QMessageBox::Save:
        qDebug() << "QMessageBox::Save"; break;
    case QMessageBox::Discard:
        qDebug() << "QMessageBox::Discard"; break;
    case QMessageBox::Cancel:
        qDebug() << "QMessageBox::Cancel"; break;
    default:
```

```
        // should never be reached
        break;
    }
```

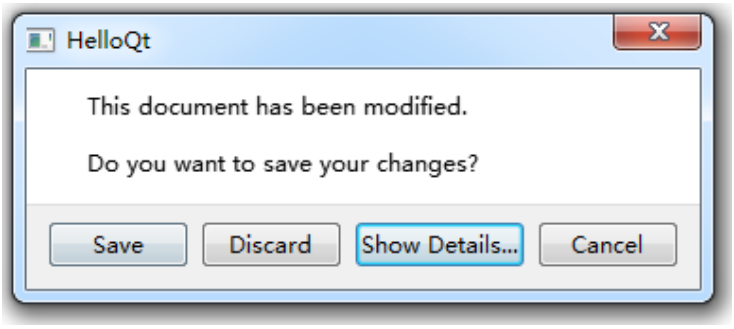
枚举变量 StandardButtons 中可取的值如下表所示：

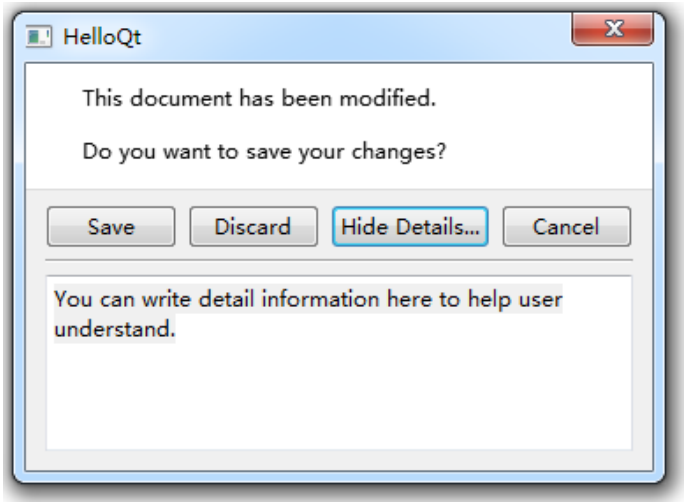
Constant	Value	Description
QMessageBox::Ok	0x00000400	An "OK" button defined with the AcceptRole.
QMessageBox::Open	0x00002000	An "Open" button defined with the AcceptRole.
QMessageBox::Save	0x00000800	A "Save" button defined with the AcceptRole.
QMessageBox::Cancel	0x00400000	A "Cancel" button defined with the RejectRole.
QMessageBox::Close	0x00200000	A "Close" button defined with the RejectRole.
QMessageBox::Discard	0x00800000	A "Discard" or "Don't Save" button, depending on the platform, defined with the DestructiveRole.
QMessageBox::Apply	0x02000000	An "Apply" button defined with the ApplyRole.
QMessageBox::Reset	0x04000000	A "Reset" button defined with the ResetRole.
QMessageBox::RestoreDefaults	0x08000000	A "Restore Defaults" button defined with the ResetRole.
QMessageBox::Help	0x01000000	A "Help" button defined with the HelpRole.
QMessageBox::SaveAll	0x00001000	A "Save All" button defined with the AcceptRole.
QMessageBox::Yes	0x00004000	A "Yes" button defined with the YesRole.
QMessageBox::YesToAll	0x00008000	A "Yes to All" button defined with the YesRole.
QMessageBox::No	0x00010000	A "No" button defined with the NoRole.
QMessageBox::NoToAll	0x00020000	A "No to All" button defined with the NoRole.
QMessageBox::Abort	0x00040000	An "Abort" button defined with the RejectRole.
QMessageBox::Retry	0x00080000	A "Retry" button defined with the AcceptRole.
QMessageBox::Ignore	0x00100000	An "Ignore" button defined with the AcceptRole.
QMessageBox::NoButton	0x00000000	An invalid button.

说明：虽然此表中给出了这些枚举值的数值，但是可能随着版本升级有所改动，因此在编程中建议直接使

用名称。

为了给用户提供更多的信息以帮助他回答问题，可以设置 detailed text 属性。如果设置了 detailed text 属性，那么“Show Details…”按钮将会被显示出来。





实现代码如下：

```
QMessageBox msgBox;
msgBox.setText("This document has been modified.");
msgBox.setInformativeText("Do you want to save your changes?");
msgBox.setDetailedText("You can write detail information here to help user understand.");
msgBox.setStandardButtons(QMessageBox::Save | QMessageBox::Discard | QMessageBox::Cancel);
msgBox.setDefaultButton(QMessageBox::Save);
int ret = msgBox.exec();
```

富文本和 Text Format 属性

detailed text 属性总是被解释为纯文本。而 main text 属性和 informative text 属性可以是纯文本，也可以是富文本。这些字符串将会根据 text format 属性的设置来解释，其默认设置为 auto-text。

请注意，有些纯文本字符串包含有 XML 元字符，这在 auto-text 富文本检测测试时可能失败，因为你的纯文本字符串被作为富文本时可能被解释错误。针对这种罕见的情况，使用 Qt::convertFromPlainText()将你的纯文本字符串转换为一种视觉效果等同的富文本，或者显式调用 setTextFormat()来设置 text format 属性。

严重程度和 Icon and Pixmap 属性

3.5.2 静态函数