

# Qt 教程一 —— 第一章：Hello, World!



第一个程序是一个简单的 Hello World 例子。它只包含你建立和运行 Qt 应用程序所需的最少的代码。上面的图片是这个程序的快照。

```

/*****
**
** Qt 教程一 - 2
**
*****/

#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    QPushButton hello( "Hello world!", 0 );
    hello.resize( 100, 30 );

    a.setMainWidget( &hello );
    hello.show();
    return a.exec();
}
```

## 一行一行地解说

```
#include <qapplication.h>
```

这一行包含了 `QApplication` 类的定义。在每一个使用 Qt 的应用程序中都必须使用一个 `QApplication` 对象。`QApplication` 管理了各种各样的应用程序的广泛资源，比如默认的字体和光标。

```
#include <qpushbutton.h>
```

这一行包含了 `QPushButton` 类的定义。[参考文档](#) 的文件的最上部分提到了使用哪个类就必须包含哪个头文件的说明。

`QPushButton` 是一个经典的图形用户界面按钮，用户可以按下去，也可以放开。它管理自己的观感，就像其它每一个 `QWidget`。一个窗口部件就是一个可以处理用户输入和绘制图形的

用户界面对象。程序员可以改变它的全部 [观感](#)和它的许多主要的属性（比如颜色），还有这个窗口部件的内容。一个QPushButton可以显示一段文本或者一个QPixmap。

```
int main( int argc, char **argv )
{
```

main()函数是程序的入口。几乎在使用 Qt 的所有情况下，main()只需要在把控制转交给 Qt 库之前执行一些初始化，然后 Qt 库通过事件来向程序告知用户的行为。

argc 是命令行变量的数量，argv 是命令行变量的数组。这是一个 C/C++特征。它不是 Qt 专有的，无论如何 Qt 需要处理这些变量（请看下面）。

```
    QApplication a( argc, argv );
```

a是这个程序的 [QApplication](#)。它在这里被创建并且处理这些命令行变量（比如在X窗口下的-display）。请注意，所有被Qt识别的命令行参数都会从argv中被移除（并且argc也因此而减少）。关于细节请看 [QApplication::argv\(\)](#)文档。

**注意：**在任何 Qt 的窗口系统部件被使用之前创建 QApplication 对象是必须的。

```
    QPushButton hello( "Hello world!", 0 );
```

这里，在 QApplication 之后，接着的是第一个窗口系统代码：一个按钮被创建了。

这个按钮被设置成显示“Hello world!”并且它自己构成了一个窗口（因为在构造函数指定 0 为它的父窗口，在这个父窗口中按钮被定位）。

```
    hello.resize( 100, 30 );
```

这个按钮被设置成 100 像素宽，30 像素高（加上窗口系统边框）。在这种情况下，我们不用考虑按钮的位置，并且我们接受默认值。

```
    a.setMainWidget( &hello );
```

这个按钮被选为这个应用程序的主窗口部件。如果用户关闭了主窗口部件，应用程序就退出了。

你不用必须设置一个主窗口部件，但绝大多数程序都有一个。

```
    hello.show();
```

当你创建一个窗口部件的时候，它是不可见的。你必须调用 show()来使它变为可见的。

```
    return a.exec();
```

这里就是 main()把控制转交给 Qt，并且当应用程序退出的时候 exec()就会返回。

在 exec()中，Qt 接受并处理用户和系统的事件并且把它们传递给适当的窗口部件。

```
}
```

你现在可以试着编译和运行这个程序了。

## 编译

编译一个C++应用程序，你需要创建一个makefile。创建一个Qt的makefile的最容易的方法是使用Qt提供的连编工具 **qmake**。如果你已经把main.cpp保存到它自己的目录了，你所要做的就是这些：

```
qmake -project  
qmake
```

第一个命令调用 **qmake**来生成一个.pro（项目）文件。第二个命令根据这个项目文件来生成一个（系统相关的）makefile。你现在可以输入make（或者nmake，如果你使用Visual Studio），然后运行你的第一个Qt应用程序！

第二种方法：

编辑一个 project 文件，

```
vi hello.pro
```

添加以下内容：

```
SOURCES+=hello.cpp
```

```
CONFIG+=qt release
```

保存退出

```
qmake
```

```
make
```

## 行为

当你运行它的时候，你就会看到一个被单一按钮充满的小窗口，在它上面你可以读到著名的词：Hello World!

# Qt 教程一 —— 第二章：调用退出



你已经在 [第一章](#)中创建了一个窗口，我们现在使这个应用程序在用户让它退出的时候退出。

我们也会使用一个比默认字体更好的一个字体。

```

/*****
**
** Qt 教程一 - 2
**
*****/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    QPushButton quit( "Quit", 0 );
    quit.resize( 75, 30 );
    quit.setFont( QFont( "Times", 18, QFont::Bold ) );

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

    a.setMainWidget( &quit );
    quit.show();
    return a.exec();
}

```

## 一行一行地解说

```
#include <qfont.h>
```

因为这个程序使用了 **QFont**，所以它需要包含 **qfont.h**。**Qt** 的字体提取和 **X** 中提供的可怕的字体提取大为不同，字体的载入和使用都已经被高度优化了。

```
QPushButton quit( "Quit", 0 );
```

这时，按钮显示“Quit”，确切的说这就是当用户点击这个按钮时程序所要做的。这不是一个巧合。因为这个按钮是一个顶层窗口，我们还是把 **0** 作为它的父对象。

```
quit.resize( 75, 30 );
```

我们给这个按钮选择了另外一个大小，因为这个文本比“Hello world!”小一些。我们也可以使用 **QFontMetrics** 来设置正确的大小。

```
quit.setFont( QFont( "Times", 18, QFont::Bold ) );
```

这里我们给这个按钮选择了一个新字体，**Times** 字体中的 18 点加粗字体。注意在这里我们调用了这个字体。

你也可以改变整个应用程序的默认字体（使用 `QApplication::setFont()`）。

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

`connect`也许是Qt中最重要的特征了。注意`connect()`是 `QObject`中的一个静态函数。不要把这个函数和socket库中的`connect()`搞混了。

这一行在两个Qt对象（直接或间接继承`QObject`对象的对象）中建立了一种单向的连接。每一个Qt对象都有signals（发送消息）和slots（接收消息）。所有窗口部件都是Qt对象。它们继承 `QWidget`，而`QWidget`继承`QObject`。

这里 `quit` 的 `clicked()`信号和 `a` 的 `quit()`槽连接起来了，所以当这个按钮被按下的时候，这个程序就退出了。

信号和槽文档详细描述了这一主题。

## 行为

当你运行这个程序的时候，你会看到这个窗口比第一章中的那个小一些，并且被一个更小的按钮充满。

（请看 [编译](#)来学习如何创建一个makefile和连编应用程序。）

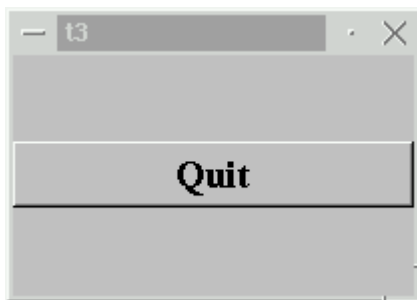
## 练习

试着改变窗口的大小。按下按钮。注意！`connect()`看起来会有一些不同。

是不是在 `QPushButton`中还有其它的你连接到`quit`的信号？提示：`QPushButton`继承了 `QPushButton`的绝大多数行为。

现在你可以进行 [第三章](#)了。

# Qt 教程一 —— 第三章：家庭价值



这个例子演示了如何创建一个父窗口部件和子窗口部件。

我们将会保持这个程序的简单性，并且只使用一个单一的父窗口部件和一个独立的子窗口部件。

```
/*
**
** Qt 教程一 - 3
**
**
***/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
#include <qvbox.h>

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    QVBox box;
    box.resize( 200, 120 );

    QPushButton quit( "Quit", &box );
    quit.setFont( QFont( "Times", 18, QFont::Bold ) );

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

    a.setMainWidget( &box );
    box.show();

    return a.exec();
}
```

## 一行一行地解说

```
#include <qvbox.h>
```

我们添加了一个头文件 `qvbox.h` 用来获得我们要使用的布局类。

```
QVBox box;
```

这里我们简单地创建了一个垂直的盒子容器。`QVBox`把它的子窗口部件排成一个垂直的行，一个在其它的上面，根据每一个子窗口部件的 `QWidget::sizePolicy()`来安排空间。

```
box.resize( 200, 120 );
```

我们它的高设置为 120 像素，宽为 200 像素。

```
QPushButton quit( "Quit", &box );
```

子窗口部件产生了。

`QPushButton`通过一个文本（“text”）和一个父窗口部件（`box`）生成的。子窗口部件总是放在它的父窗口部件的最顶端。当它被显示的时候，它被父窗口部件的边界挡住了一部分。

父窗口部件，`QVBox`，自动地把这个子窗口部件添加到它的盒子中央。因为没有其它的东西被添加了，这个按钮就获得了父窗口部件的所有空间。

```
box.show();
```

当父窗口部件被显示的时候，它会调用所有子窗口部件的显示函数（除非在这些子窗口部件中你已经明确地使用 `QWidget::hide()`）。

## 行为

这个按钮不再充满整个窗口部件。相反，它获得了一个“自然的”大小。这是因为现在的这个新的顶层窗口，使用了按钮的大小提示和大小变化策略来设置这个按钮的大小和位置。

（请看 `QWidget::sizeHint()`和 `QWidget::setSizePolicy()`来获得关于这几个函数的更详细的信息。）

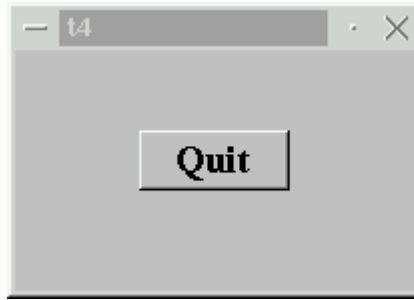
（请看 [编译](#)来学习如何创建一个makefile和连编应用程序。）

## 练习

试着改变窗口的大小。按钮是如何变化的？按钮的大小变化策略是什么？如果你运行这个程序的时候使用了一个大一些的字体，按钮的高度发生了什么变化？如果你试图让这个窗口真的变小，发生了什么？

现在你可以进行 [第四章](#)了。

# Qt 教程一 —— 第四章：使用窗口部件



这个例子显示了如何创建一个你自己的窗口部件，描述如何控制一个窗口部件的最小大小和最大大小，并且介绍了窗口部件的名称。

```

/*****
**
** Qt 教程一 - 4
**
*****/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>

class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};

MyWidget::MyWidget( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
    setMinimumSize( 200, 120 );
    setMaximumSize( 200, 120 );

    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setGeometry( 62, 40, 75, 30 );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );

    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
}

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

```



```

MyWidget w;
w.setGeometry( 100, 100, 200, 120 );
a.setMainWidget( &w );
w.show();
return a.exec();
}

```

## 一行一行地解说

```

class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};

```

这里我们创建了一个新类。因为这个类继承了 `QWidget`，所以新类是一个窗口部件，并且可以最为一个顶层窗口或者子窗口部件（像第三章里面的按钮）。

这个类只有一个成员函数，构造函数（加上从 `QWidget` 继承来的成员函数）。这个构造函数是一个标准的 Qt 窗口部件构造函数，当你创建窗口部件时，你应该总是包含一个相似的构造函数。

第一个参数是它的父窗口部件。为了生成一个顶层窗口，你指定一个空指针作为父窗口部件。就像你看到的那样，这个窗口部件默认地被认做是一个顶层窗口。

第二个参数是这个窗口部件的名称。这个不是显示在窗口标题栏或者按钮上的文本。这只是分配给窗口部件的一个名称，以后可以用来 [查找](#)这个窗口部件，并且这里还有一个 [方便的调试功能](#)可以完整地列出窗口部件层次。

```

MyWidget::MyWidget( QWidget *parent, const char *name )
    : QWidget( parent, name )

```

构造函数的实现从这里开始。像大多数窗口部件一样，它把parent和name传递给了 `QWidget` 的构造函数。

```

{
    setMinimumSize( 200, 120 );
    setMaximumSize( 200, 120 );

```

因为这个窗口部件不知道如何处理重新定义大小，我们把它的最小大小和最大大小设置为相等的值，这样我们就确定了它的大小。在下一章，我们将演示窗口部件如何响应用户的重新定义大小事件。

```

QPushButton *quit = new QPushButton( "Quit", this, "quit" );
quit->setGeometry( 62, 40, 75, 30 );
quit->setFont( QFont( "Times", 18, QFont::Bold ) );

```

这里我们创建并设置了这个窗口部件的一个名称为“quit”的子窗口部件（新窗口部件的父窗口部件是 this）。这个窗口部件名称和按钮文本没有关系，只是在这一情况下碰巧相似。

注意 quit 是这个构造函数中的局部变量。MyWidget 不能跟踪它，但 Qt 可以，当 MyWidget 被删除的时候，默认地它也会被删除。这就是为什么 MyWidget 不需要一个析构函数的原因。（另外一方面，如果你选择删除一个子窗口部件，也没什么坏处，这个子窗口部件会自动告诉 Qt 它即将死亡。）

setGeometry()调用和上一章的 move()和 resize()是一样的。

```
        connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
    }
```

因为 MyWidget 类不知道这个应用程序对象，它不得不连接到 Qt 的指针，qApp。

一个窗口部件就是一个软件组件并且它应该尽量少地知道关于它的环境，因为它应该尽可能的通用和可重用。

知道了应用程序的名称将会打破上述原则，所以在一个组件，比如 MyWidget，需要和应用程序对象对话的这种情况下，Qt 提供了一个别名，qApp。

```
int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    MyWidget w;
    w.setGeometry( 100, 100, 200, 120 );
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

这里我们举例说明了我们的新子窗口部件，把它设置为主窗口部件，并且执行这个应用程序。

## 行为

这个程序和上一章的在行为上非常相似。不同点是我们实现的方式。无论如何它的行为还是有一些小差别。试试改变它的大小，你会看到什么？

（请看 [编译](#) 来学习如何创建一个makefile和连编应用程序。）

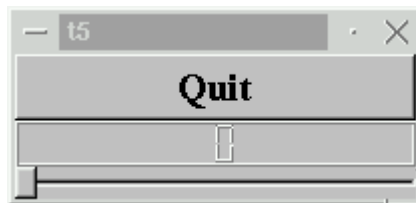
## 练习

试着在 main()中创建另一个 MyWidget 对象。发生了什么？

试着添加更多的按钮或者把除了 `QPushButton` 之外的东西放到窗口部件中。

现在你可以进行 第五章 了。

## Qt 教程一 —— 第五章：组装积木



这个例子显示了创建几个窗口部件并用信号和槽把它们连接起来，和如何处理重新定义大小事件。

```

/*****
**
** Qt 教程一 - 5
**
*****/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>

#include <qvbox.h>

class MyWidget : public QVBox
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};

MyWidget::MyWidget( QWidget *parent, const char *name )
    : QVBox( parent, name )
{
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );

    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
}
```

```

QLCDNumber *lcd = new QLCDNumber( 2, this, "lcd" );

QSlider * slider = new QSlider( Horizontal, this, "slider" );
slider->setRange( 0, 99 );
slider->setValue( 0 );

connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    MyWidget w;
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}

```

## 一行一行地解说

```

#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>

#include <qvbox.h>

```

这里显示的是三个新的被包含的头文件。`qslider.h`和`qlcdnumber.h`在这里是因为我们使用了两个新的窗口部件，`QSlider`和`QLCDNumber`。`qvbox.h`在这里是因为我们使用了Qt的自动布局支持。

```

class MyWidget : public QVBox
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};

MyWidget::MyWidget( QWidget *parent, const char *name )
    : QVBox( parent, name )
{

```

`MyWidget`现在继承了`QVBox`，而不是`QWidget`。我们通过这种方式来使用`QVBox`的布局（它可以把它的子窗口部件垂直地放在自己里面）。重新定义大小自动地被`QVBox`处理，因此现在也就被`MyWidget`处理了。

```
QLCDNumber *lcd = new QLCDNumber( 2, this, "lcd" );
```

lcd 是一个 QLCDNumber，一个可以按像 LCD 的方式显示数字的窗口部件。这个实例被设置为显示两个数字，并且是 *this* 的子窗口部件。它被命名为“lcd”。

```
QSlider * slider = new QSlider( Horizontal, this, "slider" );
slider->setRange( 0, 99 );
slider->setValue( 0 );
```

QSlider 是一个经典的滑块，用户可以通过在拖动一个东西在一定范围内调节一个整数数值的方式来使用这个窗口部件。这里我们创建了一个水平的滑块，设置它的范围是 0~99（包括 0 和 99，参见 [QSlider::setRange\(\)](#) 文档）并且它的初始值是 0。

```
connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
```

这里我们是用了 [信号/槽机制](#) 把滑块的 valueChanged() 信号和 LCD 数字的 display() 槽连接起来了。

无论什么时候滑块的值发生了变化，它都会通过发射 valueChanged() 信号来广播这个新的值。因为这个信号已经和 LCD 数字的 display() 槽连接起来了，当信号被广播的时候，这个槽就被调用了。这两个对象中的任何一个都不知道对方。这就是组件编程的本质。

槽是和普通 C++ 成员函数的方式不同，但有着普通 C++ 成员函数的方位规则。

## 行为

LCD 数字反应了你对滑块做的一切，并且这个窗口部件很好地处理了重新定义大小事件。注意当窗口被重新定义大小（因为它可以）的时候，LDC 数字窗口部件也改变了大小，但是其它的还是和原来一样（因为如果它们变化了，看起来好像很傻）。

（请看 [编译](#) 来学习如何创建一个 makefile 和连编应用程序。）

## 练习

试着改变 LCD 数字，添加更多的数字或者 [改变模式](#)。你甚至可以添加四个按钮来设置基数。

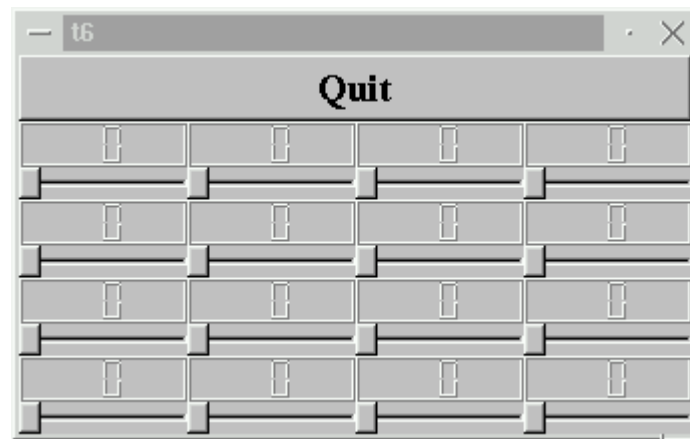
你也可以改变滑块的范围。

也许使用 [QSpinBox](#) 比滑块更好？

试着当 LCD 数字溢出的时候使这个应用程序退出。

现在你可以进行 [第六章](#)了。

# Qt 教程一 —— 第六章：组装丰富的积木！



这个例子显示了如何把两个窗口部件封装成一个新的组件和使用许多窗口部件是多么的容易。首先，我们使用一个自定义的窗口部件作为一个子窗口部件。

```
/*
**
** Qt 教程一 - 6
**
**
***/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qvbox.h>
#include <qgrid.h>

class LCDRange : public QVBox
{
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
};

LCDRange::LCDRange( QWidget *parent, const char *name )
    : QVBox( parent, name )
{
    QLCDNumber *lcd = new QLCDNumber( 2, this, "lcd" );
    QSlider * slider = new QSlider( Horizontal, this, "slider" );
    slider->setRange( 0, 99 );
    slider->setValue( 0 );
    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}
```

```

class MyWidget : public QVBox
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};

MyWidget::MyWidget( QWidget *parent, const char *name )
    : QVBox( parent, name )
{
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );

    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );

    QGrid *grid = new QGrid( 4, this );

    for( int r = 0 ; r < 4 ; r++ )
        for( int c = 0 ; c < 4 ; c++ )
            (void)new LCDRange( grid );
}

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    MyWidget w;
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}

```

## 一行一行地解说

```

class LCDRange : public QVBox
{
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
};

```

**LCDRange** 窗口部件是一个没有任何 **API** 的窗口部件。它只有一个构造函数。这种窗口部件不是很有用，所以我们一会儿会加入一些 **API**。

```

LCDRange::LCDRange( QWidget *parent, const char *name )
    : QVBox( parent, name )

```

```

{
    QLCDNumber *lcd = new QLCDNumber( 2, this, "lcd" );
    QSlider * slider = new QSlider( Horizontal, this, "slider" );
    slider->setRange( 0, 99 );
    slider->setValue( 0 );
    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}

```

这里直接利用了第五章里面的 **MyWidget** 的构造函数。唯一的不同是按钮被省略了并且这个类被重新命名了。

```

class MyWidget : public QVBox
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};

```

**MyWidget** 也是除了一个构造函数之外没有包含任何 API。

```

MyWidget::MyWidget( QWidget *parent, const char *name )
    : QVBox( parent, name )
{
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );

    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
}

```

这个按钮被放在 **LCDRange** 中，这样我们就有了一个“Quit”按钮和许多 **LCDRange** 对象。

```

QGrid *grid = new QGrid( 4, this );

```

我们创建了一个四列的 **QGrid** 对象。这个 **QGrid** 窗口部件可以自动地把自己地子窗口部件排列到行列中，你可以指定行和列的数量，并且 **QGrid** 可以发现它的新子窗口部件并且把它们安放到网格中。

```

for( int r = 0 ; r < 4 ; r++ )
    for( int c = 0 ; c < 4 ; c++ )
        (void)new LCDRange( grid );

```

四行，四列。

我们创建了一个 4\*4 个 **LCDRanges**，所有这些都是这个 **grid** 对象的子窗口部件。这个 **QGrid** 窗口部件会安排它们。

```

}

```

这就是全部了。



## 行为

这个程序显示了在同一时间使用许多窗口部件是多么的容易。其中的滑块和 LCD 数字的行为在上一章已经提到过了。还有就是，就是实现的不同。

（请看 [编译](#) 来学习如何创建一个makefile和连编应用程序。）

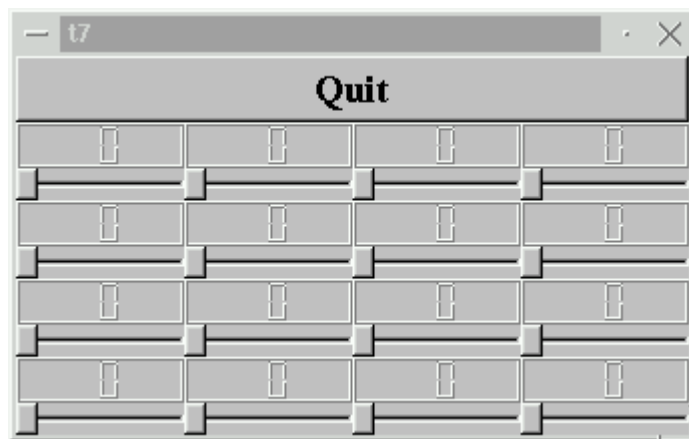
## 练习

在开始的时候使用不同的或者随机的值初始化每个滑块。

源代码中的“4”出现了 3 次。如果你改变 [QGrid](#)构造函数中调用的那个，会发生什么？改变另外两个又会发生什么呢？为什么呢？

现在你可以进行 [第七章](#)了。

# Qt 教程一 —— 第七章：一个事物领导另一个



这个例子显示了如何使用信号和槽来创建自定义窗口部件，和如何使用更加复杂的方式把它们连接起来。首先，源文件被我们分成几部分并放在放在 `t7` 目录下。

- [t7/lcdrange.h](#)包含LCDRange类定义。
- [t7/lcdrange.cpp](#)包含LCDRange类实现。
- [t7/main.cpp](#)包含MyWidget和main。

## 一行一行地解说

## t7/lcdrange.h

这个文件主要利用了第六章的 `main.cpp`，在这里只是说明一下改变了哪些。

```
#ifndef LCDRANGE_H
#define LCDRANGE_H
```

这里是一个经典的 C 语句，为了避免出现一个头文件被包含不止一次的情况。如果你没有使用过它，这是开发中的一个很好的习惯。`#ifndef` 需要把这个头文件的全部都包含进去。

```
#include <qvbox.h>
```

`qvbox.h`被包含了。`LCDRange`继承了 `QVBox`，所以父类的头文件必须被包含。我们在前几章里面偷了一点懒，我们通过包含其它一些头文件，比如 `qpushbutton.h`，这样就可以间接地包含 `qwidget.h`。

```
class QSlider;
```

这里是另外一个小伎俩，但是没有前一个用的多。因为我们在类的界面中不需要 `QSlider`，仅仅是在实现中，我们在头文件中使用一个前置的类声明，并且在`.cpp`文件中包含一个 `QSlider`的头文件。

这会使编译一个大的项目变得更快，因为当一个头文件改变的时候，很少的文件需要重新编译。它通常可以给大型编译加速两倍或两倍以上。

```
class LCDRange : public QVBox
{
    Q_OBJECT
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
```

**meta object file.** 注意`Q_OBJECT`。这个宏必须被包含到所有使用信号和/或槽的类。如果你很好奇，它定义了 **元对象文件** 中实现的一些函数。

```
    int value() const;
public slots:
    void setValue( int );

signals:
    void valueChanged( int );
```

这三个成员函数构成了这个窗口部件和程序中其它组件的接口。直到现在，`LCDRange` 根本没有一个真正的接口。

`value()`是一个可以访问 `LCDRange` 的值的公共函数。`setValue()`是我们第一个自定义槽，并且 `valueChanged()`是我们第一个自定义信号。

槽必须按通常的方式实现（记住槽也是一个C++成员函数）。信号可以在 [元对象](#) 文件中自动实现。信号也遵守C++函数的保护法则（比如，一个类只能发射它自己定义的或者继承来的信号）。

当 `LCDRange` 的值发生变化时，`valueChanged()` 信号就会被使用——你从这个名字中就可以猜到。这不会是你将会看到的命名为 `somethingChanged()` 的最后一个信号。

## t7/lcdrange.cpp

这个文件主要利用了 [t6/main.cpp](#)，在这里只是说明一下改变了哪些。

```
connect( slider, SIGNAL(valueChanged(int)),
         lcd, SLOT(display(int)) );
connect( slider, SIGNAL(valueChanged(int)),
         SIGNAL(valueChanged(int)) );
```

这个代码来自 `LCDRange` 的构造函数。

第一个 `connect` 和你在上一章中看到的一样。第二个是新的，它把滑块的 `valueChanged()` 信号和这个对象的 `valueChanged` 信号连接起来了。带有三个参数的 `connect()` 函数连接到 `this` 对象的信号或槽。

是的，这是正确的。信号可以被连接到其它的信号。当第一个信号被发射时，第二个信号也被发射。

让我们来看看当用户操作这个滑块的时候都发生了些什么。滑块看到自己的值发生了改变，并发射了 `valueChanged()` 信号。这个信号被连接到 [QLCDNumber](#) 的 `display()` 槽和 `LCDRange` 的 `valueChanged()` 信号。

所以，当这个信号被发射的时候，`LCDRange` 发射它自己的 `valueChanged()` 信号。另外，[QLCDNumber::display\(\)](#) 被调用并显示新的数字。

注意你并没有保证执行的任何顺序——`LCDRange::valueChanged()` 也许在 `QLCDNumber::display()` 之前或者之后发射，这是完全任意的。

```
int LCDRange::value() const
{
    return slider->value();
}
```

`value()` 的实现是直接了当的，它简单地返回滑块的值。

```
void LCDRange::setValue( int value )
{
    slider->setValue( value );
}
```

setValue()的实现是相当直接了当的。注意因为滑块和 LCD 数字是连接的，设置滑块的值就会自动的改变 LCD 数字的值。另外，如果滑块的值超过了合法范围，它会自动调节。

## t7/main.cpp

```
LCDRange *previous = 0;
for( int r = 0 ; r < 4 ; r++ ) {
    for( int c = 0 ; c < 4 ; c++ ) {
        LCDRange* lr = new LCDRange( grid );
        if ( previous )
            connect( lr, SIGNAL(valueChanged(int)),
                    previous, SLOT(setValue(int)) );
        previous = lr;
    }
}
```

main.cpp中所有的部分都是上一章复制的，除了MyWidget的构造函数。当我们创建 16 个 LCDRange对象时，我们现在使用 信号/槽机制连接它们。每一个的valueChanged()信号都和前一个的setValue()槽连接起来了。因为当LCDRange的值发生改变的时候，发射一个valueChanged()信号（惊奇！），我们在这里创建了一个信号和槽的“链”。

## 编译

为一个多文件的应用程序创建一个 makefile 和为一个单文件的应用程序创建一个 makefile 是没有什么不同的。如果你已经把这个例子中的所有文件都保存到它们自己的目录中，你所要做的就是这些：

```
qmake -project
qmake
```

第一个命令调用 qmake来生成一个.pro（项目）文件。第二个命令根据这个项目文件来生成一个（系统相关的）makefile。你现在可以输入make（或者nmake，如果你使用Visual Studio）。

## 行为

在开始的时候，这个程序看起来和上一章里的一样。试着操作滑块到右下角.....

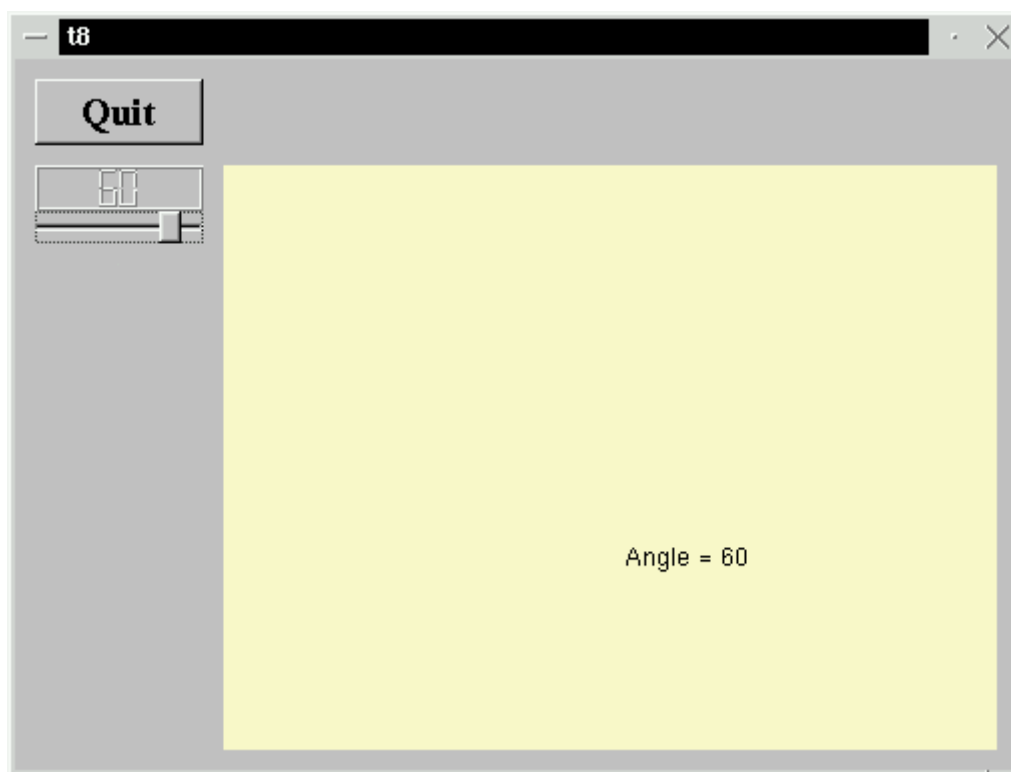
## 练习

seven LCDs back to 50. 使用右下角的滑块并设置所有的 LCD 到 50。然后设置通过点击这个滑块的左侧把它设置为 40。现在，你可以通过把最后一个调到左边来把前七个 LCD 设置回 50。

点击右下角滑块的滑块的左边。发生了什么？为什么只是正确的行为？

现在你可以进行 第八章了。

## Qt 教程一 —— 第八章：准备战斗



在这个例子中，我们介绍可以画自己的第一个自定义窗口部件。我们也加入了一个有用的键盘接口（只用了两行代码）。

- [t8/lcdrange.h](#)包含LCDRange类定义。
- [t8/lcdrange.cpp](#)包含LCDRange类实现。
- [t8/cannon.h](#)包含CannonField类定义。
- [t8/cannon.cpp](#)包含CannonField类实现。
- [t8/main.cpp](#)包含MyWidget和main。

### 一行一行地解说

#### [t8/lcdrange.h](#)

这个文件和第七章中的 `lcdrange.h` 很相似。我们添加了一个槽：`setRange()`。

```
void setRange( int minVal, int maxVal );
```

现在我们添加了设置 `LCDRange` 范围的可能性。直到现在，它就可以被设置为 0~99。

## t8/lcdrange.cpp

在构造函数中有一个变化（稍后我们会讨论的）。

```
void LCDRange::setRange( int minVal, int maxVal )
{
    if ( minVal < 0 || maxVal > 99 || minVal > maxVal ) {
        qWarning( "LCDRange::setRange(%d,%d)\n"
                  "\tRange must be 0..99\n"
                  "\tand minVal must not be greater than maxVal",
                  minVal, maxVal );
        return;
    }
    slider->setRange( minVal, maxVal );
}
```

setRange()设置了LCDRange中滑块的范围。因为我们已经把 **QLCDNumber** 设置为只显示两位数字了，我们想通过限制minVal和maxVal为 0~99 来避免QLCDNumber的溢出。（我们可以允许最小值为-9，但是我们没有那样做。）如果参数是非法的，我们使用Qt的 **qWarning()** 函数来向用户发出警告并立即返回。**qWarning()**是一个像printf一样的函数，默认情况下它的输出发送到stderr。如果你想改变的话，你可以使用 **::qInstallMsgHandler()**函数安装自己的处理函数。

## t8/cannon.h

CanonField 是一个知道如何显示自己的新的自定义窗口部件。

```
class CannonField : public QWidget
{
    Q_OBJECT
public:
    CannonField( QWidget *parent=0, const char *name=0 );
```

CanonField继承了 **QWidget**，我们使用了LCDRange中同样的方式。

```
    int angle() const { return ang; }
    QSizePolicy sizePolicy() const;

public slots:
    void setAngle( int degrees );

signals:
    void angleChanged( int );
```

目前，**CanonField** 只包含一个角度值，我们使用了 **LCDRange** 中同样的方式。

```
protected:
    void paintEvent( QPaintEvent * );
```

这是我们在 `QWidget` 中遇到的许多事件处理器中的第二个。只要一个窗口部件需要刷新它自己（比如，画窗口部件表面），这个虚函数就会被 Qt 调用。

## t8/cannon.cpp

```
CannonField::CannonField( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
```

我们又一次使用和前一章中的 `LCDRange` 同样的方式。

```
    ang = 45;
    setPalette( QPalette( QColor( 250, 250, 200) ) );
}
```

构造函数把角度值初始化为 45 度并且给这个窗口部件设置了一个自定义调色板。

这个调色板只是说明背景色，并选择了其它合适的颜色。（对于这个窗口部件，只有背景色和文本颜色是要用到的。）

```
void CannonField::setAngle( int degrees )
{
    if ( degrees < 5 )
        degrees = 5;
    if ( degrees > 70 )
        degrees = 70;
    if ( ang == degrees )
        return;
    ang = degrees;
    repaint();
    emit angleChanged( ang );
}
```

这个函数设置角度值。我们选择了一个 5~70 的合法范围，并根据这个范围来调节给定的 `degrees` 的值。当新的角度值超过了范围，我们选择了不使用警告。

如果新的角度值和旧的一样，我们立即返回。这只对当角度值真的发生变化时，发射 `angleChanged()` 信号有重要意义。

然后我们设置新的角度值并重新画我们的窗口部件。`QWidget::repaint()` 函数清空窗口部件（通常用背景色来充满）并向窗口部件发出一个绘画事件。这样的结构就是调用窗口部件的绘画事件函数一次。

最后，我们发射 `angleChanged()` 信号来告诉外面的世界，角度值发生了变化。`emit` 关键字只是 Qt 中的关键字，而不是标准 C++ 的语法。实际上，它只是一个宏。

```
void CannonField::paintEvent( QPaintEvent * )
{
    QString s = "Angle = " + QString::number( ang );
    QPainter p( this );
    p.drawText( 200, 200, s );
}
```

这是我们第一次试图写一个绘画事件处理程序。这个事件参数包含一个绘画事件的描述。`QPaintEvent` 包含一个必须被刷新的窗口部件的区域。现在，我们比较懒惰，并且只是画每一件事。

我们的代码在一个固定位置显示窗口部件的角度值。首先我们创建一个含有一些文本和角度值的 `QString`，然后我们创建一个操作这个窗口部件的 `QPainter` 并使用它来画这个字符串。我们一会儿会回到 `QPainter`，它可以做很多事。

## t8/main.cpp

```
#include "cannon.h"
```

我们包含了我们的新类：

```
class MyWidget: public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};
```

这一次我们在顶层窗口部件中只使用了一个 `LCDRange` 和一个 `CanonField`。

```
LCDRange *angle = new LCDRange( this, "angle" );
```

在构造函数中，我们创建并设置了我们的 `LCDRange`。

```
angle->setRange( 5, 70 );
```

我们设置 `LCDRange` 能够接受的范围是 5~70 度。

```
CannonField *cannonField
    = new CannonField( this, "cannonField" );
```

我们创建了我们的 `CannonField`。

```
connect( angle, SIGNAL(valueChanged(int)),
         cannonField, SLOT(setAngle(int)) );
```



```
connect( cannonField, SIGNAL(angleChanged(int)),
        angle, SLOT(setValue(int)) );
```

这里我们把 LCDRange 的 valueChanged()信号和 CannonField 的 setAngle()槽连接起来了。只要用户操作 LCDRange，就会刷新 CannonField 的角度值。我们也把它反过来连接了，这样 CannonField 中角度的变化就可以刷新 LCDRange 的值。在我们的例子中，我们从来没有直接改变 CannonField 的角度，但是通过我们的最后一个 connect()我们就可以确保没有任何变化可以改变这两个值之间的同步关系。

这说明了组件编程和正确封装的能力。

注意只有当角度确实发生变化时，才发射 angleChanged()是多么的重要。如果 LCDRange 和 CanonField 都省略了这个检查，这个程序就会因为第一次数值变化而进入到一个无限循环当中。

```
QGridLayout *grid = new QGridLayout( this, 2, 2, 10 );
//2×2, 10 像素的边界
```

到现在为止，我们没有因为几何管理把 QVBox和 QGrid窗口部件集成到一起。现在，无论如何，我们需要对我们的布局加一些控制，所以我们使用了更加强大的 QGridLayout类。QGridLayout不是一个窗口部件，它是一个可以管理任何窗口部件作为子对象的不同的类。

就像注释中所说的，我们创建了一个以 10 像素为边界的 2\*2 的数组。（QGridLayout的构造函数有一点神秘，所以最好在这里加入一些注释。）

```
grid->addWidget( quit, 0, 0 );
```

我们在网格的左上的单元格中加入一个 Quit 按钮：0,0。

```
grid->addWidget( angle, 1, 0, Qt::AlignTop );
```

我们把 angle 这个 LCDRange 放到左下的单元格，在单元格内向上对齐。（这只是 QGridLayout 所允许的一种对齐方式，而 QGrid 不允许。）

```
grid->addWidget( cannonField, 1, 1 );
```

我们把 CannonField 对象放到右下的单元格。（右上的单元格是空的。）

```
grid->setColStretch( 1, 10 );
```

我们告诉 QGridLayout右边的列（列 1）是可拉伸的。因为左边的列不是（它的拉伸因数是 0，这是默认值），QGridLayout就会在MyWidget被重新定义大小的时候试图让左面的窗口部件大小不变，而重新定义CannonField的大小。

```
angle->setValue( 60 );
```

我们设置了一个初始角度值。注意这将会引发从 LCDRange 到 CannonField 的连接。

```
angle->setFocus();
```

我们刚才做的是设置angle获得 [键盘焦点](#)，这样默认情况下键盘输入会到达LCDRange窗口部件。

LCDRange 没有包含任何 keyPressEvent(), 所以这看起来不太可能有用。无论如何，它的构造函数中有了新的一行：

```
setFocusProxy( slider );
```

LCDRange设置滑块作为它的焦点代理。这就是说当程序或者用户想要给LCDRange一个键盘焦点，滑块就会注意到它。[QSlider](#)有一个相当好的键盘接口，所以就会出现我们给LCDRange添加的这一行。

## 行为

键盘现在可以做一些事了——方向键、Home、End、PageUp 和 PageDown 都可以作一些事情。

当滑块被操作，CannonFiled 会显示新的角度值。如果重新定义大小，CannonField 会得到尽可能多的空间。

在 8 位的 Windows 机器上显示新的颜色会颤动的要命。下一章会处理这些的。

（请看 [编译](#)来学习如何创建一个makefile和连编应用程序。）

## 练习

设置重新定义窗口的大小。如果你把它变窄或者变矮会发生什么？

如果你把 AlignTop 删掉，LCDRange 的位置会发生什么变化？为什么？

如果你给左面的列一个非零的拉伸因数，当你重新定义窗口大小时会发生什么？

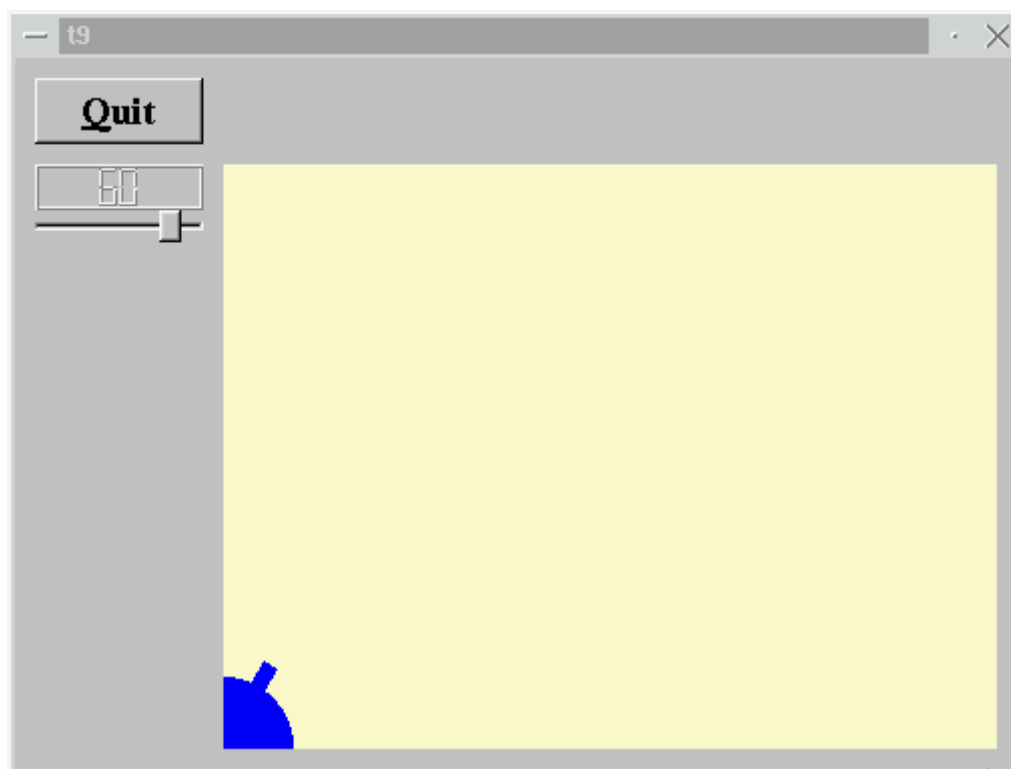
不考虑 setFocus()调用。你更喜欢什么样的行为？

试着在 [QPushButton::setText\(\)](#)调用中把“Quit”改为“&Quit”。按钮看起来变成什么样子了？如果你在程序运行的时候按下Alt+Q会发生什么？（在少量键盘中时Meta+Q。）

把 CannonField 的文本放到中间。

现在你可以进行 [第九章](#)了。

# Qt 教程一 —— 第九章：你可以使用加农炮了



在这个例子中我们开始画一个蓝色可爱的小加农炮.只 `cannon.cpp` 和上一章不同。

- `t9/lcdrange.h`包含LCDRange类定义。
- `t9/lcdrange.cpp`包含LCDRange类实现。
- `t9/cannon.h`包含CannonField类定义。
- `t9/cannon.cpp`包含CannonField类实现。
- `t9/main.cpp`包含MyWidget和main。

## 一行一行地解说

### `t9/cannon.cpp`

```
void CannonField::paintEvent( QPaintEvent * )
{
    QPainter p( this );
```

我们现在开始认真地使用 `QPainter`。我们创建一个绘画工具来操作这个窗口部件。

```
p. setBrush( blue );
```

当一个 **QPainter** 填满一个矩形、圆或者其它无论什么，它会用它的画刷填满这个图形。这里我们把画刷设置为蓝色。（我们也可以使用一个调色板。）

```
p. setPen( NoPen );
```

并且 **QPainter** 使用画笔来画边界。这里我们设置为 **NoPen**，就是说我们在边界上什么都不画，蓝色画刷会在我们画的东西的边界内画满全部。

```
p. translate( 0, rect().bottom() );
```

**QPainter::translate()**函数转化**QPainter**的坐标系，比如，它通过偏移来移动。这里我们设置窗口部件的左下角为(0,0)。x和y的方向没有改变，比如，窗口部件中的所有y坐标现在都是负数（请看 [坐标系](#)获得有关Qt的坐标系更多的信息。）

```
p. drawPie( QRect(-35, -35, 70, 70), 0, 90*16 );
```

**drawPie()**函数使用一个开始角度和弧长在一个指定的矩形内画一个饼型图。角度的度量用的是一度的十六分之一。零度在三点的的位置。画的方向是顺时针的。这里我们在窗口部件的左下角画一个四分之一圆。这个饼图被蓝色充满，并且没有边框。

```
p. rotate( -ang );
```

**QPainter::rotate()**函数绕 **QPainter**坐标系的初始位置旋转它。旋转的参数是一个按度数给定的浮点数（不是一个像上面那样给的十六分之一的度数）并且是顺时针的。这里我们顺时针旋转ang度数。

```
p. drawRect( QRect(33, -4, 15, 8) );
```

**QPainter::drawRect()**函数画一个指定的矩形。这里我们画的是加农炮的炮筒。

很难想象当坐标系被转换之后（转化、旋转、缩放或者修剪）的绘画结果。

在这种情况下，坐标系先被转化后被旋转。如果矩形 **QRect(33, -4, 15, 8)**被画到这个转化后的坐标系中，它看起来会是这样：



注意矩形被 **CannonField** 窗口部件的边界省略了一部分。当我们选装坐标系，以 60 度为例，矩形会以(0,0)为圆心被旋转，也就是左下角，因为我们已经转化了坐标系。结果会是这样：



我们做完了，除了我们还没有解释为什么 **Windows** 在这个时候没有发抖。

```
int main( int argc, char **argv )
{
    QApplication::setColorSpec( QApplication::CustomColor );
    QApplication a( argc, argv );
```

我们告诉Qt我们在这个程序中想使用一个不同的颜色分配策略。这里没有单一正确的颜色分配策略。因为这个程序使用了不常用的黄色，但不是很多颜色，CustomColor最好。这里有几个其它的分配策略，你可以在 [QApplication::setColorSpec\(\)](#) 文档中读到它们。

通常情况下你可以忽略这一点，因为默认的是好的。偶尔一些使用常用颜色的应用程序看起来比较糟糕，因而改变分配策略通常会有所帮助。

## 行为

当滑块被操作的时候，所画的加农炮的角度会因此而变化。

Quit 中的字母 Q 现在有下划线，并且 Alt+Q 会实现你所要的。如果你不知道这些，你一定没有做第八章中的练习。

你也要注意加农炮的闪烁让人很烦，特别是在一个比较慢的机器上。我们将会在下一章修正这一点。

（请看 [编译](#) 来学习如何创建一个makefile和连编应用程序。）

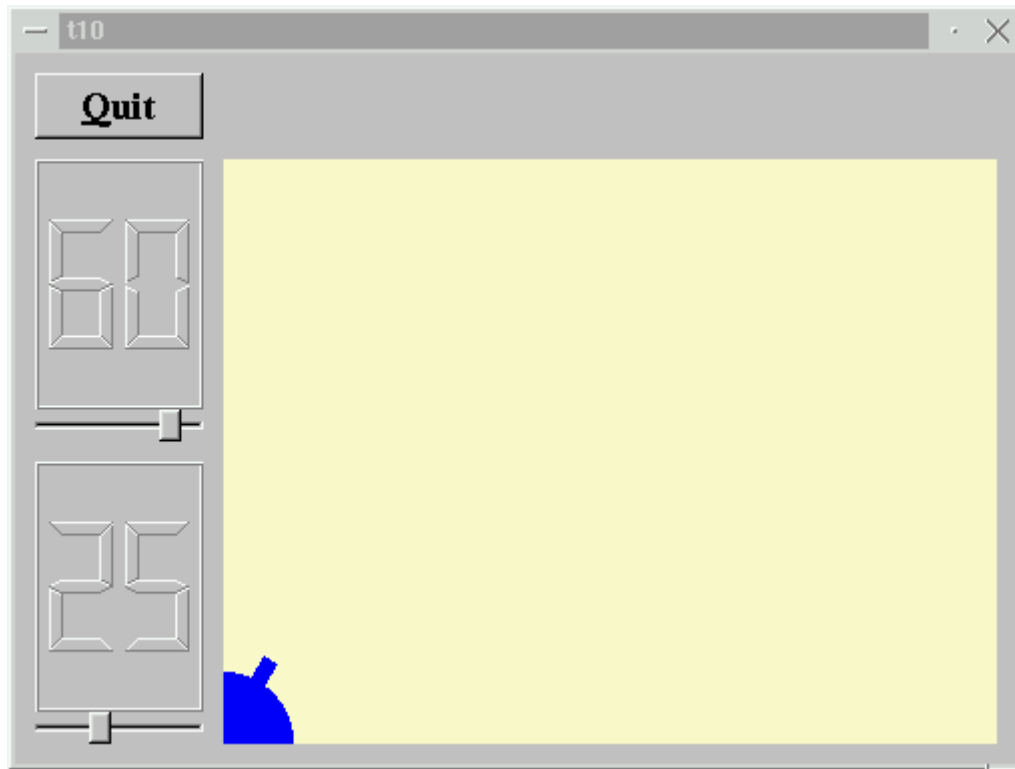
## 练习

设置一个不同的画笔代替 NoPen。设置一个调色板的画刷。

试着用“Q&uit”或者“Qu&it”作为按钮的文本来提到“&Quit”。发生了什么？

现在你可以进行 [第十章](#)了。

# Qt 教程一 —— 第十章：像丝一样滑



在这个例子中，我们介绍画一个 `pixmap` 来除去闪烁。我们也会加入一个力量控制。

- [t10/lcdrange.h](#)包含LCDRange类定义。
- [t10/lcdrange.cpp](#)包含LCDRange类实现。
- [t10/cannon.h](#)包含CannonField类定义。
- [t10/cannon.cpp](#)包含CannonField类实现。
- [t10/main.cpp](#)包含MyWidget和main。

## 一行一行地解说

### `t10/cannon.h`

CannonField 现在除了角度又多了一个力量值。

```
int    angle() const { return ang; }
int    force() const { return f; }

public slots:
    void setAngle( int degrees );
    void setForce( int newton );

signals:
    void angleChanged( int );
    void forceChanged( int );
```

力量的接口的实现和角度一样。

```
private:
    QRect cannonRect() const;
```

我们把加农炮封装的矩形的定义放到了一个单独的函数中。

```
    int ang;
    int f;
};
```

力量被存储到一个整数 f 中。

## t10/cannon.cpp

```
#include <qpixmap.h>
```

我们包含了 `QPixmap` 类定义。

```
CannonField::CannonField( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
    ang = 45;
    f = 0;
    setPalette( QPalette( QColor( 250, 250, 200 ) ) );
}
```

力量 (f) 被初始化为 0。

```
void CannonField::setAngle( int degrees )
{
    if ( degrees < 5 )
        degrees = 5;
    if ( degrees > 70 )
        degrees = 70;
    if ( ang == degrees )
        return;
    ang = degrees;
    repaint( cannonRect(), FALSE );
    emit angleChanged( ang );
}
```

我们在 `setAngle()` 函数中做了一个小的改变。它只重画窗口部件中含有加农炮的一小部分。`FALSE` 参数说明在一个绘画事件发送到窗口部件之前指定的矩形将不会被擦去。这将会使绘画过程加速和平滑。

```
void CannonField::setForce( int newton )
{
    if ( newton < 0 )
```

```

        newton = 0;
    if ( f == newton )
        return;
    f = newton;
    emit forceChanged( f );
}

```

`setForce()`的实现和 `setAngle()`很相似。唯一的不同是因为我们不显示力量值，我们不需要重画窗口部件。

```

void CannonField::paintEvent( QPaintEvent *e )
{
    if ( !e->rect().intersects( cannonRect() ) )
        return;
}

```

我们现在用只重画需要刷新得部分来优化绘画事件。首先我们检查是否不得不完全重画任何事，我们返回是否不需要。

```

QRect cr = cannonRect();
QPixmap pix( cr.size() );

```

然后，我们创建一个临时的 `pixmap`，我们用来不闪烁地画。所有的绘画操作都在这个 `pixmap` 中完成，并且之后只用一步操作来把这个 `pixmap` 画到屏幕上。

这是不闪烁绘画的本质：一次准确地在每一个像素上画。更少，你会得到绘画错误。更多，你会得到闪烁。在这个例子中这个并不重要——当代码被写时，仍然是很慢的机器导致闪烁，但以后不会再闪烁了。我们由于教育目的保留了这些代码。

```

pix.fill( this, cr.topLeft() );

```

我们用这个 `pixmap` 来充满这个窗口部件的背景。

```

QPainter p( &pix );
p.setBrush( blue );
p.setPen( NoPen );
p.translate( 0, pix.height() - 1 );
p.drawPie( QRect( -35, -35, 70, 70 ), 0, 90*16 );
p.rotate( -ang );
p.drawRect( QRect( 33, -4, 15, 8 ) );
p.end();

```

我们就像第九章中一样画，但是现在我们是在 `pixmap` 上画。

在这一点上，我们有一个绘画工具变量和一个 `pixmap` 看起来相当正确，但是我们还没有在屏幕上画呢。

```

p.begin( this );
p.drawPixmap( cr.topLeft(), pix );

```



所以我们在 CannonField 上面打开绘图工具并在这之后画这个 pixmap。

这就是全部了。在顶部和底部各有一对线，并且这个代码是 100% 不闪烁的。

```
QRect CannonField::cannonRect() const
{
    QRect r( 0, 0, 50, 50 );
    r.moveBottomLeft( rect().bottomLeft() );
    return r;
}
```

这个函数返回一个在窗口部件坐标中封装加农炮的矩形。首先我们创建一个 50\*50 大小的矩形，然后移动它，使它的左下角和窗口部件自己的左下角相等。

`QWidget::rect()`函数在窗口部件自己的坐标（左上角是 0,0）中返回窗口部件封装的矩形。

## t10/main.cpp

```
MyWidget::MyWidget( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
```

构造函数也是一样，但是已经加入了一些东西。

```
    LCDRange *force = new LCDRange( this, "force" );
    force->setRange( 10, 50 );
```

我们加入了第二个 LCDRange，用来设置力量。

```
    connect( force, SIGNAL(valueChanged(int)),
             cannonField, SLOT(setForce(int)) );
    connect( cannonField, SIGNAL(forceChanged(int)),
             force, SLOT(setValue(int)) );
```

我们把 force 窗口部件和 cannonField 窗口部件连接起来，就和我们之前对 angle 窗口部件做的一样。

```
    QVBoxLayout *leftBox = new QVBoxLayout;
    grid->addLayout( leftBox, 1, 0 );
    leftBox->addWidget( angle );
    leftBox->addWidget( force );
```

在第九章，我们把 angle 放到了布局的左下单元。现在我们要在这个单元中放入两个窗口部件，所以用一个垂直的盒子，把这个垂直的盒子放到这个网格单元中，并且把 angle 和 force 放到这个垂直的盒子中。

```
    force->setValue( 25 );
```

我们初始化力量的值为 25。

## 行为

闪烁已经走了，并且我们还有一个力量控制。

（请看 [编译](#) 来学习如何创建一个makefile和连编应用程序。）

## 练习

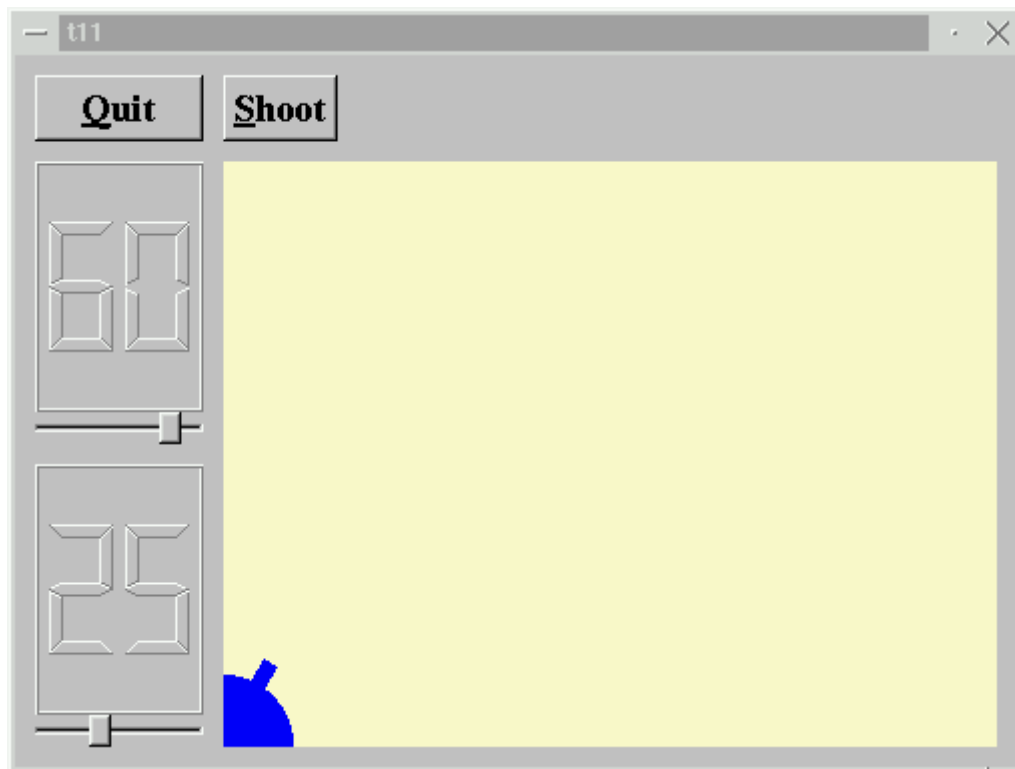
让加农炮的炮筒的大小依赖于力量。

把加农炮放到右下角。

试着加入一个更好的键盘接口。例如，用+和-来增加或者减少力量，用enter来发射。提示：[QAccel](#)和在LCDRange中新建addStep()和subtractStep()，就像 [QSlider::addStep\(\)](#)。如果你被左面和右面键所苦恼（我就是！），试着都改变！

现在你可以进行 [第十一章](#)了。

# Qt 教程一 —— 第十一章：给它一个炮弹



在这个例子里我们介绍了一个定时器来实现动画的射击。

- [t11/lcdrange.h](#)包含LCDRange类定义。
- [t11/lcdrange.cpp](#)包含LCDRange类实现。
- [t11/cannon.h](#)包含CannonField类定义。
- [t11/cannon.cpp](#)包含CannonField类实现。
- [t11/main.cpp](#)包含MyWidget和main。

## 一行一行地解说

### [t11/cannon.h](#)

CannonField 现在就有了射击能力。

```
void shoot();
```

当炮弹不在空中中，调用这个槽就会使加农炮射击。

```
private slots:  
    void moveShot();
```

当炮弹正在空中时，这个私有槽使用一个 [定时器](#)来移动射击。

```
private:  
    void paintShot( QPainter * );
```

这个函数来画射击。

```
QRect shotRect() const;
```

当炮弹正在空中的时候，这个私有函数返回封装它所占用空间的矩形，否则它就返回一个没有定义的矩形。

```
int timerCount;  
QTimer * autoShootTimer;  
float shoot_ang;  
float shoot_f;  
};
```

这些私有变量包含了描述射击的信息。timerCount 保留了射击进行后的时间。shoot\_ang 是加农炮射击时的角度，shoot\_f 是射击时加农炮的力量。

### [t11/cannon.cpp](#)

```
#include <math.h>
```

我们包含了数学库，因为我们需要使用 `sin()`和 `cos()`函数。

```
CannonField::CannonField( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
    ang = 45;
    f = 0;
    timerCount = 0;
    autoShootTimer = new QTimer( this, "movement handler" );
    connect( autoShootTimer, SIGNAL(timeout()),
            this, SLOT(moveShot()) );
    shoot_ang = 0;
    shoot_f = 0;
    setPalette( QPalette( QColor( 250, 250, 200) ) );
}
```

我们初始化我们新的私有变量并且把 `QTimer::timeout()`信号和我们的`moveShot()`槽相连。我们会在定时器超时的时候移动射击。

```
void CannonField::shoot()
{
    if ( autoShootTimer->isActive() )
        return;
    timerCount = 0;
    shoot_ang = ang;
    shoot_f = f;
    autoShootTimer->start( 50 );
}
```

只要炮弹不在空中，这个函数就会进行一次射击。`timerCount` 被重新设置为零。`shoot_ang`和 `shoot_f` 设置为当前加农炮的角度和力量。最后，我们开始这个定时器。

```
void CannonField::moveShot()
{
    QRegion r( shotRect() );
    timerCount++;

    QRect shotR = shotRect();

    if ( shotR.x() > width() || shotR.y() > height() )
        autoShootTimer->stop();
    else
        r = r.unite( QRegion( shotR ) );
    repaint( r );
}
```

`moveShot()`是一个移动射击的槽，当 `QTimer`开始的时候，每 50 毫秒被调用一次。

它的任务就是计算新的位置，重新画屏幕并把炮弹放到新的位置，并且如果需要的话，停止定时器。

首先我们使用 `QRegion` 来保留旧的 `shotRect()`。`QRegion` 可以保留任何种类的区域，并且我们可以用它来简化绘画过程。`shotRect()` 返回现在炮弹所在的矩形——稍后我们会详细介绍。

然后我们增加 `timerCount`，用它来实现炮弹在它的轨迹中移动的每一步。

下一步我们算出新的炮弹的矩形。

如果炮弹已经移动到窗口部件的右面或者下面的边界，我们停止定时器或者添加新的 `shotRect()` 到 `QRegion`。

最后，我们重新绘制 `QRegion`。这将会发送一个单一的绘画事件，但仅仅有一个到两个举行需要刷新。

```
void CannonField::paintEvent( QPaintEvent *e )
{
    QRect updateR = e->rect();
    QPainter p( this );

    if ( updateR.intersects( cannonRect() ) )
        paintCannon( &p );
    if ( autoShootTimer->isActive() &&
        updateR.intersects( shotRect() ) )
        paintShot( &p );
}
```

绘画事件函数在前一章中已经被分成两部分了。现在我们得到的新的矩形区域需要绘画，检查加农炮和/或炮弹是否相交，并且如果需要的话，调用 `paintCannon()` 和/或 `paintShot()`。

```
void CannonField::paintShot( QPainter *p )
{
    p->setBrush( black );
    p->setPen( NoPen );
    p->drawRect( shotRect() );
}
```

这个私有函数画一个黑色填充的矩形作为炮弹。

我们把 `paintCannon()` 的实现放到一边，它和前一章中的 `paintEvent()` 一样。

```
QRect CannonField::shotRect() const
{
    const double gravity = 4;

    double time      = timerCount / 4.0;
    double velocity  = shoot_f;
```

```

double radians    = shoot_ang*3.14159265/180;

double velx       = velocity*cos( radians );
double vely       = velocity*sin( radians );
double x0         = ( barrelRect.right() + 5 )*cos(radians);
double y0         = ( barrelRect.right() + 5 )*sin(radians);
double x          = x0 + velx*time;
double y          = y0 + vely*time - 0.5*gravity*time*time;

QRect r = QRect( 0, 0, 6, 6 );
r.moveCenter( QPoint( qRound(x), height() - 1 - qRound(y) ) );
return r;
}

```

这个私有函数计算炮弹的中心点并且返回封装炮弹的矩形。它除了使用自动增加所过去的时间的 `timerCount` 之外，还使用初始时的加农炮的力量和角度。

运算公式使用的是有重力的环境下光滑运动的经典牛顿公式。简单地说，我们已经选择忽略爱因斯坦理论的结果。

我们在一个y坐标向上增加的坐标系统中计算中心点。在我们计算出中心点之后，我们构造一个 6\*6 大小的 `QRect`，并移动它的中心到我们上面所计算出的中心点。同样的操作我们把这个点移动到窗口部件的坐标系统（请看 [坐标系统](#)）。

`qRound()`函数是一个在 `qglobal.h` 中定义的内嵌函数（被其它所有 Qt 头文件包含）。`qRound()` 把一个双精度实数变为最接近的整数。

## t11/main.cpp

```

class MyWidget: public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};

```

唯一的增加是 **Shoot** 按钮。

```

QPushButton *shoot = new QPushButton( "&Shoot", this, "shoot" );
shoot->setFont( QFont( "Times", 18, QFont::Bold ) );

```

在构造函数中我们创建和设置 **Shoot** 按钮就像我们对 **Quit** 按钮所做的那样。注意构造函数的第一个参数是按钮的文本，并且第三个是窗口部件的名称。

```

connect( shoot, SIGNAL(clicked()), cannonField, SLOT(shoot()) );

```

把 **Shoot** 按钮的 `clicked()`信号和 **CannonField** 的 `shoot()`槽连接起来。

## 行为

The cannon can shoot, but there's nothing to shoot at.

（请看 [编译](#) 来学习如何创建一个makefile和连编应用程序。）

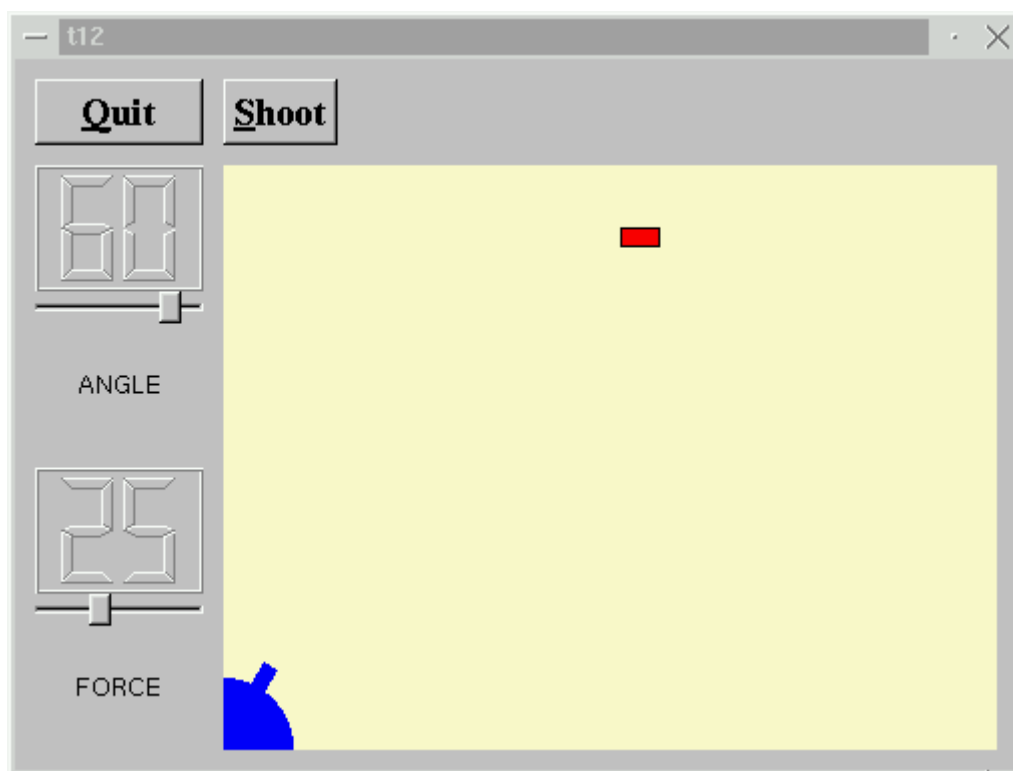
## 练习

用一个填充的圆来表示炮弹。提示：`QPainter::drawEllipse()`会对你有所帮助。

当炮弹在空中的时候，改变加农炮的颜色。

现在你可以进行 [第十二章](#)了。

# Qt 教程一 —— 第十二章：悬在空中的砖



在这个例子中，我们扩展我们的 `LCDRange` 类来包含一个文本标签。我们也会给射击提供一个目标。

- [t12/lcdrange.h](#) 包含 `LCDRange` 类定义。
- [t12/lcdrange.cpp](#) 包含 `LCDRange` 类实现。
- [t12/cannon.h](#) 包含 `CannonField` 类定义。

- [t12/cannon.cpp](#)包含CannonField类实现。
- [t12/main.cpp](#)包含MyWidget和main。

## 一行一行地解说

### t12/lcdrange.h

LCDRange 现在有了一个文本标签。

```
class QLabel;
```

我们名称声明 `QLabel`，因为我们将在这个类声明中使用一个`QLabel`的指针。

```
class LCDRange : public QVBox
{
    Q_OBJECT
public:
    LCDRange( QWidget *parent=0, const char *name=0 );
    LCDRange( const char *s, QWidget *parent=0,
               const char *name=0 );
```

我们添加了一个新的构造函数，这个构造函数在父对象和名称之外还设置了标签文本。

```
    const char *text() const;
```

这个函数返回标签文本。

```
    void setText( const char * );
```

这个槽设置标签文本。

```
private:
    void init();
```

因为我们现在有了两个构造函数，我们选择把通常的初始化放在一个私有的 `init()`函数。

```
    QLabel    *label;
```

我们还有一个新的私有变量：一个 `QLabel`。`QLabel` 是一个 Qt 标准窗口部件并且可以显示一个有或者没有框架的文本或者 `pixmap`。

### t12/lcdrange.cpp

```
#include <qlabel.h>
```

这里我们包含了 `QLabel`类定义。



```

LCDRange::LCDRange( QWidget *parent, const char *name )
    : QVBoxLayout( parent, name )
{
    init();
}

```

这个构造函数调用了 `init()` 函数，它包括了通常的初始化代码。

```

LCDRange::LCDRange( const char *s, QWidget *parent,
                    const char *name )
    : QVBoxLayout( parent, name )
{
    init();
    setText( s );
}

```

这个构造函数首先调用了 `init()` 然后设置标签文本。

```

void LCDRange::init()
{
    QLCDNumber *lcd = new QLCDNumber( 2, this, "lcd" );
    slider = new QSlider( Horizontal, this, "slider" );
    slider->setRange( 0, 99 );
    slider->setValue( 0 );

    label = new QLabel( " ", this, "label" );
    label->setAlignment( AlignCenter );

    connect( slider, SIGNAL(valueChanged(int)),
            lcd, SLOT(display(int)) );
    connect( slider, SIGNAL(valueChanged(int)),
            SIGNAL(valueChanged(int)) );

    setFocusProxy( slider );
}

```

`lcd` 和 `slider` 的设置和上一章一样。接下来我们创建一个 `QLabel` 并且让它的内容中间对齐（垂直方向和水平方向都是）。`connect()` 语句也来自于上一章。

```

const char *LCDRange::text() const
{
    return label->text();
}

```

这个函数返回标签文本。

```

void LCDRange::setText( const char *s )
{

```

```
        label->setText( s );
    }
```

这个函数设置标签文本。

## t12/cannon.h

**CannonField** 现在有两个新的信号：**hit()**和 **missed()**。另外它还包含一个目标。

```
void newTarget();
```

这个槽在新的位置生成一个新的目标。

```
signals:
    void hit();
    void missed();
```

**hit()**信号是当炮弹击中目标的时候被发射的。**missed()**信号是当炮弹移动超出了窗口部件的右面或者下面的边界时被发射的（例如，当然这种情况下它将不会击中目标）。

```
void paintTarget( QPainter * );
```

这个私有函数绘制目标。

```
QRect targetRect() const;
```

这个私有函数返回一个封装了目标的矩形。

```
QPoint target;
```

这个私有变量包含目标的中心点。

## t12/cannon.cpp

```
#include <qdatetime.h>
```

我们包含了 **QDate**、**QTime**和 **QDateTime**类定义。

```
#include <stdlib.h>
```

我们包含了 **stdlib** 库，因为我们需要 **rand()**函数。

```
newTarget();
```

这一行已经被添加到了构造函数中。它为目标创建一个“随机的”位置。实际上，**newTarget()**函数还试图绘制目标。因为我们在一个构造函数中，**CannonField** 窗口部件还是不可以见的。**Qt** 保证在一个隐藏的窗口部件中调用 **repaint()**是没有害处的。

```

void CannonField::newTarget()
{
    static bool first_time = TRUE;
    if ( first_time ) {
        first_time = FALSE;
        QTime midnight( 0, 0, 0 );
        srand( midnight.secsTo(QTime::currentTime\(\)) );
    }
    QRegion r( targetRect() );
    target = QPoint( 200 + rand() % 190,
                    10 + rand() % 255 );
    repaint( r.unite( targetRect() ) );
}

```

这个私有函数创建了一个在新的“随机的”位置的目标中心点。

我们使用 `rand()` 函数来获得随机整数。`rand()` 函数通常会在你每次运行这个程序的时候返回同样一组值。这就会使每次运行的时候目标都出现在同样的位置。为了避免这些，我们必须在这个函数第一次被调用的时候设置一个随机种子。为了避免同样一组数据，随机种子也必须是随机的。解决方法就是使用从午夜到现在的秒数作为一个假的随机值。

首先我们创建一个静态布尔型局域变量。静态变量就是在调用函数前后都保证它的值不变。

`if` 测试会成功，因为只有当这个函数第一次被调用的时候，我们在 `if` 块中把 `first_time` 设置为 `FALSE`。

然后我们创建一个 [QTime](#) 对象 `midnight`，它将会提供时间 00:00:00。接下来我们获得从午夜到现在所过的秒数并且使用它作为一个随机种子。请看 [QDate](#)、[QTime](#) 和 [QDateTime](#) 文档来获得更多的信息。

最后我们计算目标的中心点。我们把它放在一个矩形中（`x=200`，`y=35`，`width=190`，`height=255`），（例如，可能的 `x` 和 `y` 值是 `x=200~389` 和 `y=35~289`）在一个我们把窗口边界的下边界作为 `y` 的零点，并且 `y` 向上增加，`X` 轴向通常一样，左边界为零点，并且 `x` 向右增加的坐标系统中。

通过经验，我们发现这都在炮弹的射程之内。

注意 `rand()` 返回一个 `>=0` 的随机整数。

```

void CannonField::moveShot()
{
    QRegion r( shotRect() );
    timerCount++;

    QRect shotR = shotRect();
}

```

定时器时间这部分和上一章一样。

```

        if ( shotR.intersects( targetRect() ) ) {
            autoShootTimer->stop();
            emit hit();
        }

```

if 语句检查炮弹矩形和目标矩形是否相交。如果是的，炮弹击中了目标（哎哟！）。我们停止射击定时器并且发射 hit()信号来告诉外界目标已经被破坏，并返回。

注意，我们可以在这个点上创建一个新的目标，但是因为 CannonField 是一个组件，所以我们要把这样的决定留给组件的使用者。

```

        } else if ( shotR.x() > width() || shotR.y() > height() ) {
            autoShootTimer->stop();
            emit missed();
        }

```

这个 if 语句和上一章一样，除了现在它发射 missed()信号告诉外界这次失败。

```

    } else {

```

函数的其余部分和以前一样。

CannonField::paintEvent() is as before, except that this has been added:

```

        if ( updateR.intersects( targetRect() ) )
            paintTarget( &p );

```

这两行确认在需要的时候目标也被绘制。

```

void CannonField::paintTarget( QPainter *p )
{
    p->setBrush( red );
    p->setPen( black );
    p->drawRect( targetRect() );
}

```

这个私有函数绘制目标，一个由红色填充，有黑色边框的矩形。

```

QRect CannonField::targetRect() const
{
    QRect r( 0, 0, 20, 10 );
    r.moveCenter( QPoint( target.x(), height() - 1 - target.y() ) );
    return r;
}

```

这个私有函数返回封装目标的矩形。从newTarget()中所得的target点使用 0 点在窗口部件的下边界的y。我们在调用 QRect::moveCenter()之前在窗口坐标中计算这个点。

我们选择这个坐标映射的原因是在目标和窗口部件的下边界之间垂直距离。记住这些可以让用户或者程序在任何时候都可以重新定义窗口部件的大小。

## t12/main.cpp

在 `MyWidget` 类中没有新的成员了，但是我们稍微改变了一下构造函数来设置新的 `LCDRange` 的文本标签。

```
LCDRange *angle = new LCDRange( "ANGLE", this, "angle" );
```

我们设置角度的文本标签为“ANGLE”。

```
LCDRange *force = new LCDRange( "FORCE", this, "force" );
```

我们设置力量的文本标签为“FORCE”。

## 行为

加农炮会向目标射击，当它射中目标的时候，一个新的目标会自动被创建。

`LCDRange` 窗口部件看起来有一点奇怪——`QVBoxLayout` 中内置的管理给了标签太多的空间而其它的却不够。我们将会在下一章修正这一点。

（请看 [编译](#) 来学习如何创建一个 `makefile` 和连编应用程序。）

## 练习

创建一个作弊的按钮，当按下它的时候，让 `CannonField` 画出炮弹在五秒中的轨迹。

如果你在上一章做了“圆形炮弹”的练习，试着改变 `shotRect()` 为可以返回一个 `QRegion` 的 `shotRegion()`，这样你就可以真正的做到准确碰撞检测。

做一个移动目标。

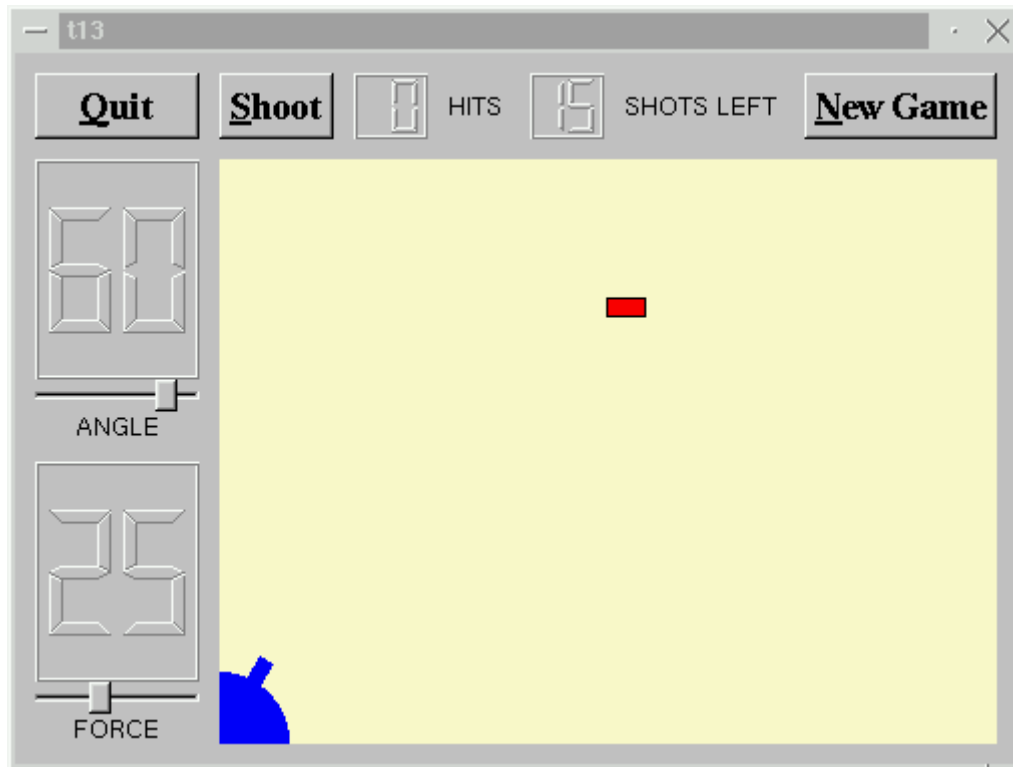
确认目标被完全创建在屏幕上。

确认加农炮窗口部件不能被重新定义大小，这样目标不是可见的。提示：`QWidget::setMinimumSize()` 是你的朋友。

不容易的是在同一时刻让几个炮弹在空中成为可能。提示：建立一个炮弹对象。

现在你可以进行 [第十三章](#)了。

# Qt 教程一 —— 第十三章：游戏结束



在这个例子中我们开始研究一个带有记分的真正可玩的游戏。我们给 `MyWidget` 一个新的名字（`GameBoard`）并添加一些槽。

我们把定义放在 `gamebrd.h` 并把实现放在 `gamebrd.cpp`。

`CannonField` 现在有了一个游戏结束状态。

在 `LCDRange` 中的布局问题已经修好了。

- [t13/lcdrange.h](#) 包含 `LCDRange` 类定义。
- [t13/lcdrange.cpp](#) 包含 `LCDRange` 类实现。
- [t13/cannon.h](#) 包含 `CannonField` 类定义。
- [t13/cannon.cpp](#) 包含 `CannonField` 类实现。
- [t13/gamebrd.h](#) 包含 `GameBoard` 类定义。
- [t13/gamebrd.cpp](#) 包含 `GameBoard` 类实现。
- [t13/main.cpp](#) 包含 `MyWidget` 和 `main`。

## 一行一行地解说

### `t13/lcdrange.h`

```
#include <qwidget.h>
```

```
class QSlider;  
class QLabel;
```

```
class LCDRange : public QWidget
```

我们继承了 `QWidget` 而不是 `QVBox`。`QVBox` 是很容易使用的，但是它也显示了它的局域性，所以我们选择使用更加强大和稍微有一些难的 `QVBoxLayout`。（和你记忆中的一样，`QVBoxLayout` 不是一个窗口部件，它管理窗口部件。）

## t13/lcdrange.cpp

```
#include <qlayout.h>
```

我们现在需要包含 `qlayout.h` 来获得其它布局管理 API。

```
LCDRange::LCDRange( QWidget *parent, const char *name )  
    : QWidget( parent, name )
```

我们使用一种平常的方式继承 `QWidget`。

另外一个构造函数作了同样的改动。`init()` 没有变化，除了我们在最后加了几行：

```
QVBoxLayout * l = new QVBoxLayout( this );
```

我们使用所有默认值创建一个 `QVBoxLayout`，管理这个窗口部件的子窗口部件。

```
l->addWidget( lcd, 1 );
```

At the top we add the `QLCDNumber` with a non-zero stretch.

```
l->addWidget( slider );  
l->addWidget( label );
```

然后我们添加另外两个，它们都使用默认的零伸展因数。

这个伸展控制是 `QVBoxLayout`（和 `QHBoxLayout`，和 `QGridLayout`）所提供的，而像 `QVBox` 这样的类却不提供。在这种情况下我们让 `QLCDNumber` 可以伸展，而其它的不可以。

## t13/cannon.h

`CannonField` 现在有一个游戏结束状态和一些新的函数。

```
bool gameOver() const { return gameEnded; }
```

如果游戏结束了，这个函数返回 `TRUE`，或者如果游戏还在继续，返回 `FALSE`。

```
void setGameOver();  
void restartGame();
```

这里是两个新槽：`setGameOver()` 和 `restartGame()`。

```
void canShoot( bool );
```

这个新的信号表明 **CannonField** 使 `shoot()`槽生效的状态。我们将在下面使用它用来使 **Shoot** 按钮生效或失效。

```
bool gameEnded;
```

这个私有变量包含游戏的状态。**TRUE** 说明游戏结束，**FALSE** 说明游戏还将继续。

## t13/cannon.cpp

```
gameEnded = FALSE;
```

这一行已经被加入到构造函数中。最开始的时候，游戏没有结束（对于玩家是很幸运的 :-）。

```
void CannonField::shoot()
{
    if ( isShooting() )
        return;
    timerCount = 0;
    shoot_ang = ang;
    shoot_f = f;
    autoShootTimer->start( 50 );
    emit canShoot( FALSE );
}
```

我们添加一个新的 `isShooting()`函数，所以 `shoot()`使用它替代直接的测试。同样，`shoot` 告诉世界 **CannonField** 现在不可以射击。

```
void CannonField::setGameOver()
{
    if ( gameEnded )
        return;
    if ( isShooting() )
        autoShootTimer->stop();
    gameEnded = TRUE;
    repaint();
}
```

这个槽终止游戏。它必须被 **CannonField** 外面的调用，因为这个窗口部件不知道什么时候终止游戏。这是组件编程中一条重要设计原则。我们选择使组件可以尽可能灵活以适应不同的规则（比如，在一个首先射中十次的人胜利的多人游戏版本可能使用不变的 **CannonField**）。

如果游戏已经被终止，我们立即返回。如果游戏会继续到我们的设计完成，设置游戏结束标志，并且重新绘制整个窗口部件。

```
void CannonField::restartGame()
```



```

{
    if ( isShooting() )
        autoShootTimer->stop();
    gameEnded = FALSE;
    repaint();
    emit canShoot( TRUE );
}

```

这个槽开始一个新游戏。如果炮弹还在空中，我们停止设计。然后我们重置 gameEnded 变量并重新绘制窗口部件。

就像 hit()或 miss()一样，moveShot()同时也发射新的 canShoot(TRUE)信号。

CannonField::paintEvent()的修改：

```

void CannonField::paintEvent( QPaintEvent *e )
{
    QRect updateR = e->rect();
    QPainter p( this );

    if ( gameEnded ) {
        p.setPen( black );
        p.setFont( QFont( "Courier", 48, QFont::Bold ) );
        p.drawText( rect(), AlignCenter, "Game Over" );
    }
}

```

绘画事件已经通过如果游戏结束，比如 gameEnded 是 TRUE，就显示文本“Game Over”而被增强了。我们在这里不怕麻烦来检查更新矩形，是因为在游戏结束的时候速度不是关键性的。

为了画文本，我们先设置了黑色的画笔，当画文本的时候，画笔颜色会被用到。接下来我们选择 Courier 字体中的 48 号加粗字体。最后我们在窗口部件的矩形中央绘制文本。不幸的是，在一些系统中（特别是使用 Unicode 的 X 服务器）它会用一小段时间来载入如此大的字体。因为 Qt 缓存字体，我们只有第一次使用这个字体的时候才会注意到这一点。

```

    if ( updateR.intersects( cannonRect() ) )
        paintCannon( &p );
    if ( isShooting() && updateR.intersects( shotRect() ) )
        paintShot( &p );
    if ( !gameEnded && updateR.intersects( targetRect() ) )
        paintTarget( &p );
}

```

我们只有在设计的时候画炮弹，在玩游戏的时候画目标（这也就是说，当游戏没有结束的时候）。

## t13/gamebrd.h

这个文件是新的。它包含最后被用来作为 MyWidget 的 GameBoard 类的定义。

```
class QPushButton;
class LCDRange;
class QLCDNumber;
class CannonField;

#include "lcdrange.h"
#include "cannon.h"

class GameBoard : public QWidget
{
    Q_OBJECT
public:
    GameBoard( QWidget *parent=0, const char *name=0 );

protected slots:
    void fire();
    void hit();
    void missed();
    void newGame();

private:
    QLCDNumber *hits;
    QLCDNumber *shotsLeft;
    CannonField *cannonField;
};
```

我们现在已经添加了四个槽。这些槽都是被保护的，只在内部使用。我们也已经加入了两个 QLCDNumbers (hits 和 shotsLeft) 用来显示游戏的状态。

## t13/gamebrd.cpp

这个文件是新的。它包含最后被用来作为 MyWidget 的 GameBoard 类的实现，

我们已经在 GameBoard 的构造函数中做了一些修改。

```
cannonField = new CannonField( this, "cannonField" );
```

cannonField 现在是一个成员变量，所以我们在使用它的时候要小心地改变它的构造函数。（Trolltech 的好程序员从来不会忘记这点，但是我就忘了。告诫程序员—如果“programmor”是拉丁语，至少。无论如何，返回代码。）

```
connect( cannonField, SIGNAL(hit()),
```

```

        this, SLOT(hit()) );
connect( cannonField, SIGNAL(missed()),
        this, SLOT(missed()) );

```

这次当炮弹射中或者射失目标的时候，我们想做些事情。所以我们将 CannonField 的 hit() 和 missed() 信号连接到这个类的两个被保护的槽。

```

connect( shoot, SIGNAL(clicked()), SLOT(fire()) );

```

以前我们直接把 Shoot 按钮的 clicked() 信号连接到 CannonField 的 shoot() 槽。这次我们想跟踪射击的次数，所以我们把它改为连接到这个类里面一个被保护的槽。

注意当你用独立的组件工作的时候，改变程序的行为是多么的容易。

```

connect( cannonField, SIGNAL(canShoot(bool)),
        shoot, SLOT(setEnabled(bool)) );

```

我们也使用 cannonField 的 canShoot() 信号来适当地使 Shoot 按钮生效和失效。

```

QPushButton *restart
    = new QPushButton( "&New Game", this, "newgame" );
restart->setFont( QFont( "Times", 18, QFont::Bold ) );

connect( restart, SIGNAL(clicked()), this, SLOT(newGame()) );

```

我们创建、设置并且连接这个 New Game 按钮就像我们对其它按钮所做的一样。点击这个按钮就会激活这个窗口部件的 newGame() 槽。

```

hits = new QLCDNumber( 2, this, "hits" );
shotsLeft = new QLCDNumber( 2, this, "shotsleft" );
QLabel *hitsL = new QLabel( "HITS", this, "hitsLabel" );
QLabel *shotsLeftL
    = new QLabel( "SHOTS LEFT", this, "shotsleftLabel" );

```

我们创建了四个新的窗口部件。注意我们不怕麻烦的把 QLabel 窗口部件的指针保留到 GameBoard 类中是因为我们不想再对它们做什么了。当 GameBoard 窗口部件被销毁的时候，Qt 将会删除它们，并且布局类会适当地重新定义它们的大小。

```

QHBoxLayout *topBox = new QHBoxLayout;
grid->addLayout( topBox, 0, 1 );
topBox->addWidget( shoot );
topBox->addWidget( hits );
topBox->addWidget( hitsL );
topBox->addWidget( shotsLeft );
topBox->addWidget( shotsLeftL );
topBox->addStretch( 1 );
topBox->addWidget( restart );

```

右上单元格的窗口部件的数量正在变大。从前它是空的，现在它是完全充足的，我们把它放到布局中来更好的看到它们。

注意我们让所有的窗口部件获得它们更喜欢的大小，改为在 **New Game** 按钮的左边加入了一个可以自由伸展的东西。

```
        newGame();
    }
```

我们已经做完了所有关于 **GameBoard** 的构造，所以我们使用 **newGame()** 来开始。(newGame() 是一个槽，但是就像我们所说的，槽也可以像普通的函数一样使用。)

```
void GameBoard::fire()
{
    if ( cannonField->gameOver() || cannonField->isShooting() )
        return;
    shotsLeft->display( shotsLeft->intValue() - 1 );
    cannonField->shoot();
}
```

这个函数进行射击。如果游戏结束了或者还有一个炮弹在空中，我们立即返回。我们减少炮弹的数量并告诉加农炮进行射击。

```
void GameBoard::hit()
{
    hits->display( hits->intValue() + 1 );
    if ( shotsLeft->intValue() == 0 )
        cannonField->setGameOver();
    else
        cannonField->newTarget();
}
```

当炮弹击中目标的时候这个槽被激活。我们增加射中的数量。如果没有炮弹了，游戏就结束了。否则，我们会让 **CannonField** 生成新的目标。

```
void GameBoard::missed()
{
    if ( shotsLeft->intValue() == 0 )
        cannonField->setGameOver();
}
```

当炮弹射失目标的时候这个槽被激活，如果没有炮弹了，游戏就结束了。

```
void GameBoard::newGame()
{
    shotsLeft->display( 15 );
    hits->display( 0 );
    cannonField->restartGame();
}
```

```
cannonField->newTarget();  
}
```

当用户点击 **Restart** 按钮的时候这个槽被激活。它也会被构造函数调用。首先它把炮弹的数量设置为 15。注意这里是我们在程序中唯一设置炮弹数量的地方。把它改变为你所想要的游戏规则。接下来我们重置射中的数量，重新开始游戏，并且生成一个新的目标。

## t13/main.cpp

这个文件仅仅被删掉了一部分。**MyWidget** 没了，并且唯一剩下的是 `main()` 函数，除了名称的改变其它都没有改变。

## 行为

射中的和剩余炮弹的数量被显示并且程序继续跟踪它们。游戏可以结束了，并且还有一个按钮可以开始一个新游戏。

（请看 [编译](#) 来学习如何创建一个 `makefile` 和连编应用程序。）

## 练习

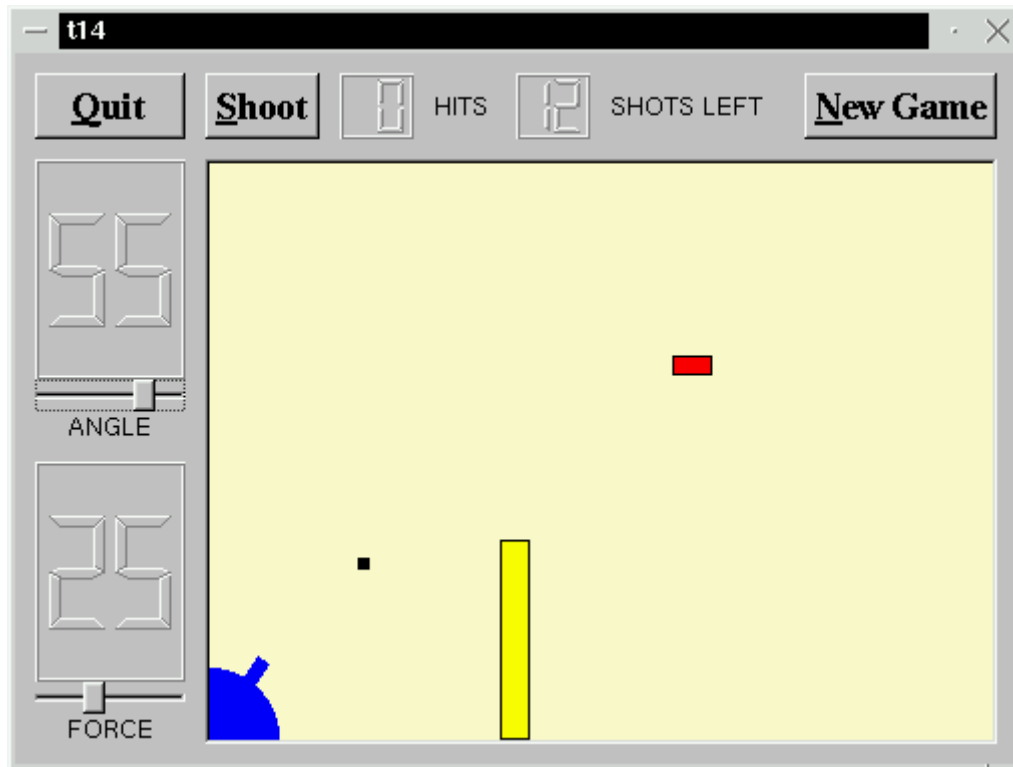
添加一个随机的风的因素并把它显示给用户看。

当炮弹击中目标的时候做一些飞溅的效果。

实现多个目标。

现在你可以进行 [第十四章](#)了。

# Qt 教程一 —— 第十四章：面对墙壁



这是最后的例子：一个完整的游戏。

我们添加键盘快捷键并引入鼠标事件到 `CannonField`。我们在 `CannonField` 周围放一个框架并添加一个障碍物（墙）使这个游戏更富有挑战性。

- [t14/lcdrange.h](#) 包含 `LCDRange` 类定义。
- [t14/lcdrange.cpp](#) 包含 `LCDRange` 类实现。
- [t14/cannon.h](#) 包含 `CannonField` 类定义。
- [t14/cannon.cpp](#) 包含 `CannonField` 类实现。
- [t14/gamebrd.h](#) 包含 `GameBoard` 类定义。
- [t14/gamebrd.cpp](#) 包含 `GameBoard` 类实现。
- [t14/main.cpp](#) 包含 `MyWidget` 和 `main`。

## 一行一行地解说

### [t14/cannon.h](#)

`CannonField` 现在可以接收鼠标事件，使得用户可以通过点击和拖拽炮筒来瞄准。`CannonField` 也有一个障碍物的墙。

```
protected:
    void paintEvent( QPaintEvent * );
    void mousePressEvent( QMouseEvent * );
    void mouseMoveEvent( QMouseEvent * );
    void mouseReleaseEvent( QMouseEvent * );
```

除了常见的事件处理器，`CannonField` 实现了三个鼠标事件处理器。名称说明了一切。

```
void paintBarrier( QPainter * );
```

这个私有函数绘制了障碍物墙。

```
QRect barrierRect() const;
```

这个私有函数返回封装障碍物的矩形。

```
bool barrelHit( const QPoint & ) const;
```

这个私有函数检查是否一个点在加农炮炮筒的内部。

```
bool barrelPressed;
```

当用户在炮筒上点击鼠标并且没有放开的话，这个私有变量为 `TRUE`。

## t14/cannon.cpp

```
barrelPressed = FALSE;
```

这一行被添加到构造函数中。最开始的时候，鼠标没有在炮筒上点击。

```
    } else if ( shotR.x() > width() || shotR.y() > height() ||  
                shotR.intersects(barrierRect()) ) {
```

现在我们有了一个障碍物，这样就有了三种射失的方法。我们来测试一下第三种。

```
void CannonField::mousePressEvent( QMouseEvent *e )  
{  
    if ( e->button() != LeftButton )  
        return;  
    if ( barrelHit( e->pos() ) )  
        barrelPressed = TRUE;  
}
```

这是一个 Qt 事件处理器。当鼠标指针在窗口部件上，用户按下鼠标的按键时，它被调用。

如果事件不是由鼠标左键产生的，我们立即返回。否则，我们检查鼠标指针是否在加农炮的炮筒内。如果是的，我们设置 `barrelPressed` 为 `TRUE`。

注意 `pos()` 函数返回的是窗口部件坐标系统中的点。

```
void CannonField::mouseMoveEvent( QMouseEvent *e )  
{  
    if ( !barrelPressed )  
        return;
```

```

    QPoint pnt = e->pos();
    if ( pnt.x() <= 0 )
        pnt.setX( 1 );
    if ( pnt.y() >= height() )
        pnt.setY( height() - 1 );
    double rad = atan(((double)rect().bottom()-pnt.y())/pnt.x());
    setAngle( qRound ( rad*180/3.14159265 ) );
}

```

这是另外一个Qt事件处理器。当用户已经在窗口部件中按下了鼠标按键并且移动/拖拽鼠标时，它被调用。（你可以让Qt在没有鼠标按键被按下的时候发送鼠标移动事件。请看 [QWidget::setMouseTracking\(\)](#)。）

这个处理器根据鼠标指针的位置重新配置加农炮的炮筒。

首先，如果炮筒没有被按下，我们返回。接下来，我们获得鼠标指针的位置。如果鼠标指针到了窗口部件的左面或者下面，我们调整鼠标指针使它返回到窗口部件中。

然后我们计算在鼠标指针和窗口部件的左下角所构成的虚构的线和窗口部件下边界的角度。最后，我们把加农炮的角度设置为我们新算出来的角度。

记住要用 `setAngle()`来重新绘制加农炮。

```

void CannonField::mouseReleaseEvent( QMouseEvent *e )
{
    if ( e->button() == LeftButton )
        barrelPressed = FALSE;
}

```

只要用户释放鼠标按钮并且它是在窗口部件中按下的时候，这个Qt事件处理器就会被调用。

如果鼠标左键被释放，我们会确认炮筒不再被按下了。

绘画事件包含了下述额外的两行：

```

    if ( updateR.intersects( barrierRect() ) )
        paintBarrier( &p );

```

`paintBarrier()`做的和 `paintShot()`、`paintTarget()`和 `paintCannon()`是同样的事情。

```

void CannonField::paintBarrier( QPainter *p )
{
    p->setBrush( yellow );
    p->setPen( black );
    p->drawRect( barrierRect() );
}

```

这个私有函数用一个黑色边界黄色填充的矩形作为障碍物。



```

QRect CannonField::barrierRect() const
{
    return QRect( 145, height() - 100, 15, 100 );
}

```

这个私有函数返回障碍物的矩形。我们把障碍物的下边界和窗口部件的下边界放在了一起。

```

bool CannonField::barrelHit( const QPoint &p ) const
{
    QWMatrix mtx;
    mtx.translate( 0, height() - 1 );
    mtx.rotate( -ang );
    mtx = mtx.invert();
    return barrelRect.contains( mtx.map(p) );
}

```

如果点在炮筒内，这个函数返回 **TRUE**；否则它就返回 **FALSE**。

这里我们使用 **QWMatrix**类。它是在头文件qwmatrix.h中定义的，这个头文件被qpainter.h包含。

**QWMatrix**定义了一个坐标系统映射。它可以执行和 **QPainter**中一样的转换。

这里我们实现同样的转换的步骤就和我们在 **paintCannon()**函数中绘制炮筒的时候所作的一样。首先我们转换坐标系统，然后我们旋转它。

现在我们需要检查点 **p**（在窗口部件坐标系统中）是否在炮筒内。为了做到这一点，我们倒置这个转换矩阵。倒置的矩阵就执行了我们在绘制炮筒时使用的倒置的转换。我们通过使用倒置矩阵来映射点 **p**，并且如果它在初始的炮筒矩形内就返回 **TRUE**。

## t14/gamebrd.cpp

```

#include <qaccel.h>

```

我们包含 **QAccel**的类定义。

```

QVBox *box = new QVBox( this, "cannonFrame" );
box->setFrameStyle( QFrame::WinPanel | QFrame::Sunken );
cannonField = new CannonField( box, "cannonField" );

```

我们创建并设置一个 **QVBox**，设置它的框架风格，并在之后创建CannonField作为这个盒子的子对象。因为没有其它的东西在这个盒子里了，效果就是 **QVBox**会在CannonField周围生成了一个框架。

```

QAccel *accel = new QAccel( this );
accel->connectItem( accel->insertItem( Key_Enter ),
                  this, SLOT( fire() ) );

```

```
accel->connectItem( accel->insertItem( Key_Return ),
                  this, SLOT(fire()) );
```

现在我们创建并设置一个加速键。加速键就是在应用程序中截取键盘事件并且如果特定的键被按下的时候调用相应的槽。这种机制也被称为快捷键。注意快捷键是窗口部件的子对象并且当窗口部件被销毁的时候销毁。**QAccel**不是窗口部件，并且在它的父对象中没有任何可见的效果。

我们定义两个快捷键。我们希望在 **Enter** 键被按下的时候调用 **fire()**槽，在 **Ctrl+Q** 键被按下的时候，应用程序退出。因为 **Enter** 有时又被称为 **Return**，并且有时键盘中两个键都有，所以我们让这两个键都调用 **fire()**。

```
accel->connectItem( accel->insertItem( CTRL+Key_Q ),
                  qApp, SLOT(quit()) );
```

并且之后我们设置 **Ctrl+Q** 和 **Alt+Q** 做同样的事情。一些人通常使用 **Ctrl+Q** 更多一些（并且无论如何它显示了如果做到它）。

**CTRL**、**Key\_Enter**、**Key\_Return**和**Key\_Q**都是Qt提供的常量。它们实际上就是Qt::Key\_Enter等等，但是实际上所有的类都继承了 **Qt**这个命名空间类。

```
QGridLayout *grid = new QGridLayout( this, 2, 2, 10 );
grid->addWidget( quit, 0, 0 );
grid->addWidget( box, 1, 1 );
grid->setColStretch( 1, 10 );
```

我们放置box（**QVBox**），不是**CannonField**，在右下的单元格中。

## 行为

现在当你按下 **Enter** 的时候，加农炮就会发射。你也可以用鼠标来确定加农炮的角度。障碍物会使你在玩游戏的时候获得更多一点的挑战。我们还会在 **CannnonField** 周围看到一个好看的框架。

（请看 [编译](#)来学习如何创建一个makefile和连编应用程序。）

## 练习

写一个空间入侵者的游戏。

（这个练习首先被 [Igor Rafienko](#)作出来了。你可以 [下载他的游戏](#)。）

新的练习是：写一个突围游戏。

# Qt 教程二

这个教程会提供一个比第一个教程更加“真实世界”的 Qt 编程实例。它介绍了 Qt 编程的许多方面，介绍了创建菜单（包括最近使用文件列表）、工具条和对话框、载入和保存用户设置，等等。

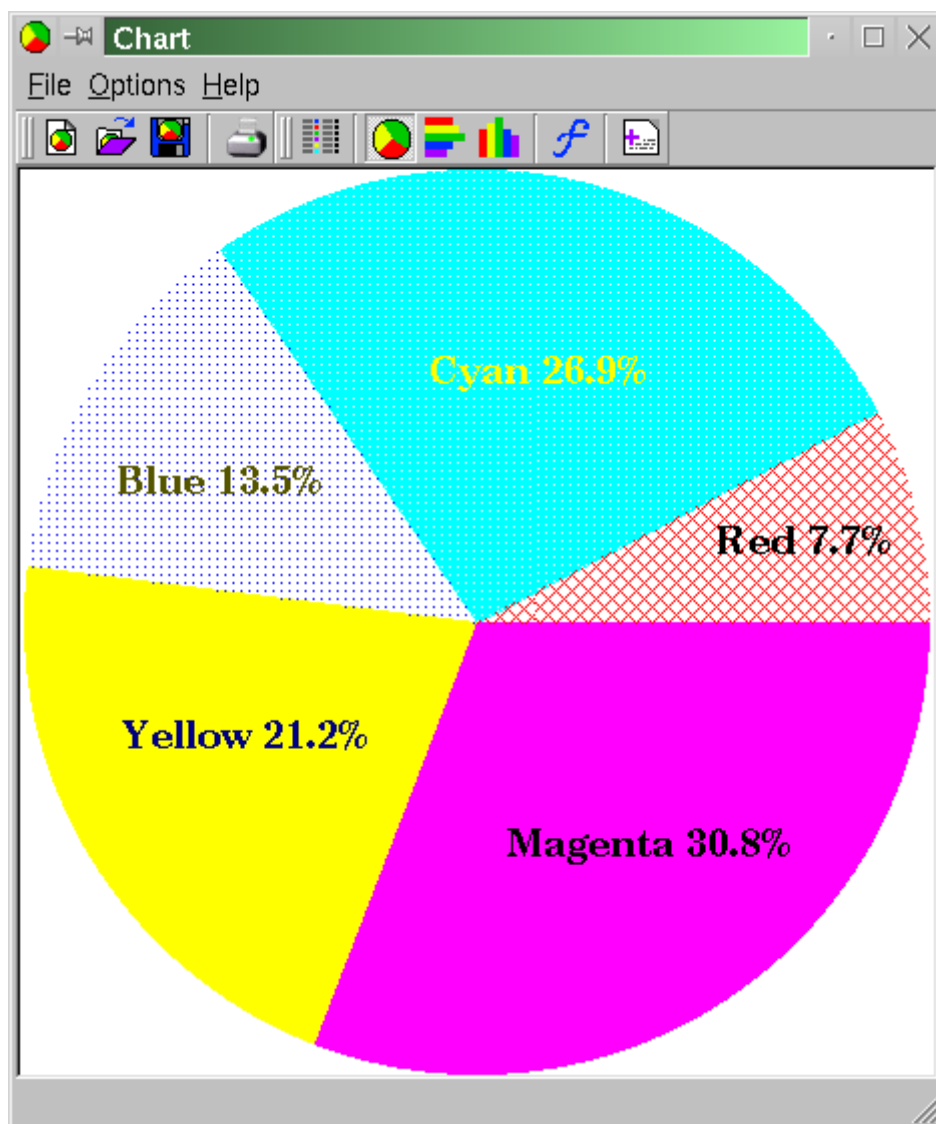
如果你对Qt很陌生，如果你还没有阅读过 [如何学习Qt](#)，请阅读一下。

- [介绍](#)
- [“大图片”](#)
- [数据元素](#)
- [主体很容易](#)
- [实现图形用户界面](#)
- [画布控制](#)
- [文件处理](#)
- [获得数据](#)
- [设置选项](#)
- [项目文件](#)
- [完成](#)

## 介绍

在这个教程中，我们将会开发一个叫做 chart 的单一应用程序，它根据用户输入的数据来显示简单的饼形和条形图表。

这个教程提供了一个应用程序开发的概述，包含了一些代码片断和与之相配的解释。应用程序完整的源程序在 `examples/chart`。



“大图片”

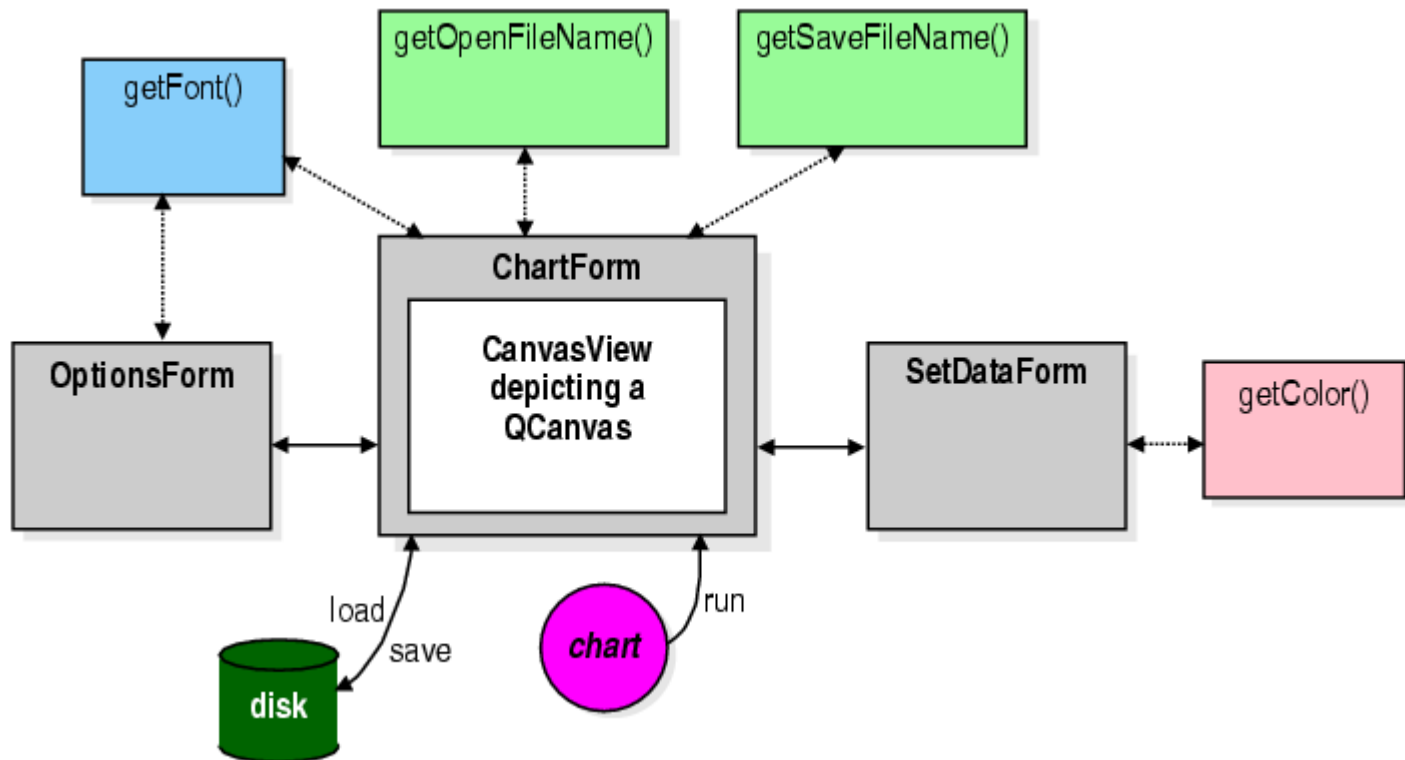


chart 程序允许用户创建、保存、载入和直观化简单的数据组。每一个用户给出的数据元素都可以被给定颜色和饼形块或条形的样式、一些标签文本和文本的位置和颜色。Element 类用来代表数据元素。

程序包含一个调入图表视窗的简单的 main.cpp。这个图表视窗有一个提供访问程序功能的菜单条和工具条。程序还提供了两个对话框，一个设置选项，另一个用来创建和编辑数据组。这两个对话框都是由图表视窗的菜单选项或者工具条按钮调用的。

图表视窗的主窗口部件是 `QCanvasView`，它显示一个我们用来画饼形图或条形图的 `QCanvas`。我们继承 `QCanvasView` 来获得一些特定的行为。同样我们因为需要比标准类提供的稍多一些，所以我们继承了 `QCanvasText` 类（用来在画布上放置文本条目）。

项目文件，chart.pro，用来创建可以用来连编应用程序的 Makefile。

## 数据元素

我们将使用一个叫 Element 的类来存储和访问数据元素。

（由 element.h 展开。）

```

private:
    double m_value;
    QColor m_valueColor;
    int m_valuePattern;
  
```

```
QString m_label;
QColor m_labelColor;
double m_propoints[2 * MAX_PROPOINTS];
```

每一个元素都有一个值。每一个值都会被使用一种特定的颜色和填充样式来图形化地显示。值也许会有一个和它们关联的标签，标签会被使用标签的颜色来画，并且对于每一种类型的图表都有一个存储在 `m_propoints` 数组中的一个（相对）位置。

```
#include <qcolor.h>
#include <qnamespace.h>
#include <qstring.h>
#include <qvaluevector.h>
```

尽管 `Element` 是一个纯粹的内部数据类，它包含了四个 Qt 类。Qt 经常被认为是一个纯粹的图形用户界面工具包，但它也提供了一些用来支持应用程序编程的绝大多数方面的非图形用户界面类。我们使用 `qcolor.h` 可以使我们在 `Element` 类中控制绘图颜色和文本颜色。

`qnamespace.h` 的用处稍微有些模糊。绝大多数 Qt 类都继承于含有各种各样的枚举的 `Qt` 这个超级类。`Element` 类不继承于 `Qt`，所以我们需要包含 `qnamespace.h` 来访问这些 Qt 枚举名称。另一个替代的方案就是使 `Element` 成为 `Qt` 的一个子类。我们包含 `qstring.h` 用来使用 Qt 的 Unicode 字符串。为了方便，我们类型定义一个 `Element` 的矢量容器，这就是我们为什么把 `qvaluevector.h` 头文件放到这里的原因。

```
typedef QVector<Element> ElementVector;
```

Qt 提供了大量的容器，一些是基于值的，比如 `QValueVector`，其它一些基于指针。（请看 [集合类](#)。）这里我们只是类型定义了一个容器类型，我们将在 `ElementVector` 中保存每一个元素数据组。

```
const double EPSILON = 0.0000001; // 必须 > INVALID.
```

元素也许只能是正的值。因为我们使用双精度实数来存储值，我们不能很容易地拿它们和零作比较。所以我们指定一个值，`EPSILON`，它和零非常接近，并且任何比 `EPSILON` 大的值可以被认为是正的和有效的。

```
class Element
{
public:
    enum { INVALID = -1 };
    enum { NO_PROPORTION = -1 };
    enum { MAX_PROPOINTS = 3 }; // 每个图表类型一个比例值
```

我们给 `Element` 定义了三个公有的枚举变量。`INVALID` 被 `isValid()` 函数使用。它是有用的，因为我们将使用一个固定大小的 `Element` 矢量，并且可以通过给定 `INVALID` 值来标明未使用的 `Element`。`NO_PROPORTION` 枚举变量用来表明用户还没有定位元素的标签，任何正的比例值都被用来作为与画布大小成比例的文本元素的位置。

如果我们存储每一个标签的实际 `x` 和 `y` 的位置，当用户每次重新定义主窗口大小（同样也对画布）的时候，文本会保留它的初始（现在是错的）位置。所以我们不存储绝对 `(x,y)` 位

置，我们存储 *比例* 位置，比如 `x/width` 和 `y/height`。然后当我们画文本的时候，我们分别用当前的宽度和高度来乘这些位置，这样不管大小如何变化，文本都会被正确定位。比如，如果标签的 `x` 位置为 300，画布有 400 像素宽，`x` 的比例值为  $300/400 = 0.75$ 。

`MAX_PROPOINTS` 枚举变量是有些疑问的。我们对于每一个图表类型的文本标签都要存储 `x` 和 `y` 的比例。并且我们已经选择把这些比例存储到一个固定大小的数组中。因为我们必须指定所需要的比例对的最大数字。如果我们改变图表类型的数字，这个值就必须被改变，这也就是说 `Element` 类和由 `ChartForm` 所提供的图表类型的数字是紧密联系的。在一个更大的应用程序中，我们也许使用一个矢量来存储这些点并且根据所能提供的图表类型的数量来动态改变它的大小。

```
Element( double value = INVALID, QColor valueColor = Qt::gray,
        int valuePattern = Qt::SolidPattern,
        const QString& label = QString::null,
        QColor labelColor = Qt::black ) {
    init( value, valueColor, valuePattern, label, labelColor );
    for ( int i = 0; i < MAX_PROPOINTS * 2; ++i )
        m_propoints[i] = NO_PROPORTION;
}
```

构造函数为 `Element` 类的所有成员变量提供了默认值。新的元素总是有没有位置的标签文本。我们是用 `init()` 函数是因为我们也提供了一个 `set()` 函数，除了比例位置它做的和构造函数一样。

```
bool isValid() const { return m_value > EPSILON; }
```

因为我们正在把 `Element` 存储到一个固定大小的矢量中，所以我们需要检测一个特定元素是否有效（比如应该被用来计算和显示）。通过 `isValid()` 函数很容易到达这一目的。

（由 `element.cpp` 展开。）

```
double Element::proX( int index ) const
{
    Q_ASSERT(index >= 0 && index < MAX_PROPOINTS);
    return m_propoints[2 * index];
}
```

这里对 `Element` 的所有成员都提供了读取函数和设置函数。`proX()` 和 `proY()` 读取函数和 `setProX()` 和 `setProY()` 设置函数用来读取和设置一个用来确定比例位置所适用的图表的类型索引。这也就是说用户可以为用于垂直条图表、水平条图表和饼形图表的相同数据组设定不同的标签位置。注意我们也使用 `Q_ASSERT` 宏来提供对图表类型索引的预先情况测试，（请看 [调试](#)）。

## 读写数据元素

（由 `element.h` 展开。）

```
Q_EXPORT QTextStream &operator<<( QTextStream&, const Element& );
Q_EXPORT QTextStream &operator>>( QTextStream&, Element& );
```

为了使我们的 Element 类更加独立，我们提供了<<和>>的操作符重载，这样 Element 就可以被文本流读写。我们也可以很容易地使用二进制流，但是使用文本可以让用户使用文本编辑器来维护他们的数据，可以更容易的使用脚本语言来产生和过滤。

(由 element.cpp 展开。)

```
#include "element.h"

#include <qstringlist.h>
#include <qtextstream.h>
```

我们对于操作符的实现需要包含 [qtextstream.h](#)和 [qstringlist.h](#)。

```
const char FIELD_SEP = ':';
const char PROPOINT_SEP = ',';
const char XY_SEP = ',';
```

我们用来存储数据的格式是用冒号来分隔字段，用换行来分隔记录。比例点用分号间隔，它们的 x 和 y 使用逗号分隔。字段的顺序是值、值的颜色、值的样式、标签颜色、标签点、标签文本。比如：

```
20:#ff0000:14:#000000:0.767033,0.412946;0,0.75;0,0:Red :with colons:!
70:#00ffff:2:#ffff00:0.450549,0.198661;0.198516,0.125954;0,0.198473:Cyan
35:#0000ff:8:#555500:0.10989,0.299107;0.397032,0.562977;0,0.396947:Blue
55:#ffff00:1:#000080:0.0989011,0.625;0.595547,0.312977;0,0.59542:Yellow
80:#ff00ff:1:#000000:0.518681,0.694196;0.794063,0;0,0.793893:Magenta or Violet
```

我们阅读 Element 数据的方式中对于文本标签中的空白符和字段间隔符都没有问题。

```
QTextStream &operator<<( QTextStream &s, const Element &element )
{
    s << element.value() << FIELD_SEP
      << element.valueColor().name() << FIELD_SEP
      << element.valuePattern() << FIELD_SEP
      << element.labelColor().name() << FIELD_SEP;

    for ( int i = 0; i < Element::MAX_PROPOINTS; ++i ) {
        s << element.proX( i ) << XY_SEP << element.proY( i );
        s << ( i == Element::MAX_PROPOINTS - 1 ? FIELD_SEP : PROPOINT_SEP );
    }

    s << element.label() << '\n';

    return s;
}
```



写元素就是一直向前。每一个成员后面都被写一个字段间隔符。点被写成由逗号间隔的 (XY\_SEP) x 和 y 的组合，每一对由 PROPOINT\_SEP 分隔符分隔。最后一个字段是标签和接着的换行符。

```
QTextStream &operator>>( QTextStream &s, Element &element )
{
    QString data = s.readLine();
    element.setValue( Element::INVALID );

    int errors = 0;
    bool ok;

    QStringList fields = QStringList::split( FIELD_SEP, data );
    if ( fields.count() >= 4 ) {
        double value = fields[0].toDouble( &ok );
        if ( !ok )
            errors++;
        QColor valueColor = QColor( fields[1] );
        if ( !valueColor.isValid() )
            errors++;
        int valuePattern = fields[2].toInt( &ok );
        if ( !ok )
            errors++;
        QColor labelColor = QColor( fields[3] );
        if ( !labelColor.isValid() )
            errors++;
        QStringList propoints = QStringList::split( PROPOINT_SEP,
fields[4] );
        QString label = data.section( FIELD_SEP, 5 );

        if ( !errors ) {
            element.set( value, valueColor, valuePattern, label,
labelColor );
            int i = 0;
            for ( QStringList::iterator point = propoints.begin();
i < Element::MAX_PROPOINTS && point != propoints.end();
++i, ++point ) {
                errors = 0;
                QStringList xy = QStringList::split( XY_SEP, *point );
                double x = xy[0].toDouble( &ok );
                if ( !ok || x <= 0.0 || x >= 1.0 )
                    errors++;
                double y = xy[1].toDouble( &ok );
                if ( !ok || y <= 0.0 || y >= 1.0 )
                    errors++;
                if ( errors )
                    x = y = Element::NO_PROPORTION;
            }
        }
    }
}
```

```

        element.setProX( i, x );
        element.setProY( i, y );
    }
}

return s;
}

```

为了读取一个元素我们读取一条记录（比如一行）。我们使用 `QStringList::split()` 来把数据分成字段。因为标签中有可能包含 `FIELD_SEP` 字符，所以我们使用 `QString::section()` 来获得从最后一个字段到这一行结尾的所有文本。如果获得了足够的字段和值，颜色和样式数据是有效的，我们使用 `Element::set()` 来把这些数据写到元素中，否则我们会设置这个元素为 `INVALID`。然后我们对点也是这样。如果 `x` 和 `y` 比例是有效的并且在范围内，我们将会为元素设置它们。如果一个或两个比例是无效的，它们将认为值为零，这样是不合适的，所以我们将会改变无效的（和超出范围的）比例点的值为 `NO_PROPORTION`。

我们的 `Element` 类现在足够用来存储、维护和读写元素数据了。我们也创建了一个元素矢量类型定义来存储一个元素的集合。

我们现在已经准备好通过我们的用户来生成、编辑和可视化他们的数据组来生成 `main.cpp` 和用户界面。

如果要获得更多的有关Qt的数据流工具请看 [QDataStream操作符格式](#)，和任何一个被提及的和你所要存储的东西相似的Qt类的源代码。

## 主体很容易

（`main.cpp`。）

```

#include <qapplication.h>
#include "chartform.h"

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

    QString filename;
    if ( app.argc() > 1 ) {
        filename = app.argv()[1];
        if ( !filename.endsWith( ".cht" ) )
            filename = QString::null;
    }
}

```

```

    ChartForm *cf = new ChartForm( filename );
    app.setMainWidget( cf );
    cf->show();
    app.connect( &app, SIGNAL(lastWindowClosed()), cf, SLOT(fileQuit()) );

    return app.exec();
}

```

我们把main()函数保持得很简单，很小。我们创建一个 `QApplication` 对象并且传递给它命令行参数。我们也允许用户通过 `chart mychart.cht` 来调用程序，所以如果他们已经添加了一个文件名，我们就把它传递给构造函数。图表窗口中的大多数行为我们将在下一步进行评论。

## 实现图形用户界面

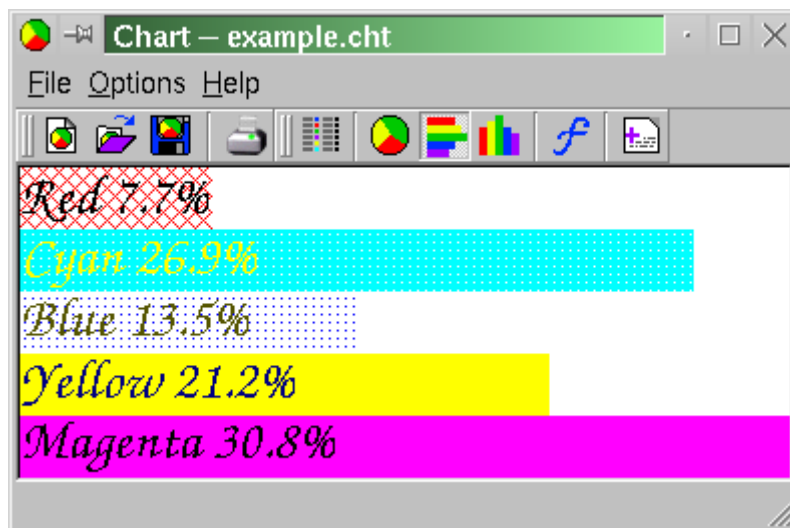


chart 程序提供了通过排列在中央窗口部件周围的菜单和工具条来访问选项，和一个通常的文档在中央的风格化的 `CanvasView`。

（由 `chartform.h` 展开。）

```

class ChartForm: public QMainWindow
{
    Q_OBJECT
public:
    enum { MAX_ELEMENTS = 100 };
    enum { MAX_RECENTFILES = 9 }; // 必须不超过 9
    enum ChartType { PIE, VERTICAL_BAR, HORIZONTAL_BAR };
    enum AddValuesType { NO, YES, AS_PERCENTAGE };

    ChartForm( const QString& filename );

```

```
~ChartForm();
```

```
int chartType() { return m_chartType; }
```

```
void setChanged( bool changed = true ) { m_changed = changed; }
```

```
void drawElements();
```

```
QPopupMenu *optionsMenu; // 为什么是公有的？请看canvasview.cpp。
```

```
private slots:
```

```
void fileNew();
```

```
void fileOpen();
```

```
void fileOpenRecent( int index );
```

```
void fileSave();
```

```
void fileSaveAs();
```

```
void fileSaveAsPixmap();
```

```
void filePrint();
```

```
void fileQuit();
```

```
void optionsSetData();
```

```
void updateChartType( QAction *action );
```

```
void optionsSetFont();
```

```
void optionsSetOptions();
```

```
void helpHelp();
```

```
void helpAbout();
```

```
void helpAboutQt();
```

```
void saveOptions();
```

```
private:
```

```
void init();
```

```
void load( const QString& filename );
```

```
bool okToClear();
```

```
void drawPieChart( const double scales[], double total, int count );
```

```
void drawVerticalBarChart( const double scales[], double total, int  
count );
```

```
void drawHorizontalBarChart( const double scales[], double total, int  
count );
```

```
QString valueLabel( const QString& label, double value, double total );
```

```
void updateRecentFiles( const QString& filename );
```

```
void updateRecentFilesMenu();
```

```
void setChartType( ChartType chartType );
```

```
QPopupMenu *fileMenu;
```

```
QAction *optionsPieChartAction;
```

```
QAction *optionsHorizontalBarChartAction;
```

```
QAction *optionsVerticalBarChartAction;
```

```
QString m_filename;
```

```
QStringList m_recentFiles;
```

```

    QCanvas *m_canvas;
    CanvasView *m_canvasView;
    bool m_changed;
    ElementVector m_elements;
    QPrinter *m_printer;
    ChartType m_chartType;
    AddValuesType m_addValues;
    int m_decimalPlaces;
    QFont m_font;
};

```

我们创建了一个 `QMainWindow` 的子类 `ChartForm`。我们的子类使用了 `Q_OBJECT` 宏来支持 Qt 的信号和槽机制。

公有接口是很少的，被显示的图表类型能够被追溯，图表可以被标记为“changed”（这样用户在退出的时候会被提示保存），并且图表可以要求拖拽自己（`drawElements()`）。我们已经把选项菜单设为公有，因为我们也会把这个菜单作为画布视图的关联菜单。

`QCanvas` 类用来绘制二维矢量图。`QCanvasView` 类用来在一个应用程序的图形用户界面中实现一个画布的视图。我们所有的绘制操作都发生在画布上，但是事件（比如鼠标点击）却发生在画布视图中。

每一个动作都被一个私有槽实现，比如 `fileNew()`、`optionsSetData()` 等等。我们也需要相当多的私有函数和数据成员，当我们执行这些实现的时候，我们来看看这些。

为了方便和编译速度的原因，图表视窗的实现被分为三个文件，`chartform.cpp` 实现图形用户界面，`chartform_canvas.cpp` 实现画布处理和 `chartform_files.cpp` 实现文件处理。我们会依次评论每一个。

## 图表视窗图形用户界面

（由 `chartform.cpp` 展开。）

```

#include "images/file_new.xpm"
#include "images/file_open.xpm"
#include "images/options_piechart.xpm"

```

`chart` 中使用的所有图像是我们已经创建好并放在 `images` 子目录中的 `.xpm` 文件。

## 构造函数

```

ChartForm::ChartForm( const QString& filename )
    : QMainWindow( 0, 0, WDestructiveClose )
...
    QAction *fileNewAction;
    QAction *fileOpenAction;

```

```
QAction *fileSaveAction;
```

对于每一个用户动作我们声明了一个 `QAction` 指针。一些动作在头文件中已经声明，因为它们需要在构造函数外被参考。

大部分用户动作适用于菜单条目和工具条按钮。Qt 允许用户创建一个单一的 `QAction` 而被添加到菜单和工具条中。这种方法保证了菜单条目和工具条按钮处于同步状态并且可以节省代码。

```
fileNewAction = new QAction(  
    "New Chart", QPixmap( file_new ),  
    "&New", CTRL+Key_N, this, "new" );  
connect( fileNewAction, SIGNAL( activated() ), this, SLOT( fileNew() ) );
```

当我们构造一个动作时，我们给它一个名字、一个可选的图标、一个菜单文本和一个加速快捷键（或者 0 如果不需要加速键）。我们也可以使它成为视图的子对象（通过 `this`）。当用户点击一个工具条按钮或者点击一个菜单选项时，`activated()` 信号会被发射。我们把这个信号和这个动作的槽连接起来，就是上面的程序代码中提到的 `fileNew()`。

图表类型是互斥的：我们可以用一个饼图或一个竖直条形图或一个水平条形图。这也就是说如果用户选择了饼图菜单选项，饼图工具条按钮也必须被自动地选中，并且其它图表菜单选项和工具条按钮必须被自动地取消选择。这种行为是通过创建一个 `QActionGroup` 来实现的并且把这些图表类型动作放到这个组中。

```
QActionGroup *chartGroup = new QActionGroup( this ); // Connected later  
chartGroup->setExclusive( true );
```

动作组成为了视图（`this`）的子对象并且 `exclusive` 行为通过 `setExclusive()` 调用实现的。

```
optionsPieChartAction = new QAction(  
    "Pie Chart", QPixmap( options_piechart ),  
    "&Pie Chart", CTRL+Key_I, chartGroup, "pie chart" );  
optionsPieChartAction->setToggleAction( true );
```

组中的每一个动作都以和其它动作一样的方式创建，除了动作的父对象是组而不是视图。因为我们的图表类型动作由开/关状态，我们为它们中的每一个调用 `setToggleAction(TRUE)`。注意我们没有连接动作，相反，稍后我们会把这个组连接到一个可以使画布重画的槽。

为什么我们不马上连接这个组呢？稍后在构造函数中我们将会读取用户选项，图表类型之一。我们将会直接设置图表类型。但那时我们还没有创建画布或者有任何数据，所以我们想做的一切就是切换画布类型工具条按钮，而不是真正地画（这时还不存在的）画布。在我们设置好画布类型之后，我们将会连接这个组。

一旦我们已经创建完所有的用户动作，我们就可以创建工具条和菜单选项来允许用户调用它们。

```
QToolBar* fileTools = new QToolBar( this, "file operations" );  
fileTools->setLabel( "File Operations" );
```

```

fileNewAction->addTo( fileTools );
fileOpenAction->addTo( fileTools );
fileSaveAction->addTo( fileTools );
...
fileMenu = new QPopupMenu( this );
menuBar()->insertItem( "&File", fileMenu );
fileNewAction->addTo( fileMenu );
fileOpenAction->addTo( fileMenu );
fileSaveAction->addTo( fileMenu );

```

工具条动作和菜单选项可以很容易地由 **QAction** 生成。

作为一个对我们的用户提供的方便，我们将会重新载入上次窗口的位置和大小并列最近使用的文件。这是通过在程序退出的时候写出这些设置，在我们构造视窗的时候再把它们都回来实现的。

```

QSettings settings;
settings.insertSearchPath( QSettings::Windows, WINDOWS_REGISTRY );
int windowWidth = settings.readNumEntry( APP_KEY + "WindowWidth", 460 );
int windowHeight = settings.readNumEntry( APP_KEY + "WindowHeight", 530 );
int windowX = settings.readNumEntry( APP_KEY + "WindowX", 0 );
int windowY = settings.readNumEntry( APP_KEY + "WindowY", 0 );
setChartType( ChartType(
    settings.readNumEntry( APP_KEY + "ChartType", int(PIE) ) ) );
m_font = QFont( "Helvetica", 18, QFont::Bold );
m_font.fromString(
    settings.readEntry( APP_KEY + "Font", m_font.toString() ) );
for ( int i = 0; i < MAX_RECENTFILES; ++i ) {
    QString filename = settings.readEntry( APP_KEY + "File" +
        QString::number( i + 1 ) );
    if ( !filename.isEmpty() )
        m_recentFiles.push_back( filename );
}
if ( m_recentFiles.count() )
    updateRecentFilesMenu();

```

**QSettings**类通过和平台无关的方式来处理用户设置。我们很简单地读写设置，把处理平台依赖性的问题留给**QSettings**来处理。**insertSearchPath()**调用没有做任何事，除非在**Windows**下被**#ifdef**过。

我们使用 **readNumEntry()**调用来得到图表视窗上次的大小和位置，并且为它的第一次运行提供了默认值。图表类型是以一个整数重新获得并把它扔给 **ChartType** 枚举值。我们创建默认标签字体，然后读取“Font”设置，如果需要的话我们使用刚才生成的默认字体。

尽管 **QSettings** 可以处理字符串列表，但是我们已经选择把最近使用的每一个文件作为单一的条目来存储，这样就可以更容易地处理和编辑这些设置。我们试着去读每一个可能的文件条目（从“File1”到“File9”），并把每一个非空条目添加到最近使用的文件的列表中。如

果有一个或多个最近使用的文件，我们通过调用 `updateRecentFilesMenu()` 来更新 File 菜单，（我们将会在后文再评论这个）。

```
connect( chartGroup, SIGNAL( selected(QAction*) ),
        this, SLOT( updateChartType(QAction*) ) );
```

现在我们已经设置图表类型（当我们把它作为一个用户设置读入的时候），把图表组和我们的 `updateChartType()` 槽连接起来是安全的。

```
resize( windowWidth, windowHeight );
move( windowX, windowY );
```

并且现在我们已经知道窗口大小和位置，我们就可以根据这些重新定义大小并移动图表视图窗口。

```
m_canvas = new QCanvas( this );
m_canvas->resize( width(), height() );
m_canvasView = new CanvasView( m_canvas, &m_elements, this );
setCentralWidget( m_canvasView );
m_canvasView->show();
```

我们创建一个新的 `QCanvas` 并且设置它的大小为图表视图窗口的客户区域。我们也创建一个 `CanvasView`（我们自己的 `QCanvasView` 的子类）来显示 `QCanvas`。我们把这个画布视图作为图表视图的主窗口部件并显示它。

```
if ( !filename.isEmpty() )
    load( filename );
else {
    init();
    m_elements[0].set( 20, red, 14, "Red" );
    m_elements[1].set( 70, cyan, 2, "Cyan", darkGreen );
    m_elements[2].set( 35, blue, 11, "Blue" );
    m_elements[3].set( 55, yellow, 1, "Yellow", darkBlue );
    m_elements[4].set( 80, magenta, 1, "Magenta" );
    drawElements();
}
```

如果我们有一个文件要载入，我们就载入它，否则我们就初始化我们的元素矢量并画一个示例图表。

```
statusBar()->message( "Ready", 2000 );
```

我们在构造函数中调用 `statusBar()` 是 *非常重要的*，因为这个调用保证了我们能够在这个主窗口中创建一个状态条。



## init()

```
void ChartForm::init()
{
    setCaption( "Chart" );
    m_filename = QString::null;
    m_changed = false;

    m_elements[0] = Element( Element::INVALID, red );
    m_elements[1] = Element( Element::INVALID, cyan );
    m_elements[2] = Element( Element::INVALID, blue );
    ...
}
```

我们使用了 `init()` 函数是因为我们想在视窗被构造的时候和无论用户载入一个存在的数据组或者创建一个新的数据组的时候初始化画布和元素（在 `m_elements` `ElementVector` 中）。

我们重新设置标题并设置当前文件名称为 `QString::null`。我们也用无效的元素来组装元素矢量。这不是必需的，但是给每一个元素一个不同的颜色对于用户来讲是更方便的，因为他们输入值的时候每一个都会已经有了一个确定的颜色（当然他们可以修改）。

## 文件处理动作

### okToClear()

```
bool ChartForm::okToClear()
{
    if ( m_changed ) {
        QString msg;
        if ( m_filename.isEmpty() )
            msg = "Unnamed chart ";
        else
            msg = QString( "Chart '%1'\n" ).arg( m_filename );
        msg += "has been changed.";
        switch( QMessageBox::information( this, "Chart -- Unsaved Changes",
            msg, "&Save", "Cancel",
            "&Abandon",
            0, 1 ) ) {
            case 0:
                fileSave();
                break;
            case 1:
            default:
                return false;
                break;
            case 2:
                ...
            ...
        }
    }
}
```

```

        break;
    }
}

return true;
}

```

`okToClear()`函数用来提示用户在有没保存的数据的时候保存它们。它也被其它几个函数使用。

## fileNew()

```

void ChartForm::fileNew()
{
    if ( okToClear() ) {
        init();
        drawElements();
    }
}

```

当用户调用 `fileNew()`动作时，我们调用 `okToClear()`来给他们一个保存任何为保存的数据的机会。无论他们保存或者放弃或者没有任何为保存的数据，我们都重新初始化元素矢量并绘制默认图表。

我们是不是也应该调用 `optionsSetData()`来弹出一个对话框，让用户通过它来创建和编辑值、颜色等等呢？你可以运行一下现在的应用程序，然后试着把 `optionsSetData()`的调用添加进去后再运行并观察它们来决定你更喜欢哪一个。

## fileOpen()

```

void ChartForm::fileOpen()
{
    if ( !okToClear() )
        return;

    QString filename = QFileDialog::getOpenFileName(
        QString::null, "Charts (*.cht)", this,
        "file open", "Chart -- File Open" );
    if ( !filename.isEmpty() )
        load( filename );
    else
        statusBar()->message( "File Open abandoned", 2000 );
}

```

我们检查它是否是`okToClear()`。如果是的话，我们使用静态的 `QFileDialog::getOpenFileName()` 函数来获得用户想要载入的文件的名称。如果我们得到一个文件名，我们就调用`load()`。

## fileSaveAs()

```
void ChartForm::fileSaveAs()
{
    QString filename = QFileDialog::getSaveFileName(
        QString::null, "Charts (*.cht)", this,
        "file save as", "Chart -- File Save As" );
    if ( !filename.isEmpty() ) {
        int answer = 0;
        if ( QFile::exists( filename ) )
            answer = QMessageBox::warning(
                this, "Chart -- Overwrite File",
                QString( "Overwrite\n\'%1\'?" ).
                    arg( filename ),
                "&Yes", "&No", QString::null, 1, 1 );
        if ( answer == 0 ) {
            m_filename = filename;
            updateRecentFiles( filename );
            fileSave();
            return;
        }
    }
    statusBar()->message( "Saving abandoned", 2000 );
}
```

这个函数调用了静态的 `QFileDialog::getSaveFileName()`来得到一个要保存数据的文件的明处那个。如果文件存在，我们使用使用一个 `QMessageBox::warning()`来提醒用户并给他们一个放弃保存的选择。如果文件被保存了我们就更新最近打开的文件列表并调用`fileSave()`（在[文件处理](#)中）来执行存储。

## 管理最近打开文件的列表

```
QStringList m_recentFiles;
```

我们用一个字符串列表来处理这个最近打开文件的列表。

```
void ChartForm::updateRecentFilesMenu()
{
    for ( int i = 0; i < MAX_RECENTFILES; ++i ) {
        if ( fileMenu->findItem( i ) )
            fileMenu->removeItem( i );
        if ( i < int(m_recentFiles.count()) )
            fileMenu->insertItem( QString( "%1 %2" ).
```

```

        arg( i + 1 ).arg( m_recentFiles[i] ),
        this, SLOT( fileOpenRecent(int) ),
        0, i );
    }
}

```

无论用户打开一个存在的文件或者保存一个新文件的时候，这个函数会被调用（通常是通过 `updateRecentFiles()`）。对于这个字符串列表中的每一个文件我们都插入一个新的菜单条目。我们在每一个文件名的前面都加上一个从 1 到 9 带下划线的数字，这样就可以支持键盘操作（比如，Alt+F, 2 就可以打开列表中的第二个文件）。我们给每一个菜单条目一个和它们在字符串列表中的索引位置相同的数值作为 `id`，并且把每一个菜单条目都和 `fileOpenRecent()` 槽相连。老的文件菜单条目会在每一个最新的文件菜单条目 `id` 来到的同时被删除。它工作是因为其它文件菜单条目都有一个由 Qt 生成的 `id`（它们都是 <0 的），然而我们所创建的菜单条目的 `id` 都是 ≥0 的。

```

void ChartForm::updateRecentFiles( const QString& filename )
{
    if ( m_recentFiles.find( filename ) != m_recentFiles.end() )
        return;

    m_recentFiles.push_back( filename );
    if ( m_recentFiles.count() > MAX_RECENTFILES )
        m_recentFiles.pop_front();

    updateRecentFilesMenu();
}

```

当用户打开一个存在的文件或者保存一个新文件的时候，它会被调用。如果文件已经存在于列表中，它就会很简单地返回。否则这个文件会被添加到列表的末尾并且如果列表太大（>9 个文件）的话，第一个（最老的）就会被移去。然后 `updateRecentFilesMenu()` 被调用在 **File** 菜单中重新创建最近使用的文件列表。

```

void ChartForm::fileOpenRecent( int index )
{
    if ( !okToClear() )
        return;

    load( m_recentFiles[index] );
}

```

当用户选择了一个最近打开的文件时，`fileOpenRecent()` 槽会伴随一个用户选择的文件的菜单 `id` 而被调用。因为我们使文件菜单的 `id` 和文件在 `m_recentFiles` 列表中的索引位置相等，我们就可以很简单的通过文件的菜单条目 `id` 来载入了。

## 退出

```

void ChartForm::fileQuit()

```

```

{
    if ( okToClear() ) {
        saveOptions();
        qApp->exit( 0 );
    }
}

```

当用户退出时，我们给他们保存任何未保存数据的机会（`okToClear()`），然后在结束之前保存它们的选项，比如窗口的大小和位置、图表类型等等。

```

void ChartForm::saveOptions()
{
    QSettings settings;
    settings.insertSearchPath( QSettings::Windows, WINDOWS_REGISTRY );
    settings.writeEntry( APP_KEY + "WindowWidth", width() );
    settings.writeEntry( APP_KEY + "WindowHeight", height() );
    settings.writeEntry( APP_KEY + "WindowX", x() );
    settings.writeEntry( APP_KEY + "WindowY", y() );
    settings.writeEntry( APP_KEY + "ChartType", int(m_chartType) );
    settings.writeEntry( APP_KEY + "AddValues", int(m_addValues) );
    settings.writeEntry( APP_KEY + "Decimals", m_decimalPlaces );
    settings.writeEntry( APP_KEY + "Font", m_font.toString() );
    for ( int i = 0; i < int(m_recentFiles.count()); ++i )
        settings.writeEntry( APP_KEY + "File" + QString::number( i + 1 ),
                               m_recentFiles[i] );
}

```

直接使用 `QSettings` 来保存用户选项。

## 自定义对话框

我们想让用户可以手工地设置一些选项并且创建和编辑值、值颜色等等。

```

void ChartForm::optionsSetOptions()
{
    OptionsForm *optionsForm = new OptionsForm( this );
    optionsForm->chartTypeComboBox->setCurrentItem( m_chartType );
    optionsForm->setFont( m_font );
    if ( optionsForm->exec() ) {
        setChartType( ChartType(
            optionsForm->chartTypeComboBox->currentItem() ) );
        m_font = optionsForm->font();
        drawElements();
    }
    delete optionsForm;
}

```

设置选项的视窗是由我们自定义的OptionsForm提供的，在 [设置选项](#) 中。这个选项视窗是一个标准的“哑的”对话框：我们创建一个实例，把所有的图形用户界面元素都和所有相关的设置都组装起来，并且如果用户点击了“OK”（exec()返回一个真值）我们就会从图形用户界面元素中读取设置。

```
void ChartForm::optionsSetData()
{
    SetDataForm *setDataForm = new SetDataForm( &m_elements, m_decimalPlaces,
this );
    if ( setDataForm->exec() ) {
        m_changed = true;
        drawElements();
    }
    delete setDataForm;
}
```

创建和编辑图表数据的视窗由我们自定义的SetDataForm提供，在 [获得数据](#) 中。这个视窗是一个“聪明的”对话框。我们传入我们想要使用的数据结构，并且对话框可以自己处理数据机构的表达。如果用户点击“OK”，对话框会更新数据结构并且exec()会返回一个真值。如果用户改变了数据时我们在optionsSetData()中所要做的时把图表标记为changed并调用drawElements()来使用新的和更新过的数据来重新绘制图表。

## 画布控制

我们在画布上画饼形区域（或者条形图表条），和所有的标签。画布是通过画布视图来呈现给用户的。drawElements()函数被调用从而在需要的时候重新绘制画布。

（由 chartform\_canvas.cpp 展开。）

### drawElements()

```
void ChartForm::drawElements()
{
    QCanvasItemList list = m_canvas->allItems();
    for ( QCanvasItemList::iterator it = list.begin(); it != list.end(); ++it )
        delete *it;
```

我们在 drawElements()中所作的第一件事是删除所有已经存在的画布条目。

```
// 360 * 16 为一个饼形，Qt 中使用的是 16 倍的度数（就是它的一个圆周为
360x16）
int scaleFactor = m_chartType == PIE ? 5760 :
    m_chartType == VERTICAL_BAR ? m_canvas->height() :
```

```
m_canvas->width();
```

接下来我们根据要绘制的图表的种类来计算比例因子。

```
double biggest = 0.0;
int count = 0;
double total = 0.0;
static double scales[MAX_ELEMENTS];

for ( int i = 0; i < MAX_ELEMENTS; ++i ) {
    if ( m_elements[i].isValid() ) {
        double value = m_elements[i].value();
        count++;
        total += value;
        if ( value > biggest )
            biggest = value;
        scales[i] = m_elements[i].value() * scaleFactor;
    }
}

if ( count ) {
    // 第二个循环是因为总量和最大的
    for ( int i = 0; i < MAX_ELEMENTS; ++i )
        if ( m_elements[i].isValid() )
            if ( m_chartType == PIE )
                scales[i] = (m_elements[i].value() * scaleFactor) /
total;
            else
                scales[i] = (m_elements[i].value() * scaleFactor) /
biggest;
}
```

我们需要知道这里有多少值、最大的值和值的总和，这样我们就可以正确地按比例创建饼形区域或条形了。我们把比例值存放在 scales 数组中。

```
switch ( m_chartType ) {
    case PIE:
        drawPieChart( scales, total, count );
        break;
    case VERTICAL_BAR:
        drawVerticalBarChart( scales, total, count );
        break;
    case HORIZONTAL_BAR:
        drawHorizontalBarChart( scales, total, count );
        break;
}
```

既然我们已经知道了必需的信息，那我们就调用相关绘制函数，传递比例值、总量和计数。

```
m_canvas->update();
```

最终我们使用 `update()` 更新画布来使所有的变化可视。

## drawHorizontalBarChart()

我们来回顾一下刚才的这个绘制函数，看到了画布条目如何被生成并放置到画布上，因为这个教程是关于 Qt 的，而不是关于绘制图表的好的（或者坏的）算法。

```
void ChartForm::drawHorizontalBarChart(  
    const double scales[], double total, int count )  
{
```

画水平条形图我们需要一个比例值的数组、总量（这样我们就可以在需要的时候计算并且画出百分比）和这一组值的计数。

```
    double width = m_canvas->width();  
    double height = m_canvas->height();  
    int proheight = int(height / count);  
    int y = 0;
```

我们重新得到画布的宽度和高度并且计算比例高度（`proheight`）。我们把初始的 `y` 位置设为 0。

```
    QPen pen;  
    pen.setStyle( NoPen );
```

我们创建一个用来绘制每一个条形（矩形）的画笔，我们把它设置为 `NoPen`，这样就不会画出边框。

```
    for ( int i = 0; i < MAX_ELEMENTS; ++i ) {  
        if ( m_elements[i].isValid() ) {  
            int extent = int(scales[i]);
```

我们在元素矢量中迭代每一个元素，忽略无效的元素。每个条的宽度（它的长度）很简单地就是它的比例值。

```
                QCanvasRectangle *rect = new QCanvasRectangle(  
                                                            0, y, extent, proheight,  
m_canvas );  
                rect->setBrush( QBrush( m_elements[i].valueColor(),  
BrushStyle(m_elements[i].valuePattern()) ) );  
                rect->setPen( pen );  
                rect->setZ( 0 );
```



```
rect->show();
```

我们为每个条形创建一个新的 `QCanvasRectangle`，它的x位置为 0（因为这是一个水平条形图，每个条形都从左边开始），y值从 0 开始，随着每一个要画的条形的高度增长，一直到我们要画的条形和画布的高度。然后我们设置条形的画刷为用户为元素指定的颜色和样式，设置画笔为我们先前生成的画笔（比如，设置为NoPen）并且我们把条形的Z轴顺序设置为 0。最后我们调用show()在画布上绘制条形。

```
QString label = m_elements[i].label();
if ( !label.isEmpty() || m_addValues != NO ) {
    double proX = m_elements[i].proX( HORIZONTAL_BAR );
    double proY = m_elements[i].proY( HORIZONTAL_BAR );
    if ( proX < 0 || proY < 0 ) {
        proX = 0;
        proY = y / height;
    }
}
```

如果用户已经为元素指定了标签或者要求把值（或者百分比）显示出来，我们也要画一个画布文本条目。我们创建我们自己的 `CanvasText` 类（请看后面），因为我们想存储每一个画布文本条目中对应元素的索引（在元素矢量中）。我们从元素中得出 x 和 y 的比例值。如果其中之一<0，那么他们还没有被用户定位，所以你必须计算它们的位置。我们标签的 x 值为 0（左）并且 y 值为条形图的顶部（这样标签的左上角就会在 x,y 位置）。

```
label = valueLabel( label, m_elements[i].value(), total );
```

然后我们调用一个助手函数 `valueLabel()`，它可以返回一个包含标签文本的字符串。（如果用户已经设置相应的选项，`valueLabel()`函数添加值或者百分比到这个文本的标签。）

```
CanvasText *text = new CanvasText( i, label, m_font,
m_canvas );

text->setColor( m_elements[i].labelColor() );
text->setX( proX * width );
text->setY( proY * height );
text->setZ( 1 );
text->show();
m_elements[i].setProX( HORIZONTAL_BAR, proX );
m_elements[i].setProY( HORIZONTAL_BAR, proY );
```

然后我们创建一个 `CanvasText` 条目，传递给它在元素矢量中这个元素的索引和所要使用的标签、字体和画布。我们设置文本条目的颜色为用户指定的颜色并且设置条目的 x 和 y 位置和画布的宽高成比例。我们设置 Z 轴顺序为 1，这样文本条目总是在条形（Z 轴顺序为 0）的上面（前面）。我们调用 `show()`函数在画布上绘制文本条目，并且设置元素的相对 x 和 y 位置。

```
}
y += proheight;
```

在绘制完条形和可能存在的标签之后，我们给 y 增加一定比例的高度用来准备绘制下一个元素。

```
    }  
  }  
}
```

## QCanvasText的子类

（由 canvastext.h 展开。）

```
class CanvasText : public QCanvasText  
{  
public:  
    enum { CANVAS_TEXT = 1100 };  
  
    CanvasText( int index, QCanvas *canvas )  
        : QCanvasText( canvas ), m_index( index ) {}  
    CanvasText( int index, const QString& text, QCanvas *canvas )  
        : QCanvasText( text, canvas ), m_index( index ) {}  
    CanvasText( int index, const QString& text, QFont font, QCanvas *canvas )  
        : QCanvasText( text, font, canvas ), m_index( index ) {}  
  
    int index() const { return m_index; }  
    void setIndex( int index ) { m_index = index; }  
  
    int rtti() const { return CANVAS_TEXT; }  
  
private:  
    int m_index;  
};
```

我们的CanvasText子类是 QCanvasText的一个非常简单的特化。我们所做的一切只是添加一个私有成员m\_index，它用来保存和这个文本相关的元素的元素矢量索引，并且提供为这个值提供一个读和写函数。

## QCanvasView的子类

（由 canvasview.h 展开。）

```
class CanvasView : public QCanvasView  
{  
    Q_OBJECT  
public:  
    CanvasView( QCanvas *canvas, ElementVector *elements,  
                QWidget* parent = 0, const char* name = "canvas view",
```

```

        WFlags f = 0 )
    : QCanvasView( canvas, parent, name, f ),
      m_elements( elements ) {}

protected:
    void viewportResizeEvent( QResizeEvent *e );
    void contentsMouseEvent( QMouseEvent *e );
    void contentsMouseMoveEvent( QMouseEvent *e );
    void contentsContextMenuEvent( QContextMenuEvent *e );

private:
    QCanvasItem *m_movingItem;
    QPoint m_pos;
    ElementVector *m_elements;
};

```

我们需要继承 `QCanvasView`，这样我们就能处理：

1. 上下文菜单请求。
2. 视窗重定义大小。
3. 用户拖拽标签到任意位置。

为了支持这些，我们存储一个到正在被移动的画布条目的指针和它的最终位置。我们也存储一个到元素矢量的指针。

## 上下文菜单请求

（由 `canvasview.cpp` 展开。）

```

void CanvasView::contentsContextMenuEvent( QContextMenuEvent * )
{
    ((ChartForm*)parent())->optionsMenu->exec( QCursor::pos() );
}

```

当用户调用一个上下文菜单（比如在绝大多数平台通过右键点击），我们把画布视图的父对象（是一个 `ChartForm`）转化为正确的类型，然后用 `exec()` 在光标位置执行选项菜单。

## 视窗重定义大小

```

void CanvasView::viewportResizeEvent( QResizeEvent *e )
{
    canvas()->resize( e->size().width(), e->size().height() );
    ((ChartForm*)parent())->drawElements();
}

```

为了改变大小我们简单地改变花布的大小，画布视图就会呈现在视窗客户端区域的宽高中，然后调用 `drawElements()` 函数来重新绘制图表。因为 `drawElements()` 画的每一件都和画布的宽高有关，所以图表就会被正确地绘制。

## 拖拽标签到任意位置

当用户想把标签拖拽到他们点击的位置时，就应该拖拽它并在新的位置释放它。

```
void CanvasView::contentsMouseEvent( QMouseEvent *e )
{
    QCanvasItemList list = canvas()->collisions( e->pos() );
    for ( QCanvasItemList::iterator it = list.begin(); it != list.end(); ++it )
        if ( (*it)->rtti() == CanvasText::CANVAS_TEXT ) {
            m_movingItem = *it;
            m_pos = e->pos();
            return;
        }
    m_movingItem = 0;
}
```

当用户点击鼠标时，我们创建一个鼠标点击“碰撞”（如果有的话）的画布条目的列表。然后我们迭代这个列表并且如果我们发现一个 `CanvasText` 条目，我们就把它设置为移动的条目并且记录下它的位置。否则我们设置为不移动条目。

```
void CanvasView::contentsMouseMoveEvent( QMouseEvent *e )
{
    if ( m_movingItem ) {
        QPoint offset = e->pos() - m_pos;
        m_movingItem->moveBy( offset.x(), offset.y() );
        m_pos = e->pos();
        ChartForm *form = (ChartForm*)parent();
        form->setChanged( true );
        int chartType = form->chartType();
        CanvasText *item = (CanvasText*)m_movingItem;
        int i = item->index();
        (*m_elements)[i].setProX( chartType, item->x() / canvas()->width() );
        (*m_elements)[i].setProY( chartType, item->y() /
canvas()->height() );
        canvas()->update();
    }
}
```

当用户拖拽鼠标的时候，移动事件就产生了。如果那里是一个可移动条目，我们从鼠标最后的位置和可移动条目原来的位置计算出位移。我们将新的位置记录为最后的位置。因为图表现在改变了，所以我们调用 `setChanged()`，这样当用户试图退出或者读入已存在的图表时或者创建新的图表，就会被提示是否保存。我们也分别地更新当前图表类型的元素的 `x` 和

y的比例位置为当前x和y与宽和高的比例。我们知道要更新哪个元件因为当我们创建每个画布文本条目的时候,我们传给它一个这个元素所对应的位置索引。我们继承了 `QCanvasText`, 这样我们就可以设置和读取这个索引值。最后我们调用`update()`来重绘画布。

`QCanvas`没有任何视觉效果。为了看到画布的内容,你必须创建一个 `QCanvasView`来呈现画布。如果条目被`show()`显示,它们就会出现在画布视图中,然后,只有当 `QCanvas::update()`被调用的时候。默认情况下`QCanvas`的背景是白色,并且会在画布上绘制默认的形状,比如 `QCanvasRectangle`、`QCanvasEllipse` 等等,因为它们被白色填充,所以非常推荐使用一个非白色的画刷颜色!

## 文件处理

(从 `chartform_files.cpp` 展开。)

### 读图表数据

```
void ChartForm::load( const QString& filename )
{
    QFile file( filename );
    if ( !file.open( IO_ReadOnly ) ) {
        statusBar()->message( QString( "Failed to load \'%1\'" ).
                                arg( filename ), 2000 );
        return;
    }

    init(); // 确保我们拥有颜色
    m_filename = filename;
    QTextStream ts( &file );
    Element element;
    int errors = 0;
    int i = 0;
    while ( !ts.eof() ) {
        ts >> element;
        if ( element.isValid() )
            m_elements[i++] = element;
    }
    file.close();
    setCaption( QString( "Chart -- %1" ).arg( filename ) );
    updateRecentFiles( filename );

    drawElements();
    m_changed = false;
}
```

载入数据组非常容易。我们打开文件并且创建一个文本流。当有数据要读的时候，我们把一个元素读入到 `element` 并且如果它是有效的，我们就把它插入到 `m_elements` 矢量。所有的细节都由 `Element` 类来处理。然后我们关闭文件并且更新标题和最近打开的文件列表。最后我们绘制图表并标明它没有被改变。

## 写图表数据

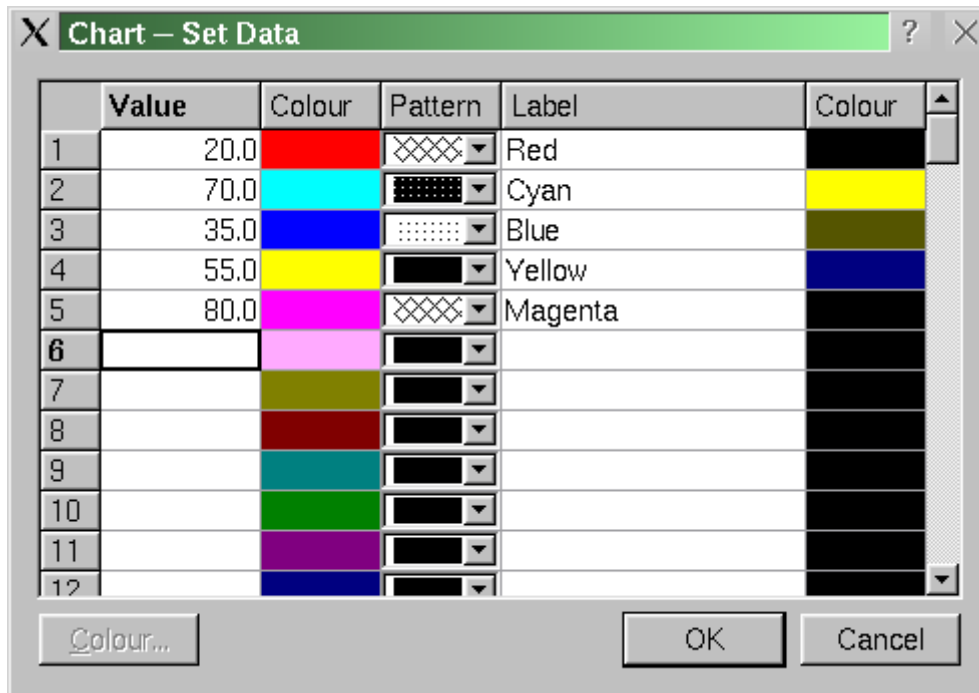
```
void ChartForm::fileSave()
{
    QFile file( m_filename );
    if ( !file.open( IO_WriteOnly ) ) {
        statusBar()->message( QString( "Failed to save \'%1\'" ).
                                arg( m_filename ), 2000 );
        return;
    }
    QTextStream ts( &file );
    for ( int i = 0; i < MAX_ELEMENTS; ++i )
        if ( m_elements[i].isValid() )
            ts << m_elements[i];

    file.close();

    setCaption( QString( "Chart -- %1" ).arg( m_filename ) );
    statusBar()->message( QString( "Saved \'%1\'" ).arg( m_filename ), 2000 );
    m_changed = false;
}
```

保存数据一样地容易。我们打开文件并且创建一个文本流。然后我们把每一个有效元素写到文本留中。所有的细节都由 `Element` 类来处理。

## 获得数据



设置数据对话框允许用户添加和编辑值，并且可以选择用来显示值的颜色和样式。用户可以输入标签文本并为每一个标签选择一个标签颜色。

（由 setdataform.h 展开。）

```
class SetDataForm: public QDialog
{
    Q_OBJECT
public:
    SetDataForm( ElementVector *elements, int decimalPlaces,
                 QWidget *parent = 0, const char *name = "set data form",
                 bool modal = TRUE, WFlags f = 0 );
    ~SetDataForm() {}

public slots:
    void setColor();
    void setColor( int row, int col );
    void currentChanged( int row, int col );
    void valueChanged( int row, int col );

protected slots:
    void accept();

private:
    QTable *table;
    QPushButton *colorPushButton;
    QPushButton *okPushButton;
    QPushButton *cancelPushButton;
```

```
protected:
    QVBoxLayout *tableButtonBox;
    QHBoxLayout *buttonBox;

private:
    ElementVector *m_elements;
    int m_decimalPlaces;
};
```

头文件很简单。构造函数中用一个指针指向元素矢量，这样这个“聪明的”对话框就可以直接显示并且编辑数据。我们将会解释我们在实现中所看到的槽的。

（由 `setdataform.cpp` 展开。）

```
#include "images/pattern01.xpm"
#include "images/pattern02.xpm"
```

我们创建了一个小的.XPM 图片用来显示 Qt 支持的每一种画刷样式。我们将会在样式组合框中使用这些的。

## 构造函数

```
SetDataForm::SetDataForm( ElementVector *elements, int decimalPlaces,
                           QWidget* parent, const char* name,
                           bool modal, WFlags f )
    : QDialog( parent, name, modal, f )
{
    m_elements = elements;
    m_decimalPlaces = decimalPlaces;
```

我们传递了绝大部分参数到 `QDialog` 超类中。我们把元素矢量指针和所要显示的小数点位数赋给成员变量，这样它们就可以被所有的 `SetDataForm` 的成员函数访问了。

```
setCaption( "Chart -- Set Data" );
resize( 540, 440 );
```

我们为对话框设置一个标题并且重定义它的大小。

```
tableButtonBox = new QVBoxLayout( this, 11, 6, "table button box layout" );
```

这个视窗的布局相当简单。按钮被组织在一个水平的布局中并且表和这个按钮布局通过使用 `tableButtonBox` 布局被竖直地组织在一起。

```
table = new QTable( this, "data table" );
table->setNumCols( 5 );
table->setNumRows( ChartForm::MAX_ELEMENTS );
```



```

table->setColumnReadOnly( 1, true );
table->setColumnReadOnly( 2, true );
table->setColumnReadOnly( 4, true );
table->setColumnWidth( 0, 80 );
table->setColumnWidth( 1, 60 ); // Columns 1 and 4 must be equal
table->setColumnWidth( 2, 60 );
table->setColumnWidth( 3, 200 );
table->setColumnWidth( 4, 60 );
QHeader *th = table->horizontalHeader();
th->setLabel( 0, "Value" );
th->setLabel( 1, "Color" );
th->setLabel( 2, "Pattern" );
th->setLabel( 3, "Label" );
th->setLabel( 4, "Color" );
tableButtonBox->addWidget( table );

```

我们创建一个有五列的新的 **QTable**，并且它的行数和元素矢量中的元素个数相同。我们让颜色和样式列只读：这是为了防止用户在这些地方输入。我们将通过让用户在颜色上点击或者定位到颜色上并且点击**Color**按钮时可以修改颜色。样式被放在一个组合框中，很简单地通过用户选择一个不同地样式就可以改变它。接下来我们设置合适地初始宽度，为每一列插入标签并且最后把这个表添加到**tableButtonBox**布局中。

```

buttonBox = new QHBoxLayout( 0, 0, 6, "button box layout" );

```

我们创建一个水平盒子布局用来保存按钮。

```

colorPushButton = new QPushButton( this, "color button" );
colorPushButton->setText( "&Color..." );
colorPushButton->setEnabled( false );
buttonBox->addWidget( colorPushButton );

```

我们创建一个 **color** 按钮并把它添加到 **buttonBox** 布局中。我们让这个按钮失效，只有当焦点在一个颜色单元格时，我们才会让它有效。

```

QSpacerItem *spacer = new QSpacerItem( 0, 0, QSizePolicy::Expanding,
                                         QSizePolicy::Minimum );
buttonBox->addItem( spacer );

```

因为我们想把 **color** 按钮和 **OK** 以及 **Cancel** 按钮分开，接下来我们创建一个间隔并把它添加到 **buttonBox** 布局中。

```

okPushButton = new QPushButton( this, "ok button" );
okPushButton->setText( "OK" );
okPushButton->setDefault( TRUE );
buttonBox->addWidget( okPushButton );

cancelPushButton = new QPushButton( this, "cancel button" );
cancelPushButton->setText( "Cancel" );

```

```
cancelPushButton->setAccel( Key_Escape );
buttonBox->addWidget( cancelButton );
```

OK 和 Cancel 按钮被创建了并被添加到 `buttonBox`。我们让 OK 按钮为这个对话框的默认按钮，并且我们为 Cancel 按钮提供了一个 Esc 加速键。

```
tableButtonBox->addLayout( buttonBox );
```

我们把 `buttonBox` 布局添加到 `tableButtonBox` 中，并且这个布局也是完整的。

```
connect( table, SIGNAL( clicked(int,int,int,const QPoint&) ),
        this, SLOT( setColor(int,int) ) );
connect( table, SIGNAL( currentChanged(int,int) ),
        this, SLOT( currentChanged(int,int) ) );
connect( table, SIGNAL( valueChanged(int,int) ),
        this, SLOT( valueChanged(int,int) ) );
connect( colorPushButton, SIGNAL( clicked() ), this, SLOT( setColor() ) );
connect( okPushButton, SIGNAL( clicked() ), this, SLOT( accept() ) );
connect( cancelButton, SIGNAL( clicked() ), this, SLOT( reject() ) );
```

现在我们来演习一下这个视窗。

- 如果用户点击了一个单元格，我们调用 `setColor()`槽，它会检查这个单元格是否保存一个颜色，如果是的，将会调用颜色对话框。
- 我们把 `QTable`的`currentChanged()`信号和我们的`currentChanged()`槽连接起来了，举例来说，这将被用在根据用户现在所在的列来决定使color按钮有效/失效。
- 我们把表格的 `valueChanged()`和我们的 `valueChanged()`槽连接起来了，我们将会用这个来显示带有正确的小数位数的值。
- 如果用户点击 Color 按钮，我们就调用 `setColor()`槽。
- OK 按钮被连接到 `accept()`槽，我们将会在这个槽里面更新元素矢量。
- Cancel按钮被连接到 `QDialog`的`reject()`槽，并且这部分中不再需要更多的代码和动作。

```
QPixmap patterns[MAX_PATTERNS];
patterns[0] = QPixmap( pattern01 );
patterns[1] = QPixmap( pattern02 );
```

我们为每一个画刷样式创建了一个图片并且把它们存储在 `patterns` 数组中。

```
QRect rect = table->cellRect( 0, 1 );
QPixmap pix( rect.width(), rect.height() );
```

我们每一个颜色单元格所占用的矩形并创建一个这样大小的空白图片。

```
for ( int i = 0; i < ChartForm::MAX_ELEMENTS; ++i ) {
    Element element = (*m_elements)[i];

    if ( element.isValid() )
        table->setText(
```

```

        i, 0,
        QString( "%1" ).arg( element.value(), 0, 'f',
                               m_decimalPlaces ) );

    QColor color = element.valueColor();
    pix.fill( color );
    table->setPixmap( i, 1, pix );
    table->setText( i, 1, color.name() );

    QComboBox *combobox = new QComboBox;
    for ( int j = 0; j < MAX_PATTERNS; ++j )
        combobox->insertItem( patterns[j] );
    combobox->setCurrentItem( element.valuePattern() - 1 );
    table->setCellWidget( i, 2, combobox );

    table->setText( i, 3, element.label() );

    color = element.labelColor();
    pix.fill( color );
    table->setPixmap( i, 4, pix );
    table->setText( i, 4, color.name() );

```

对于元素矢量中的每一个元素，我们必须填充表格。

如果元素是有效的，我们把它的值写在第一列（0列，Value），根据指定的小数点位数进行格式化。

我们读元素的值颜色并用这种颜色填充空白图片，然后我们让颜色单元格显示这个图片。我们需要能够在以后读到这个颜色（比如用户改变了颜色）。一个方法就是测试图片中的一个像素，另一个就是继承 `QTableWidgetItem`（和我们继承`CanvasText`类似）并且在里面存储这个颜色。但是我们用了一个简单的方法：我们设置这个单元格的文本为这个颜色的名字。

接下来我们用样式来填充样式组合框。我们将通过使用被选择的样式在组合框中的位置来决定用户选择了哪一个样式。`QTable`可以利用 `QComboBoxTableItem`条目，但是只支持文本，所以我们使用`setCellWidget()`来代替把 `QComboBox`的插入到表中。

接下来我们插入元素的标签。最后我们用我们设置值颜色的方法来设置标签颜色。

## 槽

```

void SetDataForm::currentChanged( int row, int col )
{
    colorPushButton->setEnabled( col == 1 || col == 4 );
    if ( col == 2 )
        ((QComboBox*)table->cellWidget( row, col ))->popup();
}

```

当用户进行定位时，表的 `currentChanged()` 信号被发射。如果用户进入 1 或 4 列时（值颜色或标签颜色），我们让 `colorPushButton` 生效，否则让它失效。

为了给键盘用户提供方便，如果用户定位到样式组合框中时，我们弹出它。

```
void SetDataForm::valueChanged( int row, int col )
{
    if ( col == 0 ) {
        bool ok;
        double d = table->text( row, col ).toDouble( &ok );
        if ( ok && d > EPSILON )
            table->setText(
                row, col, QString( "%1" ).arg(
                    d, 0, 'f', m_decimalPlaces ) );
        else
            table->setText( row, col, table->text( row, col ) + "?" );
    }
}
```

如果用户改变值，我们必须使用正确的小数位数对它进行格式化，或者指出它是无效的。

```
void SetDataForm::setColor()
{
    setColor( table->currentRow(), table->currentColumn() );
    table->setFocus();
}
```

如果用户按下 `Color` 按钮，我们调用另一个 `setColor()` 函数并把焦点返回到表中。

```
void SetDataForm::setColor( int row, int col )
{
    if ( !( col == 1 || col == 4 ) )
        return;

    QColor color = QColorDialog::getColor(
        QColor( table->text( row, col ) ),
        this, "color dialog" );
    if ( color.isValid() ) {
        QPixmap pix = table->pixmap( row, col );
        pix.fill( color );
        table->setPixmap( row, col, pix );
        table->setText( row, col, color.name() );
    }
}
```

如果当焦点在一个颜色单元格中时这个函数被调用，我们调用静态的 `QColorDialog::getColor()` 对话框来获得用户所选择的颜色。如果他们选择了一个颜色，我们就用这种颜色来填充颜色单元格的图片，并且设置单元格的文本为新的颜色的名称。

```

void SetDataForm::accept()
{
    bool ok;
    for ( int i = 0; i < ChartForm::MAX_ELEMENTS; ++i ) {
        Element &element = (*m_elements)[i];
        double d = table->text( i, 0 ).toDouble( &ok );
        if ( ok )
            element.setValue( d );
        else
            element.setValue( Element::INVALID );
        element.setValueColor( QColor( table->text( i, 1 ) ) );
        element.setValuePattern(
            ((QComboBox*)table->cellWidget( i, 2 ))->currentItem() + 1 );
        element.setLabel( table->text( i, 3 ) );
        element.setLabelColor( QColor( table->text( i, 4 ) ) );
    }

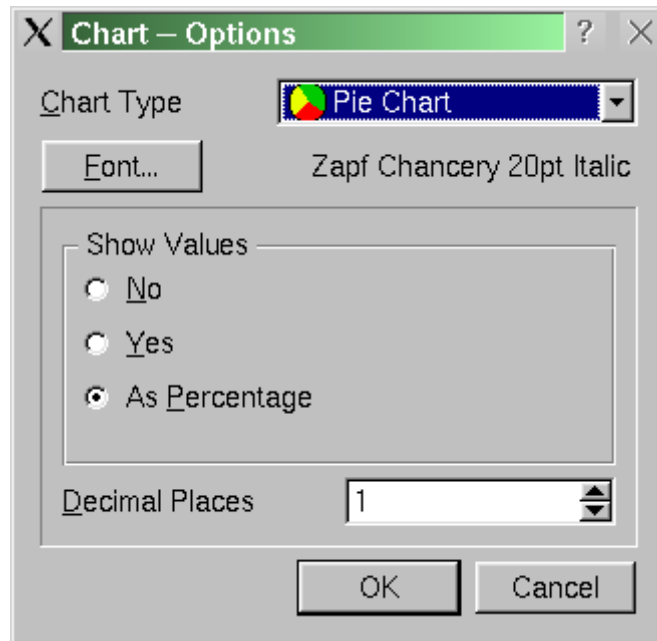
    QDialog::accept();
}

```

如果用户点击OK，我们必须更新元素矢量。我们对矢量进行迭代并把每一个元素的值设置为用户输入的值，否则如果值是无效的就设置为INVALID。我们通过颜色的名称作为参数临时构造一个 **QColor**来设置值颜色和标签颜色。样式被设置为样式组合框的当前条目与1的偏移量（因为我们的样式数字是从1开始的，但是组合框的条目是从0开始索引的）。

最后我们调用 **QDialog::accept()**。

## 设置选项



我们提供了一个选项对话框，这样用户就可以在一个地方对所有的数据组设置选项。

（由 optionsform.h 展开。）

```
class OptionsForm : public QDialog
{
    Q_OBJECT
public:
    OptionsForm( QWidget* parent = 0, const char* name = "options form",
                bool modal = FALSE, WFlags f = 0 );
    ~OptionsForm() {}

    QFont font() const { return m_font; }
    void setFont( QFont font );

    QLabel *chartTypeTextLabel;
    QComboBox *chartTypeComboBox;
    QPushButton *fontPushButton;
    QLabel *fontTextLabel;
    QFrame *addValuesFrame;
    QButtonGroup *addValuesButtonGroup;
    QRadioButton *noRadioButton;
    QRadioButton *yesRadioButton;
    QRadioButton *asPercentageRadioButton;
    QLabel *decimalPlacesTextLabel;
    QSpinBox *decimalPlacesSpinBox;
    QPushButton *okPushButton;
    QPushButton *cancelPushButton;

protected slots:
```

```

void chooseFont();

protected:
    QVBoxLayout *optionsFormLayout;
    QHBoxLayout *chartTypeLayout;
    QHBoxLayout *fontLayout;
    QVBoxLayout *addValuesFrameLayout;
    QVBoxLayout *addValuesButtonGroupLayout;
    QHBoxLayout *decimalPlacesLayout;
    QHBoxLayout *buttonsLayout;

private:
    QFont m_font;
};

```

这个对话框的布局比设置数据视窗要更复杂一些，但是我们只需要一个单一的槽。不像“聪明的”设置数据视窗那样，这是一个“哑的”对话框，它只向窗口部件的调用者提供了读和写。调用者有责任基于用户所作的改变更新事物。

（由 optionsform.cpp 展开。）

```

#include "images/options_horizontalbarchart.xpm"
#include "images/options_piechart.xpm"
#include "images/options_verticalbarchart.xpm"

```

我们包含了一些在图表类型组合框中要使用的图片。

## 构造函数

```

OptionsForm::OptionsForm( QWidget* parent, const char* name,
                          bool modal, WFlags f )
    : QDialog( parent, name, modal, f )
{
    setCaption( "Chart -- Options" );
    resize( 320, 290 );
}

```

我们把所有的参数传递给 `QDialog` 构造函数，设置一个题目并且设置一个初始大小。

视窗的布局将是一个包含图表类型标签和组合框的水平盒子布局，并且对于字体按钮和字体标签、小数点位置标签和微调框也是相似的。按钮也会被放在一个水平布局中，但是还会有一个间隔来把它们移到右边。显示值的单选按钮将会竖直地排列在一个框架中。所有地这些都被放在一个竖直盒子布局中。

```
optionsFormLayout = new QVBoxLayout( this, 11, 6 );
```

所有的窗口部件都被放在视窗的竖直盒子布局中。

```
chartTypeLayout = new QHBoxLayout( 0, 0, 6 );
```

图表类型标签和组合框将被并排放置。

```
chartTypeTextLabel = new QLabel( "&Chart Type", this );
chartTypeLayout->addWidget( chartTypeTextLabel );
```

```
chartTypeComboBox = new QComboBox( false, this );
chartTypeComboBox->insertItem( QPixmap( options_piechart ), "Pie
Chart" );
chartTypeComboBox->insertItem( QPixmap( options_verticalbarchart ),
                                "Vertical Bar Chart" );
chartTypeComboBox->insertItem( QPixmap( options_horizontalbarchart ),
                                "Horizontal Bar Chart" );
chartTypeLayout->addWidget( chartTypeComboBox );
optionsFormLayout->addLayout( chartTypeLayout );
```

我们创建图表类型标签（带有一个加速键，稍后我们会把它和图表类型组合框联系起来）。我们也创建一个图表类型组合框，用图片和文本来填充它。我们把它们两个添加到水平布局中，并把水平布局添加到视窗的垂直布局中。

```
fontLayout = new QHBoxLayout( 0, 0, 6 );

fontPushButton = new QPushButton( "&Font...", this );
fontLayout->addWidget( fontPushButton );
QSpacerItem* spacer = new QSpacerItem( 0, 0,
                                       QSizePolicy::Expanding,
                                       QSizePolicy::Minimum );

fontLayout->addItem( spacer );

fontTextLabel = new QLabel( this ); // 必须由调用者通过setFont()来设置
fontLayout->addWidget( fontTextLabel );
optionsFormLayout->addLayout( fontLayout );
```

我们创建一个水平盒子布局用来保存字体按钮和字体标签。字体按钮是被直接加入的。我们添加了一个间隔用来增加效果。字体文本标签被初始化为空（因为我们不知道用户正在使用什么字体）。

```
addValuesFrame = new QFrame( this );
addValuesFrame->setFrameShape( QFrame::StyledPanel );
addValuesFrame->setFrameShadow( QFrame::Sunken );
addValuesFrameLayout = new QVBoxLayout( addValuesFrame, 11, 6 );

addValuesButtonGroup = new QButtonGroup( "Show Values", addValuesFrame );
addValuesButtonGroup->setColumnLayout( 0, Qt::Vertical );
addValuesButtonGroup->layout()->setSpacing( 6 );
addValuesButtonGroup->layout()->setMargin( 11 );
```



```

addValuesButtonGroupLayout = new QVBoxLayout (
                                addValuesButtonGroup->layout() );
addValuesButtonGroupLayout->setAlignment( Qt::AlignTop );

noRadioButton = new QRadioButton( "&No", addValuesButtonGroup );
noRadioButton->setChecked( true );
addValuesButtonGroupLayout->addWidget( noRadioButton );

yesRadioButton = new QRadioButton( "&Yes", addValuesButtonGroup );
addValuesButtonGroupLayout->addWidget( yesRadioButton );

asPercentageRadioButton = new QRadioButton( "As &Percentage",
                                              addValuesButtonGroup );
addValuesButtonGroupLayout->addWidget( asPercentageRadioButton );
addValuesFrameLayout->addWidget( addValuesButtonGroup );

```

用户也许选择显示它们自己的标签或者在每一个标签的末尾加上值，或者加上百分比。

我们创建一个框架来存放单选按钮并且为它们创建了一个布局。我们创建了一个按钮组（这样 Qt 就可以自动地处理专有的单选按钮行为了）。接下来我们创建单选按钮，并把“No”作为默认值。

小数位标签和微调框被放在另一个水平布局中，并且按钮和设置数据视窗中的按钮的排布方式非常相似。

```

connect( fontPushButton, SIGNAL( clicked() ), this, SLOT( chooseFont() ) );
connect( okPushButton, SIGNAL( clicked() ), this, SLOT( accept() ) );
connect( cancelPushButton, SIGNAL( clicked() ), this, SLOT( reject() ) );

```

我们只需要三个连接：

1. 当用户点击字体按钮时，我们执行我们自己的 `chooseFont()`槽。
2. 如果用户点击OK，我们调用 `QDialog::accept()`，它会让调用者来从对话框的窗口部件中读取数据并且执行任何必要的动作。
3. 如果用户点击Cancel，我们调用 `QDialog::reject()`。

```

chartTypeTextLabel->setBuddy( chartTypeComboBox );
decimalPlacesTextLabel->setBuddy( decimalPlacesSpinBox );

```

我们使用 `setBuddy()`函数来连接窗口部件和标签的加速键。

## 槽

```

void OptionsForm::chooseFont ()
{
    bool ok;
    QFont font = QFontDialog::getFont( &ok, m_font, this );

```

```

        if ( ok )
            setFont( font );
    }

```

当用户点击**Font**按钮时，这个槽被调用。它简单地调用静态的 `QFontDialog::getFont()` 来获得用户选择的字体。如果他们选择了一个字体，我们调用我们的 `setFont()` 槽在字体标签中提供一个字体的文本描述。

```

void OptionsForm::setFont( QFont font )
{
    QString label = font.family() + " " +
                    QString::number( font.pointSize() ) + "pt";
    if ( font.bold() )
        label += " Bold";
    if ( font.italic() )
        label += " Italic";
    fontTextLabel->setText( label );
    m_font = font;
}

```

这个函数在字体标签中显示一个被选字体的文本描述，并且在 `m_font` 成员中保存一个字体的拷贝。我们需要这个字体为成员，这样我们就会为 `chooseFont()` 提供一个默认字体。

## 项目文件

(chart.pro.)

```

TEMPLATE = app
CONFIG += warn_on

HEADERS += element.h \
            canvastext.h \
            canvasview.h \
            chartform.h \
            optionsform.h \
            setdataform.h
SOURCES += element.cpp \
            canvasview.cpp \
            chartform.cpp \
            chartform_canvas.cpp \
            chartform_files.cpp \
            optionsform.cpp \
            setdataform.cpp \
            main.cpp

```

通过使用项目文件，我们能够把我们自己从为我们所要使用的平台创建Makefile中脱离出来。为了生成一个Makefile我们所要做的一切就是运行 `qmake`，比如：

```
qmake -o Makefile chart.pro
```

## 完成

chart应用程序显示了用Qt创建应用程序和对话框是多么的直接。创建菜单和工具条是很容易的并且Qt的 [信号和槽](#)机制相当地简化了图形用户界面事件处理。

手工创建布局可能会花费一些时间来掌握，但是有一种容易的选择：[Qt设计器](#)。[Qt设计器](#)包括了简单但强大的布局工具和一个代码编辑器。它可以自动地生成main.cpp和.pro项目文件。

chart 应用程序对于进一步开发和实验是成熟的。你也许可以考虑实现下面的一些思路：

- 使用 [QValidator](#)子类来保证只有有效的双精度实数被输入到值中。
- 添加更多的图表类型，比如线图、区域图和高低图。
- 允许用户设置上下左右边白。
- 允许用户指定一个可以像标签一样拖拽的标题。
- 提供一个绘制和标签选项的中心线。
- 提供一个选项用键（或者图例）来替换标签。
- 为所有的图表类型添加一个三维的查看选项。

# qmake 用户手册

[qmake的介绍](#)

[安装qmake](#)

[10 分钟学会使用qmake](#)

[qmake教程](#)

[qmake概念](#)

[qmake高级概念](#)

# 第一章 qmake 的介绍

## qmake 的介绍

*qmake* 是 Trolltech 公司创建的用来为不同的平台和编译器书写 Makefile 的工具。

手写 Makefile 是比较困难并且容易出错的，尤其是需要给不同的平台和编译器组合写几个 Makefile。使用 *qmake*，开发者创建一个简单的“项目”文件并且运行 *qmake* 生成适当的 Makefile。*qmake* 会注意所有的编译器和平台的依赖性，可以把开发者解放出来只关心他们的代码。Trolltech 公司使用 *qmake* 作为 Qt 库和 Qt 所提供的工具的主要连编工具。

*qmake* 也注意了 Qt 的特殊需求，可以自动的包含 `moc` 和 `uic` 的连编规则。

## 第二章 安装 qmake

### 安装 qmake

当 Qt 被连编的时候，默认情况下 *qmake* 也会被连编。

这一部分解释如何手工连编 *qmake*。如果你已经有了 *qmake*，可以跳过这里，请看 [10 分钟学会使用 qmake](#)。

### 手动安装 qmake

在手工连编 Qt 之前，下面这些环境变量必须被设置：

- QMAKESPEC

这个必须设置为你所使用的系统的平台和编译器的组合。

举例来说，加入你使用的是 Windows 和 Microsoft Visual Studio，你应该把环境变量设置为 `win32-msvc`。如果你使用的是 Solaris 和 g++，你应该把环境变量设置为 `solaris-g++`。

当你在设置 QMAKESPEC 时，可以从下面的可能的环境变量列表中进行选择：

aix-64 hpux-cc irix-032 netbsd-g++ solaris-cc unixware7-g++ aix-g++ hpux-g++  
linux-cxx openbsd-g++ solaris-g++ win32-borland aix-xlc hpux-n64 linux-g++  
openunix-cc sunos-g++ win32-g++ bsdi-g++ hpux-o64 linux-icc qnx-g++ tru64-cxx  
win32-msvc dgux-g++ hurd-g++ linux-kcc reliant-64 tru64-g++ win32-watc freebsd-g++  
irix-64 macx-pbuilder reliant-cds ultrix-g++ win32-visa hpux-acc irix-g++ macx-g++  
sco-g++ unixware-g hpux-acc irix-n32 solaris-64 unixware7-cc

`envvar` 是下面之一时，环境变量应该被设置到 `qws/envvar`:

`linux-arm-g++ linux-generic-g++ linux-mips-g++ linux-x86-g++ linux-freebsd-g++  
linux-ipaq-g++ linux-solaris-g++ qnx-rtp-g++`

- **QTDIR**

这个必须设置到 Qt 被（或者将被）安装到的地方。比如，`c:\qt` 和 `/local/qt`。

一旦环境变量被设置到 `qmake` 目录，`$QTDIR/qmake`，比如 `C:\qt\qmake`，现在根据你的编译器运行 `make` 或者 `nmake`。

当编译完成时，`qmake` 已经可以使用了。

## 第三章 10 分钟学会使用 `qmake`

### 1 创建一个项目文件

`qmake` 使用储存在项目（`.pro`）文件中的信息来决定 `Makefile` 文件中该生成什么。

一个基本的项目文件包含关于应用程序的信息，比如，编译应用程序需要哪些文件，并且使用哪些配置设置。

这里是一个简单的示例项目文件：

```
SOURCES = hello.cpp
HEADERS = hello.h
CONFIG += qt warn_on release
```

我们将会提供一行一行的简要解释，具体细节将会在手册的后面的部分解释。

```
SOURCES = hello.cpp
```

这一行指定了实现应用程序的源程序文件。在这个例子中，恰好只有一个文件，`hello.cpp`。大部分应用程序需要多个文件，这种情况下可以把文件列在一行中，以空格分隔，就像这样：

```
SOURCES = hello.cpp main.cpp
```

另一种方式，每一个文件可以被列在一个分开的行里面，通过反斜线另起一行，就像这样：

```
SOURCES = hello.cpp \  
          main.cpp
```

一个更冗长的方法是单独地列出每一个文件，就像这样：

```
SOURCES += hello.cpp
SOURCES += main.cpp
```

这种方法中使用“+=”比“=”更安全，因为它只是向已有的列表中添加新的文件，而不是替换整个列表。

**HEADERS** 这一行中通常用来指定为这个应用程序创建的头文件，举例来说：

```
HEADERS += hello.h
```

列出源文件的任何一个方法对头文件也都适用。

**CONFIG** 这一行是用来告诉 *qmake* 关于应用程序的配置信息。

```
CONFIG += qt warn_on release
```

在这里使用“+=”，是因为我们添加我们的配置选项到任何一个已经存在中。这样做比使用“=”那样替换已经指定的所有选项是更安全的。

**CONFIG** 一行中的 *qt* 部分告诉 *qmake* 这个应用程序是使用 Qt 来连编的。这也就是说 *qmake* 在连接和为编译添加所需的包含路径的时候会考虑到 Qt 库的。

**CONFIG** 一行中的 *warn\_on* 部分告诉 *qmake* 要把编译器设置为输出警告信息的。

**CONFIG** 一行中的 *release* 部分告诉 *qmake* 应用程序必须被连编为一个发布的应用程序。在开发过程中，程序员也可以使用 *debug* 来替换 *release*，稍后会讨论这里的。

项目文件就是纯文本（比如，可以使用像记事本、vim 和 xemacs 这些编辑器）并且必须存为“.pro”扩展名。应用程序的执行文件的名称必须和项目文件的名称一样，但是扩展名是跟着平台而改变的。举例来说，一个叫做“hello.pro”的项目文件将会在 Windows 下生成“hello.exe”，而在 Unix 下生成“hello”。

## 2 生成 Makefile

当你已经创建好你的项目文件，生成 **Makefile** 就很容易了，你所要做的就是先到你所生成的项目文件那里然后输入：

**Makefile** 可以像这样由“.pro”文件生成：

```
qmake -o Makefile hello.pro
```

对于 Visual Studio 的用户，*qmake* 也可以生成“.dsp”文件，例如：

```
qmake -t vcapp -o hello.dsp hello.pro
```

## 第四章 qmake 教程

### 1 qmake 教程介绍

这个教程可以教会你如何使用 *qmake*。我们建议你看完这个教程之后读一下 *qmake* 手册。

### 2 开始很简单

让我们假设你已经完成了你的应用程序的一个基本实现，并且你已经创建了下述文件：

- `hello.cpp`
- `hello.h`
- `main.cpp`

你可以在 *qt/qmake/example* 中发现这些文件。你对这个应用程序的配置仅仅知道的另一件事是它是用 Qt 写的。首先，使用你所喜欢的纯文本编辑器，在 *qt/qmake/tutorial* 中创建一个叫做 *hello.pro* 的文件。你所要做的第一件事是添加一些行来告诉 *qmake* 关于你所开发的项目中的源文件和头文件这一部分。

我们先把源文件添加到项目文件中。为了做到这点，你需要使用 **SOURCES** 变量。只要用 *SOURCES +=* 来开始一行，并且把 `hello.cpp` 放到它后面。你需要写成这样：

```
SOURCES += hello.cpp
```

我们对项目中的每一个源文件都这样做，直到结束：

```
SOURCES += hello.cpp
SOURCES += main.cpp
```

如果你喜欢使用像 **Make** 一样风格的语法，你也可以写成这样，一行写一个源文件，并用反斜线结尾，然后再起新的一行：

```
SOURCES = hello.cpp \
          main.cpp
```

现在源文件已经被列到项目文件中了，头文件也必须添加。添加的方式和源文件一样，除了变量名是 **HEADERS**。

当你做完这些时，你的项目文件就像现在这样：



```
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

目标名称是自动设置的，它被设置为和项目文件一样的名称，但是为了适合平台所需要的后缀。举例来说，加入项目文件叫做“**hello.pro**”，在 **Windows** 上的目标名称应该是“**hello.exe**”，在 **Unix** 上应该是“**hello**”。如果你想设置一个不同的名字，你可以在项目文件中设置它：

```
TARGET = helloworld
```

最后一步是设置 **CONFIG** 变量。因为这是一个 **Qt** 应用程序，我们需要把“**qt**”放到 **CONFIG** 这一行中，这样 **qmake** 才会在连接的时候添加相关的库，并且保证 **moc** 和 **uic** 的连编行也被包含到 **Makefile** 中。

最终完成的项目文件应该是这样的：

```
CONFIG += qt
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

你现在可以使用 **qmake** 来为你的应用程序生成 **Makefile**。在你的应用程序目录中，在命令行下输入：

```
qmake -o Makefile hello.pro
```

然后根据你所使用的编译器输入 **make** 或者 **nmake**。

### 3 使应用程序可以调试

应用程序的发布版本不包含任何调试符号或者其它调试信息。在开发过程中，生成一个含有相关信息的应用程序的调试版本是很有用处的。通过项目文件的 **CONFIG** 变量中添加“**debug**”就可以很简单地实现。

例如：

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

像前面一样使用 **qmake** 来生成一个 **Makefile** 并且你就能够调试你的应用程序了。

## 4 添加特定平台的源文件

在编了几个小时的程序之后，你也许开始为你的应用程序编写与平台相关的部分，并且决定根据平台的不同编写不同的代码。所以现在你有两个源文件要包含到你的项目文件中——*hello\_win.cpp* 和 *hello\_x11.cpp*。我们不能仅仅把这两个文件放到 *SOURCES* 变量中，因为那样的话会把这两个文件都加到 *Makefile* 中。所以我们在这里需要做的是根据 *qmake* 所运行的平台来使用相应的作用域来进行处理。

为 Windows 平台添加的依赖平台的文件的简单的作用域看起来就像这样：

```
win32 {
    SOURCES += hello_win.cpp
}
```

所以如果 *qmake* 运行在 Windows 上的时候，它就会把 *hello\_win.cpp* 添加到源文件列表中。如果 *qmake* 运行在其它平台上的时候，它会很简单地把这部分忽略。现在接下来我们要做的就是添加一个 X11 依赖文件的作用域。

当你做完了这部分，你的项目文件应该和这样差不多：

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
    SOURCES += hello_win.cpp
}
x11 {
    SOURCES += hello_x11.cpp
}
```

像前面一样使用 *qmake* 来生成 *Makefile*。

## 5 如果一个文件不存在，停止 *qmake*

如果某一个文件不存在的时候，你也许不想生成一个 *Makefile*。我们可以通过使用 *exists()* 函数来检查一个文件是否存在。我们可以通过使用 *error()* 函数把正在运行的 *qmake* 停下来。这和作用域的工作方式一样。只要很简单地用这个函数来替换作用域条件。对 *main.cpp* 文件的检查就像这样：

```
!exists( main.cpp ) {
    error( "No main.cpp file found" )
}
```

“!”用来否定这个测试，比如，如果文件存在，*exists( main.cpp )* 是真，如果文件不存在，*!exists( main.cpp )* 是真。

```

CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
    SOURCES += hello_win.cpp
}
x11 {
    SOURCES += hello_x11.cpp
}
!exists( main.cpp ) {
    error( "No main.cpp file found" )
}

```

像前面一样使用 *qmake* 来生成 Makefile。如果你临时改变 *main.cpp* 的名称，你会看到信息，并且 *qmake* 会停止处理。

## 6 检查多于一个的条件

假设你使用 Windows 并且当你在命令行运行你的应用程序的时候你想能够看到 `qDebug()` 语句。除非你在连编你的程序的时候使用 `console` 设置，你不会看到输出。我们可以很容易地把 `console` 添加到 `CONFIG` 行中，这样在 Windows 下，Makefile 就会有这个设置。但是如果告诉你我们只是想在当我们的应用程序运行在 Windows 下并且当 `debug` 已经在 `CONFIG` 行中的时候，添加 `console`。这需要两个嵌套的作用域：只要生成一个作用域，然后在它里面再生成另一个。把设置放在最里面的作用域里，就像这样：

```

win32 {
    debug {
        CONFIG += console
    }
}

```

嵌套的作用域可以使用冒号连接起来，所以最终的项目文件看起来像这样：

```

CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
    SOURCES += hello_win.cpp
}
x11 {
    SOURCES += hello_x11.cpp
}
!exists( main.cpp ) {
    error( "No main.cpp file found" )
}

```

```
}  
win32:debug {  
    CONFIG += console  
}
```

就这些了！你现在已经完成了 *qmake* 的教程，并且已经准备好为你的开发项目写项目文件了。

## 第五章 *qmake* 概念

### 1 介绍 *qmake*

*qmake* 是用来为不同的平台的开发项目创建 *makefile* 的 Trolltech 开发一个易于使用的工具。*qmake* 简化了 *makefile* 的生成，所以为了创建一个 *makefile* 只需要一个只有几行信息的文件。*qmake* 可以供任何一个软件项目使用，而不用管它是不是用 Qt 写的，尽管它包含了为支持 Qt 开发所拥有的额外的特征。

*qmake* 基于一个项目文件这样的信息来生成 *makefile*。项目文件可以由开发者生成。项目文件通常很简单，但是如果需要它是非常完善的。不用修改项目文件，*qmake* 也可以为 Microsoft Visual Studio 生成项目。

### 2 *qmake* 的概念

#### QMAKESPEC 环境变量

举例来说，如果你在 Windows 下使用 Microsoft Visual Studio，然后你需要把 QMAKESPEC 环境变量设置为 *win32-msvc*。如果你在 Solaris 上使用 gcc，你需要把 QMAKESPEC 环境变量设置为 *solaris-g++*。

在 *qt/mkspecs* 中的每一个目录里面，都有一个包含了平台和编译器特定信息的 *qmake.conf* 文件。这些设置适用于你要使用 *qmake* 的任何项目，请不要修改它，除非你是一个专家。例如，假如你所有的应用程序都必须和一个特定的库连接，你可以把这个信息添加到相应的 *qmake.conf* 文件中。

## 项目(.pro)文件

一个项目文件是用来告诉 *qmake* 关于为这个应用程序创建 *makefile* 所需要的细节。例如，一个源文件和头文件的列表、任何应用程序特定配置、例如一个必需要连接的额外库、或者一个额外的包含路径，都应该放到项目文件中。

### “#”注释

你可以为项目文件添加注释。注释由“#”符号开始，一直到这一行的结束。

## 3 模板

模板变量告诉 *qmake* 为这个应用程序生成哪种 *makefile*。下面是可供使用的选择：

- **app** - 建立一个应用程序的 *makefile*。这是默认值，所以如果模板没有被指定，这个将被使用。
- **lib** - 建立一个库的 *makefile*。
- **vcapp** - 建立一个应用程序的 Visual Studio 项目文件。
- **vclib** - 建立一个库的 Visual Studio 项目文件。
- **subdirs** - 这是一个特殊的模板，它可以创建一个能够进入特定目录并且为一个项目文件生成 *makefile* 并且为它调用 *make* 的 *makefile*。

### “app”模板

“app”模板告诉 *qmake* 为建立一个应用程序生成一个 *makefile*。当使用这个模板时，下面这些 *qmake* 系统变量是被承认的。你应该在你的 .pro 文件中使用它们来为你的应用程序指定特定信息。

- **HEADERS** - 应用程序中的所有头文件的列表。
- **SOURCES** - 应用程序中的所有源文件的列表。
- **FORMS** - 应用程序中的所有 .ui 文件（由 *Qt 设计器* 生成）的列表。
- **LEXSOURCES** - 应用程序中的所有 *lex* 源文件的列表。
- **YACCSOURCES** - 应用程序中的所有 *yacc* 源文件的列表。
- **TARGET** - 可执行应用程序的名称。默认值为项目文件的名称。（如果需要扩展名，会被自动加上。）
- **DESTDIR** - 放置可执行程序目标的目录。
- **DEFINES** - 应用程序所需的额外的预处理程序定义的列表。
- **INCLUDEPATH** - 应用程序所需的额外的包含路径的列表。
- **DEPENDPATH** - 应用程序所依赖的搜索路径。
- **VPATH** - 寻找补充文件的搜索路径。
- **DEF\_FILE** - 只有 Windows 需要：应用程序所要连接的 .def 文件。
- **RC\_FILE** - 只有 Windows 需要：应用程序的资源文件。
- **RES\_FILE** - 只有 Windows 需要：应用程序所要连接的资源文件。

你只需要使用那些你已经有的系统变量，例如，如果你不需要任何额外的 `INCLUDEPATH`，那么你就不需要指定它，*qmake* 会为所需的提供默认值。例如，一个实例项目文件也许就像这样：

```
TEMPLATE = app
DESTDIR  = c:\helloapp
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
DEFINES += QT_DLL
CONFIG += qt warn_on release
```

如果条目是单值的，比如 `template` 或者目的目录，我们是用“=”，但如果是多值条目，我们使用“+=”来为这个类型添加现有的条目。使用“=”会用新值替换原有的值，例如，如果我们写了 `DEFINES=QT_DLL`，其它所有的定义都将被删除。

## “lib”模板

“lib”模板告诉 *qmake* 为建立一个库而生成 `makefile`。当使用这个模板时，除了“app”模板中提到系统变量，还有一个 `VERSION` 是被支持的。你需要在为库指定特定信息的 `.pro` 文件中使用它们。

- `VERSION` - 目标库的版本号，比如，2.3.1。

## “subdirs”模板

“subdirs”模板告诉 *qmake* 生成一个 `makefile`，它可以进入到特定子目录并为这个目录中的项目文件生成 `makefile` 并且为它调用 `make`。

在这个模板中只有一个系统变量 `SUBDIRS` 可以被识别。这个变量中包含了所要处理的含有项目文件的子目录的列表。这个项目文件的名称是和子目录同名的，这样 *qmake* 就可以发现它。例如，如果子目录是“myapp”，那么在这个目录中的项目文件应该被叫做 *myapp.pro*。

## 4 CONFIG 变量

配置变量指定了编译器所要使用的选项和所需要被连接的库。配置变量中可以添加任何东西，但只有下面这些选项可以被 *qmake* 识别。

下面这些选项控制着使用哪些编译器标志：

- `release` - 应用程序将以 `release` 模式连编。如果“`debug`”被指定，它将被忽略。
- `debug` - 应用程序将以 `debug` 模式连编。
- `warn_on` - 编译器会输出尽可能多的警告信息。如果“`warn_off`”被指定，它将被忽略。
- `warn_off` - 编译器会输出尽可能少的警告信息。

下面这些选项定义了所要连编的库/应用程序的类型：

- `qt` - 应用程序是一个 Qt 应用程序，并且 Qt 库将会被连接。
- `thread` - 应用程序是一个多线程应用程序。
- `x11` - 应用程序是一个 X11 应用程序或库。
- `windows` - 只用于“app”模板：应用程序是一个 Windows 下的窗口应用程序。
- `console` - 只用于“app”模板：应用程序是一个 Windows 下的控制台应用程序。
- `dll` - 只用于“lib”模板：库是一个共享库（dll）。
- `staticlib` - 只用于“lib”模板：库是一个静态库。
- `plugin` - 只用于“lib”模板：库是一个插件，这将会使 `dll` 选项生效。

例如，如果你的应用程序使用 Qt 库，并且你想把它连编为一个可调试的多线程的应用程序，你的项目文件应该会有下面这行：

```
CONFIG += qt thread debug
```

注意，你必须使用“+=”，不要使用“=”，否则 *qmake* 就不能正确使用连编 Qt 的设置了，比如没法获得所编译的 Qt 库的类型了。

## 第六章 qmake 高级概念

### 1 qmake 高级概念

迄今为止，我们见到的 *qmake* 项目文件都非常简单，仅仅是一些 `name = value` 和 `name += value` 的列表行。*qmake* 提供了很多更强大的功能，比如你可以使用一个简单的项目文件来为多个平台生成 makefile。

### 2 操作符

到目前为止，你已经看到在项目文件中使用的 `=` 操作符和 `+=` 操作符。这里能够提供更多的可供使用的操作符，但是其中的一些需要谨慎地使用，因为它们也许会让你期待的改变的更多。

#### “=”操作符

这个操作符简单分配一个值给一个变量。使用方法如下：

```
TARGET = myapp
```

这将会设置 `TARGET` 变量为 *myapp*。这将会删除原来对 `TARGET` 的任何设置。

## “+=”操作符

这个操作符将会向一个变量的值的列表中添加一个值。使用方法如下：

```
DEFINES += QT_DLL
```

这将会把 QT\_DLL 添加到被放到 makefile 中的预处理定义的列表中。

## “-=”操作符

这个操作符将会从一个变量的值的列表中移去一个值。使用方法如下：

```
DEFINES -= QT_DLL
```

这将会从被放到 makefile 中的预处理定义的列表中移去 QT\_DLL。

## “\*=”操作符

这个操作符仅仅在一个值不存在于一个变量的值的列表中的时候，把它添加进去。使用方法如下：

```
DEFINES *= QT_DLL
```

只用在 QT\_DLL 没有被定义在预处理定义的列表中时，它才会被添加进去。

## “~=”操作符

这个操作符将会替换任何与指定的值的正则表达式匹配的任何值。使用方法如下：

```
DEFINES ~= s/QT_[DT].+/QT
```

这将会用 QT 来替代任何以 QT\_D 或 QT\_T 开头的变量中的 QT\_D 或 QT\_T。

## 3 作用域

作用域和“if”语句很相似，如果某个条件为真，作用域中的设置就会被处理。作用域使用方法如下：

```
win32 {  
    DEFINES += QT_DLL  
}
```



上面的代码的作用是，如果在 Windows 平台上使用 *qmake*，`QT_DLL` 定义就会被添加到 `makefile` 中。如果在 Windows 平台以外的平台上使用 *qmake*，这个定义就会被忽略。你也可以使用 *qmake* 执行一个单行的条件/任务，就像这样：

```
win32:DEFINES += QT_DLL
```

比如，假设我们想在除了 Windows 平台意外的所有平台处理些什么。我们想这样使用作用域来达到这种否定效果：

```
!win32 {  
    DEFINES += QT_DLL  
}
```

`CONFIG` 行中的任何条目也都是一个作用域。比如，你这样写：

```
CONFIG += warn_on
```

你将会得到一个称作“`warn_on`”的作用域。这样将会使在不丢失特定条件下可能所需的所有自定义设置的条件下，很容易地修改项目中的配置。因为你可能把你自己的值放到 `CONFIG` 行中，这将会为你的 `makefile` 而提供给你一个非常强大的配置工具。比如：

```
CONFIG += qt warn_on debug  
debug {  
    TARGET = myappdebug  
}  
release {  
    TARGET = myapp  
}
```

在上面的代码中，两个作用域被创建，它们依赖于 `CONFIG` 行中设置的是什麼。在这个例子中，`debug` 在 `CONFIG` 行中，所以 `TARGET` 变量被设置为 `myappdebug`。如果 `release` 在 `CONFIG` 行中，那么 `TARGET` 变量将会被设置为 `myapp`。

当然也可以在处理一些设置之前检查两个事物。例如，如果你想检查平台是否是 Windows 并且线程设置是否被设定，你可以这样写：

```
win32 {  
    thread {  
        DEFINES += QT_THREAD_SUPPORT  
    }  
}
```

为了避免写出许多嵌套作用域，你可以这样使用冒号来嵌套作用域：

```
win32:thread {  
    DEFINES += QT_THREAD_SUPPORT  
}
```

一旦一个测试被执行，你也许也要做 `else/elseif` 操作。这种情况下，你可以很容易地写出复杂的测试。这需要使用特殊的“`else`”作用域，它可以和其它作用域进行组合（也可以向上面一样使用冒号），比如：

```
win32:thread {
    DEFINES += QT_THREAD_SUPPORT
} else:debug {
    DEFINES += QT_NOTHREAD_DEBUG
} else {
    warning("Unknown configuration")
}
```

## 4 变量

到目前为止我们遇到的变量都是系统变量，比如 *DEFINES*、*SOURCES* 和 *HEADERS*。你也可以为你自己创建自己的变量，这样你就可以在作用域中使用它们了。创建自己的变量很容易，只要命名它并且分配一些东西给它。比如：

```
MY_VARIABLE = value
```

现在你对你自己的变量做什么是没有限制的，同样地，*qmake* 将会忽略它们，除非需要在一个作用域中考虑它们。

你也可以通过在其它任何一个变量的变量名前加 `$$` 来把这个变量的值分配给当前的变量。例如：

```
MY_DEFINES = $$DEFINES
```

现在 **MY\_DEFINES** 变量包含了项目文件在这点时 **DEFINES** 变量的值。这也和下面的语句一样：

```
MY_DEFINES = ${DEFINES}
```

第二种方法允许你把一个变量和其它变量连接起来，而不用使用空格。*qmake* 将允许一个变量包含任何东西（包括 `$(VALUE)`），可以直接在 `makefile` 中直接放置，并且允许它适当地扩张，通常是一个环境变量）。无论如何，如果你需要立即设置一个环境变量，然后你就可以使用 `$$()` 方法。比如：

```
MY_DEFINES = $$ (ENV_DEFINES)
```

这将会设置 **MY\_DEFINES** 为环境变量 **ENV\_DEFINES** 传递给 `.pro` 文件的值。另外你可以在替换的变量里调用内置函数。这些函数（不会和下一节中列举的测试函数混淆）列出如下：

## **join( variablename, glue, before, after )**

这将会在 *variablename* 的各个值中间加入 *glue*。如果这个变量的值为非空，那么就会在值的前面加一个前缀 *before* 和一个后缀 *after*。只有 *variablename* 是必须的字段，其它默认情况下为空串。如果你需要在 *glue*、*before* 或者 *after* 中使用空格的话，你必须提供它们。

## **member( variablename, position )**

这将会放置 *variablename* 的列表中的 *position* 位置的值。如果 *variablename* 不够长，这将会返回一个空串。*variablename* 是唯一必须的字段，如果没有指定位置，则默认为列表中的第一个值。

## **find( variablename, substr )**

这将会放置 *variablename* 中所有匹配 *substr* 的值。*substr* 也可以是正则表达式，而因此将被匹配。

```
MY_VAR = one two three four
MY_VAR2 = $$join(MY_VAR, " -L", -L) -Lfive
MY_VAR3 = $$member(MY_VAR, 2) $$find(MY_VAR, t.*)
```

MY\_VAR2 将会包含“-Lone -Ltwo -Lthree -Lfour -Lfive”，并且 MYVAR3 将会包含“three two three”。

## **system( program\_and\_args )**

这将会返回程序执行在标准输出/标准错误输出的内容，并且正像平时所期待地分析它。比如你可以使用这个来询问有关平台的信息。

```
UNAME = $$system(uname -s)
contains( UNAME, [lL]linux ):message( This looks like Linux ($$UNAME) to me )
```

# **5 测试函数**

*qmake* 提供了可以简单执行，但强大测试的内置函数。这些测试也可以用在作用域中（就像上面一样），在一些情况下，忽略它的测试值，它自己使用测试函数是很有用的。

## **contains( variablename, value )**

如果 *value* 存在于一个被叫做 *variablename* 的变量的值的列表中，那么这个作用域中的设置将会被处理。例如：

```
contains( CONFIG, thread ) {
    DEFINES += QT_THREAD_SUPPORT
}
```

如果 *thread* 存在于 *CONFIG* 变量的值的列表中时，那么 *QT\_THREAD\_SUPPORT* 将会被加入到 *DEFINES* 变量的值的列表中。

### **count( variablename, number )**

如果 *number* 与一个被叫做 *variablename* 的变量的值的数量一致，那么这个作用域中的设置将会被处理。例如：

```
count( DEFINES, 5 ) {
    CONFIG += debug
}
```

### **error( string )**

这个函数输出所给定的字符串，然后会使 *qmake* 退出。例如：

```
error( "An error has occurred" )
```

文本“An error has occurred”将会被显示在控制台上并且 *qmake* 将会退出。

### **exists( filename )**

如果指定文件存在，那么这个作用域中的设置将会被处理。例如：

```
exists( /local/qt/qmake/main.cpp ) {
    SOURCES += main.cpp
}
```

如果 */local/qt/qmake/main.cpp* 存在，那么 *main.cpp* 将会被添加到源文件列表中。

注意可以不用考虑平台使用“/”作为目录的分隔符。

### **include( filename )**

项目文件在这一点时包含这个文件名的内容，所以指定文件中的任何设置都将会被处理。例如：

```
include( myotherapp.pro )
```

*myotherapp.pro* 项目文件中的任何设置现在都会被处理。

### **isEmpty( variablename )**

这和使用 `count( variablename, 0 )` 是一样的。如果叫做 *variablename* 的变量没有任何元素，那么这个作用域中的设置将会被处理。例如：

```
isEmpty( CONFIG ) {  
    CONFIG += qt warn_on debug  
}
```

### **message( string )**

这个函数只是简单地在控制台上输出消息。

```
message( "This is a message" )
```

文本“*This is a message*”被输出到控制台上并且对于项目文件的处理将会继续进行。

### **system( command )**

特定指令被执行并且如果它返回一个 1 的退出值，那么这个作用域中的设置将会被处理。例如：

```
system( ls /bin ) {  
    SOURCES += bin/main.cpp  
    HEADERS += bin/main.h  
}
```

所以如果命令 *ls /bin* 返回 1，那么 *bin/main.cpp* 将被添加到源文件列表中并且 *bin/main.h* 将被添加到头文件列表中。

### **infile( filename, var, val )**

如果 *filename* 文件（当它被 *qmake* 自己解析时）包含一个值为 *val* 的变量 *var*，那么这个函数将会返回成功。你也可以不传递第三个参数（*val*），这时函数将只测试文件中是否分配有这样一个变量 *var*。

# Qt 对象模型

标准的 C++ 对象模型为对象范例提供了十分有效的运行时刻支持。但是这种 C++ 对象模型的静态性质在一定的领域是不够灵活的。图形用户界面编程就是一个同时需要运行时刻的效率和高水平的灵活性的领域。Qt 通过结合 C++ 的速度为这一领域提供了 Qt 对象模型的灵活性。

Qt 把下面这些特性添加到了 C++ 当中：

- 一种关于无缝对象通讯被称为 **信号和槽** 的非常强大的机制，
- 可查询和可设计的 **属性**，
- 强大的 **事件和事件过滤器**，
- 根据上下文进行 **国际化的字符串翻译**，
- 完善的时间间隔驱动的 **计时器** 使得在一个事件驱动的图形界面程序中很好地集成许多任务成为可能。
- 以一种自然的方式组织对象所有权的分层次和可查询的 **对象树**。
- 被守护的指针，**QGuardedPtr**，当参考对象被破坏时，可以自动地设置为无效，不像正常的 C++ 指针在它们的对象被破坏的时候变成了“摇摆指针”。

许多 Qt 的特性是基于 **QObject** 的继承，通过标准 C++ 技术实现的。其他的，比如对象通讯机制和虚拟属性系统，都需要 Qt 自己的 **元对象编译器(moc)** 提供的 **元对象系统**。

元对象系统是一种可以使语言更加适用于真正的组件图形用户界面程序的 C++ 扩展。尽管模板也可以用来扩展 C++，元对象系统提供给标准 C++ 而模板所不能提供的益处，请看 **为什么 Qt 不用模板来实现信号和槽？**。

## 对象树和对象所有权

**QObject** 在对象树中组织它们自己。当你以另外一个对象作为父对象来创建一个 **QObject** 时，它就被添加到父对象的 **children()** 列表中，并且当父对象被删除的时候，它也会被删除。这种机制很好的适合了图形用户界面应用对象的需要。例如，一个 **QAccel** (键盘快捷键) 是相关窗口的子对象，当用户关闭该窗口的时候，这个快捷键也被删除了。

静态函数 **QObject::objectTrees()** 提供了访问当前存在的所有跟对象的方法。

**QWidget**，在屏幕上显示的任何东西的基类，扩展着父-子对象关系。一个子对象通常就是一个子窗口部件，也就是说，它被显示在父对象的坐标系统中并且在图象上由父对象的边界夹住。例如，当一个应用程序在一个消息框被关闭之后删除这个消息框时，消息框的按钮和标签正如我们所想要的也被删除了，因为这些按钮和标签都是消息框的子对象。

你也可以自己删除子对象，这样它们就会把它们自己从它们的父对象中移除。例如，当用户移除工具条可以导致应用程序删除它的一个 **QToolBar** 对象，在这种情况下工具条的 **QMainWindow** 父对象会检测到这种变化并因此而重新构成屏幕空间。

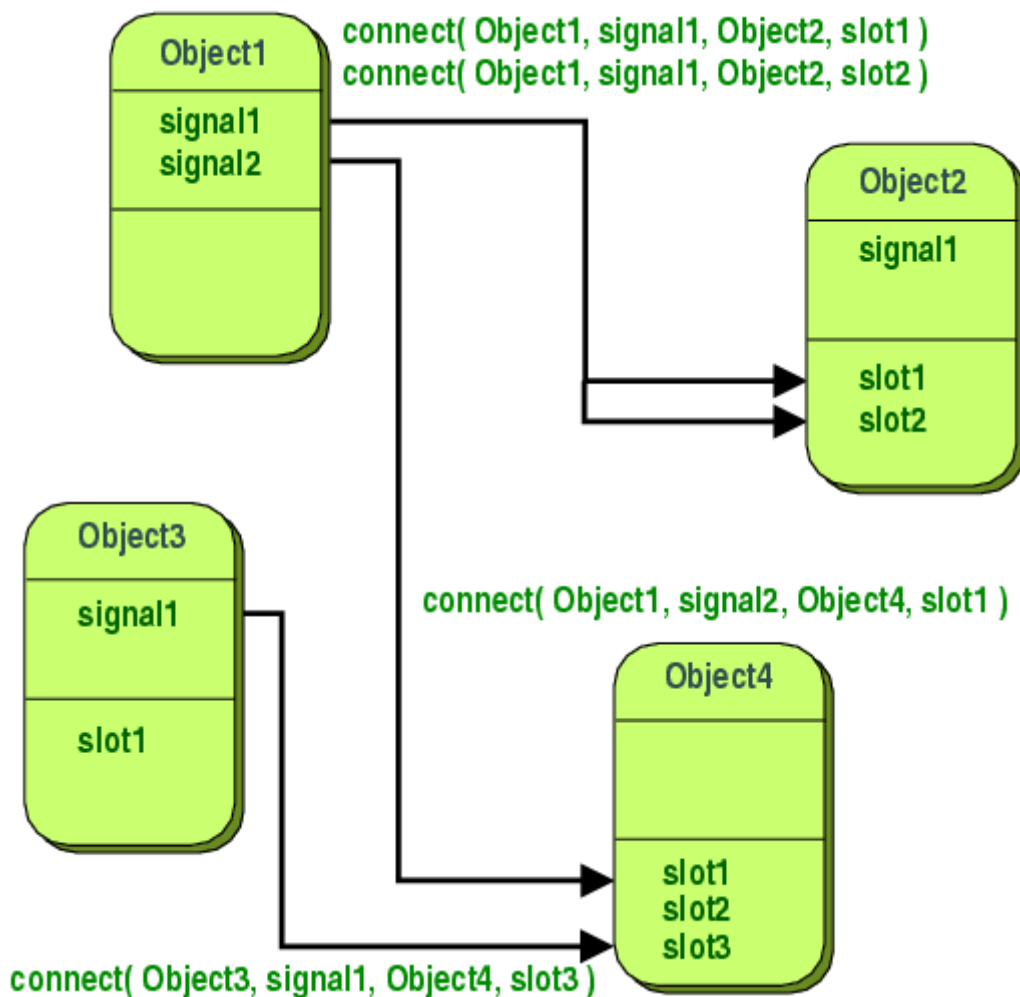
当一个应用程序看起来或者运行起来有些奇怪的时候，调试函数 `QObject::dumpObjectTree()` 和 `QObject::dumpObjectInfo()` 是经常有用的。

## 信号和槽

信号和槽用于对象间的通讯。信号/槽机制是 Qt 的一个中心特征并且也许是 Qt 与其它工具包的最不相同的部分。

在图形用户界面编程中，我们经常希望一个窗口部件的一个变化被通知给另一个窗口部件。更一般地，我们希望任何一类的对象可以和其它对象进行通讯。例如，如果我们正在解析一个 XML 文件，当我们遇到一个新的标签时，我们也许希望通知列表视图我们正在用来表达 XML 文件的结构。

较老的工具包使用一种被称作回调的通讯方式来实现同一目的。回调是指一个函数的指针，所以如果你希望一个处理函数通知你一些事件，你可以把另一个函数（回调）的指针传递给处理函数。处理函数在适当的时候调用回调。回调有两个主要缺点。首先他们不是类型安全的。我们从来都不能确定处理函数使用了正确的参数来调用回调。其次回调和处理函数是非常强有力地联系在一起，因为处理函数必须知道要调用哪个回调。

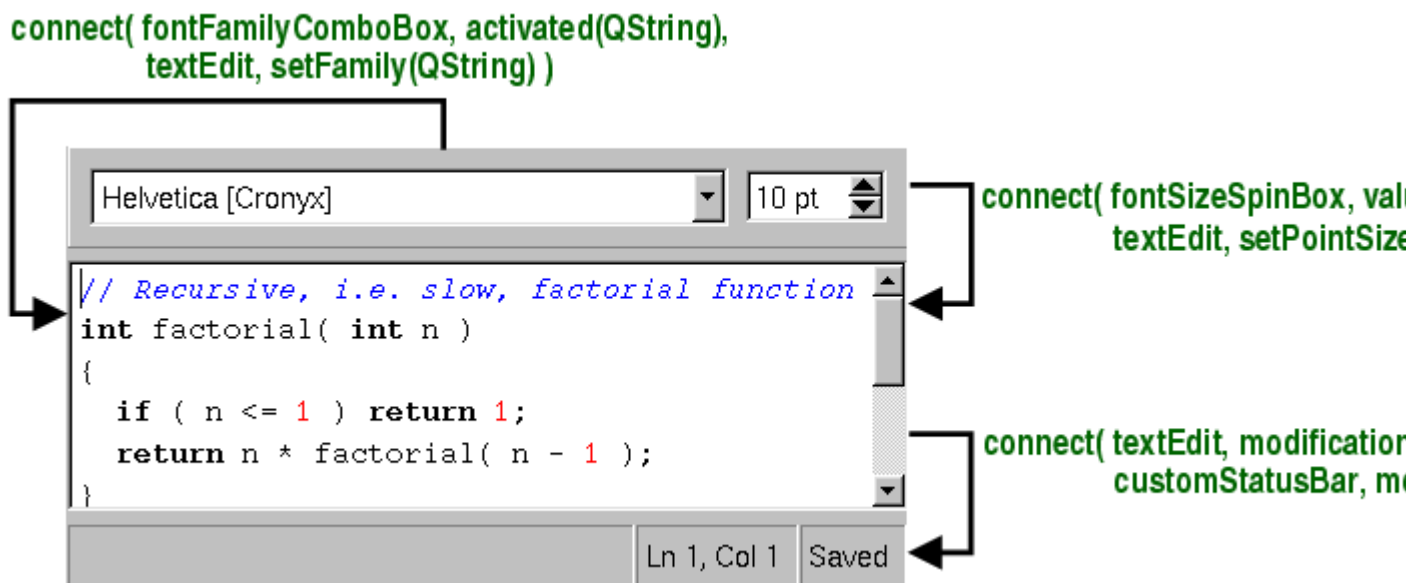


## 一个关于一些信号和槽连接的摘要图

在 Qt 中我们有一种可以替代回调的技术。我们使用信号和槽。当一个特定事件发生的时候，一个信号被发射。Qt 的窗口部件有很多预定义的信号，但是我们总是可以通过继承来加入我们自己的信号。槽就是一个可以被调用处理特定信号的函数。Qt 的窗口部件又很多预定义的槽，但是通常的习惯是你加入自己的槽，这样你就可以处理你所感兴趣的信号。

信号和槽的机制是类型安全的：一个信号的签名必须与它的接收槽的签名相匹配。（实际上一个槽的签名可以比它接收的信号签名少，因为它可以忽略额外的签名。）因为签名是一致的，编译器就可以帮助我们检测类型不匹配。信号和槽是宽松地联系在一起的：一个发射信号的类不用知道也不用注意哪个槽要接收这个信号。Qt 的信号和槽的机制可以保证如果你把一个信号和一个槽连接起来，槽会在正确的时间使用信号的参数而被调用。信号和槽可以使用任何数量、任何类型的参数。它们是完全类型安全的：不会再有回调核心转储(core dump)。

从 `QObject` 类或者它的一个子类（比如 `QWidget` 类）继承的所有类可以包含信号和槽。当对象改变它们的状态的时候，信号被发送，从某种意义上讲，它们也许对外面的世界感兴趣。这就是所有的对象通讯时所做的一切。它不知道也不注意无论有没有东西接收它所发射的信号。这就是真正的信息封装，并确保对象可以用作一个软件组件。



## 一个信号和槽连接的例子

槽可以用来接收信号，但它们是正常的成员函数。一个槽不知道它是否被任意信号连接。此外，对象不知道关于这种通讯机制和能够被用作一个真正的软件组件。

你可以把许多信号和你所希望的单一槽相连，并且一个信号也可以和你所期望的许多槽相连。把一个信号和另一个信号直接相连也是可以的。（这时，只要第一个信号被发射时，第二个信号立刻就被发射。）

总体来看，信号和槽构成了一个强有力的组件编程机制。



## 一个小例子

一个最小的 C++ 类声明如下：

```
class Foo
{
public:
    Foo();
    int value() const { return val; }
    void setValue( int );
private:
    int val;
};
```

一个小的 Qt 类如下：

```
class Foo : public QObject
{
    Q_OBJECT
public:
    Foo();
    int value() const { return val; }
public slots:
    void setValue( int );
signals:
    void valueChanged( int );
private:
    int val;
};
```

这个类有同样的内部状态，和公有方法来访问状态，但是另外它也支持使用信号和槽的组件编程：这个类可以通过发射一个信号，`valueChanged()`，来告诉外面的世界它的状态发生了变化，并且它有一个槽，其它对象可以发送信号给这个槽。

所有包含信号和/或者槽的类必须在它们的声明中提到 `Q_OBJECT`。

槽可以由应用程序的编写者来实现。这里是 `Foo::setValue()` 的一个可能的实现：

```
void Foo::setValue( int v )
{
    if ( v != val ) {
        val = v;
        emit valueChanged(v);
    }
}
```

emit valueChanged(v) 这一行从对象中发射 valueChanged 信号。正如你所能看到的，你通过使用 emit signal(arguments) 来发射信号。

下面是把两个对象连接在一起的一种方法：

```
Foo a, b;
connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));
b.setValue( 11 ); // a == undefined  b == 11
a.setValue( 79 ); // a == 79          b == 79
b.value();
```

调用 a.setValue(79) 会使 a 发射一个 valueChanged() 信号，b 将会在它的 setValue() 槽中接收这个信号，也就是 b.setValue(79) 被调用。接下来 b 会发射同样的 valueChanged() 信号，但是因为没有槽被连接到 b 的 valueChanged() 信号，所以没有发生任何事（信号消失了）。

注意只有当  $v \neq val$  的时候 setValue() 函数才会设置这个值并且发射信号。这样就避免了在循环连接的情况下（比如 b.valueChanged() 和 a.setValue() 连接在一起）出现无休止的循环的情况。

这个例子说明了对象之间可以在互相不知道的情况下一起工作，只要在最开始的时在它们中间建立连接。

预处理程序改变或者移除了 signals、slots 和 emit 这些关键字，这样就可以使用标准的 C++ 编译器。

在一个定义有信号和槽的类上运行 moc。这样就会生成一个可以和其它对象文件编译和连接成引用程序的 C++ 源文件。

## 信号

当对象的内部状态发生改变，信号就被发射，在某些方面对于对象代理或者所有者也许是很有趣的。只有定义了一个信号的类和它的子类才能发射这个信号。

例如，一个列表框同时发射 highlighted() 和 activated() 这两个信号。绝大多数对象也许只对 activated() 这个信号感兴趣，但是有时想知道列表框中的哪个条目在当前是高亮的。如果两个不同的类对同一个信号感兴趣，你可以把这个信号和这两个对象连接起来。

当一个信号被发射，它所连接的槽会被立即执行，就像一个普通函数调用一样。信号/槽机制完全不依赖于任何一种图形用户界面的事件回路。当所有的槽都返回后 emit 也将返回。

如果几个槽被连接到一个信号，当信号被发射时，这些槽就会被按任意顺序一个接一个地执行。

信号会由 moc 自动生成并且一定不要在 .cpp 文件中实现。它们也不能有任何返回类型（比如使用 void）。

关于参数需要注意。我们的经验显示如果信号和槽不使用特殊的类型，它们都可以多次使用。如果 `QScrollBar::valueChanged()` 使用了一个特殊的类型，比如 `hypothetical QRangeControl::Range`，它就只能被连接到被设计成可以处理 `QRangeControl` 的槽。简单的和 [教程 1 的第 5 部分](#) 一样的程序将是不可能的。

## 槽

当一个和槽连接的信号被发射的时候，这个槽被调用。槽也是普通的 C++ 函数并且可以像它们一样被调用；它们唯一的特点就是它们可以被信号连接。槽的参数不能含有默认值，并且和信号一样，为了槽的参数而使用自己特定的类型是很不明智的。

因为槽就是普通成员函数，但却有一点非常有意思的东西，它们也和普通成员函数一样有访问权限。一个槽的访问权限决定了谁可以和它相连：

一个 `public slots:` 区包含了任何信号都可以相连的槽。这对于组件编程来说非常有用：你生成了许多对象，它们互相并不知道，把它们的信号和槽连接起来，这样信息就可以正确地传递，并且就像一个铁路模型，把它打开然后让它跑起来。

一个 `protected slots:` 区包含了之后这个类和它的子类的信号才能连接的槽。这就是说这些槽只是类的实现的一部分，而不是它和外界的接口。

一个 `private slots:` 区包含了之后这个类本身的信号可以连接的槽。这就是说它和这个类是非常紧密的，甚至它的子类都没有获得连接权利这样的信任。

你也可以把槽定义为虚的，这在实践中被发现也是非常有用的。

信号和槽的机制是非常有效的，但是它不像“真正的”回调那样快。信号和槽稍微有些慢，这是因为它们所提供的灵活性，尽管在实际应用中这些不同可以被忽略。通常，发射一个和槽相连的信号，大约只比直接调用那些非虚函数调用的接收器慢十倍。这是定位连接对象所需的开销，可以安全地重复所有地连接（例如在发射期间检查并发接收器是否被破坏）并且可以按一般的方式安排任何参数。当十个非虚函数调用听起来很多时，举个例子来说，时间开销只不过比任何一个“`new`”或者“`delete`”操作要少些。当你执行一个字符串、矢量或者列表操作时，需要“`new`”或者“`delete`”，信号和槽仅仅对一个完整函数调用地时间开销中的一个非常小的部分负责。无论何时你在一个槽中使用一个系统调用和间接地调用超过十个函数的时间是相同的。在一台 i585-500 机器上，你每秒钟可以发射 2, 000, 000 个左右连接到一个接收器上的信号，或者发射 1, 200, 000 个左右连接到两个接收器的信号。信号和槽机制的简单性和灵活性对于时间的开销来说是非常值得的，你的用户甚至察觉不出来。

## 元对象信息

[元对象](#) 编译器（moc）解析一个 C++ 文件中的类声明并且生成初始化元对象的 C++ 代码。元对象包括所有信号和槽函数的名称，还有这些函数的指针。（要获得更多的信息，请看 [为什么 Qt 不用模板来实现信号和槽？](#)）

元对象包括一些额外的信息，比如对象的 [类名称](#)。你也可以检查一个对象是否 [继承](#) 了一个特定的类，比如：

```
if ( widget->inherits("QPushButton") ) {  
    // 是的，它是一个 Push Button、Radio Button 或者其它按钮。  
}
```

## 一个真实的例子

这是一个注释过的简单的例子（代码片断选自 [qlcdnumber.h](#)）。

```
#include "qframe.h"  
#include "qbitarray.h"  
  
class QLCDNumber : public QFrame
```

[QLCDNumber](#) 通过 [QFrame](#) 和 [QWidget](#)，还有 `#include` 这样的相关声明继承了含有绝大多数信号/槽知识的 [QObject](#)。

```
{  
    Q_OBJECT
```

`Q_OBJECT` 是由预处理器展开声明几个由 `moc` 来实现的成员函数，如果你得到了几行“`virtual function QPushButton::className not defined`”这样的编译器错误信息，你也许忘记 [运行 moc](#) 或者忘记在连接命令中包含 `moc` 输出。

```
public:  
    QLCDNumber( QWidget *parent=0, const char *name=0 );  
    QLCDNumber( uint numDigits, QWidget *parent=0, const char *name=0 );
```

它并不和 `moc` 直接相关，但是如果你继承了 [QWidget](#)，你当然想在你的构造器中获得 *parent* 和 *name* 这两个参数，而且把它们传递到父类的构造器中。

一些解析器和成员函数在这里省略掉了，`moc` 忽略了这些成员函数。

```
signals:  
    void    overflow();
```

当 [QLCDNumber](#) 被请求显示一个不可能值时，它发射一个信号。

如果你没有留意溢出，或者你认为溢出不会发生，你可以忽略 `overflow()` 信号，也就是说你可以不把它连接到任何一个槽上。

另一方面如果当数字溢出时，你想调用两个不同的错误函数，很简单地你可以把这个信号和两个不同的槽连接起来。`Qt` 将会两个都调用（按任意顺序）。

```
public slots:
```

```

void    display( int num );
void    display( double num );
void    display( const char *str );
void    setHexMode();
void    setDecMode();
void    setOctMode();
void    setBinMode();
void    smallDecimalPoint( bool );

```

一个槽就是一个接收函数，用来获得其它窗口部件状态变或的信息。`QLCDNumber` 使用它，就像上面的代码一样，来设置显示的数字。因为 `display()` 是这个类和程序的其它的部分的一个接口，所以这个槽是公有的。

几个例程把 `QScrollBar` 的 `newValue` 信号连接到 `display` 槽，所以 LCD 数字可以继续显示滚动条的值。

请注意 `display()` 被重载了，当你把一个信号和这个槽相连的时候，Qt 将会选择适当的版本。如果使用回调，你会发现五个不同的名字并且自己来跟踪类型。

一些不相关的成员函数已经从例子中省略了。

```
};
```

## 元对象系统

Qt 中的元对象系统是用来处理对象间通讯的信号/槽机制、运行时的类型信息和动态属性系统。

它基于下列三类：

1. `QObject` 类；
2. 类声明中的私有段中的 `Q_OBJECT` 宏；
3. 元对象编译器 (`moc`)。

`moc` 读取 C++ 源文件。如果它发现其中包含一个或多个类的声明中含有 `Q_OBJECT` 宏，它就会给含有 `Q_OBJECT` 宏的类生成另一个含有元对象代码的 C++ 源文件。这个生成的源文件可以被类的源文件包含 (`#include`) 到或者和这个类的实现一起编译和连接。

除了提供对象间通讯的 **信号和槽** 机制之外（介绍这个系统的主要原因），`QObject` 中的元对象代码实现其它特征：

- `className()` 函数在运行的时候以字符串返回类的名称，不需要 C++ 编译器中的本地运行类型信息 (RTTI) 的支持。
- `inherits()` 函数返回这个对象是否是一个继承于 `QObject` 继承树中一个特定类的类的实例。

- `tr()`和 `trUtf8()` 两个函数是用于 国际化中的字符串翻译。
- `setProperty()`和 `property()`两个函数是用来通过名称动态设置和获得 对象属性的。
- `metaObject()`函数返回这个类所关联的 元对象。

虽然你使用QObject作为一个基类而不使用Q\_OBJECT宏和元对象代码是可以的，但是如果Q\_OBJECT宏没有被使用，那么这里的信号和槽以及其它特征描述都不会被提供。根据元对象系统的观点，一个没有元代码的QObject的子类和它含有元对象代码的最近的祖先相同。举例来说就是，`className()`将不会返回你的类的实际名称，返回的是它的这个祖先的名称。我们强烈建议 QObject 的所有子类使用Q\_OBJECT宏，而不管它们是否实际使用了信号、槽和属性。

## 属性

Qt提供了一套和一些编译器提供商也提供的属性系统类似的完善的属性系统。然而，作为一个不依赖编译器和平台的库，Qt不能依赖像`_property`或者`[property]`那样的非标准编译器特征。我们的解决方案可以在我们支持的每一个平台上和任何标准的C++编译器一起工作。它基于元对象系统，元对象系统也通过 信号和槽提供对象通讯。

在类声明中的Q\_PROPERTY宏声明了一个属性。属性只能在继承于 QObject的子类中声明。第二个宏，Q\_OVERRIDE，可以用来覆盖一些子类中由继承得到的属性。

对于外面的世界，属性看起来和一个数据成员非常类似。然而，属性和普通的数据成员还是有一下一些不同点：

- 一个读函数。这是一直存在的。
- 一个写函数。这个是可选的：像 `QWidget::isDesktop()`这样的只读的属性就没有写函数。
- “存储”特征需要说明持续性。绝大多数属性是被存储的，但是有一小部分的虚拟属性却不用。举个例子，`QWidget::minimumWidth()`是不用存储的，因为它只是 `QWidget::minimumSize()`的一种查看，没有自己的数据。
- 一个复位函数用来把属性设置回它根据上下文的特定缺省值。这个用法还是比较罕见的，但是举个例子，`QWidget::font()`需要这个函数，因为没有调用 `QWidget::setFont()`意味着“复位到根据上下文特定的字体”。
- “可设计”特征说明它是否可以被一个图形用户界面生成器（例如 Qt设计器）设置属性。对于大多数属性都有这个特征，但不是所有，例如 `QPushButton::isDown()`。用户可以按按钮，并且应用程序设计人员可以让程序来按它自己的按钮，但是一个图形用户界面设计工具不能按按钮。

读、写和复位函数就像任何成员函数一样，继承或不继承，虚或不虚。只有一个例外就是，在多重继承的情况下，成员函数必须从第一个被继承类继承。

属性可以在不知道被使用的类的任何情况的时候通过 QObject中的一般函数进行读写。下面两个函数调用是等效的：



```
// QPushButton *b 和 QObject *o 指向同一个按钮
b->setDown( TRUE );
o->setProperty( "down", TRUE );
```

等效的是指，除了第一个函数要快一些，在编译的时候提供了更好的诊断信息。在实际应用中，第一个函数更好些。然而，因为我们可以通过 [QMetaObject](#) 获得任何一个 [QObject](#) 的所有有用属性的一个列表，[QObject::setProperty\(\)](#) 可以让你控制类中那些在编译时不可用的属性。

像 [QObject::setProperty\(\)](#) 一样，还有一个相应的 [QObject::property\(\)](#) 函数。

[QMetaObject::propertyNames\(\)](#) 返回所有可用属性的名称。[QMetaObject::property\(\)](#) 返回一个指定属性的属性数据：一个 [QMetaProperty](#) 对象。

这里有一个简单的例子说明了可以应用的绝大多数重要属性函数：

```
class MyClass : public QObject
{
    Q_OBJECT
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
};
```

这个类有一个名为“priority”的还不被 [元对象](#) 系统所知的属性。为了让这个属性被元对象系统知道，你必须用 [Q\\_PROPERTY](#) 宏来声明它。声明语法如下：

```
Q_PROPERTY( type name READ getFunction [WRITE setFunction]
            [RESET resetFunction] [DESIGNABLE bool]
            [SCRIPTABLE bool] [STORED bool] )
```

为了声明是有效的，读函数必须是常量函数并且返回值的类型是它本身或者是指向它的指针，或者是它的一个引用。可选的写函数必须返回 **void** 并且必须带有一个正确的参数，类型必须是它本身或者是指向它的指针，或者是它的一个常量引用。元对象编译器强迫这样的。

属性的类型可以是任何一个 [QVariant](#) 支持的类型或者是一个自己在类中已经定义的枚举类型。因为 [MyClass](#) 中的属性使用了枚举类型 [Priority](#)，这个类型必须也向属性系统注册。这样的话，像如下方式通过名称来设置值是可行的：

```
obj->setProperty( "priority", "VeryHigh" );
```

枚举类型必须使用 [Q\\_ENUMS](#) 宏来进行注册。这里是一个包含属性相关声明的最终类声明：

```
class MyClass : public QObject
```

```

{
    Q_OBJECT
    Q_PROPERTY( Priority priority READ priority WRITE setPriority )
    Q_ENUMS( Priority )
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
};

```

另外一个类似的宏是 `Q_SETS`。像 `Q_ENUMS` 一样，它注册了一个枚举类型，但是它额外的加了一个“set”的标记，也就是说，这个枚举数据可以被一起读或写。一个输入输出类也许有枚举数据“读”和“写”和接收“读|写”：这时最好用 `Q_SETS` 来声明一个枚举类型，而不是 `Q_ENUMS`。

`Q_PROPERTY` 段剩余的关键字是 `RESET`、`DESIGNABLE`、`SCRIPTABLE` 和 `STORED`。

`RESET` 指定一个函数可以设置属性到缺省状态（这个缺省状态可能和初始状态不同）。这个函数必须返回 `void` 并且不带有参数。

`DESIGNABLE` 声明这个属性是否适合被一个图形用户界面设计工具修改。缺省的 `TRUE` 是说这个属性可写，否则就是 `FALSE` 说明不能。你可以定义一个布尔成员函数来替代 `TRUE` 或 `FALSE`。

`SCRIPTABLE` 声明这个属性是否适合被一个脚本引擎访问。缺省是 `TRUE`，可以。你可以定义一个布尔成员函数来替代 `TRUE` 或 `FALSE`。

`STORED` 声明这个属性的值是否必须作为一个存储的对象状态而被记得。`STORED` 只对可写的属性有意义。缺省是 `TRUE`。技术上多余的属性（比如，如果 `QRect` 的 `geometry` 已经是一个属性了的 `QPoint` 的 `pos`）定义为 `FALSE`。

连接到属性系统是一个附加宏，“`Q_CLASSINFO`”，它可以用来把名称/值这样一套的属性添加到一个类的元对象中，例如：

```
Q_CLASSINFO( "Version", "3.0.0" )
```

和其它元数据一样，类信息在运行时是可以通过元对象访问的，具体请看 [QMetaObject::classInfo\(\)](#)。

## 使用元对象编译器

元对象编译器，朋友中的 `moc`，是处理 Qt 的 C++ 扩展的程序。



元对象编译器读取一个C++源文件。如果它发现其中的一个或多个类的声明中含有 **Q\_OBJECT** 宏，它就会给这个使用**Q\_OBJECT**宏的类生成另外一个包含 **元对象**代码的C++源文件。尤其是，元对象代码对信号/槽机制、运行时类型信息和动态属性系统是需要。

一个被元对象编译器生成的 C++源文件必须和这个类的实现一起被编译和连接（或者它被包含到（**#include**）这个类的源文件中）。

如果你是用 **qmake**来生成你的**Makefile**文件，当需要的时候，编译规则中需要包含调用元对象编译器，所以你不需要直接使用元对象编译器。关于元对象编译器的更多的背景知识，请看 **为什么Qt不用模板来实现信号和槽？**。

## 用法

元对象编译器很典型地和包含下面这样情况地类声明地输入文件一起使用：

```
class MyClass : public QObject
{
    Q_OBJECT
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

signals:
    void mySignal();

public slots:
    void mySlot();

};
```

除了上述提到地信号和槽，元对象编译器在下一个例子中还将实现对象属性。**Q\_PROPERTY**宏声明了一个对象属性，而**Q\_ENUMS** 声明在这个类中的 **属性系统**中可用的枚举类型的一个列表。在这种特殊的情况下，我们声明了一个枚举类型属性**Priority**，也被称为“**priority**”，并且读函数为**priority()**，写函数为**setPriority()**。

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_PROPERTY( Priority priority READ priority WRITE setPriority )
    Q_ENUMS( Priority )
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
```

```
};
```

属性可以通过 `Q_OVERRIDE` 宏在子类中进行修改。`Q_SETS` 宏声明了枚举变量可以进行组合操作，也就是说可以一起读或写。另外一个宏，`Q_CLASSINFO`，用来给类的元对象添加名称/值这样一组数据：

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_CLASSINFO( "Author", "Oscar Peterson")
    Q_CLASSINFO( "Status", "Very nice class")
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();
};
```

这三个概念：信号和槽、属性和元对象数据是可以组合在一起的。

元对象编译器生成的输出文件必须被编译和连接，就像你的程序中的其它的 C++ 代码一样；否则你的程序的连编将会在最后的连接阶段失败。出于习惯，这种操作是用下述两种方式之一解决的：

#### 方法一：类的声明放在一个头文件（.h 文件）中

如果在上述的文件 `myclass.h` 中发现类的声明，元对象编译器的输出文件将会被放在一个叫 `moc_myclass.cpp` 的文件中。这个文件将会像通常情况一样被编译，作为对象文件的结果是 `moc_myclass.o`（在 Unix 下）或者 `moc_myclass.obj`（在 Windows 下）。这个对象接着将会被包含到一个对象文件列表中，它们将会在程序的最后连编阶段被连接在一起。

#### 方法二：类的声明放在一个实现文件（.cpp 文件）中

如果上述的文件 `myclass.cpp` 中发现类的声明，元对象编译器的输出文件将会被放在一个叫 `myclass.moc` 的文件中。这个文件需要被实现文件包含（`#include`），也就是说 `myclass.cpp` 需要包含下面这行

```
#include "myclass.moc"
```

放在所有的代码之后。这样，元对象编译器生成的代码将会和 `myclass.cpp` 中普通的类定义一起被编译和连接，所以方法一中的分别编译和连接就是不需要的了。

方法一是常规的方法。方法二用在你想让实现文件自包含，或者 `Q_OBJECT` 类是内部实现的并且在头文件中不可见的这些情况下使用。

## Makefile 中自动使用元对象编译器的方法

除了最简单的测试程序之外的任何程序，建议自动使用元对象编译器。在你的程序的 `Makefile` 文件中加入一些规则，`make` 就会在需要的时候运行元对象编译器和处理元对象编译器的输出。

我们建议使用 Trolltech 的自由 makefile 生成工具，[qmake](#)，来生成你的 `Makefile`。这个工具可以识别方法一和方法二风格的源文件，并建立一个可以做所有必要的元对象编译操作的 `Makefile`。

另一方面如果，你想自己建立你的 **Makefile**，下面是如何包含元对象编译操作的一些提示。

对于在头文件中声明了 **Q\_OBJECT** 宏的类，如果你只使用 **GNU** 的 **make** 的话，这是一个很有用的 **makefile** 规则：

```
moc_%.cpp: %.h
    moc $< -o $@
```

如果你想更方便地写 **makefile**，你可以按下面的格式写单独的规则：

```
moc_NAME.cpp: NAME.h
    moc $< -o $@
```

你必须记住要把 *moc\_NAME.cpp* 添加到你的 **SOURCES**（你可以用你喜欢的名字替代）变量中并且把 *moc\_NAME.o* 或者 *moc\_NAME.obj* 添加到你的 **OBJECTS** 变量中。

（当我们给我们的 C++ 源文件命名为 **.cpp** 时，元对象编译器并不留意，所以只要你喜欢，你可以使用 **.C**、**.cc**、**.CC**、**.cxx** 或者甚至 **.c++**。）

对于在实现文件（**.cpp** 文件）中声明 **Q\_OBJECT** 的类，我们建议你使用下面这样的 **makefile** 规则：

```
NAME.o: NAME.moc

NAME.moc: NAME.cpp
    moc -i $< -o $@
```

这将会保证 **make** 程序会在编译 *NAME.cpp* 之前运行元对象编译器。然后你可以把

```
#include "NAME.moc"
```

放在 *NAME.cpp* 的末尾，这样在这个文件中的所有的类声明被完全地知道。

## 调用元对象编译器 **moc**

这里是元对象编译器 **moc** 所支持地命令行选项：

**-o file**

将输出写到 *file* 而不是标准输出。

**-f**

强制在输出文件中生成 **#include** 声明。文件的名称必须符合 [正则表达式](#) `\.[hH][^.]*`（也就是说扩展名必须以 **H** 或 **h** 开始）。这个选项只有在你的头文件没有遵循标准命名法则的时候才有用。

**-i**

不在输出文件中生成 **#include** 声明。当一个 C++ 文件包含一个或多个类声明的时候你也许应该这样使用元对象编译器。然后你应该在 **.cpp** 文件中包含（**#include**）元对象代码。如果 **-i** 和 **-f** 两个参数都出现，后出现的有效。

**-nw**

不产生任何警告。不建议使用。

-ldbg

把大量的 lex 调试信息写到标准输出。

-p *path*

使元对象编译器生成的（如果有生成的）`#include` 声明的文件名称中预先考虑到 *path/*。

-q *path*

使元对象编译器在生成的文件中的 `qt #include` 文件的名称中预先考虑到 *path/*。

你可以明确地告诉元对象编译器不要解析头文件中的成分。它可以识别包含子字符串 `MOC_SKIP_BEGIN` 或者 `MOC_SKIP_END` 的任何 C++ 注释（`//`）。它们正如你所期望的那样工作并且你可以把它们划分为若干层次。元对象编译器所看到的最终结果就好像你把一个 `MOC_SKIP_BEGIN` 和一个 `MOC_SKIP_END` 当中的所有行删除那样。

## 诊断

元对象编译器将会警告关于学多在 `Q_OBJECT` 类声明中危险的或者不合法的构造。

如果你在你的程序的最后连编阶段得到连接错误，说 `YourClass::className()` 是未定义的或者 `YourClass` 缺乏 `vtbl`，某样东西已经被做错。绝大多数情况下，你忘记了编译或者 `#include` 元对象编译器产生的 C++ 代码，或者（在前面的情况下）没有在连接命令中包含那个对象文件。

## 限制

元对象编译器并不展开 `#include` 或者 `#define`，它简单地忽略它所遇到的所有预处理程序指示。这是遗憾的，但是在实践中它通常情况下不是问题。

元对象编译器不处理所有的 C++。主要的问题是类模板不能含有信号和槽。这里是一个例子：

```
class SomeTemplate<int> : public QFrame {
    Q_OBJECT
    ...
signals:
    void bugInMocDetected( int );
};
```

次重要的是，后面的构造是不合法的。所有的这些都可以替换为我们通常认为比较好的方案，所以去掉这些限制对于我们来说并不是高优先级的。

## 多重继承需要把 `QObject` 放在第一个

如果你使用多重继承，元对象编译器假设首先继承的类是 `QObject` 的一个子类。也就是说，确信仅仅首先继承的类是 `QObject`。

```
class SomeClass : public QObject, public OtherClass {
    ...
};
```

（这个限制几乎是不可能去掉的；因为元对象编译器并不展开`#include` 或者`#define`，它不能发现基类中哪个是 `QObject`。）

## 函数指针不能作为信号和槽的参数

在你考虑使用函数指针作为信号/槽的参数的大多数情况下，我们认为继承是一个不错的替代方法。这里是一个不合法的语法的例子：

```
class SomeClass : public QObject {
    Q_OBJECT
    ...
public slots:
    // 不合法的
    void apply( void (*apply)(List *, void *), char * );
};
```

你可以在这样一个限制范围内工作：

```
typedef void (*ApplyFunctionType)( List *, void * );

class SomeClass : public QObject {
    Q_OBJECT
    ...
public slots:
    void apply( ApplyFunctionType, char * );
};
```

有时用继承和虚函数、信号和槽来替换函数指针是更好的。

## 友声明不能放在信号部分或者槽部分中

有时它也许会工作，但通常情况下，友声明不能放在信号部分或者槽部分中。把它们替换到私有的、保护的或者公有的部分中。这里是一个不合法的语法的例子：

```
class SomeClass : public QObject {
    Q_OBJECT
    ...
signals:
    friend class ClassTemplate<char>; // 错的
};
```

## 信号和槽不能被升级

把继承的成员函数升级为公有状态这一个 C++ 特征并不延伸到包括信号和槽。这里是一个不合法的例子：

```
class Whatever : public QButtonGroup {
    ...
public slots:
    void QButtonGroup::buttonPressed; // 错的
    ...
};
```

`QButtonGroup::buttonPressed()` 槽是保护的。

C++ 测验：如果你试图升级一个被重载的保护成员函数将会发生什么？

1. 所有的函数都被重载。
2. 这不是标准的 C++。

## 类型宏不能被用于信号和槽的参数

因为元对象编译器并不展开 `#define`，在信号和槽中类型宏作为一个参数是不能工作的。这里是一个不合法的例子：

```
#ifdef ultrix
#define SIGNEDNESS(a) unsigned a
#else
#define SIGNEDNESS(a) a
#endif

class Whatever : public QObject {
    ...
signals:
    void someSignal( SIGNEDNESS(int) );
    ...
};
```

不含有参数的 `#define` 将会像你所期望的那样工作。

## 嵌套类不能放在信号部分或者槽部分，也不能含有信号和槽

这里是一个例子：

```
class A {
    Q_OBJECT
```

```

public:
    class B {
        public slots:    // 错的
            void b();
        ...
    };
signals:
    class B {          // 错的
        void b();
        ...
    };
};

```

## 构造函数不能用于信号部分和槽部分

为什么一个人会把一个构造函数放到信号部分或者槽部分，这对于我们来说都是很神秘的。你无论如何也不能这样做（除去它偶尔能工作的情况）。请把它们放到私有的、保护的或者公有的部分中，它们本该属于的地方。这里是一个不合法的语法的例子：

```

class SomeClass : public QObject {
    Q_OBJECT
public slots:
    SomeClass( QObject *parent, const char *name )
        : QObject( parent, name ) { } // 错的
    ...
};

```

## 属性的声明应该放在含有相应的读写函数的公有部分之前

在包含相应的读写函数的公有部分之中和之后声明属性的话，读写函数就不能像所期望的那样工作了。元对象编译器会抱怨不能找到函数或者解析这个类型。这里是一个不合法的语法的例子：

```

class SomeClass : public QObject {
    Q_OBJECT
public:
    ...
    Q_PROPERTY( Priority priority READ priority WRITE setPriority ) // 错的
    Q_ENUMS( Priority ) // 错的
    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
    ...
};

```

根据这个限制，你应该在 `Q_OBJECT` 之后，在这个类的声明之前声明所有的属性：



```

class SomeClass : public QObject {
    Q_OBJECT
    Q_PROPERTY( Priority priority READ priority WRITE setPriority )
    Q_ENUMS( Priority )
public:
    ...
    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
    ...
};

```

## 为什么 Qt 不用模板来实现信号和槽？

一个简单的答案是，当初 Qt 被设计的时候，因为各种各样的编译器的不充分，所以在多平台应用程序中完全使用模板机制是不可能的。甚至今天，许多被广泛使用的 C++ 编译器在使用高级模板的时候还是有问题的。例如，你不能安全地依靠部分模板的示例，这对一些不平常的问题领域是必要的。因此 Qt 中模板的用法不得不保守。记住 Qt 是一个多平台的工具包，在 Linux/g++ 平台上的进步不一定能够在其它情况下获得改进。

那些在模板执行上比较弱的编译器终将得到改进。但是，即使我们所有的用户以极好的模板支持接近一个完全现代的 C++ 编译器的标准，我们也不会放弃通过使用我们的 [元对象编译器](#) 的基于字符串的途径。这里是为什么这样做的五个原因：

### 1. 语法问题

语法不是糖：我们用来表达我们的运算法则的语法较大程度上影响我们的代码的可读性和可维护性。Qt 中信号和槽所用的语法在实践中被证明是非常成功的。这种语法是直观的、容易使用的和容易读的。人们在学习 Qt 时发现这种语法帮助他们理解和使用信号和槽的概念——而不管它们的高度抽象和通用的性质。此外，在类定义中信号声明保证了信号就像被保护的 C++ 成员函数一样被保护。这帮助了程序员在刚开始的时候就获得了他们的设计权力，而不用不得不考虑设计模式。

### 2. 预编译程序是好的

Qt 的 [moc](#)（元对象编译器）提供了一种方式除了那些编译语言的工具。它可以生成任何一个标准的 C++ 编译器都能编译的额外的 C++ 代码。元对象编译器读取 C++ 源文件。如果它发现其中有一个或多个类的声明中含有“Q\_OBJECT”这个宏，它就会为这些类生成另外一个包含元对象代码的 C++ 源文件。这个由元对象编译器生成的 C++ 源文件必须和它的类实现一起编译和连接（或者它也可以被 `#included` 到个类的源文件中）。有特色的是元对象编译器不是用手工来调用的，它可以自动地被连编系统调用，所以它不需要程序员额外的付出努力。



这里有一些其它的预编译程序，比如，`rpc` 和 `idl`，它们使程序或者对象能够通过进程或者 `machine boundaries` 来进行通讯。预编译程序的选择是编写编译器，专有的语言或者使用对话框或向导这些图形编程工具来生成晦涩的代码。我们能使我们的客户使用他们所喜欢的工具，而不是把他们锁定在一个专有的 C++ 编译器或者一个特殊的集成开发环境。我们不强迫程序员把生成的代码添加到源程序仓库中，而是鼓励他们把我们的工具加入到他们的连编系统中：更加干净，更加安全和更加富有 UNIX 精神。

### 3. 灵活性为王

C++ 是一种标准化的、强大的和精心制作的多用途语言。它只是用来开发很多领域的软件项目的一种语言，生成许多种应用程序，从整个操作系统、数据库服务器和高性能的图形应用程序到普通的桌面应用程序。C++ 成功的关键之一是它着重于最大效能和最小内存占用同时保持 ANSI-C 的兼容性的可伸缩语言设计

在这些优势当中，也有一些不利方面。对于 C++，当它用来构成基于组件的图形用户界面编程的时候，静态的对象模型在使用 Objective C 途径的动态消息机制方面是明显的劣势。对于一个高端数据库服务器或者一个操作系统使用正确的图形用户界面前端工具的这一设计选择不是必须的。使用元对象编译器，我们可以把这一劣势转化为优势并且会加入当我们遇到安全的和有效的图形用户界面程序编程这一挑战的时候所需要的灵活性。

我们的方法比你用模板所能做到的一切更好。比如，我们有对象属性。并且我们可以重载信号和槽，当你在使用可以重载这一关键理念的语言进行程序设计的时候你会感到很自然。我们的信号只对一个类实例的大小增加了零个字节，也就是说我们能在不破坏二进制程序的兼容性的同时加入新的信号。因为我们不像模板那样过多地依靠内嵌，我们可以使得代码变得更小。添加一个新的连接就是增加一个简单地函数调用而不是一个复杂地模板函数。

另外一个好处就是我们可以在运行时探测对象的信号和槽。我们可以通过使用类型安全的名称调用而不用我们知道我们要连接的对象的确切类型来建立连接。这在一个基于模板的解决方案中是不可能的。这种运行时的自我检测扩充了一种新的功能，比如我们可以使用 Qt 设计器的 XML 格式的 `ui` 文件来生成和连接图形用户界面。

### 4. 调用性能不是一切

Qt 的信号和槽的执行没有基于模板的解决方案那样快。发射一个信号的时间大约和普通模板实现中的四个普通函数调用的时间差不多，Qt 要求努力控制到和十个普通函数调用差不多。这也不必惊讶，因为 Qt 机制中包括了一个通用调度器，自我测量和基本的脚本化的能力。它不过分依赖内嵌和代码扩展，并且它提供了运行时得无比安全性。Qt 的迭代 (iterator) 是安全的而那些基于模板的更快的系统确不是。甚至在你发射一个信号到多个接收器的过程中，那些接收器可以被安全地删除而不会导致你的程序崩溃。没有了这种安全，你的程序在调试自由的内存读或写错误这种困难情况下最终会崩溃。

虽然如此，一个基于模板的解决方案不是能比使用信号和槽更加提高应用程序的性能吗？虽然 Qt 通过一个信号调用槽的时候会增加一点时间开销是真的，这个调用的开销只占整个槽调用的开销的很小比例。以上的情况都是基于 Qt 的信号和槽系统使用典型的空槽。一旦你在槽里面做任何有意义的事情时，比如一些简单的字符串操作，调用的时间开销就可以忽略不计了。Qt 的系统非常的优化了，以至于任何东西都要求操作符 `new` 或者 `delete` (比

如，字符串操作或者从一个模板容器插入/删除一些东西）的时间开销要比发射一个信号多的多。

另外：如果你在一个性能为关键的任务中的一个严紧的内部回路中使用信号和槽并且你认为这种连接是瓶颈的话，建议你使用标准的监听接口模式来替代信号和槽。当这种情况发生时，总之你也许只需要一个一对一的连接。比如，你有一个对象从网络上下载数据，你使用信号来说明所需要的数据已经到达的这种设计是非常明智的。但是如果你需要向接收者一个字节一个字节地发送数据，使用监听接口要比信号和槽好。

## 5. 没有限制

因为我们有元对象编译器来处理信号和槽，我们可以向它添加一些其它模板不能做的但很有用的东西。在这之中，有我们利用生成的 `tr()` 函数进行作用域翻译，和一个自我测量和扩展的运行时的类型信息的先进的属性系统。属性系统有一个独一无二的优势：没有一个强大的和自我测量的属性系统——如果这不是不可能的——一个 Qt 设计器这样的强大的和通用的用户界面设计工具就很难被写出来。

带有元对象编译器预处理器的 C++ 从本质上给我们带来对象的 C 的灵活性或一个 Java 的运行环境，当保持 C++ 的唯一特性和可伸缩的优点。它使得 Qt 成为我们今天拥有的灵活和舒适的工具。

# 几何结构和布局

## 布局类

Qt 的布局系统提供了一个规定子窗口部件布局的简单的和强有力的方式。

当你一旦规定了合理的布局，你就会获得如下利益：

- 布置子窗口部件。
- 最高层窗口部件可感知的默认大小。
- 最高层窗口部件可感知的最小大小。
- 调整大小的处理。
- 当内容改变的时候自动更新：
  - 字体大小、文本或者子窗口部件的其它内容。
  - 隐藏或者显示子窗口部件。
  - 移去一些子窗口部件。

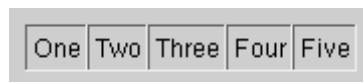
Qt 的布局类使用手写的 C++ 代码设计的，所以它们很容易被理解和使用。

手写的布局代码的不利是当你在试验设计一个视窗并且你不得不为每一个改变进行编译、连接和运行这样一个循环的时候它不是非常方便的。我们的解决方案是 [Qt 设计器](#)，一种可以快速建立和很容易地试验布局和为你生成 C++ 布局代码地图形用户界面可视化设计工具。

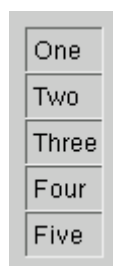
## 布局窗口部件

给你的窗口部件一个好的布局的最好的方法是使用这些布局窗口部件：[QHBox](#)，[QVBox](#) 和 [QGrid](#)。一个布局窗口部件自己自动地把它们的子窗口部件按照它们被构造地顺序进行布局。为了生成更复杂的布局，你可以嵌入一个布局窗口部件到其它的。

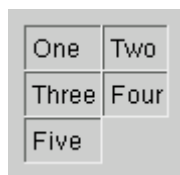
- 一个 [QHBox](#) 把它的子窗口部件从左到右排列在一个水平的行中。



- 一个 [QVBox](#) 把它的子窗口部件从上到下排列在一个竖直的列中。



- 一个 `QGrid` 把它的子窗口部件排列在一个二维的网格中。你可以规定网格可以有多少列，当前面的一行已经排满，它将会从左到右填充新的一行。网格是被确定的，当这个窗口部件被改变大小的时候，它的子窗口部件不会跑到其它的行。



上述演示的表格是由下面这些代码生成的：

```
QGrid *mainGrid = new QGrid( 2 ); // 一个 2*n 的网格
new QLabel( "One", mainGrid );
new QLabel( "Two", mainGrid );
new QLabel( "Three", mainGrid );
new QLabel( "Four", mainGrid );
new QLabel( "Five", mainGrid );
```

你可以通过对子窗口部件调用 `QWidget::setMinimumSize()` 或 `QWidget::setFixedSize()` 来调节布局。

## QLayout

当你添加一个窗口部件到一个布局中，布局过程工作如下：

1. 所有的窗口部件将最初根据它们的 `QWidget::sizePolicy()` 而被分配到一定空间中。
2. 如果任何一个窗口部件有一个伸展因素设置，而且数值大于零，那么它们就会被根据它们的 **伸展因素** 的比例分配空间。
3. 如果任何一个窗口部件有一个伸展因素设置而且数值为零，那么只有当其它窗口部件不再需要空间的时候才会得到更多的空间。在这当中，空间会首先被根据延展大小策略分配给窗口部件。
4. 任何窗口部件被分配的空间的的大小如果小于它们的最小大小（或者是在没有规定最小大小时的最小大小的提示），它们就会被按它们所需要的最小大小分配空间。（如果窗口部件的伸展因素是它们的决定因素的情况下，它们不必有最小大小或者最小大小的提示。）
5. 任何窗口部件被分配的空间的的大小如果大于它们的最大大小，它们就会被按它们所需要的最大大小分配空间。（如果窗口部件的伸展因素是它们的决定因素的情况下，它们不必有最大大小。）

### 伸展因素

窗口部件通常是在没有伸展因素设置的情况下被生成的。当它们被布置到一个布局中时，窗口部件会被根据它们的 `QWidget::sizePolicy()` 或者它们的最小大小的提示中大的那一个分配给整个空间的一部分。伸展因素是用来根据窗口部件互相的比例来改变它们所被分配的空间。

如果你使用一个 `QHBoxLayout` 来布置没有伸展参数设置的三个窗口部件，我们就会得到像下面这样的布局：



如果我们给每个窗口部件设置一个伸展因素，它们就会被按比例布置（但是不能小于最小大小的提示），举例来说：



## QLayout

如果你需要控制布局，可以使用 `QLayout` 的子类。Qt 中包括的布局类有 `QGridLayout` 和 `QBoxLayout`。（`QHBoxLayout` 和 `QVBoxLayout` 是 `QBoxLayout` 很简单的子类，它们可以更简单地使用并且使得代码更容易阅读。）

当你使用一个布局，你将不得不把每个子窗口部件不仅放到它们的父窗口部件中（这个已经在构造函数中完成），而且要把它们放到它们的布局中（典型地通过调用 `addWidget()` 完成）。这个方法中，你可以对每个窗口部件给出布局参数，规定像对齐方式、伸展因素和位置这样的属性。

下面的代码生成一个和上面的那个网格类似，但有一些改进的网格：

```
QWidget *main = new QWidget;

// 生成一个 1*1 的网格；它可以自动展开
QGridLayout *grid = new QGridLayout( main, 1, 1 );

// 根据 (行, 列) 编址来添加前四个窗口部件
grid->addWidget( new QLabel( "One", main ), 0, 0 );
grid->addWidget( new QLabel( "Two", main ), 0, 1 );
grid->addWidget( new QLabel( "Three", main ), 1, 0 );
grid->addWidget( new QLabel( "Four", main ), 1, 1 );

// 添加最后一个窗口部件到第 2 行，并占用第 0 列和第 1 列，中间对齐
grid->addMultiCellWidget( new QLabel( "Five", main ), 2, 2, 0, 1,
                        Qt::AlignCenter );

// 设置第 0 列和第 1 列的宽度比例为 2:3
grid->setColStretch( 0, 2 );
grid->setColStretch( 1, 3 );
```

你可以通过把父布局作为子布局构造函数的一个参数来把布局添加到另一个布局中。

```
QWidget *main = new QWidget;
```

```

QLineEdit *field = new QLineEdit( main );
QPushButton *ok = new QPushButton( "OK", main );
QPushButton *cancel = new QPushButton( "Cancel", main );
QLabel *label = new QLabel( "Write once, compile everywhere.", main );

// 一个窗口部件中的布局
QVBoxLayout *vbox = new QVBoxLayout( main );
vbox->addWidget( label );
vbox->addWidget( field );

// 一个布局中的布局
QHBoxLayout *buttons = new QHBoxLayout( vbox );
buttons->addWidget( ok );
buttons->addWidget( cancel );

```

如果你对默认位置感到不满意，你可以不使用父布局参数来生成一个布局然后通过调用 `addLayout()` 把这个布局插入到其它布局中。

## 自定义布局

如果内置的布局类不够充分的话，你可以定义你自己的布局类。你必须创建 `QLayout` 的一个子类来处理重新定义大小和大小的计算，也可以使用 `QGLayoutIterator` 的子类来迭代你的布局类。

如果需要更深的描述，请看 [Custom Layout](#) 页。

## 在布局中自定义窗口部件

当你创建自己的窗口部件类的时候，你也应该传递它的布局属性。如果这个窗口部件有一个 `QLayout`，这样的话就已经被处理了。如果这个窗口部件不包括任何子窗口部件，或者使用自定义布局，你需要重新实现下面这些 `QWidget` 的成员函数：

- `QWidget::sizeHint()` 返回窗口部件的优先选用的大小。
- `QWidget::minimumSizeHint()` 返回窗口部件所能有的最小大小。
- `QWidget::sizePolicy()` 返回一个 `QSizePolicy`；这个值描述的是这个窗口部件所需要的空间。

只要大小提示、最小大小提示或者大小策略发生改变，都要调用 `QWidget::updateGeometry()`。这会引起布局的重新计算。对 `updateGeometry()` 的多重调用只会引起一次重新计算。

如果你的窗口部件的优先选用的高度依赖于它的实际宽度（比如一个自动断词的标签），在 `sizePolicy()` 中设置 `hasHeightForWidth()` 标记，并且重新实现 `QWidget::heightForWidth()`。

即使你实现了 `heightForWidth()`，提供一个好的 `sizeHint()` 仍然是必需的。`sizeHint()` 提供了窗口部件的优先选用的宽度，并且它经常被不支持 `heightForWidth()` 的 `QLayout` 的子类使用（`QGridLayout` 和 `QBoxLayout` 都支持）。

当你实现这些函数并想获得更多的制导的时候，请查看和你的新窗口部件所需要的布局存在于 Qt 类中最接近的布局的实现。

## 人工布局

如果你要生成一种特殊的布局，你也可以按上面的描述来生成一个自定义窗口部件。重新实现 `QWidget::resizeEvent()` 来计算所需要分配的大小并且给每一个子窗口部件调用 `setGeometry()`。

当布局需要重新计算的时候，窗口部件会得到一个 类型是 `LayoutHint` 的事件。重新实现被通知 `LayoutHint` 事件的 `QWidget::event()`。

## 写你自己的布局管理器

这里我们提供了一个详细的例子。`CardLayout`类是从同名的Java布局管理器得到的灵感。它把项目（窗口部件或者嵌套的布局）布置到彼此的顶端，每个项目通过 `QLayout::spacing()` 偏移的。

要写自己的布局类，你必须像下面这样定义：

- 一个用来存放被布局处理的项目的数据结构。每一个项目就是一个 `QLayoutItem`。我们将在例子中使用一个 `QPtrList`。
- `addItem()`，如何向布局中添加一个项目。
- `setGeometry()`，如何构成一个布局。
- `sizeHint()`，布局优先选用的大小。
- `iterator()`，如何迭代布局。

在绝大多数情况下，你也需要实现 `minimumSize()`。

### card.h

```
#ifndef CARD_H
#define CARD_H

#include <qlayout.h>
#include <qptrlist.h>

class CardLayout : public QLayout
{
public:
    CardLayout( QWidget *parent, int dist )
        : QLayout( parent, 0, dist ) { }
```

```

    CardLayout( QLayout* parent, int dist)
        : QLayout( parent, dist ) { }
    CardLayout( int dist )
        : QLayout( dist ) { }
    ~CardLayout();

    void addItem(QLayoutItem *item);
    QSize sizeHint() const;
    QSize minimumSize() const;
    QLayoutIterator iterator();
    void setGeometry(const QRect &rect);

private:
    QList<QLayoutItem> list;
};

#endif

```

## card.cpp

```
#include "card.h"
```

首先我们为布局定义一个迭代。布局迭代被布局系统用作内部处理图形窗口部件删除操作的。它们也可以被提供给应用程序员。

这里有两个不同的相关类：[QLayoutIterator](#)是一个可以提供给应用程序员可见的类，它是 **明确地被共享**。[QLayoutIterator](#)包括一个可以任何事情地 [QGLayoutIterator](#)。我们必须生成一个知道如何迭代我们地布局类的[QGLayoutIterator](#)的子类。

在这种情况下，我们选择一个简单的实现：我们吧一个整数索引和指针存储到一个列表中。每一个 [QGLayoutIterator](#)的子类都必须实现 [current\(\)](#)、[next\(\)](#)和 [takeCurrent\(\)](#)，还有一个构造函数。在我们的例子中我们不需要析构函数。

```

class CardLayoutIterator : public QGLayoutIterator
{
public:
    CardLayoutIterator( QList<QLayoutItem> *l )
        : idx( 0 ), list( l ) { }

    QLayoutItem *current()
    { return idx < int(list->count()) ? list->at(idx) : 0; }

    QLayoutItem *next()
    { idx++; return current(); }

    QLayoutItem *takeCurrent()
    { return list->take( idx ); }
}

```



```
private:
    int idx;
    QList<QLayoutItem> *list;
};
```

我们必须实现QLayout::iterator()返回一个这个布局的 QLayoutIterator。

```
QLayoutIterator CardLayout::iterator()
{
    return QLayoutIterator( new CardLayoutIterator(&list) );
}
```

addItem()实现了布局项目的默认布置策略。它必须被实现。它被 QLayout::add()使用，被把一个布局作为父布局的 QLayout的构造函数使用，并且它被用来实现 自动添加这一特性。如果你的布局有需要参数的高级布置选项，你将必须提供像 QGridLayout::addMultiCell()一样的额外的访问函数。

```
void CardLayout::addItem( QLayoutItem *item )
{
    list.append( item );
}
```

布局对项目的增加负有责任。因为 QLayoutItem不继承 QObject，我们必须人工地删除这些项目。QLayout::deleteAllItems()函数使用我们前面定义的迭代来删除布局中的所有项目。

```
CardLayout::~CardLayout()
{
    deleteAllItems();
}
```

setGeometry()函数实际上执行了这个布局。作为参数提供的矩形不包括 margin()。如果相关的话，项目之间的距离请使用 spacing()。

```
void CardLayout::setGeometry( const QRect &rect )
{
    QLayout::setGeometry( rect );

    QListIterator<QLayoutItem> it( list );
    if (it.count() == 0)
        return;

    QLayoutItem *o;

    int i = 0;

    int w = rect.width() - ( list.count() - 1 ) * spacing();
    int h = rect.height() - ( list.count() - 1 ) * spacing();
```

```

while ( (o = it.current()) != 0 ) {
    ++it;
    QRect geom( rect.x() + i * spacing(), rect.y() + i * spacing(),
                w, h );
    o->setGeometry( geom );
    ++i;
}
}

```

`sizeHint()`和 `minimumSize()`通常情况下在实现中非常相似。这两个函数返回的大小应该包括 `spacing()`，但不包括 `margin()`。

```

QSize CardLayout::sizeHint() const
{
    QSize s( 0, 0 );
    int n = list.count();
    if ( n > 0 )
        s = QSize( 100, 70 ); // start with a nice default size
    QPtrListIterator<QLayoutItem> it( list );
    QLayoutItem *o;
    while ( (o = it.current()) != 0 ) {
        ++it;
        s = s.expandedTo( o->minimumSize() );
    }
    return s + n * QSize( spacing(), spacing() );
}

```

```

QSize CardLayout::minimumSize() const
{
    QSize s( 0, 0 );
    int n = list.count();
    QPtrListIterator<QLayoutItem> it( list );
    QLayoutItem *o;
    while ( (o = it.current()) != 0 ) {
        ++it;
        s = s.expandedTo( o->minimumSize() );
    }
    return s + n * QSize( spacing(), spacing() );
}

```

## 更多的注释

这个布局没有实现 `heightForWidth()`。

我们忽略了 `QLayoutItem::isEmpty()`，这也就是说这个布局将会把隐藏的窗口部件显示出来。

对于复杂的布局，通过存储计算结果可以使速度得到很大的提高。在这种情况下，实现 `QLayoutItem::invalidate()` 来把被存储的数据弄脏。

调用 `QLayoutItem::sizeHint()`，其它的也许更浪费时间，如果你在同一个函数中再一次稍晚的情况下需要这个值，你应该把它存储成局部变量。

你不应该在同一个函数中对同一项目调用 `QLayoutItem::setGeometry()` 两次。如果这个项目有几个子窗口部件的话，它会很浪费时间，因为它将不得不每次都执行一个完整的布局。相反，计算几何位置并且设置它。（这不仅仅是应用于布局，如果你实现了你自己的 `resizeEvent()` 你应该做同样的事情。）

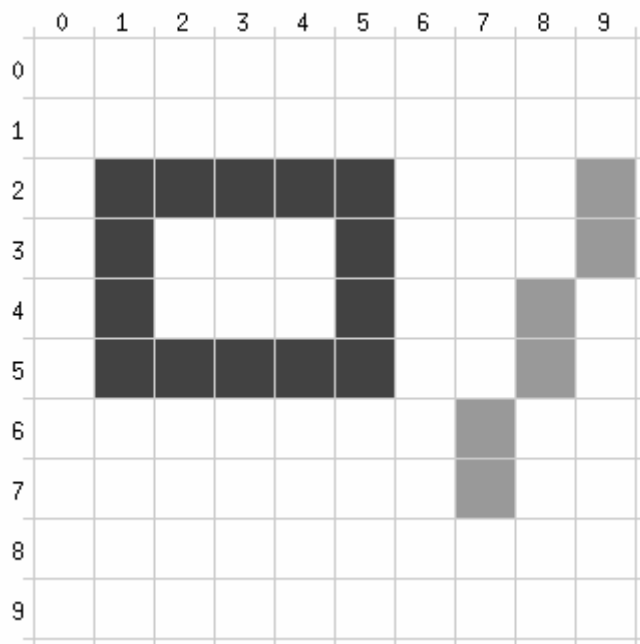
## 坐标系统

在Qt中的一个 **绘画设备** 是一个可画的二维平面。`QWidget`、`QPixmap`、`QPicture` 和 `QPrinter` 都是绘画设备。`QPainter` 是一个可以在上面画的对象。

一个绘画设备的默认坐标系统的原点在左上角。**X** 轴由左向右增加，**Y** 轴由上向下增加。对于基于像素的设备单位是像素，对于打印机是点。

### 一个例子

下面这个图显示了一个绘画设备的左上角的高度放大的部分。



这个矩形和线段是由下面的代码画出来的（在这个图中添加了网格）：

```
void MyWidget::paintEvent( QPaintEvent * )
{
    QPainter p( this );
    p.setPen( darkGray );
    p.drawRect( 1, 2, 5, 4 );
    p.setPen( lightGray );
    p.drawLine( 9, 2, 7, 7 );
}
```

applies to all the relevant functions in [QPainter](#). 请注意drawRect()所画的所有像素是在指定的大小之内（5\*4 像素）。这一点和其它工具包不同；Qt中你指定的大小包括你所要画的像素。这一点适用于所有 [QPainter](#)中相关的函数。

相似的，drawLine()调用画了这个线段的两个端点，而不仅仅是一个。

这里是一些和坐标系统关系很近的类：

- [QPoint](#)是坐标系统中的单个二维点。Qt中处理点的绝大多数函数都可以接受 [QPoint](#) 参数或者两个整数参数，比如 [QPainter::drawPoint\(\)](#)。
- [QSize](#)是单个二维矢量。在内部，[QPoint](#)和 [QSize](#)是一样的，但是一个点和一个大小不同，所以这两个类都存在。同样，绝大多数函数都可以接受一个[QSize](#)或者两个整数，比如 [QWidget::resize\(\)](#)。
- [QRect](#)是一个二维的矩形。绝大多数函数都可以接受一个 [QRect](#)或者四个整数，比如 [QWidget::setGeometry\(\)](#)。
- [QRegion](#)是点的一个任意组合，包括所有通常的组合操作，例如 [QRegion::intersect\(\)](#)，并且一些常用的函数返回一个矩形的列表，这个矩形的列表的集合等于一个区域。[QRegion](#)被例如 [QPainter::setClipRegion\(\)](#)、[QWidget::repaint\(\)](#)和 [QPaintEvent::region\(\)](#)使用。
- [QPainter](#)是一个绘图的类。它可以用同样的代码在任何一个设备上绘画。设备中是存在差异的，[QPrinter::newPage\(\)](#)是一个很好的例子，但 [QPainter](#)可以对所有的设备按同一种方式工作。
- [QPaintDevice](#)是一个[QPainter](#)可以在上面绘画的设备。这里有两种都基于像素的内部设备，和两种外部设备，[QPrinter](#)和 [QPicture](#)（它把[QPainter](#)的命令都记录在一个文件或者其它的 [QIODevice](#)，然后再运行它们）。其它的设备都可以被定义。

## 变换

尽管Qt默认的坐标系统像我们上面描述的那样工作，[QPainter](#)也支持任意的变换。

这个变换引擎是一个三步的管道，和下面的 [Foley & Van Dam and the OpenGL Programming Guide](#)两本书中的模型非常接近。参考那些深度的内容，这里我们给出了一个简单的概述和一个例子。

第一步使用通用变换矩阵。使用这个矩阵来定位你的模型中的对象。Qt提供了像 [QPainter::rotate\(\)](#)、[QPainter::scale\(\)](#)、[QPainter::translate\(\)](#)等等方法来操作这个矩阵。

`QPainter::save()`和 `QPainter::restore()`存储和恢复这个矩阵。你也可以使用 `QWMatrix`对象、`QPainter::worldMatrix()`和 `QPainter::setWorldMatrix()`来存储和使用指定的矩阵。

第二步使用窗口。这个窗口在模式坐标上说明视图的边界。矩阵定位对象并且 `QPainter::setWindow()`定位这个窗口，决定哪个坐标系统是可见的。（如果你有三维的经验，这个窗口通常是被三维投影出来的。）

第三步使用视口。视口也描述视图的边界，但是是在设备坐标上。视口和窗口描述的是同一个矩形，但是在不同的坐标系统中。

在屏幕上，默认的是你所画的通常合适的整个 `QWidget`或者 `QPixmap`。对于打印这个功能是非常重要的，因为只有很少的打印机可以在真个物理页面上打印。

所以每一个要画的对象被使用 `QPainter::worldMatrix()`变换到模式坐标中，然后被 `QPainter::window()`剪辑，最后被使用 `QPainter::viewport()`定位到所画的设备上。

没有上述一个或两个步骤也是可能做到的。举例来说，比如你的目标是画一些按比例缩放的东西，然后使用 `QPainter::scale()`来达到理想的效果。如果你使用一个大小固定的坐标系，`QPainter::setWindow()`是最好的。等等。

这里是一个使用所有这三个机制的一个简单的例子：这个函数在 `aclock/aclock.cpp`例子中画了一个钟表面板。我们建议你在读任何代码之前先编译和运行这个例子。尤其是，试着把这个窗口改变成不同的形状。

```
void AnalogClock::drawClock( QPainter *paint )
{
    paint->save();
```

首先，我们保存绘画工具的状态，这样的调用函数就保证了不会将要使用的变换干扰。

```
    paint->setWindow( -500, -500, 1000, 1000 );
```

我们设置一个 1000\*1000、原点 (0, 0)在中间的窗口的模式坐标系统。

```
    QRect v = paint->viewport();
    int d = QMIN( v.width(), v.height() );
```

这个设备也许不是正方形的，但我们希望钟表是正方形的，所以我们找到它当前的视口并计算它最短的边。

```
    paint->setViewport( v.left() + (v.width()-d)/2,
                       v.top() + (v.height()-d)/2, d, d );
```

然后我们设置一个新的正方形视口，把它放在原来的中间。

现在我们做完了我们的视图。从这点上来看，当我们在 (0, 0) 点周围画一个 1000\*1000 的区域，我们所画的东西将会在适合输出设备的最大可能的正方形区域中显示出来。

是我们开始绘图的时候了。

```
// time = QTime::currentTime();  
QPointArray pts;
```

因为我们在画一个钟表，所以我们需要知道时间。*pts* 刚好是一个用来存储一些点的有用的变量。

接下来我们画三个块，一个是时针，一个是分针，最后一个是钟表面板自己。首先我们来画时针：

```
paint->save();  
paint->rotate( 30*(time.hour()%12-3) + time.minute()/2 );
```

我们保存绘画工具，然后根据时针转动的方向旋转它。

```
pts.setPoints( 4, -20,0, 0,-20, 300,0, 0,20 );  
paint->drawConvexPolygon( pts );
```

我们把 *pts* 设置为一个四点的多边形，好像时针就在三点钟，然后画上它。因为这个旋转，时针被画到正确的位置。

```
paint->restore();
```

我们恢复被存储的绘图工具，取消旋转。我们也可以通过调用 `rotate(-30)`，但是那样也许会导致旋转错误，所以使用 `save()` 和 `restore()` 是更好的方法。接下来，按大致相同的方式画分针：

```
paint->save();  
paint->rotate( (time.minute()-15)*6 );  
pts.setPoints( 4, -10,0, 0,-10, 400,0, 0,10 );  
paint->drawConvexPolygon( pts );  
paint->restore();
```

唯一的不同是如何计算要旋转的角度和多边形的形状。

要画的最后一部分就是钟表面板自己。

```
for ( int i=0; i<12; i++ ) {  
    paint->drawLine( 440,0, 460,0 );  
    paint->rotate( 30 );  
}
```

十二个短的时针刻度之间的间隔是三十度。最后，绘图工具被旋转不是一个非常有用的方法，但是我们已经画完了，所以这也没什么。

```
paint->restore();  
}
```

最后一行的函数是恢复绘图工具，这样绘图工具的调用者就不用担心我们所做的所有变换会影响它。

## 窗口几何结构

### 概述

[QWidget](#)提供了几个处理窗口几何结构的函数。这些函数中的几个操作纯客户区域（例如不包含窗口框架的窗口），其它一些包括窗口框架。它们之间的区别在某种意义上被完成覆盖明显地最普通的方法。

包括窗口的框架：

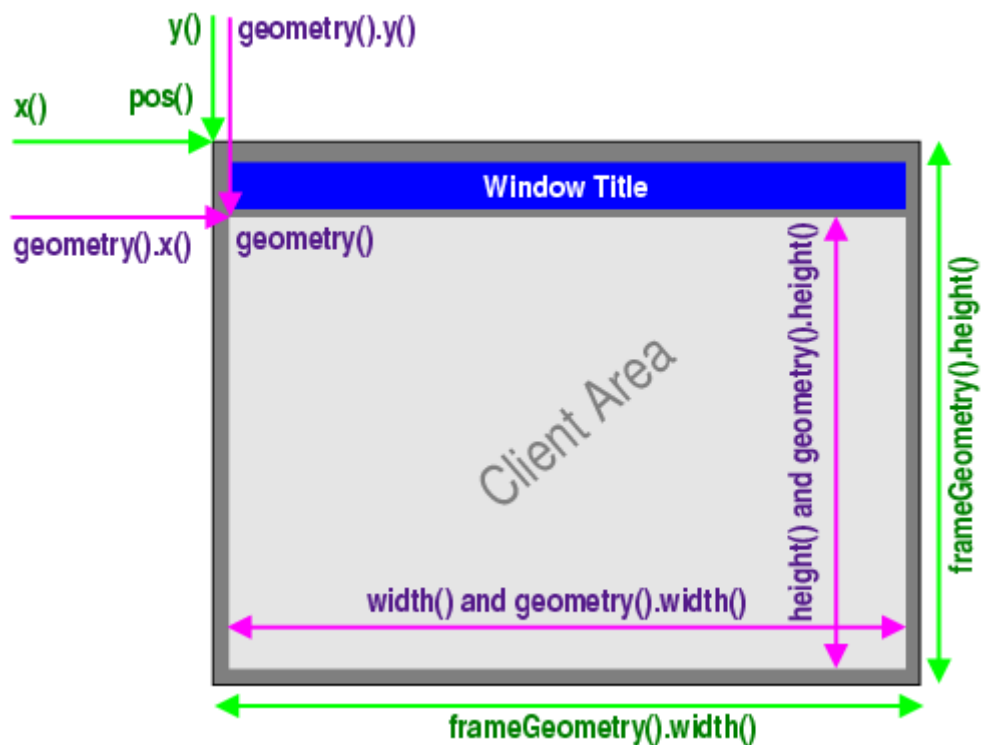
`x()`, `y()`, `frameGeometry()`, `pos()` and `move()`

不包括窗口的框架：

`geometry()`, `width()`, `height()`, `rect()` and `size()`

请注意这种区别仅仅对于被装饰的顶层窗口部件有效。对于所有的子窗口部件，框架的几何结构和这个窗口部件的客户几何结构相同。

这个图显示了我们所使用的绝大多数函数：



### Unix/X11 特性



在 Unix/X11 中，一个窗口没有框架直到窗口管理器来装饰它。这种情况发生在异步情况下调用 `show()` 之后和窗口接收到第一个绘画事件的一些点中，否则它根本就不会发生。请记住 X11 是自由策略的（其它人把它称为灵活的）。因此你不能作出关于你的窗口得到装饰框架这样的任何安全假设。基本规则：总是会有用户使用窗口管理器来打破你的假设，并且他还会向你抱怨。

此外，一个工具包不能简单地把窗口放置到屏幕上。Qt 所能做的一切是把一些确定的提示发送给窗口管理器。窗口管理器，一个单独的进程，也许会遵守、忽略或者误解它们。由于部分不清楚的客户间通信约定（`ICCCM`），在存在的窗口管理器中窗口放置的处理非常困难。

当窗口被装饰的时候，X11 没有提供标准的或者容易的方式来获得框架的几何结构。Qt 使用了可以工作在今天存在的大量的窗口管理器中的非常富有启发性和巧妙的代码来解决这一问题。如果你发现某种情况下 `frameGeometry()` 返回了虚假的结果的时候，请不要惊讶。

X11 同样也不提供最大化一个窗口的方法。因此 Qt 中的 `showMaximized()` 函数不得不模拟这一功能。它的结果完全取决于 `frameGeometry()` 的结果和窗口管理器作出正确的窗口放置的能力，但这两个都不能被保证。

## 恢复窗口几何结构

现代的应用程序的一个普通任务就是在稍后的对话中恢复一个窗口的几何结构。在 Windows 中，这基本上通过存储 `geometry()` 的结果和在下一个对话中还没有框架之前调用 `setGeometry()`。在 X11 中，这样将不会工作，因为一个没有显示的窗口还没有框架呢。窗口管理器将会在稍后装饰这个窗口。当这些发生的时候，窗口朝着依赖于装饰框架的大小的屏幕的右下角偏移。X 从理论上提供了一种可以避免这种偏移的方法。尽管，我们的测试已经显示，绝大多数窗口管理器都不能实现这一特性。

一个工作区在 `show()` 之后调用 `setGeometry()`。这样做有两个缺点，是窗口部件会在一毫秒中出现在一个错误的位置（结果是闪烁）和通常在一秒之后窗口管理器会正确地得到它。一个安全的方式是存储 `pos()` 和 `size()` 并且在 `show()` 之前调用 `resize()` 和 `move()` 来恢复几何结构，就像下面这个例子一样装饰：

```
MyWidget* widget = new MyWidget
...
QPoint p = widget->pos();    // 存储位置
QSize s = widget->size();    // 存储大小
...
widget = new MyWidget;
widget->resize( s );          // 恢复大小
widget->move( p );            // 恢复位置
widget->show();               // 显示窗口部件
```

这种方法可以在 MS-Windows 和绝大多数现存的 X11 窗口管理器上工作。



# Qt 设计器手册

前言

创建一个 Qt 应用程序

创建含有工具栏和菜单的主窗口

走近设计器

派生类和动态对话框

创建自定义窗口部件

创建数据库应用程序

定制和集成 Qt 设计器

参考：快捷键

参考：菜单选项

参考：工具栏按钮

参考：对话框

参考：向导

参考：窗口

参考：.ui 文件格式

# 前言

## 介绍

这本参考手册是有关于 *Qt 设计器* 的, *Qt 设计器* 是用来设计和实现用户界面并能够在多平台下使用的一种工具。*Qt 设计器* 可以使用户界面设计实验变得简单。在任何时候你可以要求所生成的代码去重建 *Qt 设计器* 产生的用户界面文件, 并可以根据你的喜好来改变你的设计。假如你使用的是先前的版本, 你将发现在新的版本下自己可以立即进入工作, 因为新的版本在界面上基本没有什么变化。但是你将发现根据你们的反馈而开发出的新的部件和新的或者改进的功能。

*Qt 设计器* 帮助你使用部局工具在运行时自动的移动和缩放你的部件 (Windows 中的术语 *控件*) 来构建用户界面。最终界面是既好用又好看, 使最终用户拥有一个舒适的操作环境并且能够方便的进行参数选择。*Qt 设计器* 支持信号和槽机制以使部件间能够进行有效的通信。*Qt 设计器* 包含一个代码编辑器, 使你能够在合成的代码里面嵌入自己定制的槽。那些更喜欢使用手工方法分解合成代码的朋友也能够继续使用基类, 因为从第一版的 *Qt 设计器* 开始就把这些基类移植进去了。

这本手册通过讲述开发例程来向你介绍 *Qt 设计器*。一开始的六章是设计指南, 而且各自间都尽可能设计成是独立的。接下来要介绍的是除了首章以外的每一章, 并假定你已经熟悉了第一章的内容, 该章包含使用 *Qt 设计器* 创建一个 Qt 应用程序的基础。以下是便各章的简要概述:

- 第一章, [创建一个Qt应用程序](#), 通过带着你一步一步的创建一个虽小但功能完整的应用程序来介绍 *Qt 设计器* 的使用。按照着这种方法你将学到如何创建一个窗体并且向窗体中添加部件。在你阅读这一章的过程中你将使用窗体和属性编辑器来定制你的应用程序, 并且学习怎样使用部局工具来对一个窗体进行部局。你也将学到如何使用信号和槽机制和 *Qt 设计器* 的内建代码编辑器来制造应用程序的各种功能。我们也将解释如何使用 `qmake` 来生成 `Makefile`, 以致于你能够编译和运行应用程序。
- 第二章, [创建含有工具栏和菜单的主窗口](#), 我们将创建一个简单的文本编辑器。通过写这个应用程序你将学到如何使用菜单栏和工具栏来创建一个主窗口。我们将看到如何使用 Qt 的内建功能来处理一般任务 (e.g. 复制粘贴操作) 还将看到如何为我们自己的菜单栏选项和工具栏按钮创建我们自己的功能。
- 第三章, [走近设计器](#), 提供有关 *Qt 设计器* 的信息如相关的开发应用程序, 并且还对 *Qt 设计器* 背后的一些基本原理进行解释。
- 第四章, [派生类和动态对话框](#), 将展示如何派生一个窗体; 这将让你清楚的通过执行关键代码的功能来分解用户界面。本章中还附加有关 `qmake` 和 `uic` 的信息。本章也将阐述如何使用 `QWidgetFactory` 把 `.ui` 文件放进你的应用程序从而动态的加载对话框和如何访问这些对话框的部件和派生部件。
- 第五章, [创建自定义窗口部件](#), 告诉你如何才能创建自定义部件。既有在第一版的 *Qt 设计器* 中就被介绍的简单方法, 又有像利用插件这种新的更有效的方法都在这一章里被介绍了。
- 第六章, [创建数据库应用程序](#) 介绍了 Qt 的 SQL 类并且带着你通过一个实例来演示如何执行查询和如何设置主要关系的细节, 深入讲解和处理外关键字。

- 第七章，[定制和集成Qt设计器](#)，聚焦*Qt 设计器*本身，向你展示如何定制设计器，如何使用可视化工作室集成设计器和如何创建一个Makefiles。

这剩下的章节提供了一些参考资料，用以讲述 *Qt 设计器* 的菜单选项、工具栏、快捷键以及对话框等的细节。

## 你所应该知道的

该手册假定你已经有了一些有关 C++ 和 Qt 应用程序开发框架的基础。假如你需要学习 C++ 或者 Qt，这儿有大量的 C++ 的书可供使用和少量的但是数量却在不断增长的有关 Qt 的书。你可以尝试一下大量伴随着 Qt 的联机文档和许多例程。

企业版的Qt包含了SQL模块。在 [创建数据库应用程序](#) 一章里我们演示了如何使用*Qt 设计器*来编写SQL应用程序；这一章需要一些SQL和关系数据库的知识。

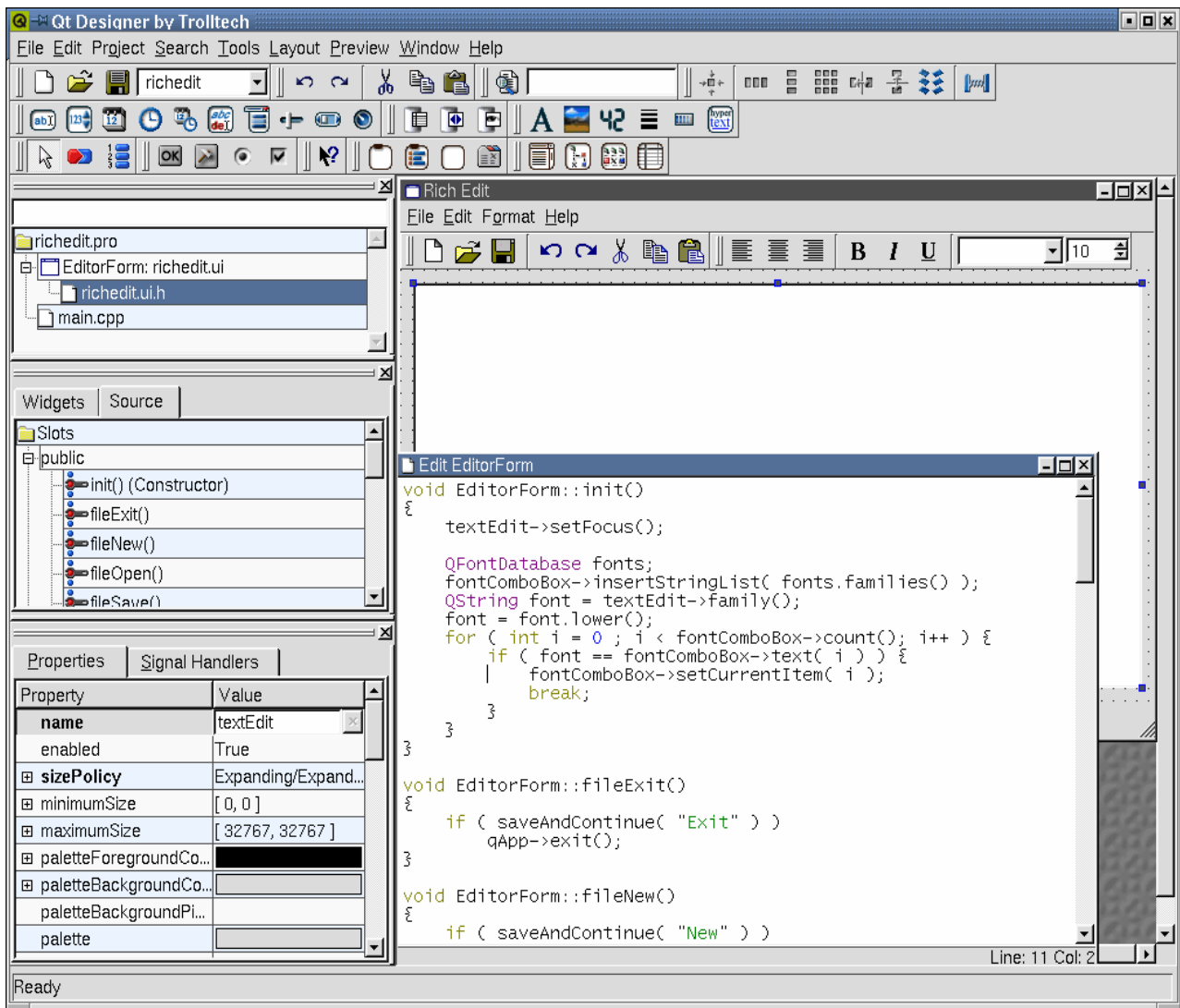
## Qt 设计器中的新东西

该版本的 *Qt 设计器* 较其前一版本来说增加了更加多的功能。自己定制的槽功能代码可以直接在 *Qt 设计器* 中编辑；操作工具栏、菜单栏就可以创建主窗口了；部局可以结合使用分解器；一些插件还允许你把大量的自定义部件进行打包并且在 *Qt 设计器* 中可以使用他们。从对用户界面微小的改善到提高效率还有很多其它增强功能被结使在里面，例如在一个应用程序里的所有窗体可以有效的共享像素映射。

该版本的*Qt 设计器*创建的工程文件使得开关在一个应用程序中的所有窗体变得十分简单，而且仍然保持了一个通常的数据库设置和映象。通过对派生类的全面支持，为直接在*Qt 设计器*中编写代码带来了很多的益处，这些知识已经全面涵盖在 [走近设计器](#) 一章中了。

还介绍了一个新的库 libqui，该库允许你在运行时从 *Qt 设计器* 的 .ui 文件中自动加载对话框。这允许你提供给你的应用程序用户相当可观的自定义界面自由度，否则就需要使用 C++ 了。

如果你仅仅想要一个简单而功能强大的单对话框可视设计工具，虽然新版本的 *Qt 设计器* 介绍了新的进阶和技术但你可以忽略这些方面并且正确的使用与 Qt 2.x 相同的方法。



Qt 设计器

## 反馈

如果你关于这个手册有一些注释、建议、批评或者适当的赞美，请访问[doc@trolltech.com](mailto:doc@trolltech.com)让我们知道。关于Qt或者Qt设计器的bug报告可以发送至[Qt-bugs@trolltech.com](mailto:Qt-bugs@trolltech.com)。你也许也想要加入专门由开发者阅读和捐献的Qt-interest邮件列表；请访问<http://www.trolltech.com>以了解更多的细节。

# 创建一个 Qt 应用程序

## 启动和退出 Qt 设计器

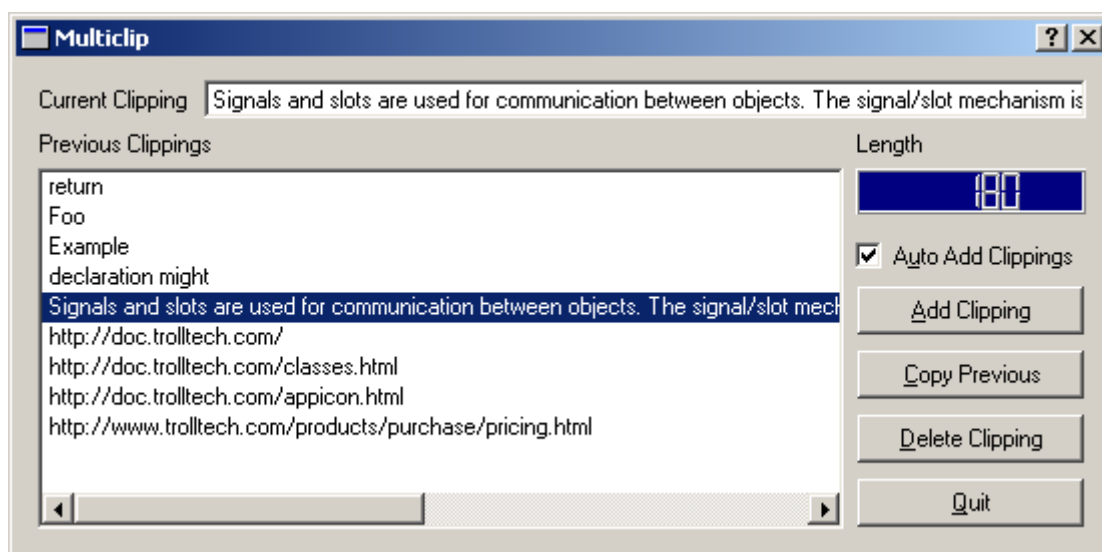
Qt 设计器和任何其他现代桌面应用程序一样已同样的方式被控制着，为了在 Windows 系统下启动 Qt 设计器，点击“开始”按钮，依次点击“程序|Qt X.x.x|Designer”（X.x.x 是 Qt 的版本号，例如 3.0.0）如果你是在 Unix 或者 Linux 操作系统下，你可以双击 Qt 设计器图标，或者在终端输入 designer。

当你完成使用 Qt 设计器，点击 File|Exit 退出时，设计器会提示你保存任何没有保存的改动，按下 F1 键或者点击“Help”菜单，你可以获得帮助。

为了使你从教程中获得最大收益，我们建议你现在开始启动 Qt 设计器，并且当你读到这些例子程序时创建它们，包括使用 Qt 设计器菜单、对话框和编辑器，大部分工作仅仅只需要键入少量的代码。

默认情况下，当你启动 Qt 设计器时，你会看到位于顶部的一个菜单栏和各种工具栏，在左边是三个窗口，第一个是文件（File）窗口，第二个是部件（Widget）和来源（Source）窗口（对象浏览器），第三个是属性（Properties）窗口。文件窗口列举了和该项目相关的文件和图像文件，要想打开任何窗体只需点击文件列表中对应的文件；部件和来源窗口列出了当前窗体的部件和槽，属性窗口用于浏览和改变窗体和部件的属性。当我们创建例子程序时，我们将介绍 Qt 设计器窗体、对话框、菜单选项和工具栏按钮的使用。

在这章我们将创建一个称为“Multiclip”的应用程序，它允许你存储多行文本到剪贴板或者从剪贴板获得多行文本。

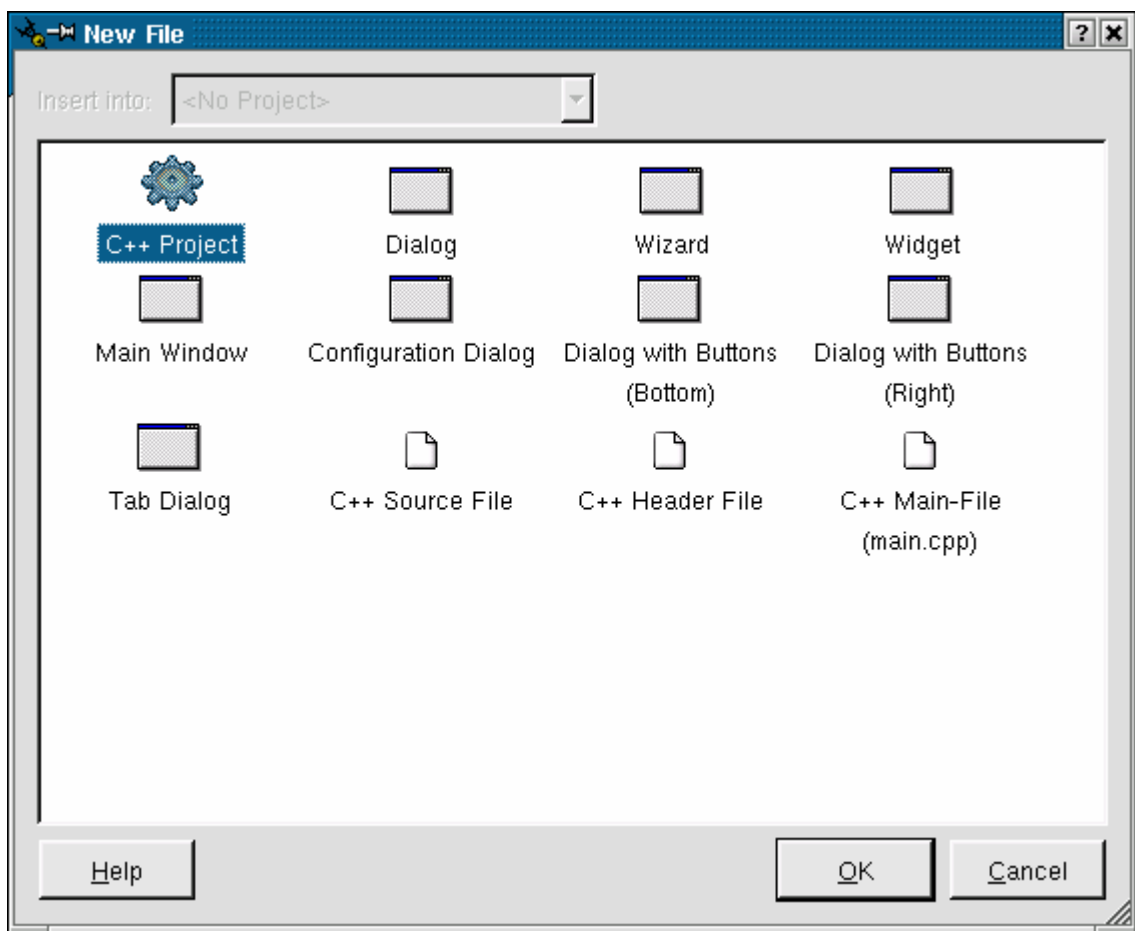


The Multiclip Application

## 创建一个项目

任何时候你创建一个新的应用程序，我们强类推荐你创建一个工程或者打开一个工程文件而不是只是创建一个单独的.ui 文件，使用一个工程有这样的好处：你为这个工程创建的所有窗体(form)可以在文件打开的对话框中仅仅通过鼠标点击加载，使用工程文件的另一个好处是就是，其允许你存贮你所有的图像文件到一个单一的文件而不是复制他们到将出现的每一个窗体中，请查看[走进设计器](#)章节的[工程管理](#)部分以获得更多的使用工程文件的好处的信息。

如果你仍未启动那现在就开始吧，点击 File|New 激活新文件对话框，点击“C++ Project”图标，点击“OK”按钮，激活项目设置对话框，你需要给出项目的文件名，我们推荐你将每一个工程文件放到其自己的子文件夹中，点击“...”按钮，激活 Save As 对话框，并浏览至你需要存放项目文件的地方，点击“Create New Folder”工具栏按钮，创建“multiclip 对话框”，双击“mulitclip”目录是其成为当前工作目录，键入一个文件名“multiclip.pro”，并点击 Save 按钮。项目设置对话框的“Project File”区域将显示你的信项目的路径名和名字，点击”OK”创建这个项目。



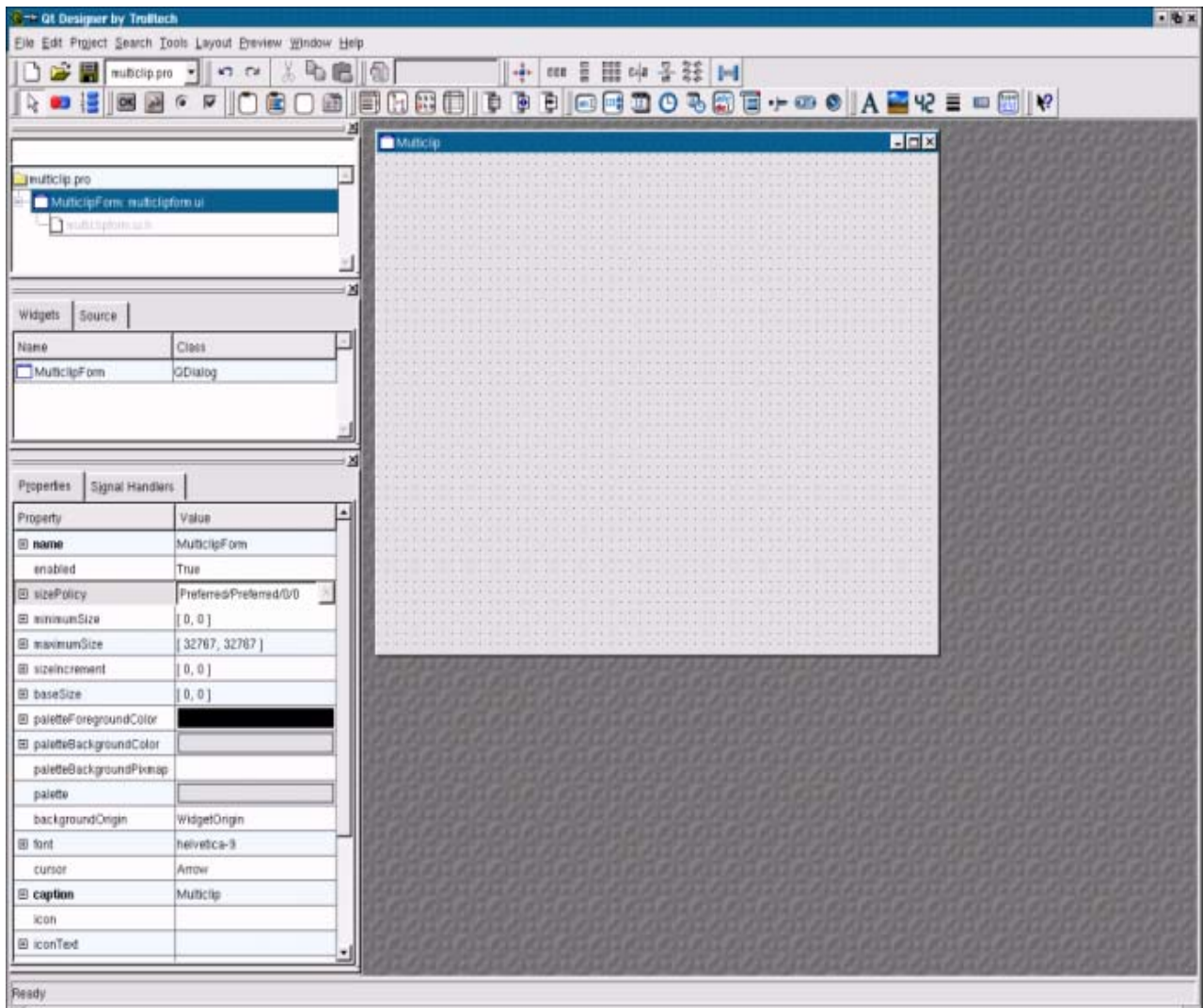
创建一个新项目



当前项目的名字显示在默认条件下左上工具栏中 Files 工具栏里，一旦你有了一个项目我们就能加入窗体并开始创建我们的应用程序，（请参看 Customizing Qt Designer 部分以获取有关改变 Qt 设计器的工具栏和窗口样式以符合你的需要的更多信息）

## 创建一个新窗体

点击 File|New 激活 New File 对话框，有几个默认的窗体出现，但是我们将使用默认的对话框形式，于是仅仅只需点击 OK，一个新的叫做“Form1”的窗体将会显示出来，注意到这个新的窗体被列表在文件列表栏，并且属性窗口显示了窗体的默认属性设置。

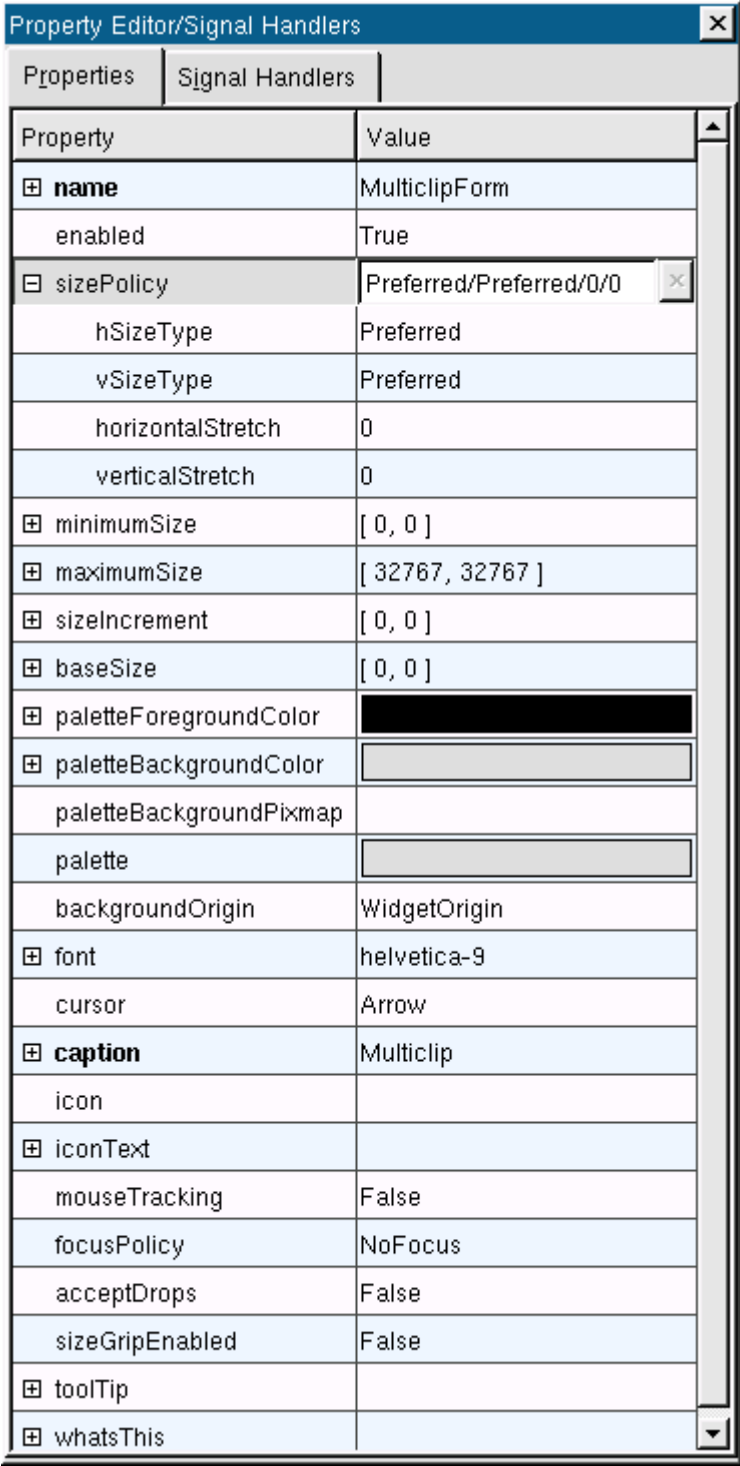


## 创建一个新窗体

点击 name 属性旁边的值，改变窗体的名字为“MultiClipForm”，改变窗体的标题为“MultiClip”，属性依据继承层次排序，标题大约在属性编辑器的中间显示，保存该窗体：点击 File|Save，键入名字“multiclip.ui”，于是点击“Save”按钮保存设置。

## 使用属性编辑器

属性编辑器有两列，属性列列出了属性的名字，值列列出了值（属性的取值），有些属性名字有一个“+”号，在他们左边的区域，这表示这个属性名是一系列相关属性的集合名字，点击窗体让属性编辑器显示窗体的属性，点击 SizePolicy 属性的加号，你将会看到四个属性缩进显示如下：sizePolicy, hSizeType, vSizeType, horizontalStretch 和 verticalStretch. 这些属性和其他属性一样以同样的方式在编辑器中修改。



Property Editor/Signal Handlers	
Properties	Signal Handlers
Property	Value
⊕ name	MulticlipForm
enabled	True
⊖ sizePolicy	Preferred/Preferred/0/0
hSizeType	Preferred
vSizeType	Preferred
horizontalStretch	0
verticalStretch	0
⊕ minimumSize	[ 0, 0 ]
⊕ maximumSize	[ 32767, 32767 ]
⊕ sizeIncrement	[ 0, 0 ]
⊕ baseSize	[ 0, 0 ]
⊕ paletteForegroundColor	
⊕ paletteBackgroundColor	
paletteBackgroundPixmap	
palette	
backgroundOrigin	WidgetOrigin
⊕ font	helvetica-9
cursor	Arrow
⊕ caption	Multiclip
icon	
⊕ iconText	
mouseTracking	False
focusPolicy	NoFocus
acceptDrops	False
sizeGripEnabled	False
⊕ tooltip	
⊕ whatsThis	

属性编辑框



有些属性具有简单是值，例如，名字(name)属性是一个文本值，宽度属性(width)（例如最小尺寸 minimmSize）是一个数字值，为了改变文本值，点击当前文本并键入你的新文本。要改比那数字值，点击改制同时键入新的数字，或者用翻动按钮(Spin button)增加或者减少当前的数字直道达到你所期望的数字，有些属性有一些固定的属性列值，例如，mouseTracking 属性是一个布尔型，能设置为真或者假，cursor 属性也有一列固定的值，如果你点击 cursor 属性，或者 mouseTracking 属性，则其值会显示在下拉组合框中，点击向下的箭头查看可用的属性取值，一些属性有复杂的取值集合，例如字体属性，如果你点击了字体属性后的省略号按钮，将会出现字体选择对话框，你可以在其中改变字体设置。其他有省略号按钮的属性设置依据属性所有的属性值会显示不同的对话框，例如，如果你要为一文本属性键入许多的文本，你可以点击省略号按钮激活多行文本编辑器对话框。

改变了的属性的名字以粗体字显示，如果你改变了属性值但是你又想恢复它到默认值，点击属性值并点击红色的”X”按钮，直到变成正确的值，有些属性有一个初始值，如“TextEdit1”，但不是默认值，如果你想恢复有初始值但是没有默认值的属性（通过点击红色的”X”按钮），如果属性，如名字，允许为空，则属性值会变成空值。

如果选中多个部件，属性编辑器会显示选中的部件的共有的属性，改变其中一个属性会导致所有选中的部件的该属性值的改变。

属性编辑器完全支持撤销和重复操作(Ctrl+Z 和 Ctrl+Y, 同样可以选择编辑菜单 Edit)

## 增加部件

这个 MultiClip 应用程序包含一个文本框显示当前剪贴板的文本（如果有的话），一个列表框显示前一个剪辑，一个长度指示器，一个校验框和按钮，如果你运行这个应用程序并改变它的大小，则所有的部件会按比例变化。

走进 Qt 设计器这一节讲述了设置一个窗体并将所需的部件放置到他们将显示的合适位置，使用布局工具正确的设置他们的大小和位置，现在我们添加 multiclip 窗体的部件

我们从当前的剪贴文本框开始，点击 Text Label 工具栏按钮并将鼠标指针移动到窗体左上方（如果你将鼠标在工具栏按钮上停留，则工具按钮名字会以标签形式显示出来），我们不想因为这些标签重新命名而自寻麻烦，因为在代码编写过程中不会设计到它，但是我们需要改变它的文本属性，改变其文本属性(text property)为“Current Clipping”。(属性编辑器的解释请看使用属性编辑浮动栏部分)

点击工具栏中的 Line Edit 按钮并在窗体的右上方点击放置该部件，使用属性编辑器改变该部件的名字为“currentLineEdit”。

1. 现在我们将添加另一个标签和列表框，点击 Text Label 工具栏按钮，并在“Current Clipping”标签下方放置，改变其文本属性(text property)为“Previous Clippings”，不要为这些部件放置的位置是否恰到好处而担心，布局工具（下部分会涉及到的）将会帮助你做好这一切。

点击 List Box 工具栏按钮，并点击鼠标放置在“Previous Clippings”标签下方，改变该列表框的名字为“clippingListBox”，默认情况下 Qt 设计器会以一个初始值“New Item”创建列表框，但我们不需要这个值（后面我们会以代码的方式组装我们的列表框），因为我们需要删除该值，在列表框处点击鼠标右键，在弹出的菜单中选中“Edit”菜单子项激活列表框的值编辑对话框，点击“Delete Item”按钮删除默认的项，然后点击“OK”按钮。（注意看值编辑器工具条）

2. 我们想知道当前剪贴文本的长度，所以我们将添加另一个标签和一个 LCD Number 部件。

点击 Text Label 工具栏按钮，并将其放置在 Line Edit 下方，改变其文本属性值为“Length”，点击 LCD Number 工具栏按钮并放置在长度标签下方，改变 LCD Number 的名字为“lengthLCDNumber”。

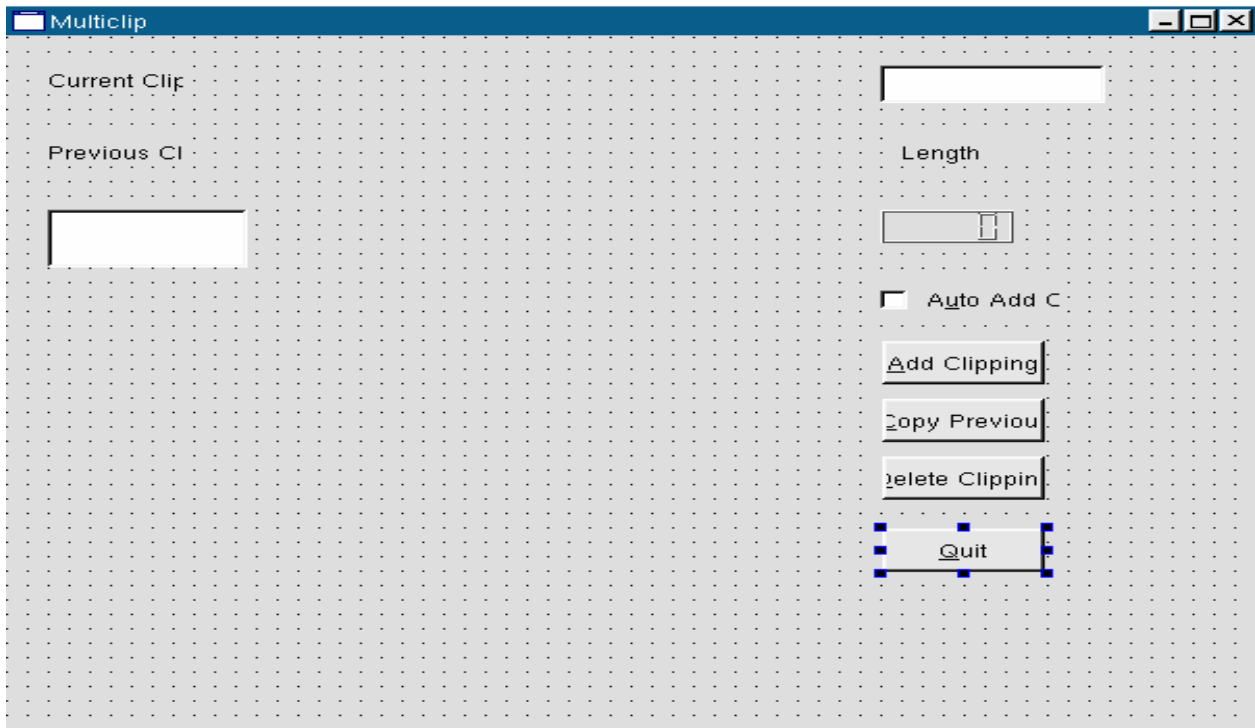
3. 多项剪辑程序能检测到剪贴板的变化并自动增加新的剪辑，无论这是是否发生，我们想让用户控制它，于是我们提供一个检测按钮来指示他们的最优选择。

点击 Check Box 工具栏按钮并放置到 LCD Number 下方，改变检测框的名字为“autoCheckBox”，改变其文本属性值为“A&uto Add Clippings”，注意到加速键属性值自动改变为 Alt+U，因为文本中的符号 & 指明了一个键盘快捷键。

4. 最后我们需要的部件是按钮，添加同种类型部件的一种方式是个添加该部件，复制它并重复的粘贴之，这里我们使用另一种方式。。

双击 Push Button 工具栏按钮，然后在检测框的下方点击放置一个按钮，然后再这个刚刚添加的按钮下方点击鼠标放置第二个按钮，如此这般，依次添加第三个和第四个按钮，点击“Pointer”（箭头型的按钮）工具栏按钮关闭自动添加同种部件的行为，改变第一个按钮的名字为“addPushButton”并将其文本置为“&Add Clipping”，改变第二个按钮的名字为“copyPushButton”并将其文本置为“&Copy Previous”改变第三个按钮的名字和文本值分别为“deletePushButton”和“&Delete Clipping”，类似的第四个按钮改变为“quitPushButton”和“&Quit”。

所有的部件的属性依据我们程序的需要改变后放置在窗体中，下一步我们将使用 Qt 设计器的布局工具正确的设置部件的大小和位置，这样当用户改变窗体的大小时部件能随着适当的变化。



## 向窗体添加部件

### 值编辑器

同时属性编辑器用于定制部件的一般属性，值编辑器用于编辑特定部件的对象值，例如一个 `QLineEdit` 仅能包含一个单行的文本，但是一个 `QListBox` 能包含任何多的项，每一项可以是一行文本、一个像素映射或者都是。为了激活一个部件的值编辑器，双击该部件，（你也可以在该部件处右击鼠标出现一个弹出的菜单，如果第一个子菜单选项是“Edit”，你能点击该项访问该部件的值编辑器对话框）不同的部件有不同的值编辑器，参看值编辑器以获得更多细节。

### 布置部件和预览

#### 布局介绍

通过对部件和部件组以垂直、水平和网格形式组合实现布局工作，以垂直和水平方式放置的部件能以布局或者分裂形式组合，唯一的不同是用户能操作这些分裂机。（各组件之间用布局工具放置组合，为了使得位置恰当，有时加入分裂机使得空间布局合理）

如果我们想一个靠着一个的放置一些部件，我们将选中他们并点击 `Lay Out Horizontally` 工具栏按钮，如果我们想以排列组件一个在另一个上面，我们可以使用 `Lay Out Vertically` 工具栏按钮，一旦我们对这些组件组合好了我们就能再次用垂直、水平和网格布局工具布置这些组件组合，一旦我们有了一个布局组合集，我们就能点击窗体本身，使用其中一个布局按钮在窗体中来优化放置写写组件组合。

一些组件会在水平方向或者垂直方向或者在这两种方向上填充可用的空间，例如按钮和线编辑为填充水平方向的空间，虽然列表视图会在两种方向上充满整个空间，获得这种布局的最简单的方式是使用 Qt 设计器的布局工具，当你在某些位置上对一些部件使用了一个布局，这些部件的布局可能不如你意，如果一个部件没有充满整个空间，试着改变其大小策略(SizePolicy)扩展之，如果一个部件占用了太多的空间，一种方法是改变其大小策略，另一种方法是使用空间器(Spacer)消耗掉多余的空间。

在运行的窗体上，空间器(Spacers)没有可视化的显示，纯粹用于在部件之间和部件组合之间插入空间，设想你有一个占用了太多空间的部件，你能消除这种布局重新设置部件大小并为空间器留有空间，于是你可以插入空间器并将其和部件一起布局，空间器会消耗掉多余的空间。如果空间器不能有效地利用空间，你可以改变其大小策略以便更好控制。

学习布局 and 空间器工具的最好方法是试着用用他们，用着这布局工具试验是很简单的，如果你对你做的改动不是满意你也可以很简单的通过点击 Edit|Undo 菜单或者按下 Ctrl +Z 键撤销这些改动，下一步我们将一步一步地对我们的多项剪辑程序布局。

## 放置部件

布局工具提供了一个将部件和组件集以水平和垂直和网格方式组合的方法，如果你用布局工具布局了带有部件的窗体，那么当用户改变窗体大小时，这些部件也会随着自动的改变尺寸，这比你使用绝对的尺寸和位置更好，因为你不必写任何代码获得窗体规模大小，并且你的用户也能充分利用他们屏幕的大小，无论他们是一个笔记本或者一个有非常大的屏幕的台式机。布局工具使用标准尺寸设置边界和部件的空间大小，这有助于使你的程序看上去一致和匀称，而这不需要你做更多的工作，比起设置绝对位置而言，布局设计工具更容易也更快上手，你只需要在窗体上合适的位置放置你的部件并使用布局工具正确的设置部件的大小和规模。

## 选择和插入部件

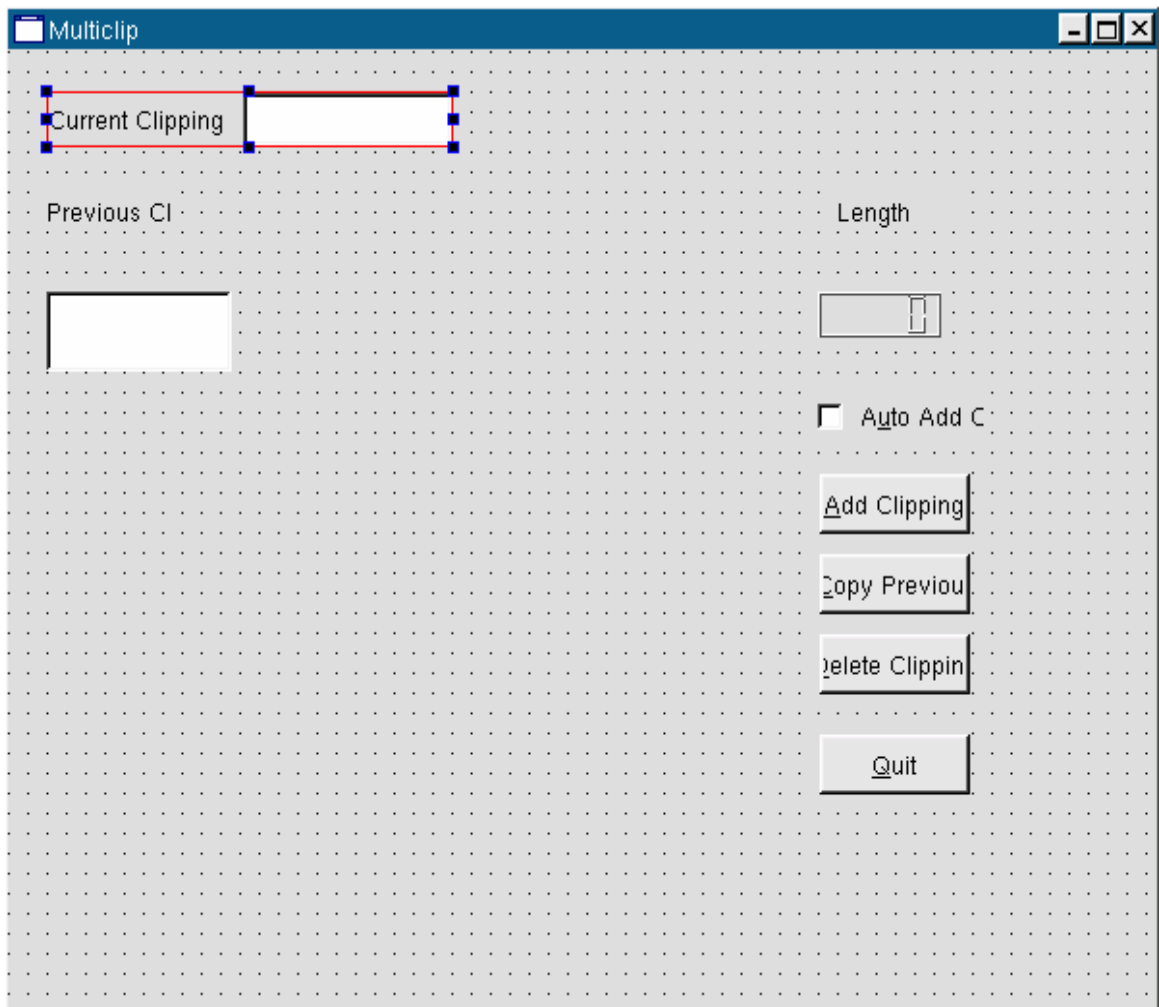
为了选择一个单独的窗口部件，或者点击该部件或者在对象浏览器窗口中点击他的名字，为了选择一组窗口部件或者点击他红色边缘线的一部分或者在对象浏览器窗口中的点击他的名字，为了选择多个部件或者多组部件，点击窗口取消任何选中的部件，然后按住 Ctrl 键，点击一个或者一组部件，然后拖动选中边框橡皮圈的边缘直到选中所有你想选中的部件或者组合部件。这项技术在选择位于另一个部件内部的部件时特别有用。例如选择一按钮组合(button group)中的单选按钮(radio button)，但是不选择该按钮组合本身，你可以点击窗体，然后按住 Ctrl+Click 键(Ctrl+鼠标点击)，选中其中一个单选按钮并拖动橡皮圈选中其他的单选按钮。

如果我们想插入一个部件到一个布局中部件之间的空隙时，我们可以点击工具栏按钮选择需要插入的新按钮，然后点击该空隙处即可。Qt Designer 会询问我们是否想打破这个布局，如果我们点击 Break Layout, 这个布局就会被打破并且我们新部件被插入了。于是我

们可以选择需要布局的部件和一组部件再次布局他们。通过点击这组部件或者点击 Break Layout 工具栏按钮或者按住 Ctrl+B 键可以达到同样的效果。

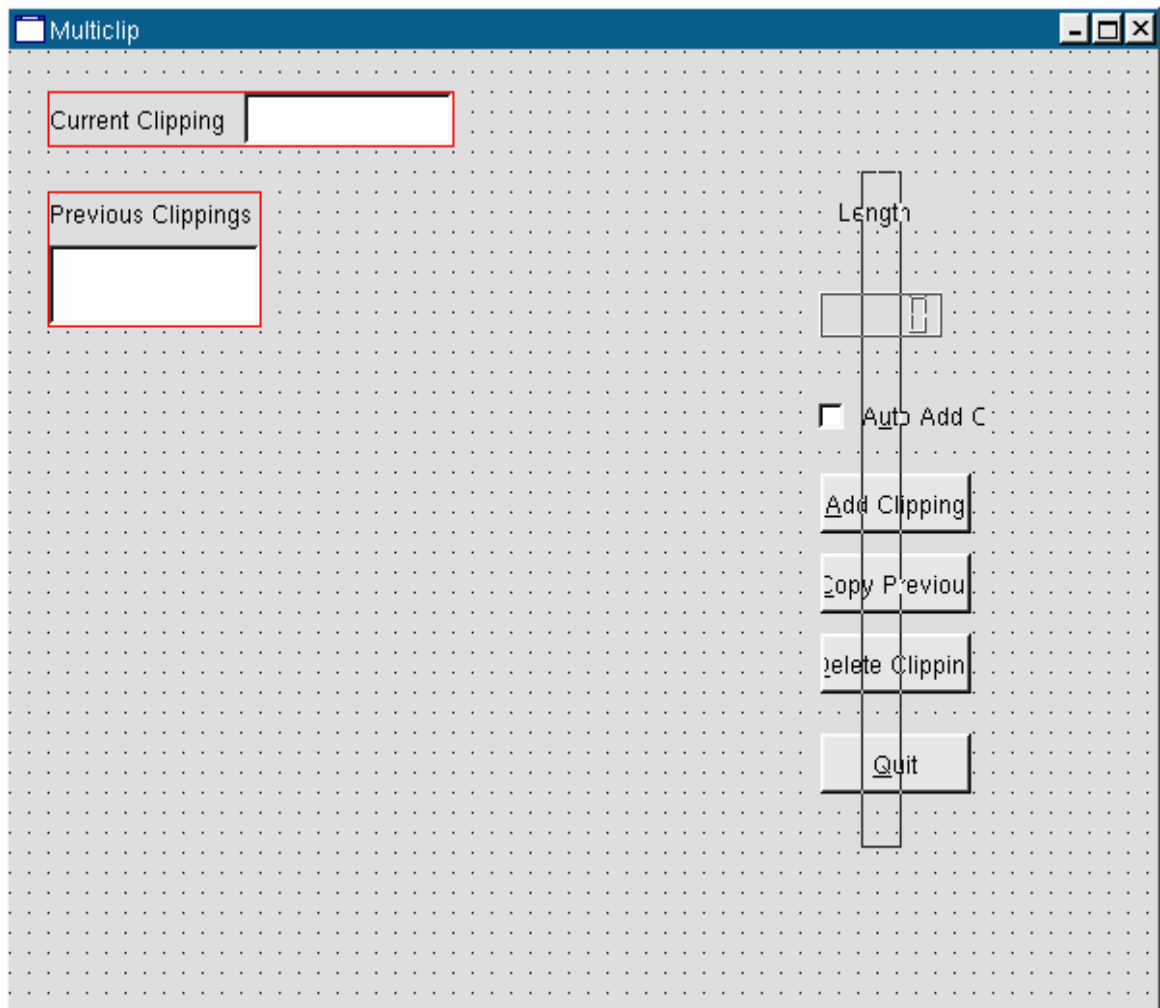
我们想要的布局方式是使当前剪辑标签和当前剪辑文本(currentLineEdit)并列放置于窗体顶部，使前一个剪辑标签和剪辑列表框(clippingListBox)充满左边的窗体，剩下的部件已列的形式放置在右边，我们想要使用一个分裂机将左右分开，并使默认情况下左边的空间大一点，部件得大小交给 Qt 设计器去做，布局控制在布局工具栏中（默认情况下，从左到右四个按钮）现在我们将布局放置在窗体上的部件。

1. 点击当前剪辑标签并按住 Shift 按钮，再点击 currentLineEdit 行编辑（Shift+Click 意思是按住 Shift 键同时点击，这可以使得 Qt 设计器能执行多项选择功能[这样的功能想必大家都会 ☺]）现在大多数布局按钮是可用的，点击 Lay Out Horizontally 水平布局工具栏按钮（如果你将鼠标停留在工具栏上，则会有个标志提示你该按钮的名字），这样这两个部件会被一细细的红色线条包围并可以一起移动，不要担心部件没有恰当的尺寸或者不再正确的位置上，因为我们能用 Qt 设计器正确的设置其大小和位置。



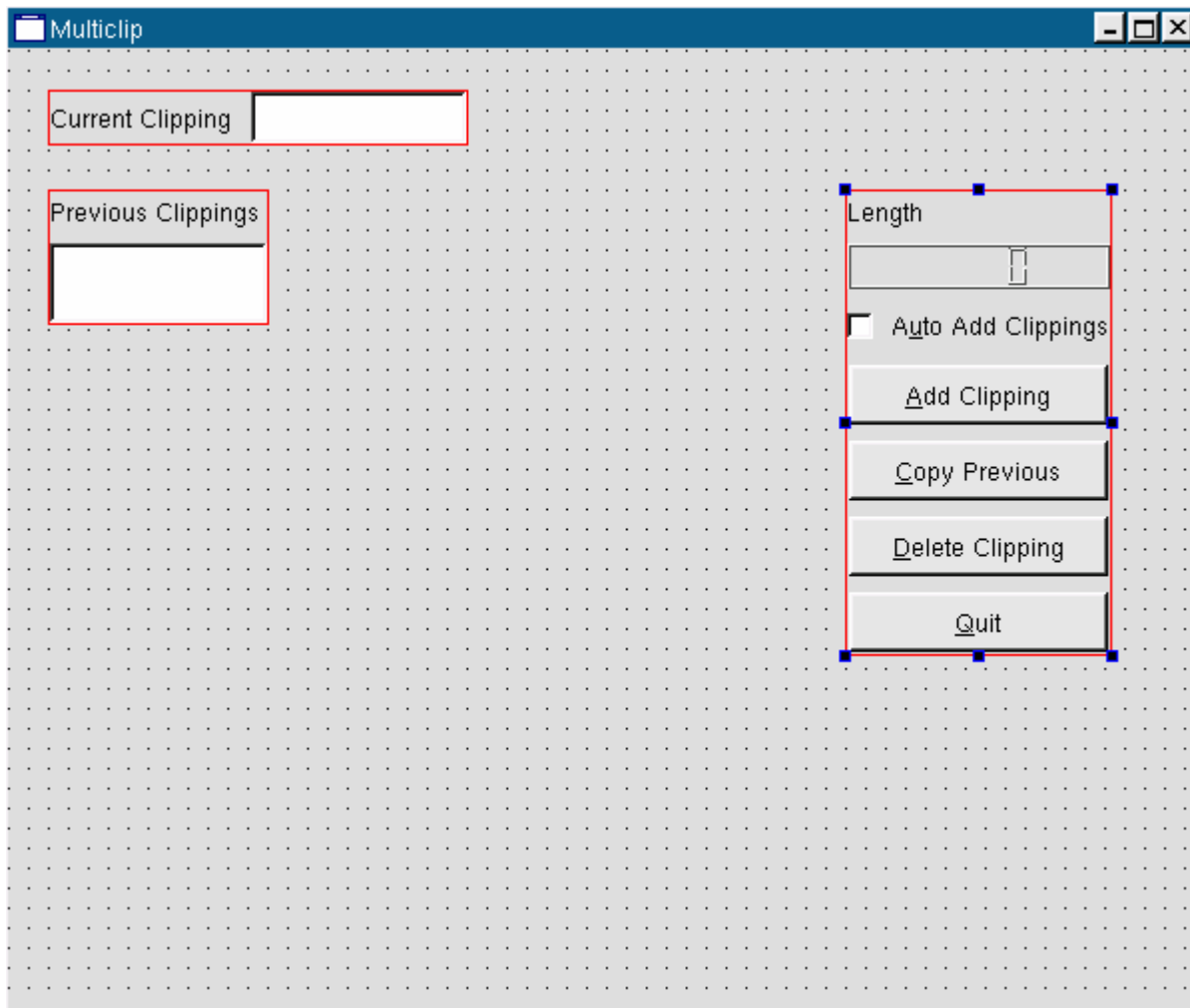
设置当前剪辑变迁和当前行编辑框

2. 点击前一个剪辑标签并按住 Shift 键，点击 ClippingListBox，点击 Lay Out Vertically 工具栏按钮。
3. 我们要使剩下的组件集合以垂直方式布局，不用按住 Shift 键一个一个点击部件，我们可以从窗体上部开始点击鼠标，拖动鼠标覆盖长度标签，LCD Number、检测框和所有的按钮，使得这些部件位于鼠标滑过的范围之内，释放鼠标，如果你没有做任何按住 Shift 并点击鼠标的操作，所有剩下的部件都会被选中，现在点击 Lay Out Vertically 工具栏按钮。



选中一组部件

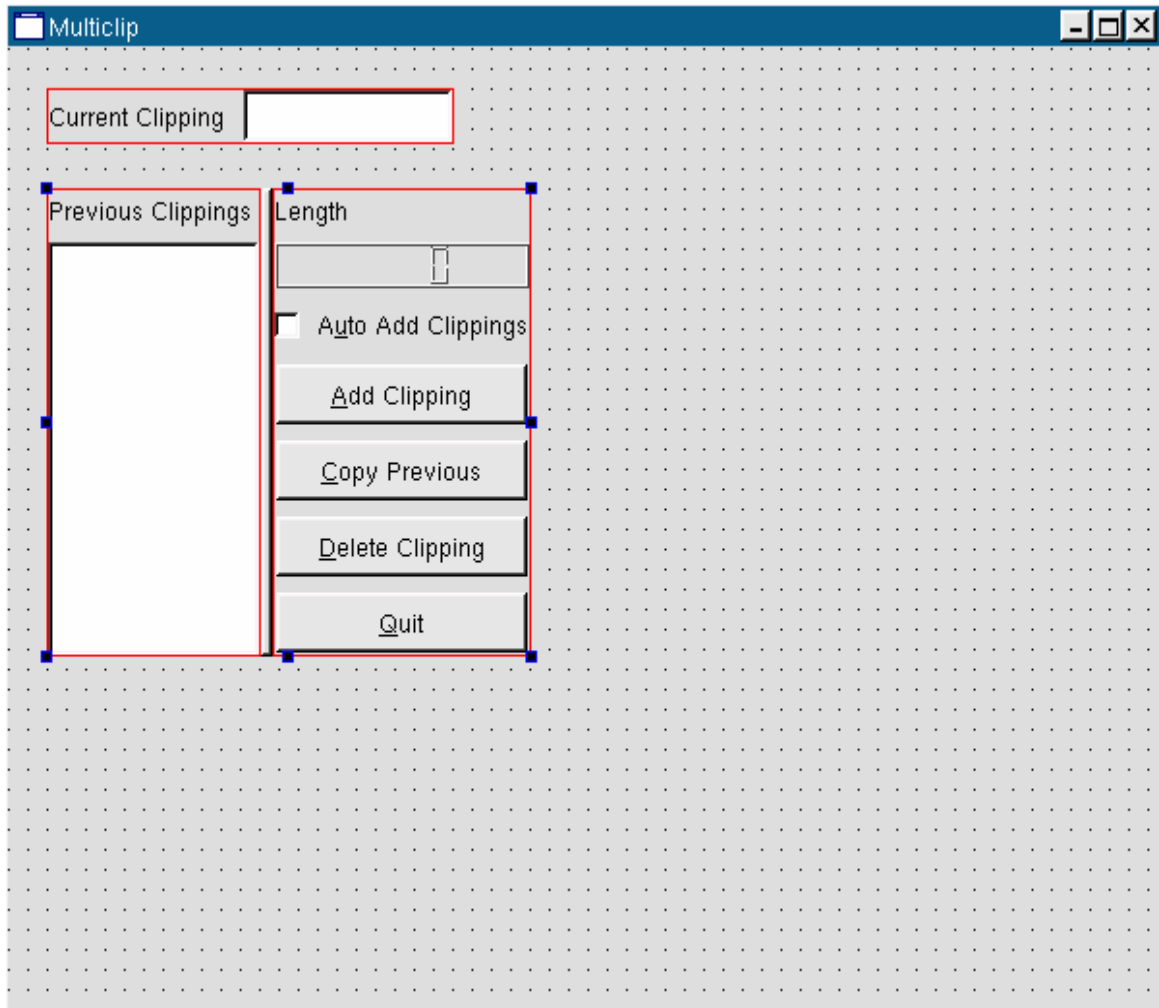
现在我们有三组必须相互联系的部件需要放置，并且各自和窗体本身相关。



## 布局部件分组

1. 按住 Shift 按钮，点击鼠标 (Shift+Click 组合键) 用于选中单独的部件，为了选中一组部件我们必须点击窗体取消任何选定的部件，然后按住 Ctrl 键，点击该组合知道选中所有你想选中并布局的部件然后释放鼠标，按钮和其他部件都已放置好并选中了，按住 Ctrl 键，点击列表框并拖动鼠标直至选中按钮，然后释放，这样两个组件集合将被选中，点击 Lay Out Horizontally (in Splitter) 工具栏按钮。(我想这部分我翻译得有点晦涩，但是大体意思我想每个热爱编程的人都能明白的，不是么？☺ 那就试试阿)

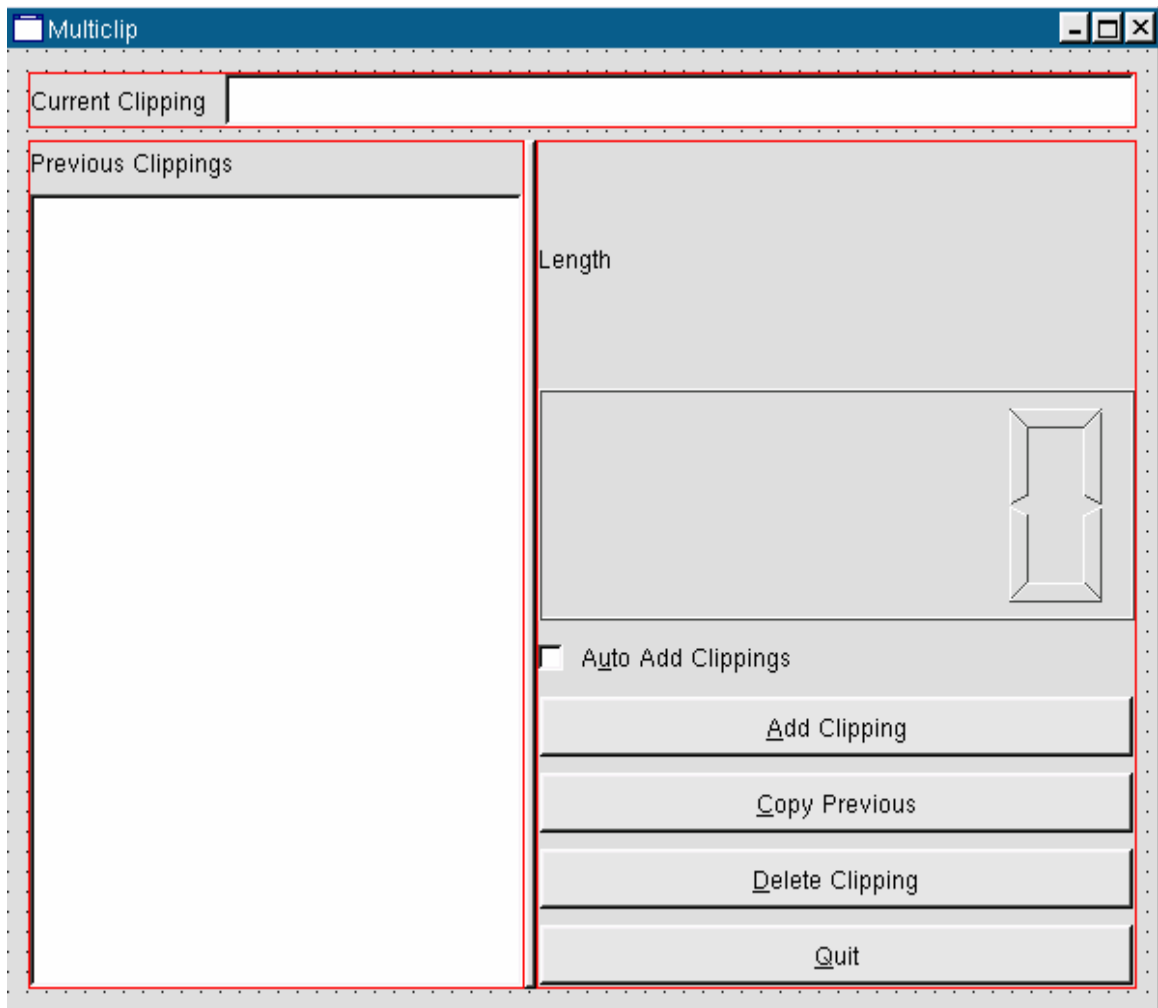




### 部件设置分组

2. 现在我们有两组了部件组合了，最上面的那个有一个当前剪辑标签和行编辑组件，以及我们刚刚创建的又一个列表框和一组按钮和其他部件。现在我们想要放置这些部件使得和窗体相关，点击窗体并点击 Lay Out Vertically 工具栏按钮，这些部件将被重新设置大小并充满整个窗体。

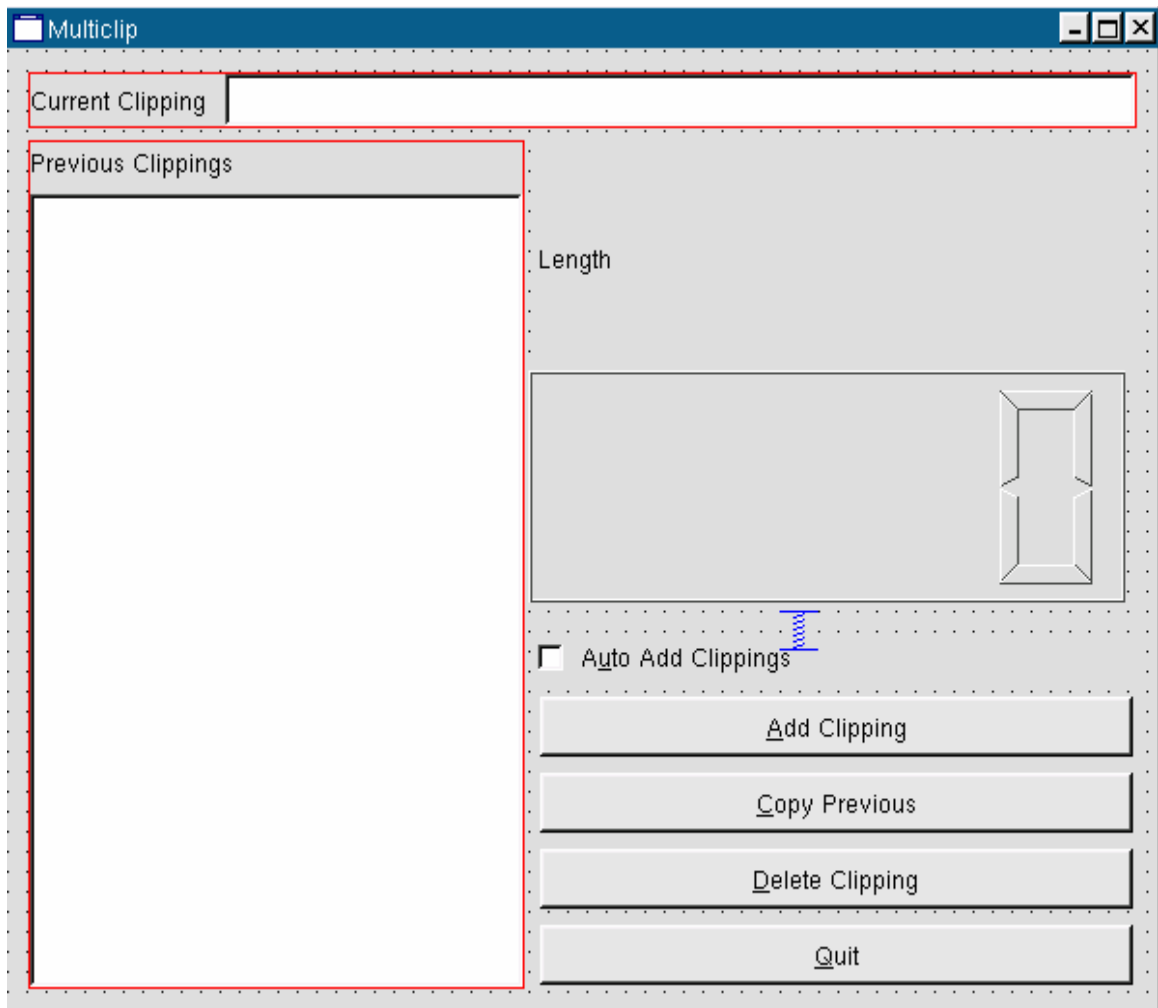




### 设置部件与窗体相关

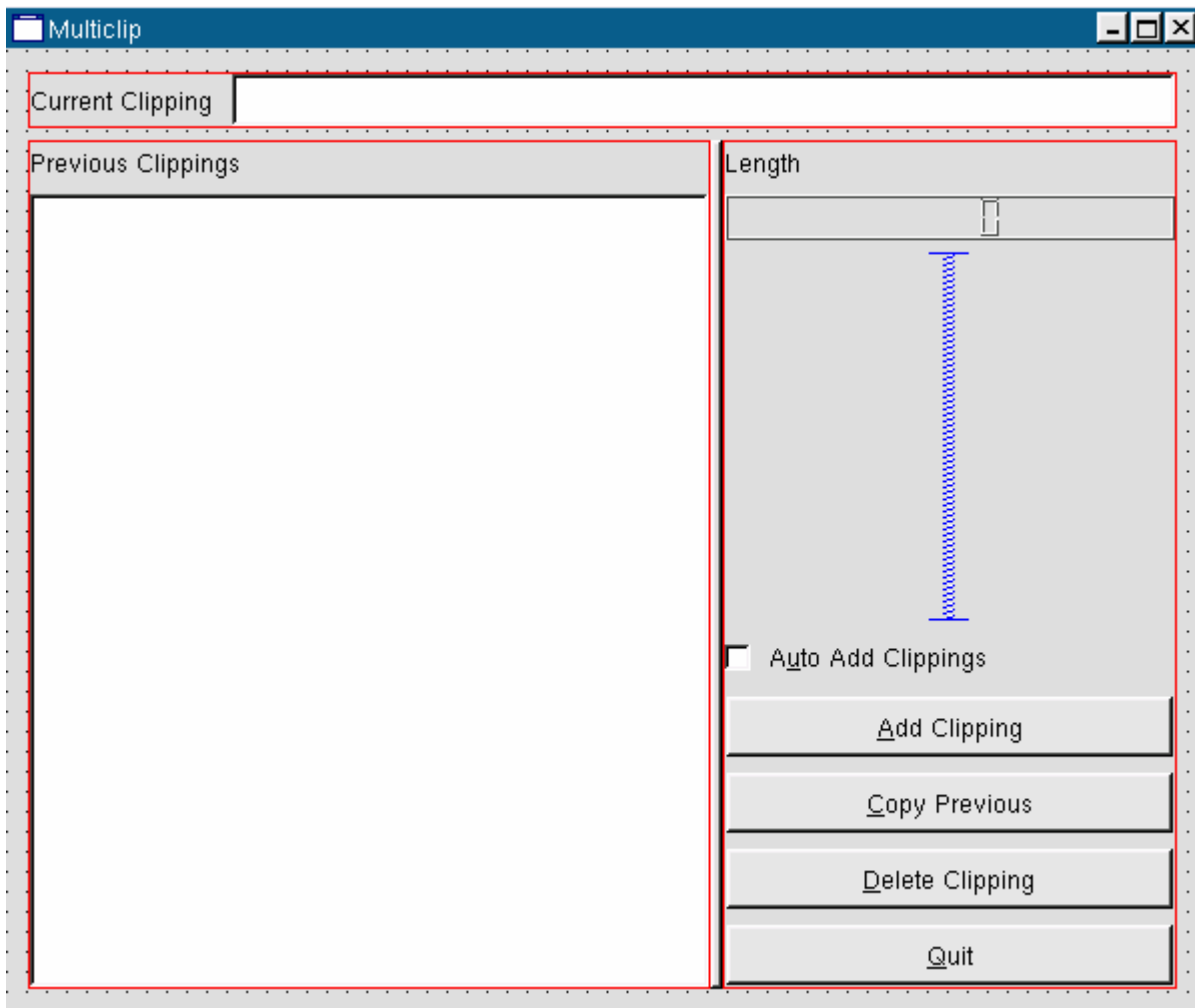
不幸的是，长度标签和 LCD Number（数字标签）占用了太多的空间，于是我们将不得不修订该布局，如果有经验你会发现你不必经常重新布局，我们可以插入空间器消耗掉额外的空间。

1. 首先我们选中需要填充的空间，点击 LCD Number(数字标签)选中它，点击 Break Layout 工具栏按钮，向上移动这个数字标签使其小一点，没必要十分精确，我们将在其下方构建一些空间。
2. 现在我们添加空间器，点击 Spacer 工具栏按钮，于是在数字标签和检测框之间你所创建的空间处点击该窗体，一个有两个选项的菜单弹出，水平和垂直，点击垂直，我们选择垂直是因为我们要空间器在垂直方向消耗多余的空间。



增加一个垂直的空间器

3. 我们需要重新组合这些按钮和其他部件为一个垂直的组件集合。拖动鼠标从窗体的底部，是的鼠标拖动范围可以达到这些按钮、检测框、空间器、数字标签和长度标签，然后释放鼠标，如果你忘记了选中其他组件，点击窗体并在此拖动鼠标直至正确选中所要选中的组件，点击 Lay Out Vertically 工具栏按钮。
4. 现在我们有如前面我们所有的那样的组合，只是添加了一些空间器。通过点击窗体选中列表框和这些按钮，拖动鼠标制止选中这两个部件组合，点击 Lay Out Horizontally(In SPlitter)用一个分裂器重新组合它们。
5. 最后一步是对窗体本身布局设置，点击该窗体并点击 Lay Out Vertically 工具栏按钮，窗体会被正确放置。



设置窗体

在我们得到的布局中有两个小的不足之处，首先是列表框和按钮占用了相等的宽度，尽管我们宁愿使列表框占用四分之三的宽度；其次是长度标签、检测框和按钮在右边延伸到了分裂器，如果在这里有一个小的空间器使它们与分裂器分开，看上去会更有魅力。

延伸列表框到分裂器的一半宽度处，这需要我们添加一个 `Init()` 函数，代码如下：

```
void MulticlipForm::init()
{
    QList< int > sizes;
    sizes << 250 << 40;
    Splitter->setSizes( sizes );
}
```

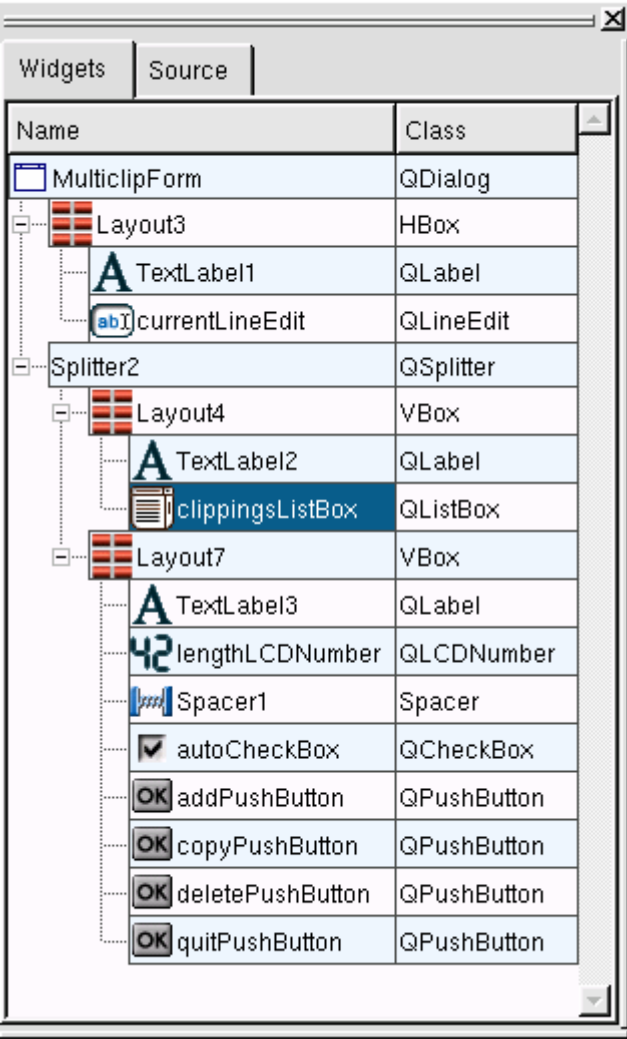
我们不想现在就添加这段代码，因为我们在本章后面将会讲述应用程序函数的实现。

我们通过改变布局组合的边缘，在分裂器周围创建一些空间，点击一个布局或者点击布局顶部红色线的一部分，或者在对象浏览器（部件和来源窗口）中点击布局的名字。（参

看对象浏览器部分，认识对象浏览器）。点击包含有列表框的部件，改变其 LayoutMargin 属性值为 6，并按下 Enter 键，点击包含有按钮和其他部件的布局，改变它的 LayoutMargin 属性值为 6，然后按下 Enter 键确认修改。

对象浏览器

通过点击 Window|Views|Object Explorer. 察看对象浏览器（部件和来源）窗口，对象浏览器有两个标签，组件标签显示了对应的层次，源码标签显示了你添加到窗体的代码，在部件标签下点击一个部件的名字，将会选中该部件，并且在属性编辑框显示其属性，在对象浏览器中查看和选择部件很容易的，特别是对那种有许多部件或者布局的窗体很有用。



对象浏览器

在 Qt 设计器的早期版本中，如果你想为一个窗体提供一份代码，你必须做个窗体的子集，并将你的代码放在子集中，这个版本的 Qt 设计器仍然支持子集的方法，但也提供了另外一种方法：直接将你的代码放在窗体中。在 Qt 设计器中编写代码方法和子集方法不尽相同，例如：你不能直接访问窗体的构造函数和析构函数，如果你的代码需要在构造函数中实现，你需要创建一个名为 void Init() 的槽函数；如果它存在了，它将在构造函数中调用，同样，如果你希望在析构函数执行前执行一段代码，你需要添加另外一个槽函数 v

oid destroy();你也可以添加增加你自己的类变量，这些变量将被放置在生成的构造函数代码中，你也可以添加前部声明和包含任何你需要的文件，为了增加一个变量或者声明，在源码标签下点击适当的子项，例如：类变量，于是点击新增加按钮并键入你的文本，例如 `QString filename`, 如果有一个或多个项存在，鼠标右键点击该项时，将会弹出一个有三个菜单子项的菜单，分别为 New、Edit 和 Delete，要编辑代码只需点击函数的名字激活代码编辑器，代码编辑和创建在后面详述。

如果你是对窗体进行子集，你要创建一个你自己的 .cpp 文件，该文件包含了你自己的构造函数，析构函数，函数，槽，声明和变量，这些都是你的需求指定的。（更多信息参看子集 (Subclassing) 信息）

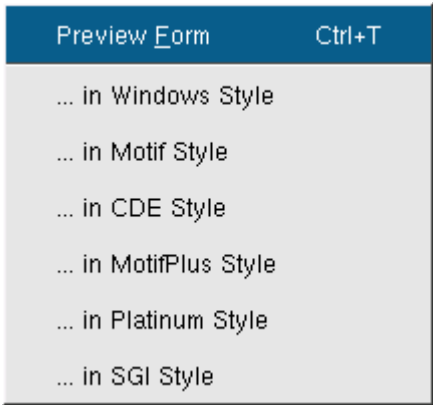
在例子中我们使用了 Qt 设计器布局工具设置我们的部件，在接下来的章节中的例子中我们将继续适用布局工具，如果你想使用绝对位置，也就是放置窗体和窗体大小以精确的像素尺寸定义，你能很容易办到，要放置一个部件，点击该部件并拖动到预期的位置，要改变其大小，点击该部件，拖拽大小调节框（有蓝色的方框显示）到合适的位置，要使得部件在窗口大小改变时停止部件尺寸改变，需要改变 `hSizeType` 和 `vSizeType`（这些包含在尺寸策略属性中）这两个属性值，为固定值。

预览

虽然 Qt 设计器能给出窗体的正确视图，但通常我们想看到在程序运行时窗体的样子，能检测出窗体的一些外貌这也是有用的，例如当窗体尺寸变化是如何比例缩放的，或者分裂器实际上是如何工作的，如果我们在创建一个多平台的应用程序，在不同的环境下预览窗体的样子这也是很有用的。

要想看到预览的样子，点击 `Preview|Preview Form` 或者按住 `Ctrl+T` 键，要离开预览模式已当前环境下的标准方式关闭窗口，为了查看应用程序在其他平台下的预览情形，可以点击 `Preview` 菜单并点击其中的菜单子项。

预览多行编辑窗体并试试分裂器和改变窗体的大小，很可能的是如果你将分裂器移动到右边减小按钮的尺寸会使得窗体更好看，这个分裂器看上去是个不错的主意，但实际上，我们需要这些按钮和其他部件在左边占用一个固定的空间，Qt 设计起使得改变这些布局非常容易，所以这样下去不会错的。（我是这么理解的）



点击分裂器然后点击 Break Layout 工具栏按钮，这个分裂器将会被移除，现在在窗体底部点击窗体本身，并拖动鼠标直至选中列表框和一些按钮，然后释放鼠标，这样列表框部件组合按钮部件组被选中了，点击 Lay Out Horizontally 工具栏按钮，点击窗体本身然后点击 Lay Out Vertically 工具栏按钮，这样窗体就按我们的要求布置好了，预览窗体（按下 Ctrl+T 键）并试着重新布置其大小。

既然这些布局工作是可视化的且最好的学习途径是实践，那么当你进一步实验时你会发现该工具很有用。点击 Break Layout 工具栏按钮移除一个布局，选择相关的部件和部件组并点击布局按钮实现一个布局，任何时候你都可以预览窗体的样式并撤销你所做的更改。

让我们做一个实验，来看看网格布局是如何工作的，点击列表框，并按住 Ctrl+B（破坏布局），点击一个按钮并点击 Ctrl+B 按钮，在底部点击窗体并拖动鼠标直至所有的部件被选中，（但是不要选中当前编辑标签和当前线编辑）；然后释放鼠标，点击 Ctrl+G（以网格方式布局）点击窗体，然后按下 Ctrl+L（以垂直方式布局）回到我们最初的设计——但是这次使用的是网格布局。

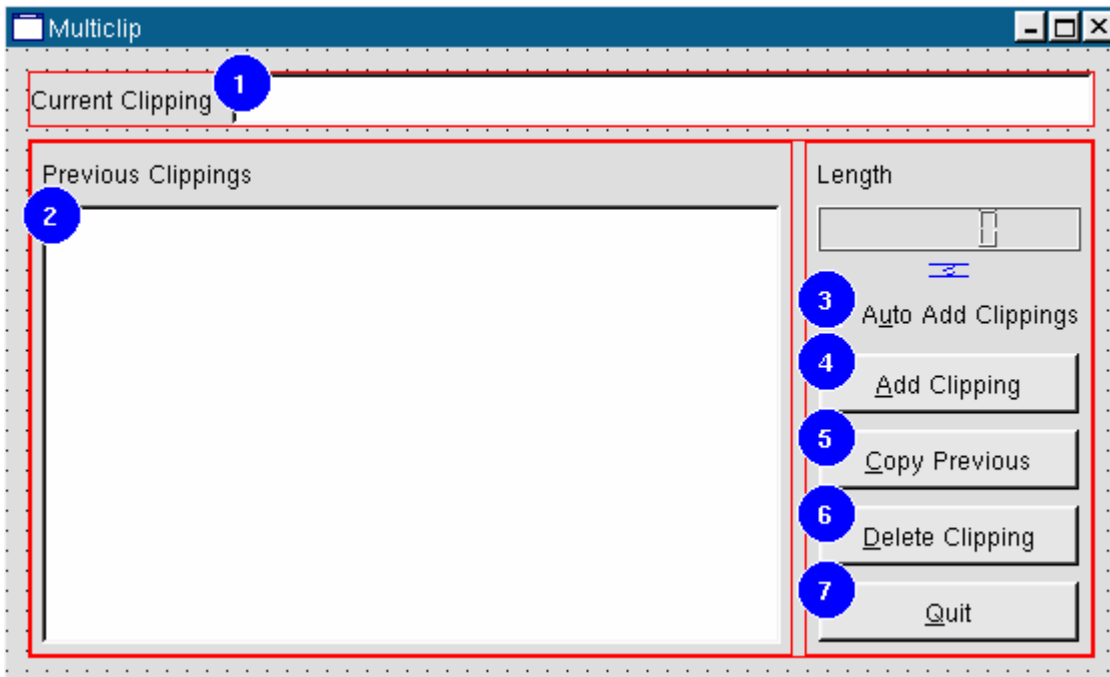
### 改变标签的次序

键盘用户按住 Tab 键然后在部件上一个一个的移动焦点，则焦点移动的顺序就是 Tab 顺序，预览多行编辑程序(按住 Ctrl+T 键)并试着对部件标签，这个标签的顺序可能不是我们所希望的，所以我们将标签模式下改变该标签次序为我们所要的顺序。

当你点击 Tab Order 工具栏按钮是，一个被蓝色圆圈包围的数字将会出现在每一个部件上，这些都能接受键盘焦点，（如果你按住 Ctrl 键同时按你的顺序点击每个部件，可以改变蓝色圆圈中数字的顺序，这一点和 VC 中的控件排序没有太大区别）。这个数字代表了每一个部件的标签顺序，起始值为 1。你可以点击这些部件改变这个标签顺序已满足你所要的新的标签顺序，如果你不小心操作错了，需要重新排序，双击该部件，这个部件的标签号码就会变成 1，于是你可以按需要需要的顺序点击其他部件，当你完成你的标签排序后你可以按下 ESC 键退出标签排序模式。如果你犯了点错，或者更喜欢修改前的标签顺序，你可以退出标签排序模式或者撤销操作（按下 Esc 键并按下 Ctrl+Z 键）来取消你刚才对标签顺序所作的修改。

点击 Tab Order 工具栏按钮，点击部件 当前剪辑线编辑(current clipping Line Edit)——即使该部件的标签号码就是 1，然后点击上次剪辑列表框(ListBox)部件，然后是点击检测框(CheckBox)，按次序从上到下（从增加剪辑到退出按钮）点击每一个按钮，按下 Esc 键完成标签模式下对标签顺序的更改，然后预览窗体并试着修改所有部件标签顺序。

注意到如果所有部件的标签顺序数字是正确的，你可以停止点击部件，只需按下 Esc 键离开标签顺序模式。



设置标签顺序

## 连接信号和槽

Qt 为两个部件之间的通讯提供了信号和槽的机制，当一个特定的事件发生时部件发射信号，我们可以将信号连接到槽，这些槽可以是预先定义的也可以是我们自己创建的槽，在以前的开发包中，这种通讯机制的实现是靠回调函数实现的。（要想知道 Qt 的信号和槽机制的详细解释请参看在线的信号和槽文档）

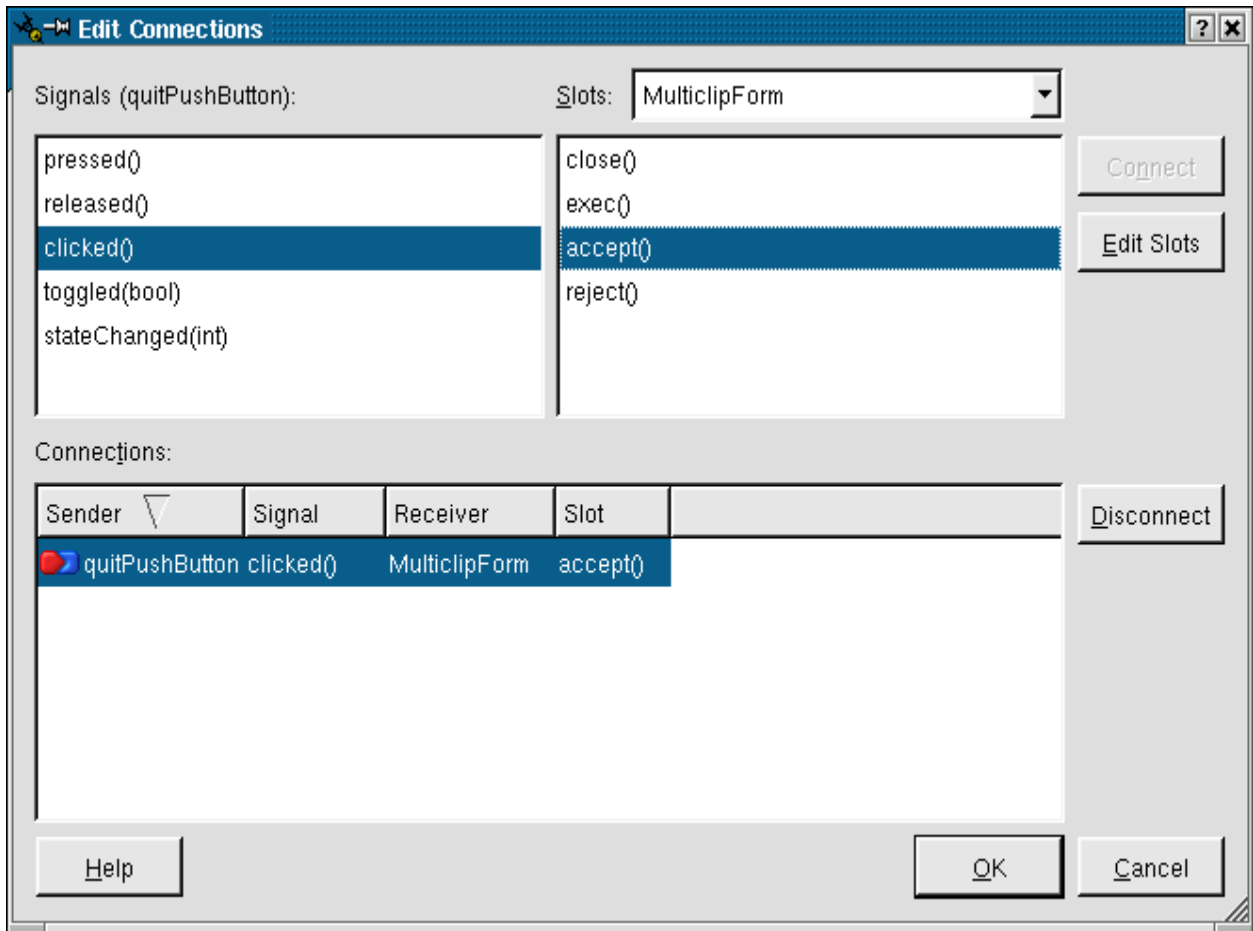
### 连接预先定义的信号和槽

一个应用程序的一些功能可以简单的通过连接预定义的信号和槽来实现。在这个多项剪辑的程序中仅仅只有一个预定义的连接可为我们所用，但是在 richedit 这个应用程序中我们将创建一个带有 Actions, ToolBar 和 Menu 的主窗口，那时我们将用到许多预定义的信号和槽来实现我们所需的大多数功能而不需要些任何代码。

我们将连接退出按钮的 `clicked()` 信号到窗体的 `accept()` 槽，这个 `accept()` 槽通知对话框的调用者（`dialog's caller`）对话框不再需要了，因为我们的对话框是我们程序的主窗口，这样就退出了这个程序，预览该窗体（`Ctrl+T`）；点击退出按钮，这个按钮处了显示出来外不任何事情，按下 `Esc` 键或者关闭窗口退出预览模式。

点击连接信号和槽（`Connect Signals/Slots`）工具栏按钮，然后点击退出按钮后释放鼠标（点击连接信号和槽工具栏后将鼠标移动到窗体上时，鼠标形状变为十字型，如果点击窗体或者窗体上的某个部件则会出现信号和槽的编辑对话框），（信号和槽的）连接编辑对话框就显示出来了，左上方的列表框列出了我们点击的部件所能发射的信号，右上方的组合框列出了窗体和窗体上的部件，这些都是接收信号的对象，由于我们是在窗体上

而不是在部件上释放鼠标，槽组合框显示的是窗体的名字，“MulticlipForm”，在组合框下方是列表框，显示了窗体上可用的槽或者组合框中是部件时显示的是部件可用的槽。注意到只有这些槽能连接到高亮显示的信号，如果你点击选中了不同的信号，比如是 `toggle d()` 信号，则可用的槽列表将会随之改变，点击信号，然后点击在连接列表中将显示的连接，点击 OK 按钮确认。



连接 `clicked()` 信号到 `accept()` 槽

当我们运行这个例子程序是我们将实现多个信号和槽的连接，包括连接我们自定义的槽，信号和槽连接（使用预定义的信号和槽）在预览模式下可以工作，按下 `Ctrl+T` 键预览窗体，点击退出按钮，此时按钮能正确工作了（即关闭对话框退出了程序）

## 创建和连接自定义的槽

在第一版的 Qt 设计器中，你可以创建你自定义的槽的信号并使他们连接起来，但是你不能直接实现你的槽，你不得不予集于该窗体，并在这个子集中对你自定义的槽编码，子集的方法依然有用，在某些情况下仍起作用 (Make sense)，但是现在你可以在 Qt 设计器中直接实现你的槽，在多的对话框和窗体的子集也不再需要了。（Qt 设计器在 `.ui.h` 文件中存贮者槽的实现，具体的细节参看 走近设计器 (*The Designer Approach*) 中的 `.ui.h` 扩展方法）。



多项剪辑应用程序需要四个槽，一个用于每个按钮，因为我们将一个信号连接到预定义的槽使得按下退出按钮时退出程序，所以仅有三个槽需要我们自定义，我们需要增加剪辑按钮实现一个槽，其作用是将当前剪辑添加到列表框中，复制以前的按钮(Copy Previous)需要一个槽实现复制列表框中选中的项到线型编辑框中（也复制到剪贴板上），删除剪辑按钮>Delete Clipping)需要一个槽实现删除党旗那剪辑和列表框中的当前项，我们也需要写一些初始化的代码使得程序启动时，将当前剪贴板上的文本（如果有的话）复制到线型编辑框中，代码可以直接在 Qt 设计器中编写，声称的已写好的代码片断可以下面的文件中获取：[qt/tools/designer/examples/multiclip/multiclip.ui.h](http://qt/tools/designer/examples/multiclip/multiclip.ui.h)

我们需要一个 Qt 的全局剪贴板对象，在整个代码中有几个地方需要调用其含义是调用 `QApplication::clipboard()` 或者是 `qApp->clipboard()`，比执行这些函数调用更好的方法，我们在窗体本身添加一个指向剪贴板的指针，点击对象浏览器中的源码标签（如果对象浏览器部可见，点击 Windows|Views|Object Explorer [由于 Qt 设计器没有汉化，再说汉化后似乎失去了其魅力，所以菜单的引用依然是 English]）源码标签显示了当前窗体中的函数、类变量、前部的声明和我们需要的一些包含文件的名字。

鼠标右键点击类变量(Class Variable)项(可能在底部，你需要点击滑动条)，然后再弹出的菜单中选择 New 子菜单，(如果存在变量，则弹出来的菜单中会有一个“Delete”删除的选项)键入 `QClipboard * cb` 并按下 Enter 键确认输入，我们将创建一个 `init()` 函数，在这个函数中我们会赋予这个指针一个 Qt 的全局剪贴板对象的值。我们也需要声明剪贴板对象的头文件，右键点击包含（声明文件）[在 Qt 设计器中对应的为 Include (in Declaration) 项]，然后再弹出的菜单上选择 New，键入“”，并按下 Enter 键确认输入，因为我们需要一个全局对象，`qApp`，我们就必须包含另一个声明文件，鼠标右键点击 包含（实现）项[在 Qt 设计器中对应为 Include (in Implementation)]，然后点击 New 菜单项，键入“”并按下 Enter 键确认输入，这个变量和声明文件将被 Qt 设计器的 .ui 文件生成后包含（从 .ui 文件生成的源文件 .h 和 .cpp 文件会包含这个变量和声明文件）。

We will invoke *Qt Designer's* code editor and write the code. 现在我们将激活 Qt 设计器的代码编辑器并编写代码。

首先我们创建一个 `init()` 函数，激活代码编辑器的一种途径是点击 Source 标签，然后点击你想编写代码的函数的名字，如果没有你需要的函数或者你想创建一个在 Source 标签中你可用的新函数，鼠标右键点击 Source 标签的槽(Slots)列表中的“protected”子项，然后点击 New 子菜单，出现一个“Edit Slots”编辑槽的对话框，改变槽的名字为 `Init()`，然后点击 OK，这样你可以点击编辑窗口中出现的函数名字并键入你的代码。

注意到并不是强迫你使用 Qt 设计器的代码编辑器，在 Qt 设计器中你可以尽管的增加、删除或者重命名你的槽，你也可以用一个外部的编辑器编写的实现代码，Qt 设计器会保存你所写的代码。下面是你需要实现的 `init()` 函数：

```
void MulticlipForm::init()
{
```

```

lengthLCDNumber->setBackgroundColor( darkBlue );
currentLineEdit->setFocus();

cb = qApp->clipboard();
connect( cb, SIGNAL( dataChanged() ), SLOT( dataChanged() ) );
if ( cb->supportsSelection() )
    connect( cb, SIGNAL( selectionChanged() ), SLOT( selectionChanged()
) );

dataChanged();
}

```

函数体中开始的两行改变了数字指示器的背景颜色并使窗体的启动焦点在线型编辑框中，我们使用了一个指向 Qt 全局剪贴板的指针，并保存在我们定义的类变量 cb 中，我们连接剪贴板的 dataChanged() 信号到一个叫做 dataChanged() 的槽，这个操我们马上就会创建，如果剪贴板支持选择（例如在 X Windows 系统中），我们也要连接剪贴板的 selectionChanged() 信号到一个我们将创建的具有相同名字的槽，最后我们将调用我们的 dataChanged() 槽函数，当程序启动时，将当前剪贴板中的文本内容（如果有的话）粘贴但线型编辑框中

既然我们提到了 dataChanged() 和 selectionChanged() 槽，下面我们将对他们编码，从 dataChanged() 开始：

```

void MulticlipForm::dataChanged()
{
    QString text;
    text = cb->text();
    clippingChanged( text );
    if ( autoCheckBox->isChecked() )
        addClipping();
}

```

我们复制了剪贴板的文本并用我们获得文本调用我们自己定义的 clippingChanged() 槽，如果用户选中了自动增加剪辑检测框，我们将调用 addClipping() 槽增加该剪辑到列表框中。

下面的代码只是在 X Windows 系统中可用，微软的 Windows 用户也能包含下面的代码确保该应用程序可以在多平台下运行。

```

void MulticlipForm::selectionChanged()
{
    cb->setSelectionMode( TRUE );
    dataChanged();
    cb->setSelectionMode( FALSE );
}

```

上面的代码中我们首先告诉剪贴板使用选择模式，我们调用 `dataChanged()` 槽获得任何选中的文本，然后设置剪贴板为默认模式。

In the another custom slot, `clippingChanged()`. 在另一个自定义的槽中, `clippingChanged()`:

```

void MulticlipForm::clippingChanged( const QString & clipping )
{
    currentLineEdit->setText( clipping );
    lengthLCDNumber->display( (int)clipping.length() );
}

```

我们用任何传递给 `clippingChanged()` 槽的文本设置当前的线型编辑框, 并用这个新文本的长度更新数字指示器。

我们将要编码实现的下一个槽实现增加剪辑功能, 当用户点击增加剪辑按钮时, 这个槽被我们的代码内部调用 (看上面的 `dataChanged()` 槽), 即使我们让 Qt 设计器能够通过编辑窗口键入代码来创建一个槽与信号的连接, 我们也让 Qt 设计器为我们创建一个结构, (为我们生成程序框架), 点击 Edit|Slots 激活编辑槽 (Edit Slots) 对话框, 点击 New Slot 按钮并把槽的缺省名字 “`new_slot()`” 改为 “`addClipping()`”, 这里不必改变访问标着和返回类型, 现在我们已经创建了我们的槽, 我们能在代码编辑器中出现的地方创建我们的槽并实现之。

增加剪辑按钮用于从当前剪辑线型编辑框中复制文本到列表框中, 同时我们更新了文本长度数字指示器。

```

void MulticlipForm::addClipping()
{
    QString text = currentLineEdit->text();
    if ( ! text.isEmpty() ) {
        lengthLCDNumber->display( (int)text.length() );
        int i = 0;
        for ( ; i < (int)clippingsListBox->count(); i++ ) {
            if ( clippingsListBox->text( i ) == text ) {
                i = -1; // Do not add duplicates
            }
        }
    }
}

```

```

        break;
    }
}
if ( i != -1 )
    clippingsListBox->insertItem( text, 0 );
}
}

```

如果有新文本需要显示，同时改变数字指示器的值为当前文本的长度，然后我们在列表框中逐个比较文本项，看当前输入的文本是否存在列表框中，如果列表框中不存在该剪辑文本项，就在列表中插入该剪辑文本。

为了使增加剪辑按钮工作，我们需要连接这个按钮的 `addClipping()` 槽，点击 `Connect Signal/Slots` 工具栏按钮，然后点击增加剪辑按钮，（将变成十字型的鼠标拖动到窗体上点击该按钮然后释放），（确认你拖向窗体而不是其他部件—拖动过程中窗体会会有一个细红色边界，如果你操作错误你只需在槽组合框中改变其名字就可以了[这里我按原文翻译，可是有些地方和单词觉得上下文不对，如果你发现了正确的请与我联系]）。连接编辑对话框将会显示，点击 `addClipping()` 槽，点击 OK 按钮确认该连接。

复制前一个文本按钮（`Copy Previous`）用于从列表框中复制选中的文本项到线型编辑框中，同时该文本也放到了剪贴板上，剩下的过程和增加剪辑按钮（`Add Clipping`）是一样的：首先我们创建一个槽，然后我们实现该槽，最后我们将按钮点击的信号和槽连接起来。

## 1. 创建槽：

点击 `Edit|Slots` 菜单子项，激活槽编辑（`Edit Slots`）对话框，点击 `New Slots` 并将缺省的槽函数名字“`new_slot()`”改变为“`copyPrevious()`”，点击 OK 确认。

## 2. 实现槽函数

```

3.         void MulticlipForm::copyPrevious()
4.         {
5.             if ( clippingsListBox->currentItem() != -1 ) {
6.                 cb->setText( clippingsListBox->currentText() );
7.                 if ( cb->supportsSelection() ) {
8.                     cb->setSelectionMode( TRUE );
9.                     cb->setText( clippingsListBox->currentText() );
10.                    cb->setSelectionMode( FALSE );
11.                }
12.            }

```

13.            }

这段复制上次剪辑的代码检测列表框中是否有选项选中，如果有选项被选中则该项会被复制到线型编辑框中，如果我们使用了一个支持选择的系统，我们将不得不重复复制，复制的次数和选择模式有关，我们不要显示的更新剪贴板，当线型编辑框的文本改变时，他发射了一个 `dataChanged()` 的信号，这个信号被我们创建的 `dataChanged()` 槽接收，我们的这个 `dataChanged()` 槽同时更新剪贴板。

#### 14. 连接到槽

点击 `Connect Signal/Slots` 工具栏按钮，点击复制上次剪辑 (`Copy Previous`) 按钮，拖动到窗体然后释放鼠标，（光标变成十字型后，点击某个部件 1，如按钮，拖动鼠标到另一个部件 2 或者窗体，则在出现的信号槽连接编辑对话框中，信号的发射者是部件 1，接受信号的槽是部件 2 或者窗体的[试试就知道了 ☺]），在弹出的连接编辑对话框中，点击 `clicked()` 信号和 `copyPrevious()` 槽，点击 OK 确认连接编辑。

用同样的方法我们对删除剪辑 (`Delete Clipping`) 按钮进行信号和槽的连接编辑。

1. 点击 `Edit|Slots` 菜单激活槽编辑对话框，点击 `New Slot` 按钮并替换缺省的槽名字为 “`deleteClipping()`”，点击 OK 按钮。

2. 删除按钮必须删除列表框中的当前项并清除线型编辑框中的文本。

```
3.         void MulticlipForm::deleteClipping()
4.         {
5.             clippingChanged( "" );
6.             clippingsListBox->removeItem( clippingsListBox->currentItem
7.             ( ) );
8.         }
```

我们以一个空字符串调用我们自己创建的 `clippingChanged()` 槽，使用列表框的 `removeItem()` 函数删除当前选项。

8. 连接删除剪辑 (`Delete Clipping`) 按钮的 `clicked()` 信号到我们创建的 `deleteClipping()` 槽，（按下 F3 键——这个点击 `Connect Signals/Slots` 工具栏按钮效果一样，点击删除剪辑按钮并拖动鼠标到窗体上，释放鼠标，出现连接编辑对话框，点击 `clicked()` 信号和 `deleteClipping()` 槽，点击 OK() 确认连接编辑）

## 编译和生成应用程序

至此在 Qt 设计器中，我们写了一个 Qt 整个应用程序大约 99%的代码，为了使应用程序编译和运行我们必须创建一个 main.cpp 文件，在这个文件中我们将显示调用我们的窗体。

创建一个新的源文件的简单方法是点击 File|New 激活“New File”新文件对话框，适当的点击“C++ Source”或者“C++ Header”图标，然后点击“OK”按钮确认，一个新的空源文件显示出来，点击 File|Save 激活另存为对话框（Save As），键入“main.cpp”然后点击 Save 按钮保存该文件。

在 main.cpp 这个 C++编辑窗口，键入以下代码。

```
#include <qapplication.h>
#include "multiclip.h"

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

    MulticlipForm clippingForm;
    app.setMainWidget( &clippingForm );
    clippingForm.show();

    return app.exec();
}
```

这个程序创建了一个 QApplication 对象和我们的多项剪辑窗体的实例，并将该窗体设置为主部件并显示该窗体，app.exec() 开始调用程序事件循环。

现在启动控制台命令（命令行格式），并将当前目录切换到 multiclip 程序下，运行 qmake 命令，一个与你的系统兼容的 Makefile 文件将会生成。

```
qmake -o Makefile multiclip.pro
```

现在你可以通过运行 `make` 或者 `nmake` 命令生成该应用程序。试着编译和运行 `Multiplic`，你会取得许多的进步，使用布局工具和编写代码的试验可以帮助你学到关于 Qt 和 Qt 设计器的更多的东西。

这一章向你介绍了使用 Qt 设计器创建一个跨平台的应用程序，我们创建了一个窗体并且用部件对其修饰，这些部件整齐优美的放置在窗体上并可以随着窗体缩放，我们已经使用了 Qt 的信号和槽的机制实现程序的功能，并生成了 `Makefile` 文件，这些增加部件到窗体并用布局工具对其放置，以及创建、编写和连接槽的技术，当你再次用 Qt 设计器创建程序是会用到，下面的章节会给出进一步的例子并介绍使用 Qt 设计器的更多技术。