

C++ GUI Programming with Qt 4 中文版第一章至第十章

C++ GUI Programming with Qt 4

By Jasmin Blanchette, Mark Summerfield

.....

Publisher: Prentice Hall

Pub Date: June 21, 2006

Print ISBN-10: 0-13-187249-4

Print ISBN-13: 978-0-13-187249-3

Pages: 560

学习，使用 Qt 已经将近两年了，最开始的时候用 Qt3，后来升级到 Qt4.1，自认为对 Qt 的 GUI 编程部分还是很熟悉的。遗憾的是 Qt 的参考书少之又少，一般就是看文档和 C++ GUI Programming with Qt 3，这本书是免费提供的，让我对 Qt，Trolltech 公司有了更多的了解。我这个人就是这样，学习什么都喜欢了解这个东东发展的历史，虽然对学习并没有太多帮助。

升级到 Qt4 以后，API 有了很多变化，苦盼一本系统的参考书，很久，C++ GUI Programming with Qt 4 才得以面世，终于盼到后却遗憾的发现这本没有提供免费的电子版，今天，一个偶然的的机会，得到 Yimin 网友的帮助，得到一份电子版，真是很谢谢他，他的 blog<http://liyimin.net/blog>。

C++ GUI Programming with Qt 4 就是这样一本参考书，从易到难，从最经典的 hello Qt 开始，到构建复杂的程序。我从现在开始阅读学习，同时也把部分心得写出来，发到这里，也希望对 Qt 的学习者们有所帮助。这对我来说有点难度，懒人那，不过一定尽最大努力坚持写完我熟悉的部分。

加油！

目 录

第一章 Qt 发展小史.....	1
1-1 从 Hello Qt 开始.....	2
1-2 连接信号和响应函数.....	3
1-3 控件的几何排列—Laying Out Widgets	4
第二章 创建对话框 (Creating Dialogs)	7
2-1 派生对话框类 (Subclassing QDialog)	7
2-2 深入信号和槽 (Signals and Slots in Depth)	12
2-3 快速设计对话框 (Rapid Dialog Design)	15
2-4 能够改变的对话框(Shape-Changing Dialogs)	20
2-5 动态对话框 (Dynamic Dialogs)	23
2-6 Qt 提供的控件和对话框类 (Built-in Widget and Dialog Classes)	25
第三章 创建主窗口 (Creating Main Windows)	27
3-1 继承 QMainWindow 类(Subclassing QMainWindow)	27
3-2 创建菜单和工具条 (Creating Menus and Toolbars)	33
3-3 创建状态条 (Setting Up the Status Bar)	38
3-4 实现文件菜单 (Implementing the File Menu)	40
3-5 使用对话框(Using Dialogs)	50
3-6 存贮设置 (Storing Settings)	57
3-7 多文档 (Multiple Documents)	59
3-8 启动画面 (Splash Screens)	62
第四章序及第三章小节	64
4-1 中央控件 (The Central Widget)	64
4-2 从 QTableWidget 继承 (Subclassing QTableWidget)	65
4-3 读取和保存 (Loading and Saving)	72
4-4 实现 Edit 菜单 (Implement the Edit menu)	75
4-5 实现其他菜单项 (Implementing the Other Menus)	81
4-6 继承类 QTableWidgetItem (Subclassing QTableWidgetItem)	86
第五章用户自定义控件 (Creating Custom Widgets) 及第四章小结 ..	97
5-1 自定义 Qt 控件 (Customizing Qt Widgets)	97
5-2 从 QWidget 继承新类 (Subclassing QWidget)	99
5-3 把自定义控件集成到 Qt Designer 中 (Integrating Custom Widgets with Qt Designer)	112
5-4 双缓冲技术 (Double Buffering) (1、简介和源代码部分)	116
5-5 双缓冲技术 (Double Buffering) (2、公有函数实现)	131
5-6 双缓冲技术 (Double Buffering) (3、事件处理函数)	136
5-7 双缓冲技术 (Double Buffering) (4、私有函数的实现)	144

5-8 双缓冲技术 (Double Buffering) (5、类 PlotSettings 实现)	148
第六章 序-布局管理 (Chapter 6. Layout Management)	151
6-1 排列窗体上的控件 (Laying Out Widgets on a Form)	151
6-2 分组布局 (Stacked Layouts)	158
6-3 分隔控件 (Splitters)	161
6-4 滚动区域 (Scrolling Areas)	165
6-5 可停靠控件和工具栏 (Dock Widgets and Toolbars)	167
6-6 多文档界面 (Multiple Document Interface)	171
第七章 (序) 事件处理- (Event Processingn)	183
7-1 重写事件处理函数 (Reimplementing Event Handlers)	183
7-2 安装事件过滤器 (Installing Event Filters)	190
7-3 系统繁忙时的响应 (Staying Responsive During Intensive Processing)	193
第八章 序 2D 和 3D 图形系统 (2D and 3D Graphics)	197
8-1 用 QPainter 绘图 (Painting with QPainter)	197
8-2 坐标变换 (Painter Transformations)	203
8-3 使用 QImage 进行高质量绘制 (High-Quality Rendering with QImage)	215
8-4 打印 (Printing)	217
8-5 用 OpenGL 绘图 (Graphics with OpenGL)	228
第九章	239
9-1 支持拖拽功能 (Enabling Drag and Drop)	239
9-2 支持自定义数据类型的拖拽 (Supporting Custom Drag Types)	245
9-3 处理剪贴板 (Clipboard Handling)	252
第十章 数据视图类 (Item View Classes)	254
10-1 使用数据视图便捷类 (Using the Item View Convenience Classes)	255
10-2 使用已有的模型类 (Using Predefined Models)	264

第一章 Qt 发展小史

Qt 的创建者 Haarard Nord (Trolltech 公司的 CEO) 和 Eirik Chambe-Eng (Trolltech 公司的总裁) 是一家瑞典公司的同事。那时 (1990) 他们在做一个项目, 这个项目需要在 Unix, Macintosh, Windows 上运行同一个 GUI, 象我们现在的开发人员一样, 工作的很累, 当时可是没有如今这么多的开发工具。一天他们工作之余去公园散步, 晒太阳, 喝咖啡。Haarard 说: "We need an object-oriented display system." 这成为了后来 Qt 最重要的思想: 提供面向对象的跨平台的 GUI 框架。看到这里小女我不仅感慨: 什么时候我们的程序员们可以在工作的时候出来走走, 只有在轻松愉快的环境中才会生产出出色的成果。在沉闷的办公室里, 只是机械的堆砌代码而已。

所做就做, Haarard 开始写代码, Eirik 负责设计, Qt 在襁褓中逐渐成长, 在开始蹒跚学步的时候 (1993 年), 他们开始让 Qt 闯荡江湖, 两个人开始了创业的艰辛历程。

对这两个年轻人, 1994 年是非常艰难的一年, 他们没有客户, 没有钱, 只有还没有完全实现的产品。关键时刻, 他们的妻子帮他们渡过了难关。

字母 Q 作为所有类的前缀, 是因为 Haarard 手写这个字母看起来特别的漂亮, 字母 t 代表 "toolkit", 在 Xt, X toolkit 等中得到灵感。

1995 年开始出现转机, 他们得到了一个合同。这一年, 他们雇佣了 Arnt Gulbrandsen, 他在 Trolltech 工作了六年, 他为 Qt 实现了优秀的文档系统。

1995 年 5 月, Qt 0.9 发布, 有商业和开源两个版本。96 年 9 月, Qt1.0 发布。

1997 年, Matthias Ettrich 开始用 Qt 开发 KDE, 使 Qt 成为 Linux 上 GUI 开发的事实上的标准。

1999 年, Qt 2 发布。

2000 年, Qtopia 发布。支持 linux 嵌入式开发。

2001 年, Qt 3 发布。

2005 年, Qt 4 发布。

十年来，Qt 就是这样从不知名的一个产品，发展到现在拥有全世界范围内成千上万的客户。

1-1 从 Hello Qt 开始

差不多所有的程序教材都从 Hello 开始，下面就是这个程序的 qt 版本。

```
1 #include <QApplication>
2 #include <QLabel>
3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     QLabel *label = new QLabel("Hello Qt!");
7     label->show();
8     return app.exec();
9 }
```

按行解析以上 9 行代码

第一，二行：是代码中需要使用的类的头文件。在 Qt4 中，可以写成<QApplication>的格式，当然也可写成“QApplication.h”。

第三行：是 main 函数的标准写法

第五行：创建一个 QApplication 对象，管理应用程序的资源。

第六行：QLabel 对象，QLabel 是一个 Qt 提供的小控件，显示一行文本。

第七行：显示 QLabel。

第八行：QApplication.exec()，让程序进入消息循环。等待可能的菜单，工具条，鼠标等的输入，进行响应。

将以上代码放到名为 hello.cpp 中，保存，编译过程如下：

qmake -project, qmake 命令创建 hello.pro，是平台无关的工程文件。

在 hello.pro 所在目录下，运行 make (unix) 或者 nmake (windows)。

第 6 行代码还可以如下替换：

```
QLabel *label = new QLabel("<h2><i>Hello</i> "  
                           "<font color=red>Qt!</font></h2>");
```

这里面包含了 html 文本，显示的字体，颜色会改变。

实际程序中，下面两行是比不可少的。

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

1-2 连接信号和响应函数

连接信号和响应函数

这个例子用来说明怎么响应信号，和 **hello** 程序的源代码相似，原来的 **Label** 用一个按钮代替，点击时退出程序。

源程序如下：

```
1 #include <QApplication>  
2 #include <QPushButton>  
3 int main(int argc, char *argv[])  
4 {  
5     QApplication app(argc, argv);  
6     QPushButton *button = new QPushButton("Quit");  
7     QObject::connect(button, SIGNAL(clicked()),  
8                     &app, SLOT(quit()));  
9     button->show();  
10    return app.exec();  
11 }
```

当有所动作或者状态改变，qt 的控件会发出消息（**signal**），例如，当点击按钮时，按钮会发送 **clicked()** 消息，这个消息可以连接到一个函数上（这个函数在这里成为 **slot**）。

这样, 当一个消息发送时, **slot** 函数可以自动执行。在这个例子中, 我们连接了按钮的 **clicked** 信号和 **QApplication** 的 **quit** 函数, 语法如第七, 八行所示。

编译以上程序, 将以上代码放在 **quit.cpp** 文件中, 保存。

依次运行

```
qmake -project
```

```
qmake quit.pro
```

```
make(unix or linux) or nmake(windows)
```

然后运行程序, 点击 **Quit** 按钮, 程序将会中止。

1-3 控件的几何排列—**Laying Out Widgets**

在这个小节中, 我们说明在一个窗口中如何排列多个控件。学习利用 **signal** 和 **slot** 的方法使控件同步。程序要求用户通过 **spin box** 或者 **slider** 输入年龄。

程序中使用了三个控件: **QSpinBox**, **QSlider** 和 **QWidget**。**QWidget** 是这个程序的主窗口。**QSpinBox** 和 **QSlider** 被放在 **QWidget** 中; 他们是 **QWidget** 的 **children**。反过来, 我们也可以称 **QWidget** 是 **QSpinBox** 和 **QSlider** 的 **parent**。**QWidget** 没有 **parent**, 因为它是程序的顶层窗口。在 **QWidget** 及其子类的构造函数中, 都有一个 **QWidget*** 参数, 用来指定它们的父控件。

源代码如下:

```
1 #include <QApplication>
2 #include <QHBoxLayout>
3 #include <QSlider>
4 #include <QSpinBox>
5 int main(int argc, char *argv[])
6 {
7     QApplication app(argc, argv);
8     QWidget *window = new QWidget;
9     window->setWindowTitle("Enter Your Age");
10    QSpinBox *spinBox = new QSpinBox;
```



```

11  QSlider *slider = new QSlider(Qt::Horizontal);
12  spinBox->setRange(0, 130);
13  slider->setRange(0, 130);
14  QObject::connect(spinBox, SIGNAL(valueChanged(int)),
15                  slider, SLOT(setValue(int)));
16  QObject::connect(slider, SIGNAL(valueChanged(int)),
17                  spinBox, SLOT(setValue(int)));
18  spinBox->setValue(35);
19  QHBoxLayout *layout = new QHBoxLayout;
20  layout->addWidget(spinBox);
21  layout->addWidget(slider);
22  window->setLayout(layout);
23  window->show();
24  return app.exec();
25 }

```

第 8, 9 行建立程序的主窗口控件, 设置标题。第 10 到 13 行创建主窗口的 **children**, 并设置允许值的范围。第 14 到第 17 行是 **spinBox** 和 **slider** 的连接, 以使之同步显示同一个年龄值。不管那个控件的值发生变化, 都会发出 **valueChanged(int)** 信号, 另一个控件的 **setValue(int)** 函数就会为这个控件设置一个新值。

第 18 行将 **spinBox** 的值设置为 35, 这时 **spinBox** 发出 **valueChanged(int)** 信号, **int** 的参数值为 35, 这个参数传递给 **slider** 的 **setValue(int)** 函数, 将 **slider** 的值也设置为 35。同理, **slider** 也会发出 **valueChanged(int)** 信号, 触发 **spinBox** 的 **setValue(int)** 函数。这个时候, 因为 **spinBox** 的当前值就是 35, 所以 **spinBox** 不会发送任何信号, 不会引起死循环。

在第 19 至 22 行, 我们使用了一个布局管理器排列 **spinBox** 和 **slider** 控件。布局管理器能够根据需要确定控件的大小和位置。**Qt** 有三个主要的布局管理器:

QHBoxLayout: 水平排列控件。

QVBoxLayout: 垂直排列控件。

QGridLayout: 按矩阵方式排列控件

第 22 行, **QWidget::setLayout()** 把这个布局管理器放在 **window** 上。这个语句将

spinBox 和 **slider** 的“父”设为 **window**，即布局管理器所在的控件。如果一个控件由布局管理器确定它的大小和位置，那个创建它的时候就不必指定一个明确的“父”控件。

现在，虽然我们还没有看见 **spinBox** 和 **slider** 控件的大小和位置，它们已经水平排列好了。

QHBoxLayout 能合理安排它们。我们不用在程序中考虑控件在屏幕上的大小和位置这些头疼的事情了，交给布局管理器就万事大吉。

在 **Qt** 中建立用户界面就是这样简单灵活。程序员的任务就是实例化所需要的控件，按照需要设置它们的属性，把它们放到布局管理器中。界面中要完成任务由 **Qt** 的 **signal** 和 **slot** 完成。

第二章 创建对话框（Creating Dialogs）

在这章介绍如何创建 Qt 的对话框。对话框是程序和用户交互的桥梁，提供了程序和用户之间对话的一种方式。

很多程序都是由一个主窗口，在这个主窗口中包含一个菜单条，多个工具条，和足够多的对话框。也有些程序本身就是一个对话框，直接相应用户的输入请求。

本章中我们首先会用代码的方式创建我们的第一个对话框，然后用 Qt Designer 工具创建对话框。Qt Designer 是一个可视化的工具，用它可以更快的创建，修改对话框。

2-1 派生对话框类（Subclassing QDialog）

第一个例子是一个用 C++ 实现的查找对话框。我们把这个对话框实现为一个类，这样它就是一个独立的控件，并有自己的信号（signal）和 slot 函数

类的源代码分别放在 finddialog.h 和 finddialog.cpp 中。首先看 finddialog.h 的代码

```
1 #ifndef FINDDIALOG_H
2 #define FINDDIALOG_H
3 #include <QDialog>
4 class QCheckBox;
5 class QLabel;
6 class QLineEdit;
7 class QPushButton;
8 class FindDialog : public QDialog
9 {
10     Q_OBJECT
11 public:
12     FindDialog(QWidget *parent = 0);
13 signals:
14     void findNext(const QString &str, Qt::CaseSensitivity cs);
```

```

15    void findPrevious(const QString &str, Qt::CaseSensitivity cs);
16 private slots:
17    void findClicked();
18    void enableFindButton(const QString &text);
19 private:
20    QLabel *label;
21    QLineEdit *lineEdit;
22    QCheckBox *caseCheckBox;
23    QCheckBox *backwardCheckBox;
24    QPushButton *findButton;
25    QPushButton *closeButton;
26 };

27 #endif

```

一共 27 行，第 1，2，27 行是为了避免头文件被多次包含。

第 3 行包含 `QDialog` 头文件，这个类从 `QDialog` 继承，`QDialog` 从 `QWidget` 继承。

第 4 至 7 行是用到的 `Qt` 中类的前向声明。通过前向声明，编译器就知道这个类已经存在，而不用写出包含的头文件。这个问题稍后还要讲。

第 8 至 26 行是类 `FindDialog` 的定义。

第 10 行，`Q_OBJECT` 是一个宏定义，如果类里面用到了 `signal` 或者 `slots`，就要声明这个宏。

第 12 行，`FindDialog(QWidget *parent = 0);` 构造函数是 `Qt` 控件类的标准格式，默认的父参数为 `NULL`，说明没有父控件。

第 13 行，`signal` 声明了这个对话框发出的两个信号，如果选择向前查找，那么对话框就发出 `findPrevious()` 信号，否则，发出 `findNext()` 信号。`signal` 也是一个宏，在编译之前，`C++` 预处理把它变成标准的 `c++` 代码。`Qt::CaseSensitivity` 是一个枚举类型，有 `Qt::CaseSensitive` 和 `Qt::CaseInsensitive` 两个值。

在类的私有部分，声明有两个 **slot** 函数。为了实现这两个函数，需要用到对话框的其他控件的信息，所以保存了一些控件的指针。**slot** 关键字和 **signal** 一样，也是一个宏。

对于私有成员变量，我们只是使用了它们的指针，没有对它们进行存取操作，编译器不需要知道它们的详细定义，所以只使用了这些类的前向声明。当然，也可以使用 `<QCheckBox>`，`<QLabel>` 等，但是，使用前向声明会让编译速度更快一些。

下面看一下 `finddialog.cpp` 源文件代码：

文件头和构造函数部分

```
1  #include <QtGui>
2  #include "finddialog.h"
3  FindDialog::FindDialog(QWidget *parent)
4      : QDialog(parent)
5  {
6      label = new QLabel(tr("Find &what:"));
7      lineEdit = new QLineEdit;
8      label->setBuddy(lineEdit);
9      caseCheckBox = new QCheckBox(tr("Match &case"));
10     backwardCheckBox = new QCheckBox(tr("Search &backward"));
11     findButton = new QPushButton(tr("&Find"));
12     findButton->setDefault(true);
13     findButton->setEnabled(false);
14     closeButton = new QPushButton(tr("Close"));
15     connect(lineEdit, SIGNAL(textChanged(const QString &)),
16            this, SLOT(enableFindButton(const QString &)));
17     connect(findButton, SIGNAL(clicked()),
18            this, SLOT(findClicked()));
19     connect(closeButton, SIGNAL(clicked()),
20            this, SLOT(close()));
21     QHBoxLayout *topLeftLayout = new QHBoxLayout;
```

```

22  topLeftLayout->addWidget(label);
23  topLeftLayout->addWidget(lineEdit);
24  QVBoxLayout *leftLayout = new QVBoxLayout;
25  leftLayout->addLayout(topLeftLayout);
26  leftLayout->addWidget(caseCheckBox);
27  leftLayout->addWidget(backwardCheckBox);
28  QVBoxLayout *rightLayout = new QVBoxLayout;
29  rightLayout->addWidget(findButton);
30  rightLayout->addWidget(closeButton);
31  rightLayout->addStretch();
32  QHBoxLayout *mainLayout = new QHBoxLayout;
33  mainLayout->addLayout(leftLayout);
34  mainLayout->addLayout(rightLayout);
35  setLayout(mainLayout);
36  setWindowTitle(tr("Find"));
37  setFixedHeight(sizeHint().height());
38 }

```

到这里 `FindDialog` 的构造函数就完成了。在传见控件和布局时我们使用了 `new`，一般情况下，我们还需要写析构函数 `delete` 这些控件。

但是在 `Qt` 中这是不需要的，当父控件销毁时，`Qt` 自动删除它所有的子控件和布局。

下面是 `FindDialog` 类的两个 `slot` 函数：

```

39 void FindDialog::findClicked()
40 {
41     QString text = lineEdit->text();
42     Qt::CaseSensitivity cs =
43         caseCheckBox->isChecked() ? Qt::CaseSensitive
44                                     : Qt::CaseInsensitive;
45     if (backwardCheckBox->isChecked()) {
46         emit findPrevious(text, cs);

```

```

47     } else {
48         emit findNext(text, cs);
49     }
50 }
51 void FindDialog::enableFindButton(const QString &text)
52 {
53     findButton->setEnabled(!text.isEmpty());
54 }

```

当用户点击 `findButton` 按钮，`findClicked()` 就会调用，根据 `backwardCheckBox` 状态，他发出 `findPrevious()` 或者 `findNext()` 信号。`emit` 也是一个 Qt 的宏。

当用户改变 `lineEdit` 中的文本，`enableFindButton()` slot 函数就会调用。如果输入了文本，那么让 `findButton` 有效，否则就无效。

最后，创建 `main.cpp` 测试 `FindDialog` 对话框。

```

1 #include <QApplication>
2 #include "finddialog.h"
3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     FindDialog *dialog = new FindDialog;
7     dialog->show();
8     return app.exec();
9 }

```

运行 `qmake` 编译程序。由于在 `FindDialog` 中包含了 `Q_OBJECT` 宏，由 `qmake` 生成的 `makefile` 会保换特殊的规则运行 `moc`（Qt 的原对象编译器）。

为了确保 `moc` 正确工作，类定义必须放在头文件而不能放在实现文件中。由 `moc` 生成的代码中包含这个头文件，并加入它自己实现的 C++ 代码。

使用了 `Q_OBJECT` 宏的类必须运行 `moc`。如果使用 `qmake`，那么 `makefile` 里自动包含相关的规则。如果忘记了运行 `moc`，就会发生连接错误。不同的编译器给出的提示信息不同，有的会非常晦涩。GCC 给出的错误信息如下：

```
finddialog.o: In function 'FindDialog::tr(char const*, char const*)':  
/usr/lib/qt/src/corelib/global/qglobal.h:1430: undefined reference to  
'FindDialog::staticMetaObject'
```

Visual C++中的输出是这样：

```
finddialog.obj : error LNK2001: unresolved external symbol  
"public:~virtual int __thiscall MyClass::qt_metacall(enum QMetaObject  
::Call,int,void * *)"
```

这时需要重新运行 `qmake`，更新 `makefile`，然后编译程序。

运行程序，如果看到了快捷键，测试 `ALT+W,ALT+C,ALT+B,ALT+F` 引发相应的处理程序。

使用 `TAB` 键在将焦点改变到不同的控件上。默认的 `TAB` 键是控件创建的顺序。

`QWidget::setTabOrder()`可以改变这个顺序。

提供合适的 `tab` 顺序和快捷键可以让用户不用鼠标也可以运行程序，通过键盘可以快速控制程序。

2-2 深入信号和槽 (Signals and Slots in Depth)

信号和槽是 `Qt` 编程的一个重要部分。这个机制可以在对象之间彼此并不了解的情况下将它们的行为联系起来。在前几个例程中，我们已经连接了信号和槽，声明了控件自己的信号和槽，并实现了槽函数，发送了信号。现在来更深入地了解这个机制。

槽和普通的 `c++` 成员函数很像。它们可以是虚函数 (`virtual`)，也可被重载 (`overload`)，可以是公有的 (`public`)，保护的 (`protective`)，也可是私有的 (`private`)，它们可以象任何 `c++` 成员函数一样被调用，可以传递任何类型的参数。不同在于一个槽函数能和一个信号相连接，只要信号发出了，这个槽函数就会自动被调用。

`connect` 函数语法如下：

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

`sender` 和 `receiver` 是 `QObject` 对象指针，`signal` 和 `slot` 是不带参数的函数原型。

`SIGNAL()`和 `SLOT()`宏的作用是把他们转换成字符串。

在目前有的例子中，我们已经连接了不同的信号和槽。实际使用中还要考虑如下一些规则：

1、一个信号可以连接到多个槽：

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

当信号发出后，槽函数都会被调用，但是调用的顺序是随机的，不确定的。

2、多个信号可以连接到一个槽

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

任何一个信号发出，槽函数都会执行。

3、一个信号可以和另一个信号相连

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

第一个信号发出后，第二个信号也同时发送。除此之外，信号与信号连接上和信号和槽连接相同。

4、连接可以被删除

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

这个函数很少使用，一个对象删除后，Qt 自动删除这个对象的所有连接。

信号和槽函数必须有着相同的参数类型，这样信号和槽函数才能成功连接：

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

如果信号里的参数个数多于槽函数的参数，多余的参数被忽略：

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

如果参数类型不匹配，或者信号和槽不存在，在 debug 状态时，Qt 会在运行期间给出警告。如果信号和槽连接时包含了参数的名字，Qt 将会给出警告。

以前我们列举的例子中都是控件的信号和槽。但是信号和槽机制在 `QObject` 中就实现了，可以实现在任何从 `QObject` 继承的子类中。

```
class Employee : public QObject
{
    Q_OBJECT
public:
    Employee() { mySalary = 0; }
    int salary() const { return mySalary; }
public slots:
    void setSalary(int newSalary);
signals:
    void salaryChanged(int newSalary);
private:
    int mySalary;
};

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

注意，只有 `newSalary != mySalary` 时才发出 `salary-Changed()` 信号，这样避免了死循环的出现。

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Qt 的 Meta-Object 系统

Qt 的一个最主要的成功是对 C++ 扩展，即把彼此独立的软件模块连接起来，而不需要模块间的任何细节。

这个机制就是 **Meta-Object** 系统，它提供了两个关键的用途：信号和槽和 **introspection**（内省）。**introspection** 功能允许应用程序在运行时得到 **QObject** 它子类的“**meta-information**”，这对实现信号和槽是很必要的，包括全部信号和槽的列表，和类的名字。这个机制还提供了属性（在 **Qt Designer** 中使用）和文本翻译（国际化）支持。它们构成了 **QSA**（**Qt Script for Application**）的基础。

标准 **C++** 不提供 **Qt meta-object** 系统需要的动态 **meta-information**。**Qt** 提供了一个独立的工具 **moc**，通过定义 **Q_OBJECT** 宏实现到 **C++** 函数的转变。**moc** 是用纯 **c++** 实现的，因此可以使用在任何 **C++** 编译器中。

这个机制工作过程如下：

Q_OBJECT 声明了一些 **QObject** 子类必须实现的内省函数：**metaObject()**，**TR()**，**qt_metacall()**等。

Qt 的 **moc** 工具实现 **Q_OBJECT** 宏声明的函数和所有的信号。

QObject 成员函数 **connect()**和 **disconnect()**使用这些内省函数实现信号和槽的连接。

以上这些是通过 **qmake**，**moc** 和 **QObject** 自动处理的，程序员通常不用考虑它们。如果你感到对此好奇，可以查看 **QMetaObject** 类文档和 **moc** 实现的 **c++** 代码。

2-3 快速设计对话框（**Rapid Dialog Design**）

通常程序员们都是用 **c++** 源代码编写 **Qt** 应用程序，**Qt** 也是很容易用来编写的。然而，许多程序员更喜欢用可视化的方法设计对话框，这样能更快速更容易对对话框进行修改。

Qt Designer 满足了程序员的这一要求，提供了可视化设计对话框的方法。它可以给一个应用程序提供全部或者部分对话框。用 **Qt Designer** 设计的对话框和用 **c++** 代码写成的对话框是一样的，可以用做一个常用的工具，并不对编辑器产生影响。

在这一节中，我们使用 **Qt Designer** 创建 **Go-to-Cell** 对话框，无论用编写代码的方式还是用 **Qt Designer**，创建对话框都有如下基本的步骤：

- 1、创建和初始化子控件。
- 2、把子控件放到布局管理器中。
- 3、设置 **tab** 顺序。
- 4、创建信号和槽。

5、实现对话框的自己的槽函数。

在 windows 平台 Qt 的安装目录的 bin 目录下，点击 `desinger.exe`，或者在 unix 平台，在命令行上输入 `designer`。当 Qt Designer 启动后，它会列出一个控件模板的列表，选择一个模板，进入设计。

原文中对 Qt Designer 的介绍略去不想翻译了，只要稍有点界面编程基础的都可以轻松使用。如果确实需要，以后再补上。

我个人不喜欢使用这个东东，因为要多一个文件要维护，当然如果要频繁修改所设计的对话框，那这种方法还是很方便的。但不管怎么样，最终都要修改源代码。所以我还是比较喜欢用源代码的方式把控件手工写出来。

我想主要介绍把对话框设计好以后，保存为.ui 文件后的处理。

假如设计好的文件保存在 `gotocell` 目录中，命名为 `gotocelldialog.ui` 中，然后在同一个目录下创建一个 `main.cpp` 文件，编码如下：

```
#include <QApplication>
#include <QDialog>
#include "ui_gotocelldialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Ui::GoToCellDialog ui;
    QDialog *dialog = new QDialog;
    ui.setupUi(dialog);
    dialog->show();
    return app.exec();
}
```

保存后，在该目录下运行 `qmake`，创建.pro 文件，然后运行 `qmake -project` 生成 makefile 文件。`qmake` 可以发现 `gotocelldialog.ui` 文件，然后就会调用 `uic` (Qt 的用户界面编译器)，`uic` 工具把 `gotocelldialog.ui` 转换成 c++ 代码，保存在 `ui_gotocelldialog.h` 中。

在 `ui_gotocelldialog.h` 中，包含了 `Ui::GoToCellDialog` 类的定义，这个类和 `gotocelldialog.ui` 等价。这个类声明成员变量存储对话框的子控件和布局管理器，`setupUi()` 函数初始化对话框。

这个类的定义看起来有点象下面这个样子：

```
class Ui::GoToCellDialog
{
public:
    QLabel *label;
    QLineEdit *lineEdit;
    QSpacerItem *spacerItem;
    QPushButton *okButton;
    QPushButton *cancelButton;
    ...
    void setupUi(QWidget *widget) {
        ...
    }
};
```

这个类没有父类。使用时创建一个 `QDialog`，把它传递给 `setupUi()` 函数。

运行这个程序，对话框将会显示出来，但是有些功能它还不能实现：

- 1、Ok 按钮是不可用状态的
- 2、Cancel 按钮不作任何事情
- 3、编辑框除可以输入许可的字符或者数字外，还可以输入任何文本

我们可以编写代码，让这个对话框变得有用起来。最直接的方法是创建一个新类，继承 `QDialog` 和 `Ui::GoToCellDialog`，补上缺少的功能。（这说明任何软件问题可以通过添加一层间接包装来简单解决）。通常命名新类规则是把去掉 `uic` 生成的类名去掉 `Ui::` 前缀。

创建 `gotocelldialog.h` 头文件，写下如下代码：

```
#ifndef GOTOCELLDIALOG_H
#define GOTOCELLDIALOG_H

#include <QDialog>

#include "ui_gotocelldialog.h"
```

```

class GoToCellDialog : public QDialog, public Ui::GoToCellDialog
{
    Q_OBJECT
public:
    GoToCellDialog(QWidget *parent = 0);
private slots:
    void on_lineEdit_textChanged();
};
#endif

```

新建 `gotocelldialog.cpp` 源文件，实现这个类：

```

#include <QtGui>
#include "gotocelldialog.h"

GoToCellDialog::GoToCellDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);
    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));
    connect(okButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
}

void GoToCellDialog::on_lineEdit_textChanged()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}

```

在构造函数中，我们调用 `setupUi()` 初始化这个对话框。由于多继承，我们可以直接使用 `Ui::GoToCellDialog` 的成员。创建了用户界面以后，我们可以把子控件的信号和槽函数连接起来。

在构造函数中，我们还创建一个许可器（`validator`）限制编辑框输入的范围。Qt 提供了三个许可器类：`QIntValidator`，`QDoubleValidator` 和 `QRegExpValidator`。这里我们

使用了 `QRegExpValidator`，使用的表达式为“`[A-Za-z][1-9][0-9]{0,2}`”这个表达式的意思是第一个字符输入为大写或者小写字母，第二个字符为一个数字范围是 1 到 9，第三个字符是一个数字范围为 0 到 9。在 `QRegExpValidator` 的构造函数中，第二个参数为 `this`，把当前类作为它的父控件，这样就可以不用删除它，父控件析构时可以被自动删除。

Qt 的父子机制在 `QObject` 中实现的。当我们创建一个带有父的对象（如一个子控件，一个许可器，布局管理等）时，父对象把子对象放到自己的子对象列表中。父对象被删除时，它查找自己的子对象并把每一个删除掉。这些子对象再把自己的子对象删除掉，如此递归，知道删除所有对象。

这种父子对象的机制简化了内存管理，减少了内存泄漏的危险。需要程序员删除的对象就是我们使用 `new` 创建的没有父对象的对象。如果在父对象存在时删除了它的一个子对象，Qt 将会在父列表中自动删除。（需要记住的是 Qt 只是删除有父的对象，父对象还是需要手动删除的，还有就是那些用 `new` 申请的没有指定父的内存，一般情况下，在对话框里的子控件，许可器和布局管理器由 Qt 自己管理，其他还要程序员小心删除）

对于控件来讲，父对象还有一个意义：子控件在父对象的显示区域内显示。当父控件删除后，子控件不但在内存中被删除，它也同时在屏幕上消失。

在构造函数的最后两行，把 `QDialog` 的 `accept()` 函数连接到 OK 按钮的点击信号，把 Cancel 按钮的点击信号连接到 `reject()` 函数。这两个槽函数都关闭这个对话框，但是 `accept()` 返回 `QDialog::Accepted`（值为 1），`reject()` 返回值为 `QDialog::Rejected`（值为 0）。不同的返回值可以判断用户点击了那个按钮。

`on_lineEdit_textChanged()` 槽函数控制 Ok 按钮的可用状态，通过编辑框中的输入字符，如果字符有效 Ok 按钮则有效，否则为不可用状态。

`QLineEdit::hasAcceptableInput()` 根据我们在构造函数中设置的许可器返回 `bool` 值。

这就完成了这个对话框，现在重写这个 `main.cpp` 文件：

```
#include <QApplication>
#include "gotocelldialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    GoToCellDialog *dialog = new GoToCellDialog;
    dialog->show();
}
```

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

编译这个程序（`qmake -project; qmake gotocell.pro`）然后运行。输入“A12”，**Ok** 按钮变为可用。试着输入一行随意字符，观察许可器的反映。点击 **Cancel** 按钮关闭这个对话框。

使用 **qt Designer** 可以不改变源程序的情况下改变对话框的设计。如果对话框用 **C++** 代码编写，改变它将会很费力的。使用 **Qt Designer**，**uic** 自动重新生成源文件。不会浪费任何时间。

2-4 能够改变的对话框(Shape-Changing Dialogs)

前面几章我们设计的对话框都是不能改变它的样子的。但是有时需要对话框根据要求进行适当的改变。两个最常用的需要改变的对话框是可扩展对话框和多页对话框。这两种类型的可以通过代码编写，也可以用 **Qt Designer** 设计。

可扩展对话框通常外观简单，带有一个可扩展按钮来切换对话框的简单外观和可扩展外观。这种对话框通常为了迎合普通用户和高端用户而设计的，如果没有特别请求隐藏高级应用部分。在这一节，我们使用 **Qt Designer** 设计一个可扩展对话框。

对话框是一个表格程序的排序对话框，对用户选择的一些列按要求排列。对话框的简单外观允许用户输入一个简单排序关键词，扩展部分允许输入两个额外的排序关键词。一个 **More** 按钮使用户在简单外观和扩展外观进行切换。

我们使用 **Qt Designer** 创建这个可扩展的对话框，在运行时刻隐藏高级功能，这个看起来很复杂的对话框用 **Qt Designer** 可以很容易实现。首先设计好第一个关键词，第二个和第三个关键词通过复制就可以得到：

- 1、启动 **File|New** 菜单，选择“**Dialog with Buttons Right**”模板。
- 2、创建 **More** 按钮，并将它托到右边的垂直布局管理器中，放到垂直空白的下面。设置按钮的文本属为“&More”，它的 **checkable** 属性为“**true**”，设置 **Ok** 按钮的 **default** 属性为 **true**。
- 3、创建一个组合框，两个标签，两个下拉组合框和一个水平空白，先把它放在对话框的任何地方。
- 4、把组合框拖动大些，把 3 中其他控件拖动到其中，按比例调整位置。

- 5、第二个下拉框宽度调整为第一个下拉框的二倍。
- 6、设置组合框的 **title** 属性为"&Primary Key"，第一个标签的 **text** 属性为"Column:"，第二个标签的 **text** 属性为"Order:"。
- 7、设置第一个下拉框的第一个项目文本项为"None"。
- 8、设置第二个下拉框的项目为"Ascending"和"Descending"两个项目，即升序和降序排列。
- 9、选择组合框，设置它的布局为 **Grid**。

如果设计过程中出现错误，可以选择 **Edit|Undo** 或者 **Form|Break Layout**，重新进行排列。当然只要看起来不是很难看，也可以是其他的样子，只要易于理解就是 **ok**。

现在加入第二个，第三个关键词：

- 1、把对话框拖动到足够大。
- 2、复制第一个组合框，粘贴两次，一次拖动到下面。
- 3、把复制的两个组合框的 **title** 属性为"&Secondary Key"和"&Tertiary Key"。
- 4、在第一个关键词和第二个关键词组合框之间添加一个垂直空白。
- 5、调整添加的控件。
- 6、选择这个对话框，降它设置为 **Grid** 管理。
- 7、设置两个垂直空白的 **sizeHint** 属性为[20,0]。

按照下图命名每一个控件。命名对话框为 **sortDialog**，窗口标题为"Sort"。

然后设置控件的 **tab** 顺序。从上到下点击下拉框，然后点击 **Ok**，**Cancel**，**More** 按钮。

以上是对话框的设计。然后用 **Qt Designer** 建立控件的信号连接。因为我们创建对话框时使用了"Dialog with Buttons Right"模板，**Ok** 和 **Cancel** 按钮已经连接到了对话框的 **accept()**和 **reject()**槽函数。连接可以在 **Qt designer** 的 **signal/slot** 编辑窗口查看。我们需要自己建立的连接是连接 **More** 按钮和 **secondary-GroupBox**。将按钮的 **toggled(bool)**信号和组合框的 **setVisible(bool)**连接。选择 **Edit|Signal/Slots**，将编辑状态变为连接态，拖动 **More** 按钮到 **secondary-GroupBox** 上，弹出信号编辑对话框。创建一个 **sort** 目录，保存对话框文件到 **sort** 目录的 **sortdialog.ui**，使用多继承的方式使用这个对话框。

首先新建一个 **sortdialog.h** 头文件，代码如下：

```
#ifndef SORTDIALOG_H
#define SORTDIALOG_H
```

```

#include <QDialog>

#include "ui_sortdialog.h"

class SortDialog : public QDialog, public Ui::SortDialog
{
    Q_OBJECT
public:
    SortDialog(QWidget *parent = 0);
    void setColumnRange(QChar first, QChar last);
};

#endif

```

然后新建 sortdialog.cpp 源文件:

```

1 #include <QtGui>
2 #include "sortdialog.h"
3 SortDialog::SortDialog(QWidget *parent)
4     : QDialog(parent)
5 {
6     setupUi(this);
7     secondaryGroupBox->hide();
8     tertiaryGroupBox->hide();
9     layout()->setSizeConstraint(QLayout::SetFixedSize);
10    setColumnRange('A', 'Z');
11 }
12 void SortDialog::setColumnRange(QChar first, QChar last)
13 {
14     primaryColumnCombo->clear();
15     secondaryColumnCombo->clear();
16     tertiaryColumnCombo->clear();
17     secondaryColumnCombo->addItem(tr("None"));
18     tertiaryColumnCombo->addItem(tr("None"));

```

```

19 primaryColumnCombo->setMinimumSize(
20     secondaryColumnCombo->sizeHint());
21 QChar ch = first;
22 while (ch <= last) {
23     primaryColumnCombo->addItem(QString(ch));
24     secondaryColumnCombo->addItem(QString(ch));
25     tertiaryColumnCombo->addItem(QString(ch));
26     ch = ch.unicode() + 1;
27 }
28 }

```

在构造函数中，隐藏了第二个和第三个关键词部分。设置对话框的 `sizeConstraint` 的属性为 `QLayout::setFixedSize`，这样用户就不能随便改变对话框的大小。

下面是 `main.cpp` 文件：

```

#include <QApplication>
#include "sortdialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    SortDialog *dialog = new SortDialog;
    dialog->setColumnRange('C', 'F');
    dialog->show();
    return app.exec();
}

```

编译运行这个程序，点击 **More** 按钮，查看对话框的改变。

另一种可以改变的对话框是多页对话框。这类对话框也可以用两种方式创建。相关的类有 `QTabWidget`，`QStackedWidget`，`QListWidget`，`QTreeWidget` 等以后介绍。

2-5 动态对话框（Dynamic Dialogs）

动态对话框是在程序运行时用 Qt Designer 的.ui 文件创建。不用 uic 工具把.ui 文件变成等价的 c++ 代码，而是在程序时使用类 QUiLoader 加载.ui 文件，例如下面的代码：

```
QUILoader uiLoader;
QFile file("sortdialog.ui");
QWidget *sortDialog = uiLoader.load(&file);
if (sortDialog) {
    ...
}
```

子控件可以用 QObject::findChild<T>() 得到

```
QComboBox *primaryColumnCombo =
    sortDialog->findChild<QComboBox *>("primaryColumnCombo");
if (primaryColumnCombo) {
    ...
}
```

findChild<T>() 是模板成员函数，得到类型为 T 的给定名字的子控件的指针。由于编译器的原因，用 MSVC6 是得不到的。如果使用的是 MSVC6，那么可以使用全局函数 qFindChild<T>()。

QUILoader 类在一个单独的链接库中，如果在一个应用程序中使用了 QUILoader，必须在这个程序的.pro 文件中添加下面这样的代码：

```
CONFIG      += uitools
```

使用动态对话框不用重新编译程序就能够改变对话框的布局。它们可以用来创建“细客户”的程序，只有两个内建的对话框，其他的对话框都是按照不同需要创建的。（这段的翻译有点直，原文如下：Dynamic dialogs make it possible to change the layout of a form without recompiling the application. They can also be used to create thin-client applications, where the executable merely has a front-end form built-in and all other forms are created as required.）

2-6 Qt 提供的控件和对话框类（Built-in Widget and Dialog Classes）

Qt 提供了许多控件和对话框类，可以满足多种情况的需要。这一节将对它们进行介绍。有些特殊的控件如：QMenuBar, QToolBar 和 QStatusBar 主窗口类控件在第三章介绍，QSplitter 和 QScrollArea 在第六章介绍。大部分 Qt 提供的控件都会在这本书中出现。在下面即将介绍的控件中，用透明的方式显示其外观。

Qt 提供四种方式的按钮：QPushButton, QToolButton, QCheckBox, 和 QRadioButton. QPushButton 和 QToolButton 主要用来提供点击动作，可以做为套索按钮（点击时显示按下的状态，再次点击后恢复）。QCheckBox 可以用来表示开关选项。QRadioButton 一般是多个组合起来一起使用，提供一个单一的选择。

Qt 提供的容器类控件可以容纳其他的控件。QFrame 可以单独使用，可以在其上画直线等，它也被其他许多控件类继承，如 QToolBox 和 QLabel。

QTabWidget 和 QToolBox 是多页对话框，每一页都是一个子控件，页数从 0 开始。

列表视图一般处理大量数据，经常需要使用滚动条来显示全部内容。滚动条机制的基类是 QAbstractScrollArea，是视图类和其他滚动控件的基类。

Qt 还提供只显示信息的控件，QLabel 是用的最多的，它可以用来显示文本，显示带有 html 格式的文本，还可以显示图片。

QTextBrowser 显示图片，表格，多文本连接等。Qt Assistant 就是使用 QTextBrowser 显示用户文档。

Qt 提供这样一些数据输入的控件。QLineEdit 只可输入许可器允许的字符。QTextEdit 是 QAbstractScrollArea 的子类，可以输入多行文本。

Qt 还提供了多种普通的对话框，可以方便的选择颜色，字体，文件，打印文档等。Windows，Mac OS X 等不同平台上的普通对话框尽可能和平台控件风格一致。

Qt 提供了很多信息显示对话框和错误提示对话框。程序的进行状态可以用 `QProgressDialog` 和 `QProgressBar` 显示。`QInputDialog` 可以方便的让用户输入一行文本或者数字。

这些控件和对话框提供了很多方便的函数，大部分特殊的要求可以通过设置控件属性或者通过信号和槽连接完成。

有时候有些用户需要从零开始新建一个新自定义控件。在 Qt 中可以使用所有平台无关的绘制函数。自定义控件还可以集成到 Qt Designer 中，象 Qt 原有提供的控件一样使用。第五章将会介绍怎么创建自定义控件。

第三章 创建主窗口 (Creating Main Windows)

这一章介绍如何用 **qt** 创建程序的主窗口。最后，读者能够生成一个有着全部菜单，工具条，状态条和许多对话框的完整的用户界面。

应用程序的主窗口是用户界面的框架。**SpreadSheet** 应用程序的主窗口如图所示。这个程序使用了第二章创建的 **Find**，**Go-to-Cell** 和 **Sort** 对话框。

很多 **GUI** 应用程序都能够处理文件的读写，数据处理等功能。在第四章，我们继续使用 **SpreadSheet** 为例子进行说明。

3-1 继承 **QMainWindow** 类(Subclassing **QMainWindow**)

一个应用程序的主窗口要从 **QMainWindow** 继承。我们在第二章看到的创建对话框的方法可以用来创建主窗口，**QDialog** 和 **QMainWindow** 都是继承自 **QWidget** 类。

主窗口可用 **Qt Designer** 创建。但是在这一章，我们使用 **c++** 代码实现。如果你喜欢使用可视化的工具，可以参考在线手册“**Creating Main Windows in Qt Designer**”。

SpreadSheet 应用程序的主窗口类定义文件和实现文件分别在 **mainwindow.h** 和 **mainwindow.cpp** 中，首先看头文件：

```
#ifndef MAINWINDOW_H

#define MAINWINDOW_H

#include <QMainWindow>

class QAction;

class QLabel;

class FindDialog;

class Spreadsheet;

class MainWindow : public QMainWindow
```

```

{

    Q_OBJECT

public:

    MainWindow();

protected:

    void closeEvent(QCloseEvent *event);

private slots:

    void newFile();

    void open();

    bool save();

    bool saveAs();

    void find();

    void goToCell();

    void sort();

void about();

void openRecentFile();

void updateStatusBar();

void spreadsheetModified();

private:

void createActions();

```



```
void createMenus();

void createContextMenu();

void createToolBars();

void createStatusBar();

void readSettings();

void writeSettings();

bool okToContinue();

bool loadFile(const QString &fileName);

bool saveFile(const QString &fileName);

void setCurrentFile(const QString &fileName);

void updateRecentFileActions();

QString strippedName(const QString &fullFileName);

Spreadsheet *spreadsheet;

FindDialog *findDialog;

QLabel *locationLabel;

QLabel *formulaLabel;

QStringList recentFiles;

QString curFile;

enum { MaxRecentFiles = 5 };

QAction *recentFileActions[MaxRecentFiles];
```

```

    QAction *separatorAction;

    QMenu *fileMenu;

    QMenu *editMenu;


    QToolBar *fileToolBar;

    QToolBar *editToolBar;

    QAction *newAction;

    QAction *openAction;


    QAction *aboutQtAction;

};

#endif

```

我们定义 **MainWindow** 类继承自 **QMainWindow**。因为它有自己的信号和槽，所以声明了 **Q_OBJECT** 宏。

closeEvent()是 **QWidget** 的虚函数，当用户关闭窗口时自动调用。在 **MainWindow** 中它被重新实现，这样我们就可以提出用户一些常见的问题，如：保存所作的改变？，提示用户存盘。

有些菜单项，如 **File|New**，**Help|About** 等被声明为 **MainWindow** 的私有的相应函数。多数的槽函数返回值为 **void**，但是 **save()**和 **saveAs()**返回的值为 **bool** 型。当一个槽函数由信号引发时它的返回值被忽略，但是如果槽函数做为普通函数调用，这个返回值就可以象其他普通函数一样被得到。

在这个类中还声明了很多其他的私有槽函数和私有函数实现用户界面的功能。除此之外还有很多私有变量，这些在使用的时候会解释。

下面来看源文件代码：

```
#include <QtGui>

#include "finddialog.h"

#include "gotocelldialog.h"

#include "mainwindow.h"

#include "sortdialog.h"

#include "spreadsheet.h"

MainWindow::MainWindow()

{

    spreadsheet = new Spreadsheet;

    setCentralWidget(spreadsheet);

    createActions();

    createMenus();

    createContextMenu();

    createToolBars();

    createStatusBar();

    readSettings();

    findDialog = 0;
```

```

        setWindowIcon(QIcon(":/images/icon.png"));

        setCurrentFile("");

    }

```

在包含文件中由<QtGUI>，这包含了我们在这个类中使用的很多 Qt 类。其他是第二章中定义的头文件，这里也使用了。

在构造函数中，我们开始创建 **SpreadSheet** 控件，并把这个控件做为主窗口的中心控件。这个控件占据主窗口的中间部分。**SpreadSheet** 是一个 **QTableWidget** 控件，有一些简单的列表功能，将会在第四章实现。

然后我们调用 **createActions()**，**createMenus()**，**createContext-Menu()**，**createToolBars()**和 **createStatusBar()**创建主窗口的其他部分。**readSettings()**读取程序保存在磁盘上的一些设置。

我们把 **findDialog** 指针为空，当 **MainWindow::find()**第一次被调时，将会创建一个 **FindDialog** 对象。

最后，我们设置窗口的图标为 **icon.png**。Qt 支持多种格式的图片文件，包括 **BMP, GIF, JPEG, PNG, PNM, XBM, XPM** 。在 **QWidget::setWindowIcon()**中设置的图标显示在程序主窗口的左上角。不过，Qt 没有提供一个平台无关的程序的桌面图标。相关平台的处理方式可参考 <http://doc.trolltech.com/4.1/appicon.html>.中说明。

GUI 程序通常会使用很多图片。提供图片的方式很多，主要有：

- 1、 把图片存储在文件中，程序运行时加载它们
- 2、 在源代码中包含 **XPM** 文件（这种文件是有效的 **c++** 文件）
- 3、 使用 Qt 提供的资源管理方案。

这里我们使用 Qt 提供的资源管理方案，因为它能够在运行时方便的加载图片文件，并支持以上文件格式。这里假设图片文件保存在应用程序源代码目录的字母里 **images** 里面。

使用这个方案时，需要创建一个资源文件，并在 **.pro** 文件中添加这个资源文件的有关信息。在这个例子中，定义资源文件为 **spreadsheet.qrc**，在 **.pro** 文件中加入如下信息：

```
RESOURCES    = spreadsheet.qrc
```

在资源文件中使用了简单的 XML 格式：

```
<!DOCTYPE RCC><RCC version="1.0">
```

```
<qresource>
```

```
<file>images/icon.png</file>
```

```
...
```

```
<file>images/gotocell.png</file>
```

```
</qresource>
```

```
</RCC>
```

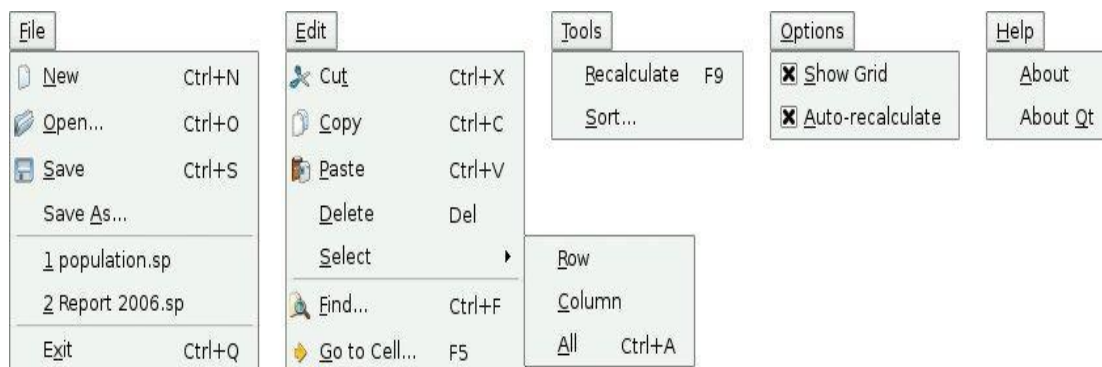
资源文件被编译到程序的可执行文件中，故它们不会丢失。使用资源时使用前缀：**/**。例如：**/images/icon.png**。除图片外，资源可以是任何格式的文件，这将在第 12 章里介绍。

3-2 创建菜单和工具条（Creating Menus and Toolbars）

很多 GUI 程序都有菜单，上下文菜单，工具条等。用户通过菜单浏览程序提供的所有功能。上下文菜单和工具条能够使用户快速得到平时频繁使用得功能。

Qt 使用“行为”（**action**）这个概念提供菜单和工具条。一个“行为（**action**）”是一个可以加入任何菜单或者工具条的项目。用 Qt 创建菜单和工具条需要如下步骤

- 1、建立行为（action）
- 2、创建菜单，并使它与一个行为关联
- 3、创建工具条，并使它与一个行为关联



下面是 Spreadsheet 程序中 createActions()函数得实现：

```
void MainWindow::createActions()
{
    newAction = new QAction(tr("&New"), this);
    newAction->setIcon(QIcon(":/images/new.png"));
    newAction->setShortcut(tr("Ctrl+N"));
    newAction->setStatusTip(tr("Create a new spreadsheet file"));
    connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
    //其他相关 action
    for (int i = 0; i < MaxRecentFiles; ++i)
    {
        recentFileActions[i] = new QAction(this);
        recentFileActions[i]->setVisible(false);
        connect(recentFileActions[i], SIGNAL(triggered()), this, SLOT(openRecentFile()));
    }

    selectAllAction = new QAction(tr("&All"), this);
    selectAllAction->setShortcut(tr("Ctrl+A"));
    selectAllAction->setStatusTip(tr("Select all the cells in the " "spreadsheet"
```

```

));
|   connect(selectAllAction, SIGNAL(triggered()), spreadsheet, SLOT(selectAll
|   ());
|
|   showGridAction = new QAction(tr("&Show Grid"), this);
|   showGridAction->setCheckable(true);
|   showGridAction->setChecked(spreadsheet->showGrid());
|   showGridAction->setStatusTip(tr("Show or hide the spreadsheet's ""grid")
|   );
|   connect(showGridAction, SIGNAL(toggled(bool)), spreadsheet, SLOT(setSh
|   owGrid(bool)));
|
|   aboutQtAction = new QAction(tr("About &Qt"), this);
|   aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));
|
|   connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
| }

```

以第一个 action 为例：这个 action 为 **New**，有一个加速键（**N**），一个父对象（主窗口），一个图标（**new.jpg**），一个快捷键（**Ctrl+N**），还有一个提示信息。我们连接这个 action 的 **triggered()** 信号和主窗口得私有槽函数 **newFile()**，这个函数将在下一节实现。用户在选择 **File|New** 菜单项，点击了 **New** 工具栏，或者在键盘敲了 **Ctrl+N** 时，**newFile()** 被调用。

Open，**Save**，**SaveAs** 这些行为和 **New** 行为相似，所以略去这个部分说明 **recentFileActions** 的实现。

recentFileActions 是一个 action 数组。里面的 action 被隐藏起来并连接到 **openRecentFile()** 槽函数。以后我们会讨论这些最近使用的文件是怎么可见的和被使用的。现在看看行为 **Options** 菜单里的 **ShowGrid**。

ShowGrid 是一个可选取的行为，菜单的旁边有一个选择的记号。在工具栏上这一项是个套索形式的工具条。当它被按下时，SpreadSheet 组件显示一个网格。

ShowGrid 和 Auto_Recalculate 是独立的行为。Qt 的类 QActionGroup 也提供多选一的行为。

现在我们实现菜单函数 createMenus()

```
void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(saveAsAction);
    separatorAction = fileMenu->addSeparator();
    for (int i = 0; i < MaxRecentFiles; ++i)
        fileMenu->addAction(recentFileActions[i]);
    fileMenu->addSeparator();
    fileMenu->addAction(exitAction);
    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(cutAction);
    editMenu->addAction(copyAction);
    editMenu->addAction(pasteAction);
    editMenu->addAction(deleteAction);
    selectSubMenu = editMenu->addMenu(tr("&Select"));
    selectSubMenu->addAction(selectRowAction);
    selectSubMenu->addAction(selectColumnAction);
    selectSubMenu->addAction(selectAllAction);
    editMenu->addSeparator();
    editMenu->addAction(findAction);
    editMenu->addAction(goToCellAction);
}
```



```

|
| toolsMenu = menuBar()->addMenu(tr("&Tools"));
| toolsMenu->addAction(recalculateAction);
| toolsMenu->addAction(sortAction);
| optionsMenu = menuBar()->addMenu(tr("&Options"));
| optionsMenu->addAction(showGridAction);
| optionsMenu->addAction(autoRecalcAction);
| menuBar()->addSeparator();
| helpMenu = menuBar()->addMenu(tr("&Help"));
| helpMenu->addAction(aboutAction);
| helpMenu->addAction(aboutQtAction);
|
| }

```

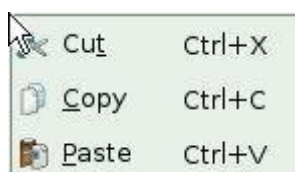
在 Qt 中，菜单是类 `QMenu` 的实例。`addMenu()` 函数创建一个带有文本的 `QMenu` 控件，把它添加到菜单条中。`QMainWindow::menuBar()` 函数返回一个 `QMenuBar` 指针，当程序第一次调用 `menuBar()` 时，菜单条就被创建好了

我们首先新建了 **File** 菜单，给它添加了 **New, Open, Save, SaveAs** 行为。一个分割条（**separator**）把功能相近的菜单组合起来并和其他菜单分开。使用 **for** 循环添加 **recentFileActions** 行为数组，然后又添加了 **exitAction** 行为。

我们保存了一个 **separator** 的指针，是因为这样可以控制它是否可见，如果最近文件没有时，这个 **separator** 就隐藏起来。

相同的方式创建 **Edit, Option, Help** 等菜单。

上下文菜单实现 `createContextMenu()`:



```
void MainWindow::createContextMenu()
{
    spreadsheet->addAction(cutAction);
    spreadsheet->addAction(copyAction);
    spreadsheet->addAction(pasteAction);
    spreadsheet->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

工具条 createToolbars()



```
void MainWindow::createToolbars()
{
    fileToolBar = addToolBar(tr("&File"));
    fileToolBar->addAction(newAction);
    fileToolBar->addAction(openAction);
    fileToolBar->addAction(saveAction);
    editToolBar = addToolBar(tr("&Edit"));
    editToolBar->addAction(cutAction);
    editToolBar->addAction(copyAction);
    editToolBar->addAction(pasteAction);
    editToolBar->addSeparator();
    editToolBar->addAction(findAction);
    editToolBar->addAction(goToCellAction);
}
```

3-3 创建状态条 (Setting Up the Status Bar)

完成菜单和工具条后，我们开始创建 Spreadsheet 应用程序的状态条。

在通常情况下，状态条提示两条信息：当前的格子位置，和当前格子的公式。状态条还能够根据情况显示程序当前的运行状态和其他临时的信息。

在 MainWindow 的构造函数中，调用 createStatusBar() 创建状态条。代码如下：

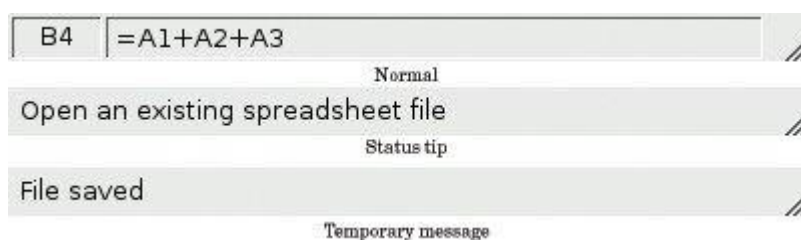
```

void MainWindow::createStatusBar()
{
    |   locationLabel = new QLabel(" W999 ");
    |   locationLabel->setAlignment(Qt::AlignHCenter);
    |   locationLabel->setMinimumSize(locationLabel->sizeHint());
    |   formulaLabel = new QLabel;
    |   formulaLabel->setIndent(3);
    |   statusBar()->addWidget(locationLabel);
    |   statusBar()->addWidget(formulaLabel, 1);
    |   connect(spreadsheet, SIGNAL(currentCellChanged(int, int, int, int)),
    |           this, SLOT(updateStatusBar()));
    |   connect(spreadsheet, SIGNAL(modified()),
    |           this, SLOT(spreadsheetModified()));
    |   updateStatusBar();
}

```

`QMainWindow::statusBar()`函数返回状态条的指针（和 `QMainWindow::menuBar()`一样，状态条在第一次调用这个函数的时候创建）。状态条的指示控件是 `QLabel`，根据程序的状态改变显示的信息。我们给 `formulaLabel` 设置了一个缩进，使它显示的信息离左边有一定的偏移。当 `QLabel` 控件加到状态条上时，它们的父对象就自动变成了状态条。

下图显示了状态条的两个文本框对控件的不同需求。格子位置占用的控件较小，当窗口的大小改变时，其他多余的空间由 `formulaLabel` 占据，这是因为我们在代码中 `statusBar()->addWidget(formulaLabel, 1)`，设置了 `formulaLabel` 的托放因数为 1。而 `locationLabel` 的托放因数为默认的 0，这表明它的大小是固定的。



`QStatusBar` 放置指示控件时，它首先查看控件的 `QWidget::sizeHint()` 得到控件的合适大小。然后托放那些可托放的控件填到其他可用的空间。一个控件的理想大小是控件自己根据它显示的内容决定的，当显示内容改变时，这个大小也会改变。为了避免频繁改变 `locationLabel` 的大小，我们设置它的最小值为它可能显示的最大的文本（“W199”），对齐方式为水平居中对齐（`Qt::AlignHCenter`）。

在函数的最后，我们连接 `Spreadsheet` 控件的信号到 `MainWindow` 的槽函数：`updateStatusBar()`和 `spreadsheetModified()`。

```
void MainWindow::updateStatusBar()
{
    locationLabel->setText(spreadsheet->currentLocation());
    formulaLabel->setText(spreadsheet->currentFormula());
}
```

`updateStatusBar()`更新位置和公式的显示。只要用户移动图标到一个新的格子这个函数就会被调用。在 `createStatusBar()`的最后它做为普通函数调用初始化状态条的显示，这样做是因为最开始 `Spreadsheet` 不发送 `currentCellChanged()`信号。

```
void MainWindow::spreadsheetModified()
{
    setWindowModified(true);
    updateStatusBar();
}
```

`spreadsheetModified()`槽函数设置 `windowModified` 属性为 `true`，用来更新窗口标题。然后调用 `updateStausBar()`反映当前状态的变化。

3-4 实现文件菜单（Implementing the File Menu）

在这一节，我们实现与文件菜单有关的槽函数和相关的私有函数，以使文件菜单可以工作，同时管理最近打开文件列表。

```
void MainWindow::newFile()
```

```

{

    if (okToContinue()) {

        spreadsheet->clear();

        setCurrentFile("");

    }

}

```

`newFile()`槽函数在用户点击了 **File|New** 菜单或者工具条上的 **New** 按钮后调用。

`okToContinue()` 是一个私有函数，在这里如果需要存盘，程序会询问用户 “Do you want to save your changes ?（是否存盘提示）”，如果用户选择了 **Yes** 或者 **No**，函数返回 `true`，如果用户选择了 **Cancel**，返回 `false`。`Spreadsheet::clear()`函数清楚所有 `spreadsheet` 控件的格子和公式。`setCurrentFile()`也是一个私有函数，它更新窗口标题，重新设置 `curFile` 变量，更新最近打开的文件列表，为用户开始编辑没有名字的新文档做好准备。

```

bool MainWindow::okToContinue()

{

    if (isWindowModified()) {

        int r = QMessageBox::warning(this, tr("Spreadsheet"),

            tr("The document has been modified. "

                "Do you want to save your changes?"),

```

```

        QMessageBox::Yes | QMessageBox::Default,

        QMessageBox::No,

        QMessageBox::Cancel | QMessageBox::Escape);

    if (r == QMessageBox::Yes) {

        return save();

    } else if (r == QMessageBox::Cancel) {

        return false;

    }

}

return true;

}

```

在 `okToContinue()` 函数中，检查 `windowModified` 属性的状态，如果为 `true`，那么就会显示如下的消息框。这个消息框有 `Yes`，`No`，和 `Cancel` 按钮。`QMessageBox::Default` 说明 `Yes` 为默认的按钮，`QMessageBox::Escape` 说明按键 `Esc` 和 `Cancel` 按钮等效



咋一看，`QMessageBox::warning()` 看起来有些复杂，实际是很简单明了的。

```
QMessageBox::warning(parent, title, message, button0, button1, ...);
```

`QMessageBox` 还提供其他函数如：`information()`，`question()`和 `critical()`，每一个函数都有他们自己特殊的显示图标：



槽函数 `open()` 相应菜单 `File|Open`，它首先也是调用 `okToContinue()` 处理为保存的信息。然后使用 `QFileDialog::getOpenFileName()`，这个函数弹出一个对话框，让用户选择一个文件的名称，如果用户选择了一个文件，那么函数返回文件的名称，如果用户点击了 `Cancel` 按钮，则返回一个空字符串。

```
void MainWindow::open()
{

    if (okToContinue()) {

        QString fileName = QFileDialog::getOpenFileName(this,

                                                         tr("Open Spreadsheet"), ".",

                                                         tr("Spreadsheet files (*.sp)"));

        if (!fileName.isEmpty())
```

```

        loadFile(fileName);

    }

}

```

QFileDialog::getOpenFileName()的第一个参数是它的父控件。父子关系对于对话框来说和其他控件有些不同，一个对话框总是显示为一个窗口，如果它有父控件，那么它一般显示在父控件的中上位置，**A child dialog also shares its parent's taskbar entry.**（怎么准确翻译那，好像是共享父控件的一些东西，taskbar）

第二个参数是对话框使用的标题。第三个参数是显示的初始目录，++表示的是程序的当前目录。

第四个参数用来说明文件过滤器，即确定文件类型。文件过滤器由一个描述性的文本和通配符格式组成。如果我们在 spreadsheet 程序中除了支持自定义的文件格式外，还支持了 Comma-separated values 文件和 Lotus 1-2-3 文件，那么过滤器就要这样：

```

tr("Spreadsheet files (*.sp)\n"

"Comma-separated values files (*.csv)\n"

"Lotus 1-2-3 files (*.wk1 *.wks)")

```

loadFile()是一个私有函数，用来加载文件。把这段代码独立出来是因为在打开最近文件时我们还要使用它：

Spreadsheet::readFile()来读取硬盘的文件。如果读取成功，调用 setCurrentFile() 更新窗口标题。否则，该函数给出一个错误的提示框。通常，在低级别的控件中给出相信的错误信息是个好的习惯，这样可以清楚知道出错的原因。

```

bool MainWindow::loadFile(const QString &fileName)
{
    if (!spreadsheet->readFile(fileName)) {
        statusBar()->showMessage(tr("Loading canceled"), 2000);
        return false;
    }
}

```



```

setCurrentFile(fileName);

statusBar()->showMessage(tr("File loaded"), 2000);

return true;

}

```

不管成功与否，程序的状态条上都显示 2 秒（2000 毫秒）的状态信息，告诉用户程序的正在做的事情。

菜单 **File|Save** 是 `save()` 函数相应的。如果文件已经有了名字，或者是在磁盘上打开的，或者已经保存过，函数直接调用 `saveFile()`，文件名字不变。否则调用 `saveAs()`。

```

bool MainWindow::save()
{
    if (curFile.isEmpty()) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

bool MainWindow::saveFile(const QString &fileName)
{
    if (!spreadsheet->writeFile(fileName)) {
        statusBar()->showMessage(tr("Saving canceled"), 2000);
        return false;
    }

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
}

```

```

    return true;
}

bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
                                                    tr("Save Spreadsheet"), ".",
                                                    tr("Spreadsheet files (*.sp)"));

    if (fileName.isEmpty())
        return false;

    return saveFile(fileName);
}

```

菜单 **File|SaveAs** 相应函数为 `saveAs()`。`QFileDialog::getSaveFileName()`提示用户输入文件名。如果用户点击了 **Cancel** 按钮，函数返回 `false`，并将状态传递给调用者。如果文件已经存在，`getSaveFileName()`询问用户是否要覆盖。在 `getSaveFileName()`的一个默认参数就是是否要覆盖，默认参数为 `QFileDialog::DontConfirmOverwrite`。

当用户点击了 **File|Close** 菜单或者窗口标题栏上的关闭按钮，`QWidget::close()`就会被调用。并发送 `close()`信号。重新实现 `QWidget::closeEvent()`能够拦截这个消息，以便确定是否真的要关闭窗口，防止误操作。

```

void MainWindow::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}

```

如果需要存盘或者用户选择了 **Cancel**，那么就忽视这个事件，不关闭窗口。通常如果接受了这个事件，Qt 就会隐藏这个窗口。私有函数 `writeSettings()` 保存应用程序当前的设置。当最后一个窗口也关闭后，应用程序中止。如果不需要这个功能，可以设置 `QApplication` 的 `quitOnLastWindowClosed` 属性为 `false`。这样，程序会一直运行，直到我们调用函数 `QApplication::quit()`。

`setCurrentFile()` 函数中，我们让 `curFile` 这个私有变量保存当前正在编辑的文件的名字。这个变量保存的是全路径名，我们用函数 `strippedName()` 删除掉文件的路径，再在窗口的标题栏显示这个文件的名字。

```
void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    setWindowModified(false);
    QString shownName = "Untitled";
    if (!curFile.isEmpty()) {
        shownName = strippedName(curFile);
        recentFiles.removeAll(curFile);
        recentFiles.prepend(curFile);
        updateRecentFileActions();
    }

    setWindowTitle(tr("%1[*] - %2").arg(shownName)
                  .arg(tr("Spreadsheet")));
}

QString MainWindow::strippedName(const QString &fullFileName)
{
    return QFileInfo(fullFileName).fileName();
}
```

每一个 `QWidget` 都有一个 `windowModified` 属性，如果有文件没有保存，那么就设置为 `true`。否则设置为 `false`。在 Mac OS X 平台，如果有没有保存的文件，在窗口的标题栏

的关闭按钮旁有一个小点。在其他平台，在文件名后面加一个 “*” 表示。只要我们保持更新 `windowModified` 属性，把 “[*]” 放在合适的地方，Qt 就能够自动处理。

传递给 `setWindowTitle()` 的文本是：

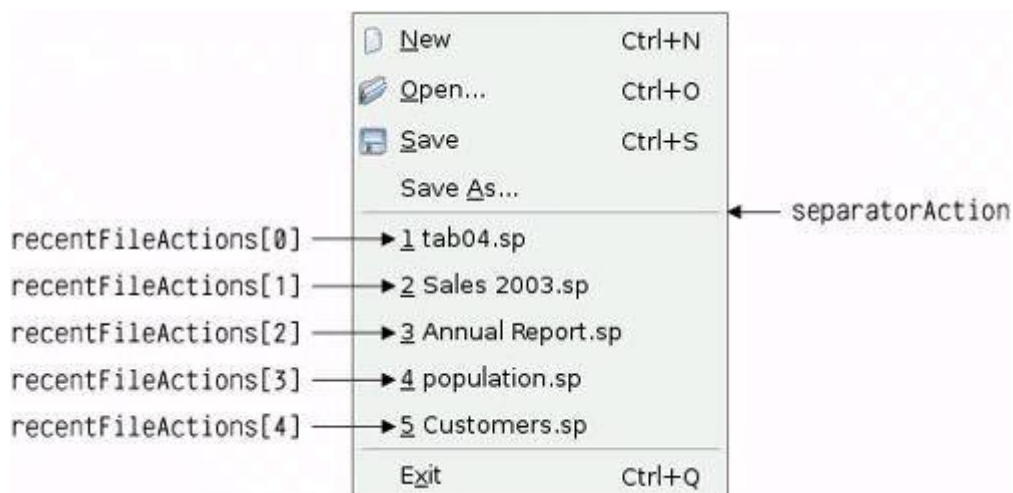
```
tr("%1[*] - %2").arg(shownName).arg(tr("Spreadsheet"))
```

`QString::arg()` 函数用自己的参数代替文本中的数字 `%n`，并返回结果字符串。上面语句有两个 `.arg()`，分别用来代替 `%1`，`%2`。如果文件名为 “`budget.sp`”，且没有加载翻译文件，那么显示的字符串就是 “`budget.sp[*] - Spreadsheet`”。也可以简写如下：

```
setWindowTitle(shownName + tr("[*] - Spreadsheet"));
```

但是使用 `arg()` 更加灵活且容易实现国际化。

打开文件后，我们要更新 `recentFiles`（最近打开文件列表）。使用 `removeAll()` 函数删除列表里的这个文件名，然后把它加在列表的前面。最后调用 `updateRecentFileActions()` 更新 **File** 菜单项。



首先我们用一个 `java` 样式的迭代器删除不存在的文件，因为有些文件可能在列表中但已经被删除掉了。`recentFiles` 的变量类型是 `QStringList`。第 11 章详细介绍容器，迭代器及它们与 `c++` 标准模板库（STL）的关系。

```
void MainWindow::updateRecentFileActions()
{
    QMutableStringListIterator i(recentFiles);
    while (i.hasNext()) {
```

```

        if (!QFile::exists(i.next()))
            i.remove();
    }

    for (int j = 0; j < MaxRecentFiles; ++j) {
        if (j < recentFiles.count()) {
            QString text = tr("&%1 %2")
                            .arg(j + 1)
                            .arg(strippedName(recentFiles[j]));
            recentFileActions[j]->setText(text);
            recentFileActions[j]->setData(recentFiles[j]);
            recentFileActions[j]->setVisible(true);
        } else {
            recentFileActions[j]->setVisible(false);
        }
    }
    separatorAction->setVisible(!recentFiles.isEmpty());
}

```

再看文件列表，后一部分我们使用了数组索引方式。每一个文件用一个&号，数字序号，一个空格，和文件名组成，行为名字就是这个字符串。例如，如果第一个文件是 C:\My Documents\tab04.sp，那么第一个行为显示的文本就是 “&1 tab04.sp”。

每一个行为都有一个大 data 项，存储 QVariant 类型的数据。QVariant 能够存储很多 c++数据类型和 Qt 数据类型，将在第 11 章进行介绍。这里我们存储文件的全名，这样在将来我们打开文件时就可以很方便的找到它。

如果用户选择了一个最近打开的文件，openRecentFile()就被调用。okToContinue()用来检查是否需要存盘。这个函数特别的地方就是用 QObject::sender() 得到信号的发送者。

```

void MainWindow::openRecentFile()
{

```

```

if (okToContinue()) {
    QAction *action = qobject_cast<QAction *>(sender());
    if (action)
        loadFile(action->data().toString());
}
}

```

`qobject_cast<T>()`实现基于 moc 生成的元信息的动态类型转换。它返回一个 `QObject` 类的子类对象的指针，如果这个对象不能转换成类型 `T`，返回一个空指针。和标准 c++ 的 `dynamic_cast<T>` 不同，`qobject_cast<T>()` 只在动态库内使用。在这个例子中，我们把一个 `QObject` 指针变为一个 `QAction` 指针。如果转换成功，调用 `loadFile()`，打开保存在 `QAction` 的 `data` 属性中保存的文件。

需要说明的是，因为我们知道发送者是一个 `QAction` 对象，如果使用 `static_cast<T>` 或者一个传统的 C 样式的类型转换都能正确。

3-5 使用对话框(Using Dialogs)

在这一节中，我们介绍 Qt 中对话框的调用：初始化对话框，显示对话框和与用户交互。我们将会使用在第二章创建的 **Find**，**Go-to-Cell** 对话框和 **Sort** 对话框。我们还会创建一个关于 (**About**) 对话框。

首先我们看一下 **Find** 对话框。我们希望用户能够在 **Find** 对话框和 **Spreadsheet** 应用程序的主窗口之间自由切换，所以 **Find** 对话框应该是无模式的。一个无模式的对话框就是在程序运行过程中不依赖其他窗口是否显示的对话框。

创建无模式对话框后，一般要连接信号和槽函数来响应用户输入。

当用户想在表格中查找文本时，**Find** 对话框就会显示。用户点击了 **Edit|Find** 菜单，`find` 槽函数就会调用，弹出 **Find** 对话框。这时对话框的有以下三种可能：

```

void MainWindow::find()
{

```

```

if (!findDialog) {
    findDialog = new FindDialog(this);
    connect(findDialog, SIGNAL (findNext(const QString &,
                                         Qt::CaseSensitivity)),
            spreadsheet, SLOT (findNext(const QString &,
                                         Qt::CaseSensitivity)));
    connect(findDialog, SIGNAL(findPrevious(const QString &,
                                             Qt::CaseSensitivity)),
            spreadsheet, SLOT(findPrevious(const QString &,
                                             Qt::CaseSensitivity)));
}
findDialog->show();
findDialog->activateWindow();
}

```

- 1、 第一次调用 Find 对话框
- 2、 用户已经调用过，但是给关闭了
- 3、 用户已经调用过，且仍然显示

如果 Find 对话框还不存在，那么创建对话框，连接 `findNext()`和 `findPrevious()`两个信号到相应的 `Spreadsheet` 槽函数。当然我们也可以在 `MainWindow` 的构造函数中创建，但是在需要的时候再创建可以加快程序的启动时间，而且，如果在程序运行期间没有调用这个对话框，还可以节约内存。

接着我们调用 `show()`和 `activateWindow()`确保窗口是可见的，激活的。单独调用 `show()`是能够显示并激活窗口的。但是如果调用时，Find 对话框是可见的，`show()`就不做任何事情，调用 `activateWindow()`就有必要了。所以后面几行还可以这样写：

```

        if (findDialog->isHidden()) {
            findDialog->show();
        } else {
            findDialog->activateWindow();
        }
    }

```

接着我们来看 **Go-to-Cell** 对话框是一个模式对话框。我们需要用户弹出对话框，在切换到程序的其他窗口前关闭它。模式对话框就是弹出后，在关闭之前，它阻止程序的其他消息和其他进程的干扰，也不能切换到其他窗口。我们以前使用过的文件对话框和消息提示对话框都是模式对话框。

使用 `show()` 显示的对话框是无模式对话框。用 `exec()` 显示的对话框是模式对话框。如果对话框被接受，`QDialog::exec()` 函数返回 `true` (`QDialog::Accepted`)，否则返回 `false` (`QDialog::Rejected`)。在第二章创建 **Go-to-Cell** 对话框时，我们连接了 `ok` 按钮到 `accept()`，`cancel` 按钮连接到了 `reject()`。如果用户点击了 `Ok` 按钮，我们就把当前的网格设为编辑框中的值。

```

void MainWindow::goToCell()
{
    GoToCellDialog dialog(this);
    if (dialog.exec()) {
        QString str = dialog.lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                    str[0].unicode() - 'A');
    }
}

```


函数 `QTableWidget::setCurrentCell()` 需要两个参数：一个行序号和一个列序号。在 Spreadsheet 程序中，网格 A1 对应(0,0)，B27 对应(26,1)。为了从 `QLineEdit::text()` 中返回的 `QString` 得到行序号，使用 `QString::mid()` 函数（得到从 `mid()` 中指定的位置到字符串的最后）然后用 `QString::toInt()` 得到整数值后减 1。至于列序号，我们得到字符串的第一个字符减去字母"A"的 unicode 的数字值。在创建这个对话框的时候，我们使用了 `QRegExpValidator` 确保能够得到正确的格式。

`goToCell()` 和以前创建控件的代码不同，这一次是在栈上创建 `GoToCellDialog`。如果多写一行代码，可以用 `new` 和 `delete` 实现：

```
void MainWindow::goToCell()
{
    GoToCellDialog *dialog = new GoToCellDialog(this);
    if (dialog->exec()) {
        QString str = dialog->lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                     str[0].unicode() - 'A');
    }
    delete dialog;
}
```

在栈上创建对话框是一个常用的编程模式，因为我们使用完这个控件以后就不再需要了，在调用完函数后能够自动析构它。

现在我们看 `sort` 对话框。`Sort` 对话框是一个模式对话框，使用户能够按列排序选中的区域。

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableWidgetItemSelectionRange range = spreadsheet->selectedRange();
    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());
    if (dialog.exec()) {
```

```

SpreadsheetCompare compare;

compare.keys[0] =

    dialog.primaryColumnCombo->currentIndex();

compare.keys[1] =

    dialog.secondaryColumnCombo->currentIndex() - 1;

compare.keys[2] =

    dialog.tertiaryColumnCombo->currentIndex() - 1;

compare.ascending[0] =

    (dialog.primaryOrderCombo->currentIndex() == 0);

compare.ascending[1] =

    (dialog.secondaryOrderCombo->currentIndex() == 0);

compare.ascending[2] =

    (dialog.tertiaryOrderCombo->currentIndex() == 0);

spreadsheet->sort(compare);

}

}

```

在 sort()函数中，我们使用了同 goToCell()同样的模式：

- 1、在栈上创建对话框并初始化。
- 2、用 exec()显示对话框。
- 3、如果用户点击了 ok，得到用户在对话框控件中的输入并使用这个字符串。

函数 setColumnRange()得到选定的列，下图是一个排序的例子，B 列为主排序列，A 列为第二排序列，按降序排列。例如使用途中的选定区域，range.leftColumn()得到 0，'A'+0='A'，range.rightColumn()得到 2，'A'+2='C'。

	A	B	C	
1	George	Washington	1789-1797	
2	John	Adams	1797-1801	
3	Thomas	Jefferson	1801-1809	
4	James	Madison	1809-1817	
5	James	Monroe	1817-1825	
6	John Quincy	Adams	1825-1829	
7	Andrew	Jackson	1829-1837	
8				

(a) Before sort

	A	B	
1	John	Adams	179
2	John Quincy	Adams	182
3	Andrew	Jackson	182
4	Thomas	Jefferson	180
5	James	Madison	180
6	James	Monroe	181
7	George	Washington	178
8			

(b) After sort

compare 对象存储第一，第二，第三排序列和它们排序顺序（在下一章我们将会对 SpreadsheetCompare 进行定义）。在 Spreadsheet::sort() 中会使用到这个对象。keys 数组存储的是要排列的列序号。例如，如果选定的区域是从 C2 到 E5，列 C 的位置就是 0，ascending 数组是每一列排序的顺序。QComboBox::currentIndex() 得到当前选定项目的序号，顺序是从 0 开始。对于第一，第二排序列。还需要用当前值减去 "None" 项目的值。

sort() 函数已经可以工作了，但是有点脆弱。它假定了 Sort 对话框只能用这种固定的方式实现，有下拉框，需要 None 项目。如果我们重新设计 Sort 对话框，我们还要重写这段代码。如果只在一个地方这样调用了，维护一次也就够了。但是如果在多个地方都使用了这个对话框，那么维护这些代码就成了程序员的恶梦。

一个更加强壮的方法是让 SortDialog 自己创建 Spreadsheetcompare 对象，这样可以大大减少了 MainWindow::sort() 的代码。

```
void MainWindow::sort()
{
    SortDialog dialog(this);

    QTableWidgetItemSelectionRange range = spreadsheet->selectedRange();

    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());

    if (dialog.exec())
        spreadsheet->performSort(dialog.comparisonObject());
}
```

```
}
```

这样控件之间的耦合度就小多了，如果多次使用了这个对话框，这是一个非常正确的选择。

一个让程序更加强壮的方法是在初始化 SortDialog 对话框的时候传递 Spreadsheet 对象的指针，是对话框能够直接操作 Spreadsheet。这样 SortDialog 只是作为一个控件，使 SortDialog 更加通用，MainWindow::sort()函数也更加简单：

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    dialog.setSpreadsheet(spreadsheet);
    dialog.exec();
}
```

这个 sort()函数和第一个 sort()函数相比：这里调用函数不需要知道对话框的实现细节，也不需给对话框提供任何数据结构。当对话框需要适应数据的实时改变时这样实现很必要。第一种方法调用函数很脆弱，同样如果数据结构改变了，最后一种方法也会失败。

有些程序员坚持用一种方式使用对话框。这样的好处是简单，易于实现，但是同时就失去了其他实现模式的优点。至于到底用那种模式则需要根据实际情况而定。

最后我们实现 About 对话框。我们也可以象创建 Find, Go-to-Cell 对话框一样实现一个用户子定义的对话框来显示程序的有关信息，但是由于大多 About 对话框的样式都是一样的，所以 Qt 给出了一个简单的解决方案。

```
void MainWindow::about()
{
    QMessageBox::about(this, tr("About Spreadsheet"),
        tr("<h2>Spreadsheet 1.1</h2>"
            "<p>Copyright &copy; 2006 Software Inc."
            "<p>Spreadsheet is a small application that "
            "demonstrates QAction, QMainWindow, QMenuBar, "
            "QStatusBar, QTableWidgetItem, QToolBar, and many other "
            "Qt classes."));
```

}

调用 `QMessageBox::about()` 静态函数可以得到下图这样的 About 对话框。除了对话框的图标外，这和 `QMessageBox::warning()` 显示的对话框很相似。



到目前为止我们已经使用了几个 `QMessageBox` 和 `QFileDialog` 的静态函数。这些函数创建一个对话框，进行初始化然后调用 `exec()` 显示出来。当然，首先创建 `QMessageBox` 或者 `QFileDialog`，然后显式调用 `exec()` 或者 `show()` 也是可以的，并且一样方便。

3-6 存储设置 (Storing Settings)

在 `MainWindow` 构造函数中，我们调用 `readSettings()` 得到应用程序保存的设置选项。同样在 `closeEvent()` 中我们调用 `writeSettings()` 保存当前应用程序的设置。这是 `MainWindow` 需要实现的最后两个成员函数。

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");
    settings.setValue("geometry", geometry());
    settings.setValue("recentFiles", recentFiles);
    settings.setValue("showGrid", showGridAction->isChecked());
    settings.setValue("autoRecalc", autoRecalcAction->isChecked());
}
```

在 `writeSetting()` 中保存程序主窗口的几何信息（位置和大小），最近打开的文件列表，是否显示网格和是否自动计算属性。

在缺省情况下，`QSettings` 根据平台特性存储应用程序的设置。在 **Windows** 中使用注册表；在 **Unix** 中把数据存贮在文本文件中；在 **Mac OS X** 平台上使用 **Core Foundation Preference API**。

在构造函数中传递软件厂商和应用程序的名字，这些信息用来确定特定平台下应用程序设置文件的位置。

`QSettings` 使用键值对存贮设置。键相当于一个文件系统目录，子键通过路径样式的语法确定（例如 `findDialog/matchCase`），或者使用 `beginGroup()` 和 `endGroup()` 对。

```
settings.beginGroup("findDialog");
settings.setValue("matchCase", caseCheckBox->isChecked());
settings.setValue("searchBackward", backwardCheckBox->isChecked());
settings.endGroup();
```

对应的值可是 `bool`, `double`, `QString`, `QStringList`, 或者是其他 `QVariant` 支持的数据类型，也包括注册过的用户自定义类型。

```
void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");
    QRect rect = settings.value("geometry",
                                QRect(200, 200, 400, 400)).toRect();
    move(rect.topLeft());
    resize(rect.size());
    recentFiles = settings.value("recentFiles").toStringList();
    updateRecentFileActions();
    bool showGrid = settings.value("showGrid", true).toBool();
    showGridAction->setChecked(showGrid);

    bool autoRecalc = settings.value("autoRecalc", true).toBool();
    autoRecalcAction->setChecked(autoRecalc);
}
```

`readSettings()`函数读取 `writeSettings()`保存的程序设置。函数 `value()`中的第二个参数是在没有这项设置时取的默认值。一般默认值在第一次运行程序时使用。在读取最近程序列表时，没有第二个参数，则程序第一次运行时为空。

Qt 提供了 `QWidget::setGeometry()`函数做为 `QWidget::geometry()`的补充。但是在 X11 上由于窗口管理器多样的原因不能准确实现。所以我们使用 `move()`和 `resize()`。在 <http://doc.trolltech.com/4.1/geometry.html> 中有详细解释。

`MainWindow` 中要保存的设置，在 `readSettings()`和 `writeSettings()`只是一种可行方法之一。`QSettings` 对象可以在程序运行过程中的任何时间任何位置读取和修改这些设置。

到现在为止，我们已经完成了 `MainWindow` 的实现。在一下的几个小节中，我们将要讨论让 `Spreadsheet` 程序支持多文档，怎样显示启动画面。在下一章中，我们将会实现程序功能，如处理公式，排序等。

3-7 多文档（Multiple Documents）

现在我们开始实现 `Spreadsheet` 程序的 `main()`函数：

```
include <QApplication>

include "mainwindow.h"

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

{

    QApplication app(argc, argv);

    MainWindow mainWin;

    mainWin.show();

    return app.exec();

}
```

这个 `main()`函数和以前实现的稍有不同：我们在堆栈上创建了 `MainWindow` 实例。在程序中止的时候，`MainWindow` 自动销毁。

使用以上的 `main()`函数，`Spreadsheet` 程序提供一个主窗口，一次只能处理一个文档。如果我们希望在同时处理多个文档，我们就要同时启动多个 `Spreadsheet` 程序。这对用户来

说很不方便，他们更喜欢在一个应用程序实例中打开多个窗口，就如同一个 web 浏览器可以同时打开多个窗口一样。

为了处理多文档，我们需要对 **Spreadsheet** 进行一点修改。首先 **File** 菜单要进行修改：

- **File|New** 创建一个带有空文档的新的主窗口，而不是重新使用已经存在的主窗口。
- **File|Close** 关闭当前主窗口。
- **File|Exit** 关闭所有窗口。

在原来的程序版本中没有 **Close** 菜单项，因为这个菜单项和 **Exit** 时一样。

新的 `main()` 函数变为这样：

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

    MainWindow *mainWin = new MainWindow;

    mainWin->show();

    return app.exec();
}
```

由于有了多个主窗口，我们需要用 `new` 创建 `MainWindow` 实例，这样当保存后就可以使用 `delete` 删除 `MainWindow` 实例。

槽函数 `MainWindow::newFile()` 要改成这样：

```
void MainWindow::newFile()
{
    MainWindow *mainWin = new MainWindow;

    mainWin->show();
}
```

我们只是简单的创建一个新的 `MainWindow` 实例。奇怪的是我们没有保存新窗口的指针，这是因为 Qt 会为我们记录所有窗口的地址。

在 `MainWindow::createActions()` 中，我们需要 `Close` 和 `Exit` 行为：

```
void MainWindow::createActions()
{
    ...

    closeAction = new QAction(tr("&Close"), this);
```



```

closeAction->setShortcut(tr("Ctrl+W"));

closeAction->setStatusTip(tr("Close this window"));

connect(closeAction, SIGNAL(triggered()), this, SLOT(close()));

exitAction = new QAction(tr("E&xit"), this);

exitAction->setShortcut(tr("Ctrl+Q"));

exitAction->setStatusTip(tr("Exit the application"));

connect(exitAction, SIGNAL(triggered()),

        qApp, SLOT(closeAllWindows()));

...

}

```

函数 `QApplication::closeAllWindows()` 关闭所有应用程序的窗口，除非有些窗口拒绝了
这个关闭事件。这个功能是很需要的。因为只要关闭一个窗口，`MainWindow::closeEvent()`
就会调用，这样我们就不用担心有些文档没有存盘。

到现在，看起来我们的程序已经能够处理多文档窗口了。但是还有一个隐藏的问题：
如果用户不停的创建关闭主窗口，那么机器最终会耗尽所有内存。因为我们在 `newFile()` 中
不停的创建主窗口 但是却没有删除它。用户关闭一个窗口只是把它隐藏，窗口实例一致在
内存中。在内存中不用的主窗口越来越多，这个问题就很严重了。

解决这个问题很简单，在构造函数中我们把窗口属性设置为 `Qt::WA_DeleteOnClose` 就
可以了：

```

MainWindow::MainWindow()

{

    ...

    setAttribute(Qt::WA_DeleteOnClose);

    ...

}

```

这样 Qt 就在窗口关闭的同时销毁它。QWidgets 有许多可以影响行为的属性，
`Qt::WA_DeleteOnClose` 只是其中的一个。

内存泄漏只是我们需要处理的问题之一。在我们的原来程序设计中，我们假定只有一个
主窗口。如果创建了多个窗口，那么每一个窗口都已个最近打开的文档和自己的设置。很
明显，最近打开的文档应该是对程序全局有效的。我们可以声明 `recentFiles` 为静态变量，这

样在整个程序运行期间就只有一份拷贝。这样我们就需要在调用 `updateRecentFileActions()` 函数时，所有的主窗口都要调用，代码实现如下：

```
foreach (QWidget *win, QApplication::topLevelWidgets()) {  
    if (MainWindow *mainWin = qobject_cast<MainWindow *>(win))  
        mainWin->updateRecentFileActions();  
}
```

上面的代码用到了 Qt 的 `foreach`（这将在第 11 章介绍）遍历所有程序窗口，类型为 `MainWindow` 的窗口全部调用 `updateRecentFileActions()`。选项 `ShowGrid` 和 `AutoRecalculate` 也要这样处理进行同步，确保同一个文件不会调用两次。

一个主窗口只能处理一个文档的程序称为 **SDI**（single document interface）程序。能处理多个文档的程序称之为 **MDI**（Multiple document interface）程序。Qt 能够在所有操作系统平台上支持 **SDI** 和 **MDI** 程序。

3-8 启动画面（**Splash Screens**）

许多应用程序在启动时显示一个画面。在程序启动很慢时，程序员用这种方法可以让启动时间感觉不那么长，还有用这个画面满足市场的一些要求。给 Qt 应用程序加一个启动画面很简单，需要使用的类是 `QSplashScreen`。

在窗口没有显示之前，`QSplashScreen` 显示一个图片，他还可以在图片上显示文字信息提示用户当前程序初始化的进度。一般情况下，启动画面代码在 `main()` 函数中，加在调用 `QApplication::exec()` 之前。

下面的一个程序的 `main()` 函数使用 `QSplashScreen` 显示一个启动画面，同时显示加载的模块信息和网络连接情况。

```
int main(int argc, char *argv[])  
{  
    QApplication app(argc, argv);  
    QSplashScreen *splash = new QSplashScreen;  
    splash->setPixmap(QPixmap(":/images/splash.png"));  
    splash->show();  
    Qt::Alignment topRight = Qt::AlignRight | Qt::AlignTop;
```

```
splash->showMessage(QObject::tr("Setting up the main window..."),
                    topRight, Qt::white);

MainWindow mainWin;

splash->showMessage(QObject::tr("Loading modules..."),
                    topRight, Qt::white);

loadModules();

splash->showMessage(QObject::tr("Establishing connections..."),
                    topRight, Qt::white);

establishConnections();

mainWin.show();

splash->finish(&mainWin);

delete splash;

return app.exec();
}
```

Spreadsheet 程序的用户界面部分我们已经完成了。在下一章中我们将会实现表格的核心功能。

第四章序及第三章小节

第三章已经翻译完了，看着从 3-8 到 3-1 倒序排列的文章，还是很有成就感的。

在这一章中上传了一些必要的图片，这样对理解程序会很有好处。我翻译这本书的本意是给自己以后做参考，之所以加入到 **blog** 大军中是因为有趣和好奇，也想用这种方式激励自己。到现在我已经博了一个多月了，点击量一直稳步上升。我无法想象看我的文章的都是一些什么样的人，看了之后是何感想。如果有些朋友觉得这些文字还有点用，那么我建议参考原书看这些文章，尽管我的翻译时态度很认真，但我的翻译水平只停留在我自己翻译完三天内看懂而已。如果有朋友需要原书可以留邮箱给我，当然也可以去 **google** 和百度上去搜，我也是这样得到的。看看 **google** 和百度的热火，就知道现在也是一个“搜”时代，只有不知道，没有搜不到。

话说第三章，我以前看过，这次翻译了一遍以后仍然觉得很有帮助，特别是刚刚开始 **GUI** 编程的初学者和 **Qt** 的初学者，可以仔细看看。在这一章用到了很多 **Qt** 的类及其常用的功能，这在程序设计中是很常用到的，当然这些对于 **Qt** 的庞大类库来说只是冰山一角，但是如果你顺藤摸瓜，查找每一个类的说明文档，就会发现很多有用或者有意思的功能。

还有一个很有参考意义的是程序的结构设计，对于设计一个小型程序，这一章的设计模式简单实用规范整洁，我认为很值得学习。

当然如果现在就想编译运行程序，一定是编译不过的，因为缺少很多东西，但是如果把设计到 **Spreadsheet** 具体功能的代码去掉，也是可以的。第四章就是讲述 **Spreadsheet** 的具体功能的实现，在这一章里，还会介绍怎样读取保存文件，实现剪贴板的操作等很多内容。

4-1 中央控件（The Central Widget）

在 **QMainWindow** 的中心区域可以放置各类控件。例如下面列举的：

- 1、使用标准 **Qt** 控件：标准的 **Qt** 控件如 **QTableWidget** 或者 **QTextEdit** 可以作为中央控件。这时，读取和保存文件等程序功能需要在其他地方实现（例如在 **QMainWindow** 的子类中）

- 2、使用用户自定义控件：一些特殊程序需要在一个用户控件中显示数据。例如，图标编辑程序就要把 **IconEditor** 做为中央控件。第五章将会介绍怎么样在 **Qt** 中实现自定义的用户控件。
- 3、使用带有布局管理器的空白控件：有时候，一些程序的中央控件由多个控件组成。这时，可以用一个控件做为其他控件的父控件，使用布局管理器管理其他子控件的位置和大小。
- 4、使用分隔条：另一种使用多个控件的方式是使用 **QSplitter**（分隔条）。**QSplitter** 可以水平方式或垂直方式排列子控件，用中间的分隔线控制控件的大小分隔条里面可以包含各种控件，包括其他的分隔条。
- 5、使用 **MDI** 工作控件。在 **MDI** 程序中，中央控件由 **QWorkspace** 控件占据。每一个 **MDI** 窗口是这个控件的一个子控件。布局，分隔条和 **MDI** 工作空间可以同标准 **Qt** 控件一起使用，也可以和自定义控件使用，第六章会详细介绍。

在 **Spreadsheet** 程序中，一个 **QTableWidget** 子类做为它的中央控件。

QTableWidget 已经提供了大部分我们需要的表格功能，但是它不支持剪贴板，不能理解如 "**=A1+A2+A3**" 这样的公式。我们将在 **Spreadsheet** 类中实现这些功能。

4-2 从 **QTableWidget** 继承（**Subclassing QTableWidget**）

类 **Spreadsheet** 从 **QTableWidget** 继承。**QTableWidget** 是一个表示二维离散数组的表格。它根据给定坐标显示当前用户指定的网格。当用户在一个空的网格中输入一些文本时，**QTableWidget** 自动创建一个 **QTableWidgetItem** 对象保存输入的文本。

现在我们来实现这个类，首先是头文件 **spreadsheet.h**，首先前向声明两个类 **Cell** 和 **SpreadsheetCompare**。

```
#ifndef SPREADSHEET_H
#define SPREADSHEET_H
#include <QTableWidget>

class Cell;

class SpreadsheetCompare;

class Spreadsheet : public QTableWidget
{
    Q_OBJECT
public:
```

```

    Spreadsheet(QWidget *parent = 0);

    bool autoRecalculate() const { return autoRecalc; } //内联函数

    QString currentLocation() const;

    QString currentFormula() const;

    QTableWidgetItemSelectionRange selectedRange() const;

    void clear();

    bool readFile(const QString &fileName);

    bool writeFile(const QString &fileName);

    void sort(const SpreadsheetCompare &compare);

public slots:

    void cut();

    void copy();

    void paste();

    void del();

    void selectCurrentRow();

    void selectCurrentColumn();

    void recalculate();

    void setAutoRecalculate(bool recalc);

    void findNext(const QString &str, Qt::CaseSensitivity cs);

    void findPrevious(const QString &str, Qt::CaseSensitivity cs);

signals:

    void modified();

private slots:

    void somethingChanged();

private:

    enum { MagicNumber = 0x7F51C883, RowCount = 999, ColumnCount =

26 };

    Cell *cell(int row, int column) const;

    QString text(int row, int column) const;

    QString formula(int row, int column) const;

```

```

        void setFormula(int row, int column, const QString &formula);

        bool autoRecalc;
};

class SpreadsheetCompare
{
public:

    bool operator()(const QStringList &row1,
                    const QStringList &row2) const;

    enum { KeyCount = 3 };

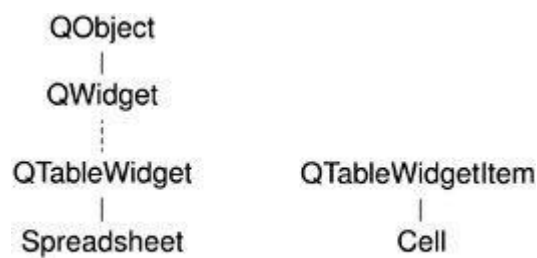
    int keys[KeyCount];

    bool ascending[KeyCount];
};

#endif

```

Figure 4.1. Inheritance trees for Spreadsheet and Cell



文本，对齐等这个 **QTableWidget** 网格的属性存储在 **QTableWidgetItem** 类里。**QTableWidgetItem** 类不是一个控件类，而是一个单纯保存数据的类。类 **Cell** 从 **QTableWidgetItem** 继承的，将在下一节介绍。

在第三章我们实现 **MainWindow** 类的时候我们用到了 **Spreadsheet** 的一些函数。如在 **MainWindow::newFile** 中调用 **clear()** 将表格置空。我们也用到了 **QTableWidget** 的一些函数，如 **setCurrentCell()** 和 **setShowGrid()** 就多次调用过。**Spreadsheet** 提供了很多槽函数来相应 **Edit**, **Tools** 和 **Options** 等菜单的动作。信号 **modified()** 在表格发生变化时给出通知。

私有槽函数 `somethingChanged()` 在 `Spreadsheet` 类内部使用。在类的私有部分，我们声明了三个常数，四个函数和一个变量。在头文件的最后定义了类 `SpreadsheetCompare`。现在我们看一下源文件 `spreadsheet.cpp`:

```
#include <QtGui>
#include "cell.h"
#include "spreadsheet.h"

Spreadsheet::Spreadsheet(QWidget *parent)
    : QTableWidget(parent)
{
    autoRecalc = true;
    setItemPrototype(new Cell);
    setSelectionMode(ContiguousSelection);
    connect(this, SIGNAL(itemChanged(QTableWidgetItem *)),
            this, SLOT(somethingChanged()));
    clear();
}

void Spreadsheet::clear()
{
    setRowCount(0);
    setColumnCount(0);
    setRowCount(RowCount);
    setColumnCount(ColumnCount);
    for (int i = 0; i < ColumnCount; ++i) {
        QTableWidgetItem *item = new QTableWidgetItem;
        item->setText(QString(QChar('A' + i)));
        setHorizontalHeaderItem(i, item);
    }
    setCurrentCell(0, 0);
}

Cell *Spreadsheet::cell(int row, int column) const
```



```

{
    return static_cast<Cell *>(item(row, column));
}

QString Spreadsheet::text(int row, int column) const
{
    Cell *c = cell(row, column);
    if (c) {
        return c->text();
    } else {
        return "";
    }
}

QString Spreadsheet::formula(int row, int column) const
{
    Cell *c = cell(row, column);
    if (c) {
        return c->formula();
    } else {
        return "";
    }
}

void Spreadsheet::setFormula(int row, int column,
                             const QString &formula)
{
    Cell *c = cell(row, column);
    if (!c) {
        c = new Cell;
        setItem(row, column, c);
    }
    c->setFormula(formula);
}

```

```

}
QString Spreadsheet::currentLocation() const
{
    return QChar('A' + currentColumn())
        + QString::number(currentRow() + 1);
}
QString Spreadsheet::currentFormula() const
{
    return formula(currentRow(), currentColumn());
}
void Spreadsheet::somethingChanged()
{
    if (autoRecalc)
        recalculate();
    emit modified();
}

```

通常，用户在一个空的网格中输入文本时，`QTableWidget` 将会自动创建 `QTableWidgetItem` 对象来保存这些文本。在 `spreadsheet` 程序中，我们使用 `Cell` 代替 `QTableWidgetItem`。在构造函数中，`setItemProtoType()` 完成这个替换。实现方式是当需要创建一个新的项目时，`QTableWidget` 克隆传递给 `setItemProtoType()` 函数中的项目。

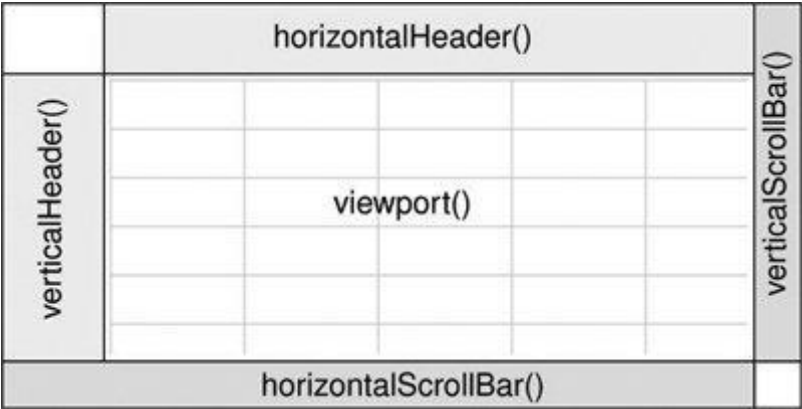
在构造函数中，我们设置选择方式 `QAbstractItemView::ContiguousSelection` 使表格能够选择一个单一的网格。连接表格控件的信号 `itemChanged()` 和 `somethingChanged()` 槽函数，这样当用户编辑了一个网格时，`somethingChanged()` 能够被调用。最后，我们调用 `clear()` 清空表格，设置列标头。

在构造函数中调用 `clear()` 用来初始化表格。在 `MainWindow::newFile()` 中也调用了这个函数。如果使用函数 `QTableWidget::clear()` 也可清除所有的网格和选择，但这样不能改变标题头的个数。我们首先把表格从新定义为 `0×0`，这样全部清除了表格和标题头。然后把表格重新定义为 `ColumnCount×RowCount` (`26× 999`)，让水平标题头为

QTableWidgetItem 类型，文本为"A"到"Z "。垂直标题栏会自动设置为 1，2，到 999。最后把光标移动到 A1。

QTableWidget 由几个子控件组成。它的最上面有一个水平的 QHeaderView，最左边有一个垂直的 QHeaderView 和两个 QScrollBars。中间区域是一个特殊的 viewport 控件，这个控件可以显示网格。这些组成控件可以通过 QTableView 和 QAbstractScrollArea 的函数进行操作。QAbstractScrollArea 提供了一个可以滚动的 viewport 和两个滚动条。它的子类 QScrollArea 会在第六章介绍到。

Figure 4.2. QTableWidget 's constituent widgets



在 Items 中保存数据:

在 Spreadsheet 应用程序中，每一个非空的网格都是一个独立的 QTableWidgetItem 对象。这种在 Item 中保存数据的方法被 QListWidget 和 QTreeWidget 所采用，对应这两个控件的 Item 类分别为 QListWidgetItem 和 QTreeWidgetItem。

Qt 的 Item 类还可以作为数据存储使用。比如，QTableWidgetItem 也保存了一些属性如文本，字体，颜色，图标等，还有一个指向 QTableWidget 的指针。这个 Item 还可以保存 QVariant 类型的数据，包括注册的自定义类型。把这个类作为基类，我们还可以提供其他功能。

其他的工具是在 item 类可以提供一个空指针来保存用户数据。在 Qt 中更加好用的方法是使用 setData()，把 QVariant 类型的数据保存起来。如果需要一个空类型指针，也可以继承 item 类，添加一个空类型指针成员数据。

对于那些更为复杂的数据处理，如大量的数据，复杂的数据项，数据库数据和多种数据显示方式，Qt 提供了一套 model/view 类将数据和显示分离出来，第十章介绍了这个特性。

私有函数 `cell()` 返回给定的行数和列数的 `Cell` 对象。它和 `QTableWidget::item()` 是一样的，只是它返回的是 `Cell` 类型的指针，`QTableWidget::item()` 返回的是 `QTableWidgetItem` 类型的指针。

私有函数 `text()` 返回给定的网格的文本。如果 `cell()` 返回空指针，网格为空，则返回空字符。

函数 `formula()` 返回的是网格的公式。大多数情况下，网格的公式和文本是一样的。例如，公式 "hello" 和字符 "hello" 是一样的，如果用户输入了 "hello"，网格的文本就显示为 hello。但是下面会是例外：

- 1、如果公式是一个数字，那么网格的文本也是数字。
- 2、如果公式是单引号开头，公式的其他部分就是文本。如公式 '12345，网格公式就是 "12345" 。
- 3、如果公式由等号 "=" 开头，代表一个数学公式。如果 A1 为 12，A2 为 6，那么公式 "=A1+A2" 就是 18。

把公式转换为值的任务是由类 `Cell` 完成的。此时需要记住的是网格中显示的文本是经过公式计算的结果，而不是公式本身。

私有函数 `setFormula()` 用来给一个指定的网格设置公式。如果网格有 `Cell` 对象，那就使用这个对象。否则，我们创建一个新的 `Cell` 对象。最后我们调用 `Cell` 自己的 `setFormula()` 函数，在网格上显示公式结果。我们不用删除 `Cell` 对象，在适当的时候，`QTableWidget` 会自动删除这些对象。

函数 `currentLocation()` 返回当前网格的位置，字母显示的列和行号。在 `MainWindow::updateStatusBar()` 调用在状态条上显示位置。

函数 `currentFormula()` 返回当前网格的公式。 `MainWindow::updateStatusBar()` 调用了这个函数。

私有槽函数 `somethingChanged()` 中，如果 `auto-recalculate` 为真，那么重新计算整个表格。然后发送 `modified()` 信号。

4-3 读取和保存（Loading and Saving）

我们使用 `QFile` 和 `QDataStream` 来实现 `Spreadsheet` 文件的保存和读取。这两个类都是提供了平台无关的二进制 I/O。

首先是保存文件的代码：

```

bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::WriteOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                               tr("Cannot write file %1: %2.")
                               .arg(file.fileName())
                               .arg(file.errorString()));
        return false;
    }
    QDataStream out(&file);
    out.setVersion(QDataStream::Qt_4_1);
    out << quint32(MagicNumber);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
    }
    QApplication::restoreOverrideCursor();
    return true;
}

```

函数 `writeFile()` 由 `MainWindow::saveFile()` 调用把文件保存到磁盘上。如果保存成功返回 `true`，否则返回 `false`。

首先我们使用给定的程序名创建一个 `QFile` 对象，调 `open()` 打开这个文件准备写入。同时创建 `QDataStream` 对象将数据写入文件中。

在写数据之前，我们将程序的光标换成等待形式，数据写完后恢复原来的鼠标。函数退出时，`QFile` 的析构函数把文件自动关闭。

QDataStream 支持基本的 C++ 类型，也支持多种 Qt 类型。语法和标准 C++<iostream> 类是一样的。例如：

Out<<x<<y<<z; 把变量 x, y, z 写入数据流。

In>>x>>y>>z; 从数据流中读取数据到 x, y, z 中。

在不同的平台上，基本的 C++ 类型如 short, char, int, long, long long 会有不同的字长。最好把它们转换为 qint8, quint8, qint16, quint16, qint32, quint32, qint64, quint64，这些类型能确保字长是不随平台改变的。

Spreadsheet 程序的文件格式非常简单。Spreadsheet 程序开头部分是一个 32 位的标识数字（MagciNumber，在 spreadsheet.h 中定义的，一个二进制的随机数），这个数字后面是一系列的数据块，友一个行号，列号和公式组成。为了节省空间，不保存空的网格。

数据类型的二进制表示由类 QDataStream 决定。如：quint16 表示位两个字节。一个 QString 类型表示是字符创的长度和每一个字母的 Unicode 码组成。

自 Qt1.0 以来，Qt 数据类型的二进制表示有了很大变化。在未来的 Qt 版本中还可能有更多的改变，QDataStream 使用最近的 Qt 版本，但是它可以读取以前的版本。为了程序用新的 Qt 版本重新编译后能够更好的兼容，我们显式的给出 QDataStream 使用的版本为 7（QDataStream::Qt_4_1 定义为常量 7）

QDataStream 可以支持多种类型。如 QFile, QBuffer, QProcess, QTcpSocket 或者 QUdpSocket。Qt 还提供了类 QTextStream 能够读写文本文件。第 12 章详细介绍这些类。

读取文件如下：

```
bool Spreadsheet::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                               tr("Cannot read file %1: %2.")
                               .arg(file.fileName())
                               .arg(file.errorString()));
        return false;
    }
}
```

```

}
QDataStream in(&file);
in.setVersion(QDataStream::Qt_4_1);
quint32 magic;
in >> magic;
if (magic != MagicNumber) {
    QMessageBox::warning(this, tr("Spreadsheet"),
                          tr("The file is not a Spreadsheet file."));
    return false;
}
clear();
quint16 row;
quint16 column;
QString str;
QApplication::setOverrideCursor(Qt::WaitCursor);
while (!in.atEnd()) {
    in >> row >> column >> str;
    setFormula(row, column, str);
}
QApplication::restoreOverrideCursor();
return true;
}

```

函数 `readFile()` 和 `writeFile()` 很相似。这次文件的打开方式为 `QIODevice::ReadOnly` 而不是 `QIODevice::writeOnly`。设置 `QDataStream` 的版本为 7。写文件和读文件的版本必须一致。

如果文件的 **magic number** 号是正确的，调用 `clear()` 清空所有的表格，因为文件中只是保存了非空的网格数据，不能保证所有的网格都会设置，然后再读取网格数据。

4-4 实现 **Edit** 菜单（Implement the Edit menu）

现在我们开始实现菜单 **Edit** 相应的槽函数。

```
void Spreadsheet::cut()
{
    copy();
    del();
}
```

槽函数 `cut()` 相应 **Edit|Cut** 菜单，这里调用了两个函数，因为剪切的操作和拷贝然后删除是等价的。

```
void Spreadsheet::copy()
{
    QTableWidgetItemSelectionRange range = selectedRange();
    QString str;
    for (int i = 0; i < range.rowCount(); ++i) {
        if (i > 0)
            str += " ";
        for (int j = 0; j < range.columnCount(); ++j) {
            if (j > 0)
                str += " ";
            str += formula(range.topRow() + i, range.leftColumn() + j);
        }
    }
    QApplication::clipboard()->setText(str);
}

QTableWidgetItemSelectionRange Spreadsheet::selectedRange() const
{
    QList<QTableWidgetItemSelectionRange> ranges = selectedRanges();
    if (ranges.isEmpty())
        return QTableWidgetItemSelectionRange();
    return ranges.first();
}
```


函数 `copy()` 相应 `Edit|Copy` 菜单。首先得到当前的选择项（如果没有明确选择，返回当前的网格），然后把选择项的公式按顺序记录下来。行之间用换行符隔开，同一行中每一列之间用 `TAB` 隔开。



`QApplication::clipboard()` 可以得到系统的剪贴板。调用 `QClipboard::setText()` 把文本放到剪贴板上，这样应用程序中和其他需要文本的 `Qt` 程序就可以使用这些文本。用换行符和 `tab` 的形式把行列分开也被许多应用程序支持。

`QTableWidget::selectedRanges()` 返回所有的选择范围列表。在 `Spreadsheet` 构造函数中我们设置了选择模式为 `QAbstractItemView::contiguousSelection`，因此选择范围只能有一个。为了程序使用方便，定义了 `selectedRange()` 函数返回当前的选择范围。如果有选择范围，则返回第一个且也是唯一的一个选择范围。如果没有明确选择范围，则当前的网格为一个选择（由于 `ContiguousSelection` 选择模式）。但是为了程序中可能出现的 `bug`，也处理了选择为空的情况。

```
void Spreadsheet::paste()
{
    QTableWidgetItemSelectionRange range = selectedRange();
    QString str = QApplication::clipboard()->text();
    QStringList rows = str.split(' ');
    int numRows = rows.count();
    int numColumns = rows.first().count(' ') + 1;
    if (range.rowCount() * range.columnCount() != 1
        && (range.rowCount() != numRows
            || range.columnCount() != numColumns)) {
```

```

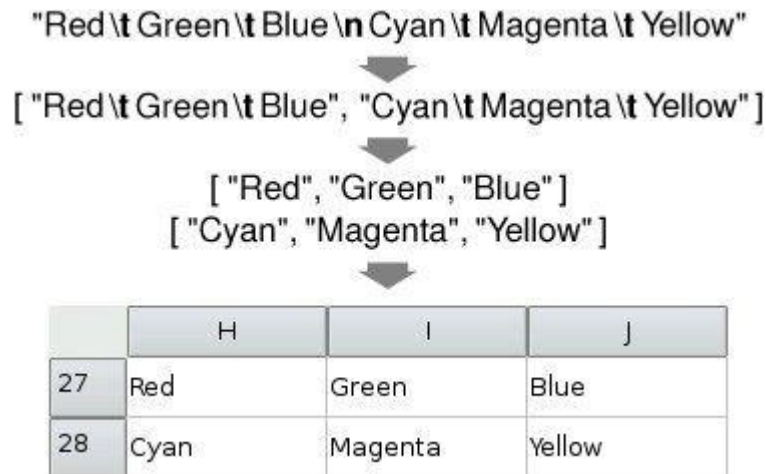
        QMessageBox::information(this, tr("Spreadsheet"),
            tr("The information cannot be pasted because the copy "
                "and paste areas aren't the same size.));
        return;
    }
    for (int i = 0; i < numRows; ++i) {
        QStringList columns = rows[i].split(' ');
        for (int j = 0; j < numColumns; ++j) {
            int row = range.topRow() + i;
            int column = range.leftColumn() + j;
            if (row < RowCount && column < ColumnCount)
                setFormula(row, column, columns[j]);
        }
    }
    somethingChanged();
}

```

菜单 **Edit|Paste** 的槽函数为 `paste()`。我们首先得到剪贴板里的文本，然后调用 `QString::split()` 按行把文本分成 `QStringList`。每一行为一个字符串。

接着我们确定拷贝区域的范围。行数为 `QStringList` 里 `QString` 的个数。列数为第一行中 `tab` 的个数加一。如果只有一个网格被选中，我们使用左上角的那个粘贴区域，否则使用当前选择范围为粘贴区域。

粘贴文本时，再一次调用 `QString::split()` 把一行文本分裂为每一列文本的组合。



```
void Spreadsheet::del()
{
    foreach (QTableWidgetItem *item, selectedItems())
        delete item;
}
```

函数 `del()` 相应菜单 `Edit|Delete`。它通过删除表格里选定的 `Cell` 对象清除网格。

`QTableWidgetItem` 发现 `QTableWidgetItem` 被删除后会自动重新绘制所有可见区域。删除网格后，如果调用 `cell()`，将会返回一个空指针。

```
void Spreadsheet::selectCurrentRow()
{
    selectRow(currentRow());
}

void Spreadsheet::selectCurrentColumn()
{
    selectColumn(currentColumn());
}
```

以上两个函数分别相应菜单 `Edit|Select|Row` 和 `Edit|Select|Column`。通过调用 `QTableWidgetItem::selectRow()` 和 `QTableWidgetItem::selectColumn()`。 `Edit|Select|All` 菜单操作由 `QTableWidgetItem` 的父 `QTableWidgetItemView::selectAll()` 实现的。

```

void Spreadsheet::findNext(const QString &str, Qt::CaseSensitivity cs)
{
    int row = currentRow();
    int column = currentColumn() + 1;
    while (row < RowCount) {
        while (column < ColumnCount) {
            if (text(row, column).contains(str, cs)) {
                clearSelection();
                setCurrentCell(row, column);
                activateWindow();
                return;
            }
            ++column;
        }
        column = 0;
        ++row;
    }
    QApplication::beep();
}

```

函数 **findNext()** 从当前网格开始向右查找，查找完当前行后向下一行在继续查找，直到发现匹配的文本为止。如果发现了一个匹配，清除当前选择，把匹配的网格做为当前网格，并把相应的窗口激活。如果没有发现则程序 **beep**，说明查找完成但没有成功找到匹配的网格。

```

void Spreadsheet::findPrevious(const QString &str,
                               Qt::CaseSensitivity cs)
{
    int row = currentRow();
    int column = currentColumn() - 1;
    while (row >= 0) {
        while (column >= 0) {

```

```

        if (text(row, column).contains(str, cs)) {
            clearSelection();
            setCurrentCell(row, column);
            activateWindow();
            return;
        }
        --column;
    }
    column = ColumnCount - 1;
    --row;
}
QApplication::beep();
}

```

函数 `findPrevious()` 和 `findNext()` 很相似，只是搜索顺序是向前向上的，在 **A1** 停止。

4-5 实现其他菜单项（**Implementing the Other Menus**）

在这一节我们将会实现 **Tools** 和 **Options** 菜单的相应函数。

Figure 4.7. The Spreadsheet application's Tools and Options menus



```

void Spreadsheet::recalculate()

{
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            if (cell(row, column))
                cell(row, column)->setDirty();
        }
    }
}

```

```

    }
    viewport()->update();
}

```

槽函数 `recalculate()` 相应 `Tools|Recalculate` 菜单，又是 `Spreadsheet` 也会自动调用这个函数。遍历所有的行和列，在每一个网格上调用 `setDirty()` 给他们设置重新计算状态，然后 `QTableWidget` 调用每一个网格项的 `text()` 函数重新在表格中更新网格显示值，这个值是重新计算过的。

调用视图的 `update()` 重新绘制网格。在 `update()` 函数中，`QTableWidget` 会调用每一个可见网格的 `text()` 函数得到需要显示的值，因为前面调用过 `setDirty()`，所以显示值会重新计算。计算时也可能需要其他不可见网格项的值，这些不可见网格的值也会重新计算。这个计算是由 `Cell` 类实现的。

```

void Spreadsheet::setAutoRecalculate(bool recalc)
{
    autoRecalc = recalc;
    if (autoRecalc)
        recalculate();
}

```

上面这个函数相应了菜单 `Options|Auto-Recalculate`。如果它设置为开，那么立即重新计算全部表格更新。在程序运行的其他时间，如果 `somethingChanged()`, `recalculate()` 也会重新调用。

我们需要为菜单 `Options|Show Grid` 写任何代码，`QTableWidget::setShowGrid()` 已经为我们实现了。需要实现的是 `Spreadsheet::sort()`，由 `MainWindow::sort()` 调用。

```

void Spreadsheet::sort(const SpreadsheetCompare &compare)
{
    QList<QStringList> rows;
    QTableWidgetSelectionRange range = selectedRange();
    int i;
    for (i = 0; i < range.rowCount(); ++i) {
        QStringList row;
        for (int j = 0; j < range.columnCount(); ++j)

```

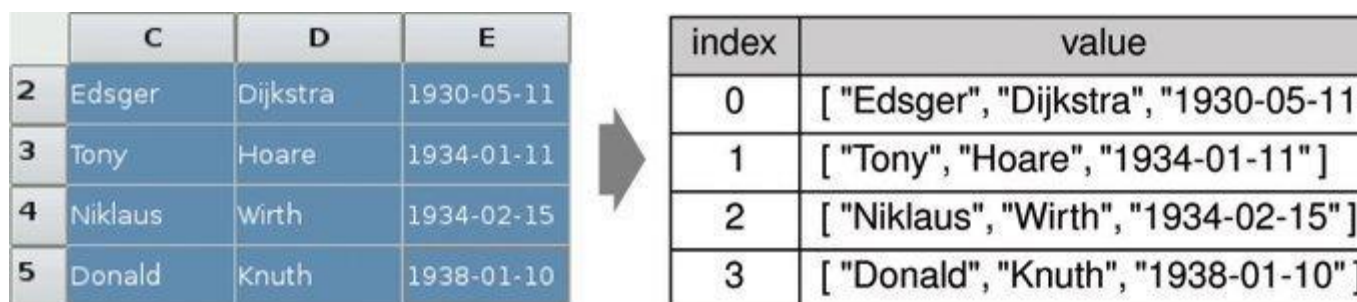
```

        row.append(formula(range.topRow() + i,
                           range.leftColumn() + j));
    rows.append(row);
}
qStableSort(rows.begin(), rows.end(), compare);
for (i = 0; i < range.rowCount(); ++i) {
    for (int j = 0; j < range.columnCount(); ++j)
        setFormula(range.topRow() + i, range.leftColumn() + j,
                    rows[i][j]);
}
clearSelection();
somethingChanged();
}

```

排序操作只对当前选择网格项有效。根据排序的关键字和排序顺序按行重新排列选择的网格项。我们把每一行的数据用 `QStringList` 的形式保存在列表中，为了简单起见，我们使用了 Qt 的 `qStableSort()` 算法，并且只对公式排序而不是对值排序。Qt 的标准算法和数据结构在第十一章介绍。


Figure 4.8. Storing the selection as a list of rows



函数 `qStableSort()` 接受一个起始遍历器和一个结束遍历器和一个比较函数。这个比较函数有两个参数（两个字符串），如果第一个参数小于第二个，返回 `true`，否则返回 `false`。我们传递的比较对象不是一个函数，但是它也是可用的，很快我们会看到这个实现。

Figure 4.9. Putting the data back into the table after sorting

index	value		C	D	E
0	["Donald", "Knuth", "1938-01-10"]				
1	["Edsger", "Dijkstra", "1930-05-11"]				
2	["Niklaus", "Wirth", "1934-02-15"]				
3	["Tony", "Hoare", "1934-01-11"]				



	C	D	E
2	Donald	Knuth	1938-01-10
3	Edsger	Dijkstra	1930-05-11
4	Niklaus	Wirth	1934-02-15
5	Tony	Hoare	1934-01-11

用 `qStableSort()` 排序后，把数据再写回到表格中，清空选择集，然后调用 `somethingChanged()`。

在 `spreadsheet.h` 中，我们这样实现 `SpreadsheetCompare` 类：

```
class SpreadsheetCompare
{
public:
    bool operator()(const QStringList &row1,
                    const QStringList &row2) const;

    enum { KeyCount = 3 };
    int keys[KeyCount];
    bool ascending[KeyCount];
};
```

类 `SpreadsheetCompare` 的特殊地方是它实现了一个 `()` 操作符。这允许我们把类作为一个函数使用，这样的类被称为函数类（`functors`）。为了理解这个函数类的工作情况，首先举一个简单的例子：

```
class Square
{
public:
    int operator()(int x) const { return x * x; }
}
```

`Square` 类只提供一个函数，`operator()(int)`。返回参数 `int` 的平方。把这个函数命名为 `operator()(int)` 而不是其他如 `compute(int)` 这样子，我们可以得到一个特殊的用途，可以把这个类的对象象函数一样使用。


```
Square square;

int y = square(5);
```

现在我们看一个使用 SpreadsheetCompare 的例子：

```
QStringList row1, row2;

QSpreadsheetCompare compare;

...

if (compare(row1, row2)) {
    // row1 is less than row2
}
```

这个比较对象就想一个 `compare()` 函数一样，但是我们还可以得到所有的排序关键字和排序顺序，它们做为数据成员保存在 `compare` 对象中。

另一个实现的方式就是在全局变量中保存排序键和排序顺序信息，直接使用比较函数 `compare()`。但是使用全局变量进行信息的交换在程序设计中是不提倡的，这有可能导致一些问题。函数类在使用模板函数 `qStableSort()` 交换信息时是一个更有效的方式。

下面是这个操作符函数的实现：

```
bool SpreadsheetCompare::operator()(const QStringList &row1,
                                    const QStringList &row2) const
{
    for (int i = 0; i < KeyCount; ++i) {
        int column = keys[i];
        if (column != -1) {
            if (row1[column] != row2[column]) {
                if (ascending[i]) {
                    return row1[column] < row2[column];
                } else {
                    return row1[column] > row2[column];
                }
            }
        }
    }
}
```

```

return false;
}

```

如果第一个行小于第二行，函数返回 **TRUE**，否则返回 **FALSE**。**qStableSort** 就使用这个结果执行排序操作。

SpreadsheetCompare 对象中的键值和排序队列是在 **MainWindow::sort()** 中初始化的。每一个键都包含一个列索引，或者为 **-1**（空值）。

我们按照顺序比较每一行中的相应的网格项。只要发现不同，就返回 **true** 或者 **false**。如果两行都相等，也返回 **false**。**qStableSort()** 使用排序前的顺序解决这个问题。如果排序前的顺序是 **row1** 和 **row2**，且经比较相等，在结果中 **row1** 始终就排在 **row2** 前面。这就是 **qStableSort()** 和 **qSort()** 之间的不同。

我们已经实现了类 **Spreadsheet**。在下一节中我们实现 **Cell** 类。这个类用来存贮网格项的公式，重新实现了 **QTableWidgetItem::data()** 函数，**Spreadsheet** 间接调用了这个函数，在 **QTableWidgetItem::text()**，这个函数根据网格项的公式计算出显示文本。

4-6 继承类 **QTableWidgetItem**（**Subclassing QTableWidgetItem**）

类 **Cell** 继承自 **QTableWidgetItem**。这个类不但可以在 **Spreadsheet** 程序中工作良好，但是并不仅限于这个类，在理论上，它可以被用在任何 **QTableWidgetItem** 子类中。下面是头文件：

```

#ifndef CELL_H
#define CELL_H
#include <QTableWidgetItem>
class Cell : public QTableWidgetItem
{
public:
    Cell();
    QTableWidgetItem *clone() const;
    void setData(int role, const QVariant &value);
    QVariant data(int role) const;
    void setFormula(const QString &formula);
    QString formula() const;

```

```

    void setDirty();
private:
    QVariant value() const;
    QVariant evalExpression(const QString &str, int &pos) const;
    QVariant evalTerm(const QString &str, int &pos) const;
    QVariant evalFactor(const QString &str, int &pos) const;
    mutable QVariant cachedValue;
    mutable bool cacheIsDirty;
};
#endif

```

类 `Cell` 在 `QTableWidgetItem` 基础上增加了两个私有变量：

cachedValue: 以 `QVariant` 的形式保存网格项的值，这个值可能是 `double` 型，也可能是 `QString` 类型。

cacheIsDirty: 如果保存的值需要更新则置为 `true`。

变量 `cachedValue` 和 `cacheIsDirty` 前声明了 C++ 的关键字 `mutable`。这可以允许我们在常函数中修改这个变量大值。我们也可以在每次调用 `text()` 时计算变量的值，但是这样毫无疑问效率会很差。

注意，类定义里面没有声明 `Q_OBJECT` 宏。`Cell` 是一个纯粹的 C++ 类，没有信号和槽。事实上，`QTableWidgetItem` 也是一个纯粹 C++ 类，而不是从 `QObject` 继承来的。为了保证最小的代价和高效，Qt 所有 `item` 类都不是从 `QObject` 继承的。如果需要信号和槽，可以在使用它们的控件中定义，或者使用多继承。

下面是源文件：

```

#include <QtGui>
#include "cell.h"
Cell::Cell()
{
    setDirty();
}

```

```

QTableWidgetItem *Cell::clone() const
{
    return new Cell(*this);
}

void Cell::setFormula(const QString &formula)
{
    setData(Qt::EditRole, formula);
}

QString Cell::formula() const
{
    return data(Qt::EditRole).toString();
}

void Cell::setData(int role, const QVariant &value)
{
    QTableWidgetItem::setData(role, value);
    if (role == Qt::EditRole)
        setDirty();
}

void Cell::setDirty()
{
    cacheIsDirty = true;
}

QVariant Cell::data(int role) const
{
    if (role == Qt::DisplayRole) {
        if (value().isValid()) {
            return value().toString();
        } else {
            return "####";
        }
    }
}

```

```

    } else if (role == Qt::TextAlignmentRole) {
        if (value().type() == QVariant::String) {
            return int(Qt::AlignLeft | Qt::AlignVCenter);
        } else {
            return int(Qt::AlignRight | Qt::AlignVCenter);
        }
    } else {
        return QTableWidgetItem::data(role);
    }
}

const QVariant Invalid;
QVariant Cell::value() const
{
    if (cacheIsDirty) {
        cacheIsDirty = false;
        QString formulaStr = formula();
        if (formulaStr.startsWith("(")) {
            cachedValue = formulaStr.mid(1);
        } else if (formulaStr.startsWith('=')) {
            cachedValue = Invalid;
            QString expr = formulaStr.mid(1);
            expr.replace(" ", "");
            expr.append(QChar::Null);
            int pos = 0;
            cachedValue = evalExpression(expr, pos);
            if (expr[pos] != QChar::Null)
                cachedValue = Invalid;
        } else {
            bool ok;
            double d = formulaStr.toDouble(&ok);

```

```

        if (ok) {
            cachedValue = d;
        } else {
            cachedValue = formulaStr;
        }
    }
}

return cachedValue;
}

QVariant Cell::evalExpression(const QString &str, int &pos) const
{
    QVariant result = evalTerm(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '+' && op != '-')
            return result;

        ++pos;

        QVariant term = evalTerm(str, pos);
        if (result.type() == QVariant::Double
            && term.type() == QVariant::Double) {
            if (op == '+') {
                result = result.toDouble() + term.toDouble();
            } else {
                result = result.toDouble() - term.toDouble();
            }
        } else {
            result = Invalid;
        }
    }

    return result;
}

```

```

}
QVariant Cell::evalTerm(const QString &str, int &pos) const
{
    QVariant result = evalFactor(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '*' && op != '/')
            return result;
        ++pos;
        QVariant factor = evalFactor(str, pos);
        if (result.type() == QVariant::Double
            && factor.type() == QVariant::Double) {
            if (op == '*') {
                result = result.toDouble() * factor.toDouble();
            } else {
                if (factor.toDouble() == 0.0) {
                    result = Invalid;
                } else {
                    result = result.toDouble() / factor.toDouble();
                }
            }
        } else {
            result = Invalid;
        }
    }
    return result;
}

QVariant Cell::evalFactor(const QString &str, int &pos) const
{
    QVariant result;

```

```

bool negative = false;
if (str[pos] == '-') {
    negative = true;
    ++pos;
}
if (str[pos] == '(') {
    ++pos;
    result = evalExpression(str, pos);
    if (str[pos] != ')')
        result = Invalid;
    ++pos;
} else {
    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    QString token;
    while (str[pos].isLetterOrNumber() || str[pos] == '.') {
        token += str[pos];
        ++pos;
    }
    if (regExp.exactMatch(token)) {
        int column = token[0].toUpper().unicode() - 'A';
        int row = token.mid(1).toInt() - 1;
        Cell *c = static_cast<Cell *>(
            tableWidget()->item(row, column));
        if (c) {
            result = c->value();
        } else {
            result = 0.0;
        }
    } else {
        bool ok;

```



```

        result = token.toDouble(&ok);
        if (!ok)
            result = Invalid;
    }
}
if (negative) {
    if (result.type() == QVariant::Double) {
        result = -result.toDouble();
    } else {
        result = Invalid;
    }
}
return result;
}

```

在构造函数中，我们只是把存储器设为“脏”。这里不需要传递一个父参数。因为用 `QTableWidget::setItem()` 插入 `Cell` 对象时，`QTableWidget` 自动得到它的所有权。在 `QTableWidgetItem` 中，每一个 `QVariant` 都以一种“角色”保存一类数据。最常用的角色是 `Qt::EditRole` 和 `Qt::DisplayRole`。`Qt::EditRole` 表示用来编辑网格中的数据，`Qt::DisplayRole` 只是显示数据。通常这两个角色中的数据都是一样的。但是在 `Cell` 中，`Qt::EditRole` 表示要编辑的公式，`Qt::DisplayRole` 表示网格要显示的值（公式计算的结果）。

当需要一个新的网格时，`QTableWidget` 调用函数 `clone()`，例如，当用户在一个空的网格中输入数据。传递给 `QTableWidget::setItemPrototype()` 就是由 `clone()` 得到的项目。我们使用了 C++ 自动创建的 `Cell` 的默认拷贝构造函数，这对于成员级别的拷贝已经足够了。

函数 `setFormula()` 设置网格的公式。它为调用 `Qt::EditRole` 的 `setData()` 函数提供了方便。在 `Spreadsheet::setFormula()` 函数中调用了 `setFormula()` 函数。

在 `Spreadsheet::formula()` 中调用了函数 `Cell::formula()`。和 `setFormula()` 一样，它也是一个方便函数，得到网格项的 `Qt::EditRole` 数据。

修改网格的数据 `setData()` 时，如果输了一个新的公式，那么将 `cacheIsDirty` 设置为 `true`，以便在下一此调用 `text()` 时重新计算显示值。

尽管在 `Spreadsheet::text()` 中用了 `Cell`，但在 `Cell` 中没有定义 `text()`。`text()` 函数是 `QTableWidgetItem` 提供的一个方便函数，等价于 `data(Qt::DisplayRole).toString()`。

`setDirty()` 用来强制计算网格的值，它只是将 `cacheIsDirty` 为 `true`，说明 `cachedValue` 中的值需要更新，需要时要重新计算。

函数 `data()` 重新进行了实现。如果用 `Qt::DisplayRole` 调用，返回显示的文本。如果用 `Qt::EditRole` 调用则返回公式。如果用 `Qt::TextAlignmentRole` 调用，返回给你合适的对其方式。在 `Qt::isplayRole` 方式中，调用 `value()` 得到计算的网格值。如果值无效，则显示字符串 `####`。

在 `data()` 中使用的 `Cell::value()` 函数返回一个 `QVariant` 类型值。一个 `QVariant` 类型可以保存多种类型的数据，并且提供不同数据类型之间的转换。例如，在一个保存 `double` 型的变量中调运女冠 `toString()` 则得到 `double` 的字符串表示。`QVariant` 用一个 `"invalid"` 数据进行默认的初始化。

函数 `value()` 返回网格的显示值。如果 `cacheIsDirty` 为 `true`，则需要重新计算。如果公式用单引号开头，如 `"'12344"`，网格值为单引号后面的文本。如果公式由等号 `"="` 开头，得到等号后面的字符串并删除其中所有的空格然后调用 `evalExpression()` 计算表达式的值。参数 `pos` 是传引用。它表示表达式开始分解的字符串位置。调用 `evalExpression()` 后，如果表达式解析成功，`pos` 的值应为我们附加的 `QChar::Null`，否则失败，置 `cachedValue` 为 `Invalid`。如果公式不是由单引号或者等号开头，首先试着把公式转换为浮点数，如果转换成功，返回值就是得到的浮点数。否则直接返回公式。

给 `toDouble()` 一个 `bool` 型的指针。我们能够区分返回值为 `0.0` 时是成功与否。如果转换失败，返回值为 `0.0`，但是 `bool` 值为 `false`。在我们不需要考虑转换成功与否的时候，

返回 0.0 值还是有必要的。为了性能和可移植性，Qt 不使用 C++ 表达式报告失败的情况。但这不影响你在 Qt 中使用它们，只要你的编译器支持就可以。

我们声明 `value()` 为常函数，为了编译器允许在 `value()` 中改变 `cachedValue` 和 `cacheIsValid` 的值，我们不得不把这两个变量声明为 `mutable`。如果把 `value()` 改为非常函数，那么 `mutable` 关键字就可以去掉，但是因为我们在 `data()` 常函数中调用的 `value()`，编译不会成功。

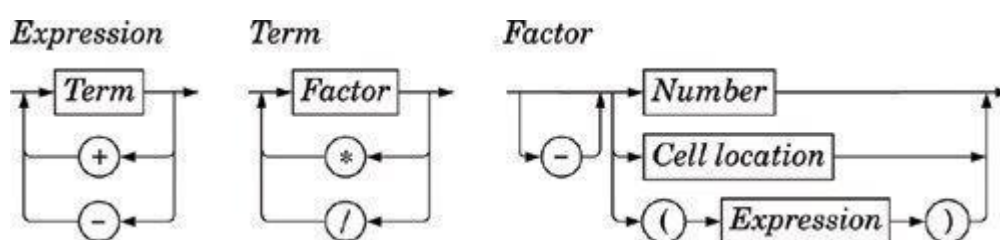
到目前为止我们已经完成了大部分 Spreadsheet 程序，还有一部分就是公式的解析。下面的部分介绍 `evalExpression()` 和两个辅助函数 `evalTerm()` 和 `evalFactor()`。代码有些复杂，为了程序的完整性把它们包含了进来。这些代码和 GUI 编程没有任何关系，因此你也可以跳过直接阅读第五章。

函数 `evalExpression()` 返回表格表达式的值。一个表达式由一个或者多个项组成，这些项之间由“+”或者“+-”符号隔开。每一算式项由一个或者多个因数“*”或者“/”隔开。把表达式分解为加减法项，把加减法解析为乘除法项，这样我们就能确保表达式按照正确的优先级进行计算

例如表达式“2*C5+D6”，“2*C5”是第一项，其中 2 是第一个因数，C5 是第二个因数。“D6”是第二项，只有一个因数。一个因数可以是一个数字，也可以是一个网格的位置，有时候前面还有一个一元减号。

图 4.10 定义了表格表达式的语法结构。对于语义中的每一个符号（表达式，算式项，因数项），都有一个相应的成员函数进行解析，算式结构严格遵照语法。这种解析方式被称为递归解析。

图 4.10 syntax diagram for spreadsheet expressions



先从 `evalExpression()` 开始。这个函数解析一个表达式。

首先，调用 `evalTerm()` 得到第一个算式的值。如果接下来的符号为“+”或者“+-”那么继续调用 `evalTerm()`。否则这个表达式就是由一个算式组成的，它的值就是表达式的值。当得到前两个算式的值后，我们根据操作符计算这两个算式的结果。如果两个算式都是 `double` 型的，结果也为 `double` 型的。否则结果为 `Invalid`。继续计算直到表达式中没有算式为止。因为加法和减法都是左结合的，所以计算的结果是正确的。

函数 `evalTerm()` 和 `evalExpression()` 很像，只是它处理的是乘除法。还有一个不同地方就是必须避免除数为 0 的情况，在很多处理器上都是错误的算式。由于四舍五入的误差，一般不用判断浮点数的值，测试是否等于 0.0 就可以了。

`evalFactor()` 有些复杂。首先我们判断因数前面是否有负号，然后判断是否有左括号，如果发现括号，那么就把括号内的内容作为一个表达式，调用 `evalExpression()`。在解析括号内的表达式时，`evalExpression()` 调用 `evalTerm()`，再调用 `evalFactor()`。这就是解析的递归部分。

如果因数不是一个内嵌的表达式，我们就得到它的下一个语法符号，它可能是一个网格的位置或者是一个数字。如果符号匹配 `QRegExp`，则意味它是一个网格位置，得到这个网格的值。这个网格应该在表格的某一个地方，它的值如果依赖其他的网格，会触发更多的对 `value()` 的调用，对所有依赖的网格都解析。如果因数不是网格，那么把它看作一个数字。如果 A1 的公式为“=A1”，或者 A1 的公式为“=A2”，A2 的公式为“=A1”时该怎么办那？虽然我们没有代码检测这些圆形依赖关系，解析器也可以返回一个无效的 `QVariant` 值，因为在调用 `evalExpression()` 之前，我们以把 `cacheIsDirty` 置为 `false`，`cachedValue` 为 `Invalid`。如果 `evalExpression()` 不停的调用某一个网格的 `value()`，它会返回 `Invalid`，整个表达式的值为 `Invalid`。

我们就这样完成了公式的解析。也可以增加对因数的类型的定义，直接对它进行扩展处理表格预定义的函数，如 `sum()`，`avg()`，另一个简单的扩展也可以把“+”好用字符串式的连接实现，这不需要更改代码。

第五章用户自定义控件（Creating Custom Widgets）及第四章小结

第四章实现的是应用程序的功能。我们实现了一个简单的表格编辑器，并且和 GUI 一起完成了一个具有简单功能的应用程序。应该和第三章结合在一起看。第二，三，四章的程序夹在一起是一个运行良好的程序。

第五章是独立的一章，创建用户自定义控件。用户自定义的控件可以通过继承现有的 Qt 控件实现，也可以直接从 QWidget 继承。这两种方法我们都进行介绍。介绍自定义控件如何放到 Qt Designer 的控件列表中象 Qt 自己的控件一样使用。最后介绍一个使用双缓冲的自定义控件，双缓冲是实现高速绘制图形的一种技术。

5-1 自定义 Qt 控件（Customizing Qt Widgets）

在某些情况下，我们发现有些 Qt 控件通过设置它的属性或者函数不能满足我们的要求，还需要更多的要求。一个简单且直接的解决方法就是从这些 Qt 继承然后让它们满足我们的需要。

图 5—1 the HexSpinBox widget



在本节中，我们开发一个十六进制的旋转盒来说明怎样来自定义 Qt 的控件。

QSpinBox 只支持十进制整数，但是继承它是新类能够接受和显示十六进制数值是非常简单的。

```
#ifndef HEXSPINBOX_H
#define HEXSPINBOX_H
#include <QSpinBox>
class QRegExpValidator;
class HexSpinBox : public QSpinBox
{
    Q_OBJECT
public:
```

```

    HexSpinBox(QWidget *parent = 0);
protected:
    QValidator::State validate(QString &text, int &pos) const;
    int valueFromText(const QString &text) const;
    QString textFromValue(int value) const;
private:
    QRegExpValidator *validator;
};
#endif

```

类 HexSpinBox 继承了很多 QSpinBox 的功能。它提供了一个典型的构造函数，重写了 QSpinBox 的三个虚函数。

```

#include <QtGui>
#include "hexspinbox.h"
HexSpinBox::HexSpinBox(QWidget *parent)
    : QSpinBox(parent)
{
    setRange(0, 255);
    validator = new QRegExpValidator(QRegExp("[0-9A-Fa-f]{1,8}"), this);
}
QValidator::State HexSpinBox::validate(QString &text, int &pos) const
{
    return validator->validate(text, pos);
}
QString HexSpinBox::textFromValue(int value) const
{
    return QString::number(value, 16).toUpper();
}
QString HexSpinBox::textFromValue(int value) const
{

```

```

        return QString::number(value, 16).toUpper();
    }
    int HexSpinBox::valueFromText(const QString &text) const
    {
        bool ok;
        return text.toInt(&ok, 16);
    }

```

我们设置默认的数值范围是 0 到 255（0X00 到 0XFF），在 QSpinBox 中默认的范围是 0 到 99，在十六进制中，前者合理多了。

用户可以点击上下箭头修改旋转盒的当前值，也可在编辑框里直接输入一个值。如果是字母，我们限制用户的只能输入合理的十六进制数字。为了做到这一点，我们使用一个 QRegExpValidator，它只允许输入数字 0 到 9，A 到 F，和小写字母 a 到 f。QSpinBox 调用函数 validate() 确定是否输入的文本是合法的。它会产生三个可能的结果：Invalid（不合法），Intermediate（输入的文本是一个合理值的合理部分），Acceptable（文本是合理的）。QRegExpValidator 有一个合适的 validate() 函数，所以我们就返回调用他的结果。在理论上，如果超过了范围，我们需要返回 Invalid 或者 Intermediate，但是 QSpinBox 能够帮助我们做这些。

函数 textFromValue() 把一个整数变换为一个字符串。当用户点击上下箭头时，QSpinBox 调用这个函数更新旋转盒的编辑部分。16 作为基数，QString::number() 把数值转换为小写的十六进制，QString::toUpper() 将得到的结果转换为大写。函数 valueFromText() 实现了逆转换，把字符串转换为整数。当用户在旋转盒的编辑框中输入一个数值时由 QSpinBox 调用。使用 16 作为基数，把当前的文本转换为整数值，如果文本不能转换为十六进制数值，ok 置为 false，toInt() 返回 0 值。这里我们不需要考虑这个可能性，因为 validator 只允许输入合法的十六进制字符。我们也可以不传递 ok 的地址，使用一个空指针也可以。

自定义其他 Qt 控件也遵循一样的步骤，选择一个合理的 Qt 控件，把它作为基类，然后重新实现一些虚函数改变它的行为以满足我们的需要。

5-2 从 QWidget 继承新类（Subclassing QWidget）

许多 Qt 的控件或者象 `HexSpinBox` 这些自定义控件都是现有的控件的一个组合。由 Qt 控件组合而成的用户控件可以用 Qt Designer 实现：

1. 用模板“Widget”新建一个控件框架
2. 在框架中加入需要的控件，并对控件进行布局
3. 进行信号和槽连接
4. 如果还需要更多的信号和槽，可以在继承 `QWidget` 和 `uic` 生成的类中添加相关代码

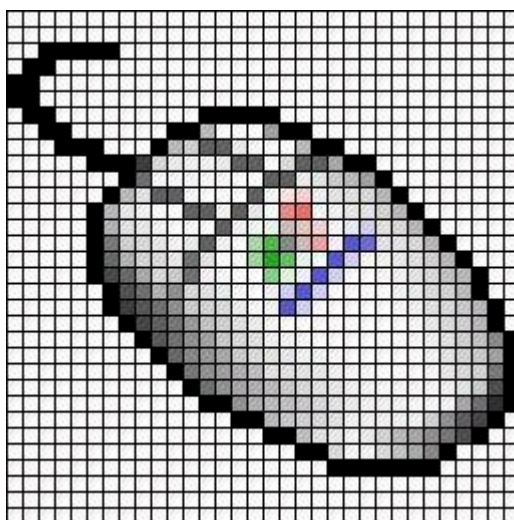
这种控件的组合也可以用代码实现。不管那种方法，结果都是创建一个直接从 `QWidget` 继承的类。

如果控件不需要信号和槽，也不用实现任何虚函数，也可以不用继承，只要把现有的控件组合起来就可以。在第一章的 `Age` 程序 中就是这样。当然我们也可以写一个 `QWidget` 的基类，在新类的构造函数中创建 `QSpinBox` 和 `QSlider`。

如果所需要的用户控件，找不到合适的 Qt 控件可用，也没有办法通过组合或者调整现有的控件，我们也可以自己创建出来。我们可以从 `QWidget` 继承一个新类，重新实现一些事件处理函数实现绘制新的控件，相应鼠标的点击，我们可以完全控制控件的外观和行为。Qt 的许多控件如 `QLabel`，`QPushButton` 和 `QTableWidget` 就是这样实现的。如果他们不存在，也可以使用 `QWidget` 提供的函数创建出来，并保持平台的无关性。

我们创建一个 `IconEditor` 控件来说明这个方法。`IconEditor` 控件如下图所示，这个控件可以在图标编辑程序中使用。

Figure 5-2 the IconEditor Widget



下面是头文件：

```
#ifndef ICONEDITOR_H
#define ICONEDITOR_H

#include <QColor>
#include <QImage>
#include <QWidget>

class IconEditor : public QWidget
{
    Q_OBJECT

    Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
    Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
    Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)

public:
    IconEditor(QWidget *parent = 0);
    void setPenColor(const QColor &newColor);
    QColor penColor() const { return curColor; }

    void setZoomFactor(int newZoom);
    int zoomFactor() const { return zoom; }
    void setIconImage(const QImage &newImage);
    QImage iconImage() const { return image; }
    QSize sizeHint() const;

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

private:
    void setImagePixel(const QPoint &pos, bool opaque);
```

```

    QRect pixelRect(int i, int j) const;

    QColor curColor;

    QImage image;

    int zoom;
};

#endif

```

类 `IconEditor` 使用宏 `Q_PROPERTY()` 定义了三个自定义属性: `penColor`, `iconImage`, `zoomFactor`。每一个属性都有一个数据类型, 一个读函数和一个写函数。例如, 属性 `penColor` 类型为 `QColor`, 读写函数分别为 `penColor()` 和 `setPenColor()`。如果在 `Qt Designer` 中使用这个控件, 自定义属性就会出现在 `Qt Designed` 的属性编辑窗口中。属性的数据类型可以是 `QVariant` 支持的各种类型。要使这些属性有效, `Q_OBJECT` 宏是必须的。

`IconEditor` 实现了 `QWidget` 的三个保护成员函数。此外还声明了三个变量保存这些属性的值。

源文件如下:

```

#include <QtGui>
#include "iconeditor.h"

IconEditor::IconEditor(QWidget *parent)
    : QWidget(parent)
{
    setAttribute(Qt::WA_StaticContents);
    setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);
    curColor = Qt::black;
    zoom = 8;
    image = QImage(16, 16, QImage::Format_ARGB32);
    image.fill(qRgba(0, 0, 0, 0));
}

QSize IconEditor::sizeHint() const
{

```

```

    QSize size = zoom * image.size();
    if (zoom >= 3)
        size += QSize(1, 1);
    return size;
}

void IconEditor::setPenColor(const QColor &newColor)
{
    curColor = newColor;
}

void IconEditor::setIconImage(const QImage &newImage)
{
    if (newImage != image) {
        image = newImage.convertToFormat(QImage::Format_ARGB32);
        update();
        updateGeometry();
    }
}

void IconEditor::setZoomFactor(int newZoom)
{
    if (newZoom < 1)
        newZoom = 1;
    if (newZoom != zoom) {
        zoom = newZoom;
        update();
        updateGeometry();
    }
}

void IconEditor::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

```

```

if (zoom >= 3) {
    painter.setPen(palette().foreground().color());
    for (int i = 0; i <= image.width(); ++i)
        painter.drawLine(zoom * i, 0,
                           zoom * i, zoom * image.height());
    for (int j = 0; j <= image.height(); ++j)
        painter.drawLine(0, zoom * j,
                           zoom * image.width(), zoom * j);
}

for (int i = 0; i < image.width(); ++i) {
    for (int j = 0; j < image.height(); ++j) {
        QRect rect = pixelRect(i, j);
        if (!event->region().intersect(rect).isEmpty()) {
            QColor color = QColor::fromRgba(image.pixel(i, j));
            painter.fillRect(rect, color);
        }
    }
}

}

QRect IconEditor::pixelRect(int i, int j) const
{
    if (zoom >= 3) {
        return QRect(zoom * i + 1, zoom * j + 1, zoom - 1, zoom - 1);
    } else {
        return QRect(zoom * i, zoom * j, zoom, zoom);
    }
}

void IconEditor::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {

```

```

        setImagePixel(event->pos(), true);
    } else if (event->button() == Qt::RightButton) {
        setImagePixel(event->pos(), false);
    }
}

void IconEditor::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        setImagePixel(event->pos(), true);
    } else if (event->buttons() & Qt::RightButton) {
        setImagePixel(event->pos(), false);
    }
}

void IconEditor::setImagePixel(const QPoint &pos, bool opaque)
{
    int i = pos.x() / zoom;
    int j = pos.y() / zoom;
    if (image.rect().contains(i, j)) {
        if (opaque) {
            image.setPixel(i, j, penColor().rgba());
        } else {
            image.setPixel(i, j, qRgba(0, 0, 0, 0));
        }
        update(pixelRect(i, j));
    }
}

```

在构造函数中，属性 `Qt::WA_StaticContents` 和 `setSizePolicy` 的调用稍后再介绍。画笔的颜色为黑色，放大倍数为 8，意思是图标中的每一个像素占用了 8×8 个小格子的空间。

图标数据保存在 `image` 成员变量中可以用函数 `iconImage()` 和 `setIconImage()` 读取。图标编辑程序可以调用 `setIconImage()` 打开图标文件，调用 `iconImage()` 得到图标把它保存到磁盘上。变量 `image` 的类型为 `QImage`，初始化为透明的图片， 16×16 个像素，32 位的 ARGB 格式，这个图片格式支持半透明显示。

`QImage` 中的图片是平台无关的。它可以显示 1 位，8 位或者 32 位像素的图片。一个 32 位的图片用 8 个位显示红，绿，蓝三个分量。剩下的 8 位是图片 `alpha` 通道值，表示透明度。例如一个纯红色的红，绿，蓝和 `alpha` 四个分量分别为 255，0，0，和 255。在 Qt 中，这个颜色可以这样表示：`QRgb red = qRgba(255, 0, 0, 255)`，由于图片不是透明的，也可以如下简单表示 `QRgb red = qRgb(255, 0, 0)`。

`QRgb` 实际上是一个 `unsigned int` 类型，内联函数 `qRgb()`，`qRgba()` 只是把分量值合成一个 32 为整数。`QRgb red = 0xffff0000`。第一个 `ff` 为 `alpha` 分量值，第二个 `ff` 为红色的分量值。在 `IconEditor` 中我们设置 `alpha` 分量为 0，得到一个透明的图片。

Qt 提供了两种颜色有关的类：`QRgb` 和 `QColor`。在 `QImage` 中使用的 `QRgb` 只是一个 32 位的像素数据。`QColor` 是一个有很多功能的类，在 Qt 中使用的很多。在这个控件中，我们只是在处理 `QImage` 的时候使用了 `QRgb`，其他地方都是用的 `QColor`，`penColor` 属性也是使用的 `QColor` 类型。

函数 `IconEditor::sizeHint()` 是 `QWidget` 的虚函数，返回控件的最适当的大小。这里进行了重写，图片的大小乘以放大倍数，如果放大倍数大于 3，则在四个方向上再加上一个像素，用来显示网格。如果放大倍数小于 3，根本没有位置显示网格，所以也就没有必要加一个像素。

控件的 `sizeHint` 在布局中非常有用。布局管理器根据控件的 `sizeHint` 排列子控件。`IconEditor` 控件为了能在布局时有一个好的位置，就必须提供提供一个 `sizeHint`。

除了控件的 `sizeHint`，控件还有一个 `sizePolicy` 属性，布局管理器根据这个属性拉伸或者缩小空间尺寸。在 `IconEditor` 构造函数中，`setSizePolicy()` 的参数位 `QSizePolicy::Minimum`，布局管理器就会把控件的最小尺寸做为 `sizeHint`。即控件可以被拉伸，但是不能缩小到小于它的最小尺寸值。这个值可以在 Qt Designer 中的 `sizePolicy` 属性里修改。`sizePolicy` 的各种取值的含义在第六章中介绍。

函数 `setPenColor()` 设置当前画笔的颜色。新绘制的像素显示新的画笔颜色。

函数 `setIconImage()` 重新设置编辑的图片。调用 `convertToFormat()` 构成一个 32 位具有 alpha 值的图片数据。程序的其他地方都假设编辑的图片数据保存的是 32 位 ARGB 值。调用 `QWidget::update()` 强制控件显示新的图片，`QWidget::updateGeometry()` 通知布局管理器用新的 `sizeHint` 重新调整控件的大小。

函数 `setZoomFactor()` 设置图片的放大倍数。为了避免 0 位除数，不允许放大倍数小于 1。放大倍数改变后，也要调用 `update()` 和 `updateGeometry()` 重新显示图片，调整控件大小。

函数 `penColor()`，`iconImage()`，`zoomFactor()` 在头文件中做为内联函数实现。现在来看 `paintEvent()` 函数。这个函数是 `IconEditor` 最重要的一个函数，在控件需要重新绘制的时候调用。在类 `QWidget` 中这个函数不作任何事情，控件是一片空白，和第三章的 `closeEvent()` 一样，`paintEvent()` 函数也是一个事件处理函数。Qt 有很多事件处理函数，每一个函数相应一个类型的事件，第七章将会深入介绍事件处理。

Qt 中很多情况下都会产生绘制事件，调用 `paintEvent()` 函数：

1. 当控件第一次显示时，Qt 自动产生绘制事件使空间绘制自身。
2. 当控件尺寸发生变化时，系统产生绘制事件
3. 如果控件被其他的窗口遮住，窗口移走时，产生绘制被遮住部分的事件。

如果我们调用了 `QWidget::update()` 和 `QWidget::repaint()` 函数时，也会产生一个绘制事件。这两个函数也有所不同。`repaint()` 立刻产生绘制事件，重新绘制控件。而调用 `update()` 后，只是提交给 Qt 一个产生绘制事件的计划。如果控件在屏幕上不可见，那么这两个函数都是什么都不做。如果 `update()` 被调用了多次后，Qt 就把这几个连续的绘制事件合为一个事件避免闪烁。在 `IconEditor` 中，我们总是使用 `update()` 产生绘制事件。在代码中，首先创建一个控件的 `QPainter` 对象。如果放大倍数大于等于 3，调用 `QPainter::drawLine()` 函数绘制水平垂直线形成网格。

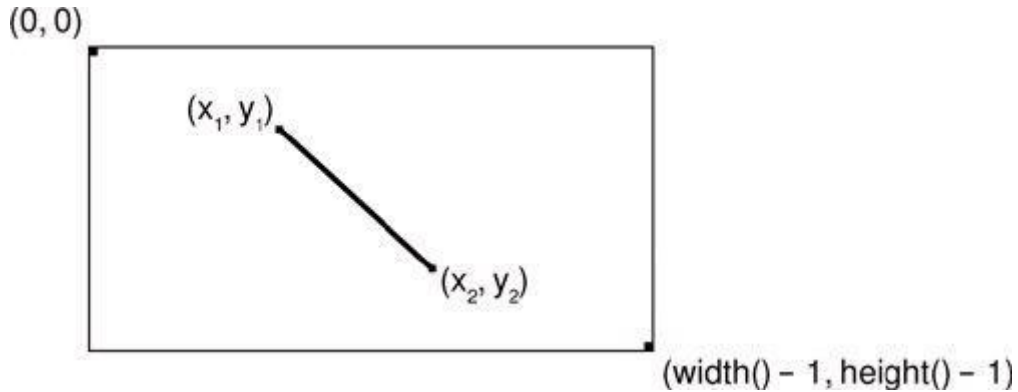
`QPainter::drawLine()` 语法如下：

```
painter.drawLine(x1, y1, x2, y2);
```

(`x1`, `y1`) 是线的一个端点，(`x2`, `y2`) 是另一个端点。函数还有一个重载形式，两个 `QPoints` 做为参数。

在 Qt 中，控件左上角的坐标为 $(0, 0)$ ，右下角的坐标为 $(\text{width}() - 1, \text{height}() - 1)$ 。这和常规笛卡儿坐标很像，只是方向向下。在第八章中，我们会介绍利用坐标变换改变 QPainter 的坐标系，如平移，放缩，旋转，剪切。

5-3 drawing a line with QPainter



在 `drawLine()` 之前，用 `setPen()` 设置线的颜色。我们可以用代码设置线的颜色，如黑色或者灰色，但是使用控件的调色板是一个更好的方法。

每一个控件都有一个调色板设置控件不同位置的颜色。例如，控件的背景（一般是亮灰色），文本的颜色（一般为黑色）。缺省情况下，一个控件的调色板的颜色和所使用系统的窗口颜色设置一样。使用调色板的颜色，IconEditor 的外观和用户的喜好一致。

一个控件的调色板由三个颜色组构成：激活的，未激活的和不可用的。使用那个颜色组由控件当前的状态决定：

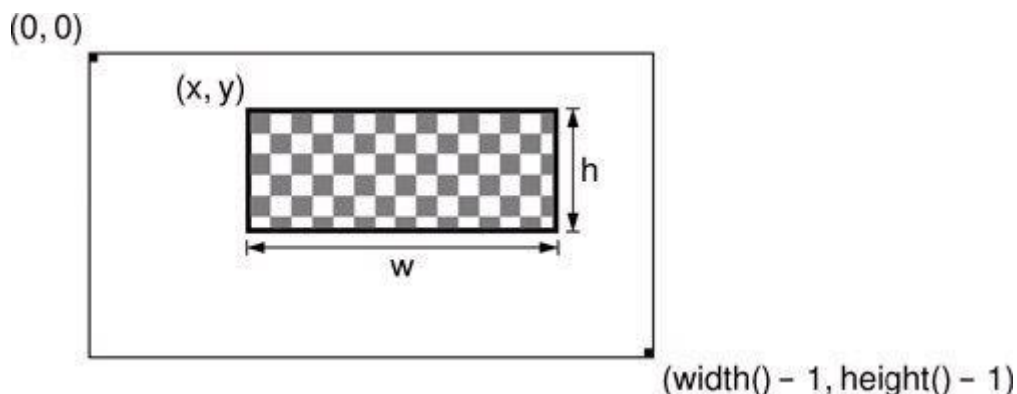
1. 控件所在窗口未当前激活的窗口，使用激活组的颜色；
2. 控件在非当前窗口的其他窗口，使用未激活组的颜色；
3. 控件在窗口中为不可用状态时，使用不可用组的颜色；

`QWidget::palette()` 函数返回当前控件的调色板对象。颜色组由枚举 `QPalette::ColorGroup` 决定。

在需要一个合适的刷子颜色画图时，正确的方法是使用当前 `QWidget::palette()` 返回的调色板和一个特定的角色（role），如 `QPalette::foreground()`。每一个角色都返回一个刷子，一般我们使用这个刷子就可以了，有时也需要使用刷子的颜色，如在 `paintEvent()` 函数就是这样。通过这种方法得到的刷子与控件的状态一致，一般不需要确定颜色组。

函数 `paintEvent()` 绘制了图像。`IconEditor::pixelRect()` 返回的 `QRect` 定义了需要重新绘制的区域。这样我们就不用重新绘制那些不在这个区域里的像素。

Figure5-4 Darwing a line with QPainter



`QPainter::fillRect()`绘制一个有放大倍数的像素。需要一个 `QRect` 和 `QBrush` 类型的参数。使 `QColor` 作为刷子，我们得到一个固体填充的模式。

```
QRect IconEditor::pixelRect(int i, int j) const
{
    if (zoom >= 3) {
        return QRect(zoom * i + 1, zoom * j + 1, zoom - 1, zoom - 1);
    } else {
        return QRect(zoom * i, zoom * j, zoom, zoom);
    }
}
```

函数 `pixelRect()`返回一个 `QRect`，传递给 `QPainter::fillRect()`。参数 `i` 和 `j` 是 `QImage` 中像素的坐标，而不是控件的坐标。只有放大倍数为 1 时，这两者的坐标系才是一致的。

`QRect` 的构造函数语法为 `QRect(x, y, width, height)`，`(x, y)` 是矩形左上角的坐标，`width` 和 `height` 是矩形的长和宽。如果放大倍数大于等于 3，为了不覆盖住网格线，我们少画一个像素。

```
void IconEditor::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        setImagePixel(event->pos(), true);
    } else if (event->button() == Qt::RightButton) {
```

```
setImagePixel(event->pos(), false);
}
}
```

当用户点击鼠标时，系统产生鼠标点击事件。重载 `QWidget::mousePressEvent()`，我们可以按照我们的意愿回应这个事件，在鼠标位置设置或者清除图像像素。

如果用户点击鼠标左键，调用私有函数 `setImagePixel(, true)` 设置当前像素为当前画笔的颜色。如果用户点击鼠标右键，也调用 `setImagePixel(, false)` 清除当前位置的像素。

```
void IconEditor::mouseMoveEvent(QMouseEvent *event)
{
if (event->buttons() & Qt::LeftButton) {
setImagePixel(event->pos(), true);
} else if (event->buttons() & Qt::RightButton) {
setImagePixel(event->pos(), false);
}
}
```

鼠标移动事件由函数 `mouseMoveEvent()` 处理。缺省情况下，这些事件在用户拿着鼠标时产生。调用 `QWidget::setMouseTracking()` 改变这个行为，但是这个例子中我们不需要。

点击鼠标左键设置像素，点击右键清除像素。同样一直按住鼠标或者鼠标焦点在像素位置时也进行设置和清除像素。由于可以同时点击多个鼠标键，`QMouseEvent::buttons()` 返回的值是和鼠标键按位或运算得到的。使用 `&` 运算可以确定点击的鼠标键，如果是这样，就调用 `setImagePixel()`。

```
void IconEditor::setImagePixel(const QPoint &pos, bool opaque)
{
int i = pos.x() / zoom;
int j = pos.y() / zoom;
if (image.rect().contains(i, j)) {
if (opaque) {
image.setPixel(i, j, penColor().rgba());
```

```

} else {
image.setPixel(i, j, qRgb(0, 0, 0));
}
update(pixelRect(i, j));
}
}

```

函数 `setImagePixel()` 由 `mousePressEvent()` 和 `mouseMoveEvent()` 调用进行设置和清除像素。参数 `pos` 是控件上鼠标的位置。

首先坐标值除以放大倍数是把控件坐标系的鼠标位置转换为图像坐标里的位置。然后我们检查当前的点是否在有效区域内，使用的函数是 `QImage::rect()` 和 `QRect::contains()`，判断 `i` 是否在 0 和 `image.width()-1` 之间，和 `j` 是否在 0 和 `image.height()-1` 之间。

根据 `opaque` 参数，我们或者设置或者清除图像像素。将像素值为透明就可以清除像素。`QImage::setPixel` 需要把画笔的 `QColor` 转换为 32 位的 ARGB 值。最后，我们调用 `update()` 重新绘制 `QRect` 区域。

成员函数我们义已经介绍完了，现在让我们回到构造函数中的 `Qt::WA_StaticContents` 属性。这个属性的含义是当控件大小改变时，控件的内容不会跟着放缩。从左上角开始保持不变。这样当控件尺寸改变时，不需要重新绘制已经绘制的区域。通常情况下，控件尺寸改变时，Qt 会产生一个控件全部可见区域的绘制事件。如果控件的属性设置为 `Qt::WA_StaticContents` 属性，绘制事件的区域就会限制在以前没有显示的部分。如果控件变小，那么没有绘制事件产生。

Figure 5-5 Resizing a `Qt::WA_StaticContents` widgets



`IconEditor` 控件已经完成了。我们可以写代码使用 `IconEditor` 作为一个独立的窗口或者时 `QMainWindow` 的一个中央控件，或者作为一个布局里的子控件，或者是 `QScrollArea` 中的子控件。在下一节中，我们把 `IconEditor` 控件集成到 `Qt Designer` 中。

5-3 把自定义控件集成到 Qt Designer 中 (Integrating Custom Widgets with Qt Designer)

要想在 Qt Designer 中使用自定义控件，必须要使 Qt Designer 能够知道我们的自定义控件的存在。有两种方法可以把新自定义控件的信息通知给 Qt Designer: “升级”法和插件法。

升级法最为简便快捷。顾名思义，升级法就是把 Qt 自有的控件进行升级改造一番。找一个 Qt 自有的控件，如果它和我们新加的自定义控件有着相似的 API，那么只要在 Qt Designer 的对话框里面完成一些新控件的信息就一切大吉，新控件就可以在 Qt Designer 中使用了。但是在编辑的时候和预览时，还是和原来的 Qt 控件没有什么两样。

现在把 HexSpinBox 控件用升级方法集成到 Qt Designer 中：

1. 用 Qt Designer 创建一个新的窗体，把控件箱里的 QSpinBox 添加到窗体中。
2. 右击旋转盒，选择“Promote to Custom Widget”上下文菜单。
3. 在弹出的对话框中，类名处填写“HexSpinBox”，头文件填写“hexspinbox.h”

好了。在 uic 生成的包含有 QSpinBox 的控件文件中，包含文件变为“hexspinbox.h”，并且初始化为一个 HexSpinBox 的实例，而不是 QSpinBox。在 Qt Designer 中，QSpinBox 表示的控件为 HexSpinBox，并且可以设置所有的 QSpinBox 的属性。

Figure 5.6. Qt Designer's custom widget dialog



升级法的缺点是不能在 Qt Designer 中设置自定义控件自己的特有属性，也不能够绘制自己。这些问题可以用插件法解决。

插件法需要创建一个动态库，使 Qt Designer 能够在实时加载，创建控件的实例。这样，Qt Designer 就可以在编辑窗体或者预览的时候使用自定义控件。Qt Designer 用 Qt

的 meta-object 系统动态获得自定义控件的全部属性。现在以 **IconEditor** 为例，用插件法把 **IconEditor** 集成到 Qt Designer 中。

首先，我们从 **QDesignerCustomWidgetInterface** 继承一个类，重写一些虚函数。我们假定这个类的源代码在 **iconeditorplugin** 目录中，**IconEditor** 类的代码在与它平行的目录 **iconeditor** 中。

这里是插件类的定义：

```
#include <QDesignerCustomWidgetInterface>

class IconEditorPlugin : public QObject, public QDesignerCustomWidgetInterface
{
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetInterface)
public:
    IconEditorPlugin(QObject *parent = 0);
    QString name() const;
    QString includeFile() const;
    QString group() const;
    QIcon icon() const;
    QString toolTip() const;
    QString whatsThis() const;
    bool isContainer() const;
    QWidget *createWidget(QWidget *parent);
};
```

IconEditorPlugin 是一个包装 **IconEditor** 控件的类厂，使用了双继承，父类为 **QObject** 和 **QDesignerCustomWidgetInterface**。宏 **Q_INTERFACES()** 告诉 moc 第二个基类为一个插件接口类。**Qt Designer** 使用类中的函数创建 **IconEditor** 的实例并得到有关它的信息。

源文件如下：

```

IconEditorPlugin::IconEditorPlugin(QObject *parent)
    : QObject(parent)
{
}

QString IconEditorPlugin::name() const
{
    return "IconEditor";
}

QString IconEditorPlugin::includeFile() const
{
    return "iconeditor.h";
}

QString IconEditorPlugin::group() const
{
    return tr("Image Manipulation Widgets");
}

QIcon IconEditorPlugin::icon() const
{
    return QIcon(":/images/iconeditor.png");
}

QString IconEditorPlugin::toolTip() const
{
    return tr("An icon editor widget");
}

QString IconEditorPlugin::whatsThis() const
{
    return tr("This widget is presented in Chapter 5 of <i>C++ GUI "
        "Programming with Qt 4</i> as an example of a custom Qt "
        "widget.");
}

```

```

bool IconEditorPlugin::isContainer() const
{
    return false;
}

QWidget *IconEditorPlugin::createWidget(QWidget *parent)
{
    return new IconEditor(parent);
}

Q_EXPORT_PLUGIN2(iconeditorplugin, IconEditorPlugin)

```

构造函数是一个空函数。

函数 `name()` 返回控件的名称。

函数 `includeFile()` 得到控件的头文件，这个头文件包含在 moc 产生的代码中

函数 `group()` 返回的是控件所属的工具箱的名字。如果 Qt Designer 中没有这个名字，就会为这个控件创建一个新的组别。

函数 `icon()` 返回控件在 Qt Designer 中用的图标。这里我们假设 IconEditorPlugin 有关联的资源文件，里面有一个图标编辑器的图像。

在 Qt Designer 的控件箱中，当鼠标移动到自定义控件时，显示 `toolTip()` 返回的字符串做为提示。

函数 `whatsThis()` 返回 Qt Designer 显示的“What’s This”提问。

函数 `isContainer()` 返回 true 说明这个控件可以包含其他控件。例如，QFrame 可以包含其他控件，则它是一个容器控件。很多 Qt 控件都可以包含其他控件，但是如果 `isContainer()` 返回 false，Qt Designer 就不允许这个控件包含其他控件了。

Qt Designer 调运函数 `createWidget()` 创建控件实例，指定父控件。

宏 `Q_EXPORT_PLUGIN2()` 必须在源文件的最后声明，这个宏使 Qt Designer 能够得到这个插件。第一个参数是这个插件的名字，第二个参数是实现这个插件类的名字。

.pro 文件如下：

```

TEMPLATE      = lib

CONFIG      += designer plugin release

HEADERS      = ../iconeditor/iconeditor.h \

```

```
iconeditorplugin.h
SOURCES      = ../iconeditor/iconeditor.cpp \
              iconeditorplugin.cpp
RESOURCES    = iconeditorplugin.qrc
DESTDIR      = $(QTDIR)/plugins/designer
```

.pro 文件假定 QTDIR 位于 Qt 的安装目录。在运行 make 或者 nmake 后，程序自动它安装到 Qt Designer 的插件目录中。安装成功后，我们就能象其他控件一样在 Qt Designer 中使用它了

如果想在 Qt Designer 集成多个控件，你可以为每个控件创建一个上面装佯的插件库，也可以使用 QDesignerCustomWidgetCollectionInterface 一次性创建。

5-4 双缓冲技术（Double Buffering）（1、简介和源代码部分）

这一节实在是有些长，翻译完后统计了一下，快到 2w 字了。考虑到阅读的方便和网络的速度，打算把这节分为 5 个部分，第一部分为双缓冲技术的一个简介和所有的代码，如果能够看懂代码，不用看译文也就可以了。第二部分为 Plotter 控件的公有函数的实现，第三部分为 Plotter 的事件处理函数的实现，第四部分为 Plotter 控件的私有函数实现，第五部分为辅助类 PlotSettings 的实现。

这里给出一些常用的中英文对照（不一定准确，我这样用的）：

Rubber band（橡皮筋线，或者橡皮线）， pixmap（图像，双缓冲中用到的图像，有时也直呼 pixmap）， off-screen pixmap（离线图像）

Plot（plot，这一节实现的就是一个绘制曲线的控件 Plotter，有时原文也叫 plot，有点小名的意思，没有翻译，直接呼之）

废话少说，以下是译文：

双缓冲技术是 GUI 编程中常用的技术。所谓的双缓冲就是把需要绘制的控件保存到一个图像中，然后在把图像拷贝到需要绘制的控件上。在 Qt 的早期版本中，为了用户界面更加清爽，经常用这个技术来消除闪烁。

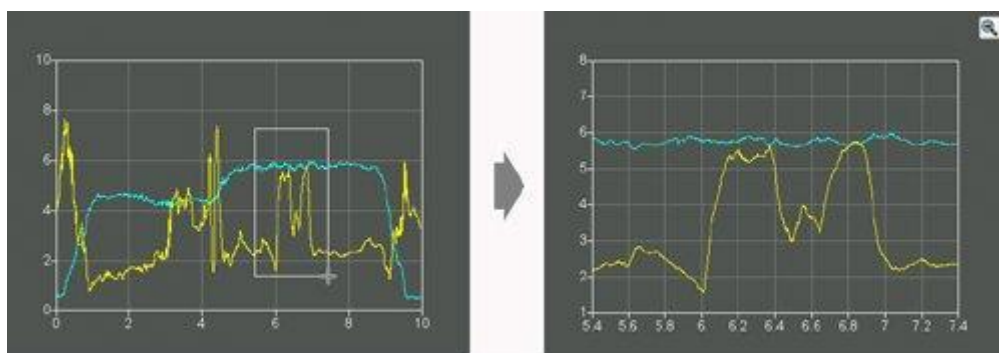
在 Qt4 中，QWidget 能够自动处理闪烁，因此我们不用再担心这个问题。尽管如此，如果控件绘制复杂且需要经常刷新，双缓冲技术还是很有用的。我们可以把控件永久保存在一个图像中，随时准备下一次绘制事件的到来，一旦接到一个控件的绘制事件，就把图片拷贝到

控件上。如果我们要做的只是小范围的修改，这个技术更是尤为有用，如要绘制一条橡皮筋线，就不必刷新整个控件了。

在本章的最后一节，我们实现的是一个叫做 **Plotter** 的自定义控件。这个控件使用了双缓冲技术，也涉及到了 **Qt** 编程的其他方面：如键盘的事件处理，布局和坐标系统。

Plotter 控件用来显示一条或者多条曲线，这些曲线由一组向量坐标表示。用户可以在显示的曲线上画一个橡皮筋线，**Plotter** 控件对橡皮筋线包围的区域进行放大。用户用鼠标左键在控件上选择一个点，然后拖动鼠标走到另一点，然后释放鼠标，就在控件上绘制一条橡皮筋线。

Figure 5.7 Zooming in on the Plotter Widget



用户可以多次用橡皮筋线进行放大，也可以用 **ZoomOut** 按钮缩小，然后用 **ZoomIn** 按钮再放大。**ZoomOut** 和 **ZoomIn** 按钮只是在控件第一次放大或者缩小操作后变得可见，如果用户不缩放图形，则这两个按钮会一直不可见，这样可以使绘图区域不那么混乱。

Plotter 控件可以存储任何数量的曲线的数据。同时它还维护一个 **PlotSettings** 对象的堆栈区域，每一个 **PlotSettings** 对象都是对应一个特定的放缩值。

首先看一下头文件的代码(对头文件的解析在代码中用注释的形式给出)：

```
#ifndef PLOTTER_H
#define PLOTTER_H
#include <QMap> //包含的 Qt 的头文件
#include <QPixmap>
#include <QVector>
#include <QWidget>
class QToolButton; //两个前向声明
class PlotSettings;
```

```

class Plotter : public QWidget
{
    Q_OBJECT
public:
    Plotter(QWidget *parent = 0);
    void setPlotSettings(const PlotSettings &settings);
    void setCurveData(int id, const QVector<QPointF> &data);
    void clearCurve(int id);
    QSize minimumSizeHint() const; //重写 QWidget::minimumSizeHint()
    QSize sizeHint() const;      //重写 QWidget::sizeHint()
public slots:
    void zoomIn(); //放大曲线
    void zoomOut(); //缩小显示曲线
protected: //重写的事件处理函数
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
    void keyPressEvent(QKeyEvent *event);
    void wheelEvent(QWheelEvent *event);
private:
    void updateRubberBandRegion();
    void refreshPixmap();
    void drawGrid(QPainter *painter);
    void drawCurves(QPainter *painter);
    enum { Margin = 50 };
    QToolButton *zoomInButton;
    QToolButton *zoomOutButton;
    QMap<int, QVector<QPointF> > curveMap; //曲线数据

```

```

    QVector<PlotSettings> zoomStack; //PlotSettings 堆栈区域

    int curZoom;

    bool rubberBandIsShown;

    QRect rubberBandRect;

    QPixmap pixmap; //显示在屏幕的控件的一个拷贝,任何绘制总是先在 pixmap 进行,
    然//后拷贝到控件上
};

//PlotSettings 确定 x, y 轴的范围, 和刻度的个数

class PlotSettings
{
public:
    PlotSettings();

    void scroll(int dx, int dy);
    void adjust();
    double spanX() const { return maxX - minX; }
    double spanY() const { return maxY - minY; }

    double minX;
    double maxX;
    int numXTicks;
    double minY;
    double maxY;
    int numYTicks;

private:
    static void adjustAxis(double &min, double &max, int &numTicks);
};

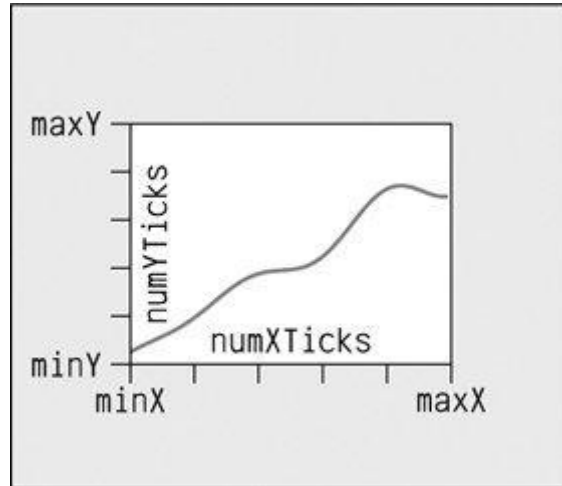
#endif

```

图 5-8 表示了 `Plotter` 控件和 `PlotSettings` 的关系。

通常，`numXTicks` 和 `numYTicks` 是有一个的误差，如果 `numXTicks` 为 5，实际上 `Plotter` 会在 x 轴上绘制 6 个刻度。这样可以简化以后的计算（至于怎么样简化的，就看程序和后文吧吧）。

Figure 5-8 `PlotSettings`'s member variables



现在来看源文件（代码有些长，先用代码格式给出完整源文件代码）：

```
#include <QtGui>
#include <cmath>

#include "plotter.h"

Plotter::Plotter(QWidget *parent)
    : QWidget(parent)
{
    setBackgroundRole(QPalette::Dark);
    setAutoFillBackground(true);
    setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    setFocusPolicy(Qt::StrongFocus);
    rubberBandIsShown = false;

    zoomInButton = new QToolButton(this);
    zoomInButton->setIcon(QIcon(":/images/zoomin.png"));
```

```

zoomInButton->adjustSize();

connect(zoomInButton, SIGNAL(clicked()), this, SLOT(zoomIn()));

zoomOutButton = new QToolButton(this);

zoomOutButton->setIcon(QIcon(":/images/zoomout.png"));

zoomOutButton->adjustSize();

connect(zoomOutButton, SIGNAL(clicked()), this, SLOT(zoomOut()));

setPlotSettings(PlotSettings());
}

void Plotter::setPlotSettings(const PlotSettings &settings)
{
    zoomStack.clear();

    zoomStack.append(settings);

    curZoom = 0;

    zoomInButton->hide();

    zoomOutButton->hide();

    refreshPixmap();
}

void Plotter::zoomOut()
{
    if (curZoom > 0) {
        --curZoom;

        zoomOutButton->setEnabled(curZoom > 0);

        zoomInButton->setEnabled(true);

        zoomInButton->show();

        refreshPixmap();
    }
}

```

```

}

void Plotter::zoomIn()
{
    if (curZoom < zoomStack.count() - 1) {
        ++curZoom;
        zoomInButton->setEnabled(curZoom < zoomStack.count() - 1);
        zoomOutButton->setEnabled(true);
        zoomOutButton->show();
        refreshPixmap();
    }
}

void Plotter::setCurveData(int id, const QVector<QPointF> &data)
{
    curveMap[id] = data;
    refreshPixmap();
}

void Plotter::clearCurve(int id)
{
    curveMap.remove(id);
    refreshPixmap();
}

QSize Plotter::minimumSizeHint() const
{
    return QSize(6 * Margin, 4 * Margin);
}

```

```

QSize Plotter::sizeHint() const
{
    return QSize(12 * Margin, 8 * Margin);
}

void Plotter::paintEvent(QPaintEvent * /* event */)
{
    QStylePainter painter(this);
    painter.drawPixmap(0, 0, pixmap);
    if (rubberBandIsShown) {
        painter.setPen(palette().light().color());
        painter.drawRect(rubberBandRect.normalized()
                        .adjusted(0, 0, -1, -1));
    }
    if (hasFocus()) {
        QStyleOptionFocusRect option;
        option.initFrom(this);
        option.backgroundColor = palette().dark().color();
        painter.drawPrimitive(QStyle::PE_FrameFocusRect, option);
    }
}

void Plotter::resizeEvent(QResizeEvent * /* event */)
{
    int x = width() - (zoomInButton->width()
                    + zoomOutButton->width() + 10);
    zoomInButton->move(x, 5);
    zoomOutButton->move(x + zoomInButton->width() + 5, 5);
    refreshPixmap();
}

```

```

void Plotter::resizeEvent(QResizeEvent * /* event */)
{
    int x = width() - (zoomInButton->width()
        + zoomOutButton->width() + 10);

    zoomInButton->move(x, 5);

    zoomOutButton->move(x + zoomInButton->width() + 5, 5);

    refreshPixmap();
}

void Plotter::resizeEvent(QResizeEvent * /* event */)
{
    int x = width() - (zoomInButton->width()
        + zoomOutButton->width() + 10);

    zoomInButton->move(x, 5);

    zoomOutButton->move(x + zoomInButton->width() + 5, 5);

    refreshPixmap();
}

void Plotter::mousePressEvent(QMouseEvent *event)
{
    QRect rect(Margin, Margin,
        width() - 2 * Margin, height() - 2 * Margin);

    if (event->button() == Qt::LeftButton) {
        if (rect.contains(event->pos())) {
            rubberBandIsShown = true;

            rubberBandRect.setTopLeft(event->pos());

            rubberBandRect.setBottomRight(event->pos());

            updateRubberBandRegion();

            setCursor(Qt::CrossCursor);
        }
    }
}

```



```

}

void Plotter::mouseMoveEvent(QMouseEvent *event)
{
    if (rubberBandIsShown) {
        updateRubberBandRegion();
        rubberBandRect.setBottomRight(event->pos());
        updateRubberBandRegion();
    }
}

void Plotter::mouseReleaseEvent(QMouseEvent *event)
{
    if ((event->button() == Qt::LeftButton) && rubberBandIsShown) {
        rubberBandIsShown = false;
        updateRubberBandRegion();
        unsetCursor();

        QRect rect = rubberBandRect.normalized();

        if (rect.width() < 4 || rect.height() < 4)
            return;

        rect.translate(-Margin, -Margin);

        PlotSettings prevSettings = zoomStack[curZoom];
        PlotSettings settings;

        double dx = prevSettings.spanX() / (width() - 2 * Margin);
        double dy = prevSettings.spanY() / (height() - 2 * Margin);

        settings.minX = prevSettings.minX + dx * rect.left();
        settings.maxX = prevSettings.minX + dx * rect.right();
        settings.minY = prevSettings.maxY - dy * rect.bottom();
        settings.maxY = prevSettings.maxY - dy * rect.top();

        settings.adjust();

        zoomStack.resize(curZoom + 1);
        zoomStack.append(settings);
    }
}

```

```

        zoomIn();
    }
}

void Plotter::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
        case Qt::Key_Plus:
            zoomIn();

            break;

        case Qt::Key_Minus:
            zoomOut();

            break;

        case Qt::Key_Left:
            zoomStack[curZoom].scroll(-1, 0);
            refreshPixmap();

            break;

        case Qt::Key_Right:
            zoomStack[curZoom].scroll(+1, 0);
            refreshPixmap();

            break;

        case Qt::Key_Down:
            zoomStack[curZoom].scroll(0, -1);
            refreshPixmap();

            break;

        case Qt::Key_Up:
            zoomStack[curZoom].scroll(0, +1);
            refreshPixmap();

            break;

        default:

```

```

        QWidget::keyPressEvent(event);
    }
}

void Plotter::wheelEvent(QWheelEvent *event)
{
    int numDegrees = event->delta() / 8;
    int numTicks = numDegrees / 15;
    if (event->orientation() == Qt::Horizontal) {
        zoomStack[curZoom].scroll(numTicks, 0);
    } else {
        zoomStack[curZoom].scroll(0, numTicks);
    }
    refreshPixmap();
}

void Plotter::updateRubberBandRegion()
{
    QRect rect = rubberBandRect.normalized();
    update(rect.left(), rect.top(), rect.width(), 1);
    update(rect.left(), rect.top(), 1, rect.height());
    update(rect.left(), rect.bottom(), rect.width(), 1);
    update(rect.right(), rect.top(), 1, rect.height());
}

void Plotter::refreshPixmap()
{
    pixmap = QPixmap(size());
    pixmap.fill(this, 0, 0);
    QPainter painter(&pixmap);
    painter.initFrom(this);
    drawGrid(&painter);
}

```

```

drawCurves(&painter);

update();
}

void Plotter::drawGrid(QPainter *painter)
{
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);

    if (!rect.isValid())
        return;

    PlotSettings settings = zoomStack[curZoom];

    QPen quiteDark = palette().dark().color().light();
    QPen light = palette().light().color();

    for (int i = 0; i <= settings.numXTicks; ++i) {
        int x = rect.left() + (i * (rect.width() - 1)
                               / settings.numXTicks);

        double label = settings.minX + (i * settings.spanX()
                                         / settings.numXTicks);

        painter->setPen(quiteDark);
        painter->drawLine(x, rect.top(), x, rect.bottom());

        painter->setPen(light);
        painter->drawLine(x, rect.bottom(), x, rect.bottom() + 5);
        painter->drawText(x - 50, rect.bottom() + 5, 100, 15,
                         Qt::AlignHCenter | Qt::AlignTop,
                         QString::number(label));
    }

    for (int j = 0; j <= settings.numYTicks; ++j) {
        int y = rect.bottom() - (j * (rect.height() - 1)
                                 / settings.numYTicks);

        double label = settings.minY + (j * settings.spanY()
                                         / settings.numYTicks);
    }
}

```

```

        / settings.numYTicks);

painter->setPen(quiteDark);

painter->drawLine(rect.left(), y, rect.right(), y);

painter->setPen(light);

painter->drawLine(rect.left() - 5, y, rect.left(), y);

painter->drawText(rect.left() - Margin, y - 10, Margin - 5, 20,
                  Qt::AlignRight | Qt::AlignVCenter,
                  QString::number(label));
}

painter->drawRect(rect.adjusted(0, 0, -1, -1));
}

void Plotter::drawCurves(QPainter *painter)
{
    static const QColor colorForIds[6] = {
        Qt::red, Qt::green, Qt::blue, Qt::cyan, Qt::magenta, Qt::yellow
    };

    PlotSettings settings = zoomStack[curZoom];

    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);

    if (!rect.isValid())
        return;

    painter->setClipRect(rect.adjusted(+1, +1, -1, -1));

    QMapIterator<int, QVector<QPointF>> i(curveMap);

    while (i.hasNext()) {
        i.next();

        int id = i.key();

        const QVector<QPointF> &data = i.value();

        QPolygonF polyline(data.count());

        for (int j = 0; j < data.count(); ++j) {

```

```

    double dx = data[j].x() - settings.minX;

    double dy = data[j].y() - settings.minY;

    double x = rect.left() + (dx * (rect.width() - 1)
                               / settings.spanX());

    double y = rect.bottom() - (dy * (rect.height() - 1)
                                / settings.spanY());

    polyline[j] = QPointF(x, y);
}

painter->setPen(colorForIds[uint(id) % 6]);

painter->drawPolyline(polyline);
}
}

```

```

////////////////////////////////////

```

```

PlotSettings::PlotSettings()

```

```

{
    minX = 0.0;
    maxX = 10.0;
    numXTicks = 5;
    minY = 0.0;
    maxY = 10.0;
    numYTicks = 5;
}

```

```

void PlotSettings::scroll(int dx, int dy)

```

```

{
    double stepX = spanX() / numXTicks;

    minX += dx * stepX;
    maxX += dx * stepX;

    double stepY = spanY() / numYTicks;

```

```

    minY += dy * stepY;
    maxY += dy * stepY;
}

void PlotSettings::adjust()
{
    adjustAxis(minX, maxX, numXTicks);
    adjustAxis(minY, maxY, numYTicks);
}

void PlotSettings::adjustAxis(double &min, double &max,
                             int &numTicks)
{
    const int MinTicks = 4;
    double grossStep = (max - min) / MinTicks;
    double step = pow(10.0, floor(log10(grossStep)));
    if (5 * step < grossStep) {
        step *= 5;
    } else if (2 * step < grossStep) {
        step *= 2;
    }
    numTicks = int(ceil(max / step) - floor(min / step));
    if (numTicks < MinTicks)
        numTicks = MinTicks;
    min = floor(min / step) * step;
    max = ceil(max / step) * step;
}

```

5-5 双缓冲技术（Double Buffering）（2、公有函数实现）

```
#include <QtGui>
```

```
#include <cmath>
using namespace std;
#include "plotter.h"
```

以上代码为文件的开头，在这里把 **std** 的名空间加入到当前的全局命名空间。这样在使用 **<cmath>** 里的函数时，就不用前缀 **std::** 了，如可以直接使用函数 **floor()**，而不用写成 **std::floor()**。

```
Plotter::Plotter(QWidget *parent) : QWidget(parent)
{
    setBackgroundRole(QPalette::Dark);
    setAutoFillBackground(true);
    setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    setFocusPolicy(Qt::StrongFocus);
    rubberBandIsShown = false;

    zoomInButton = new QToolButton(this);
    zoomInButton->setIcon(QIcon(":/images/zoomin.png"));
    zoomInButton->adjustSize();
    connect(zoomInButton, SIGNAL(clicked()), this, SLOT(zoomIn()));

    zoomOutButton = new QToolButton(this);
    zoomOutButton->setIcon(QIcon(":/images/zoomout.png"));
    zoomOutButton->adjustSize();
    connect(zoomOutButton, SIGNAL(clicked()), this, SLOT(zoomOut()));

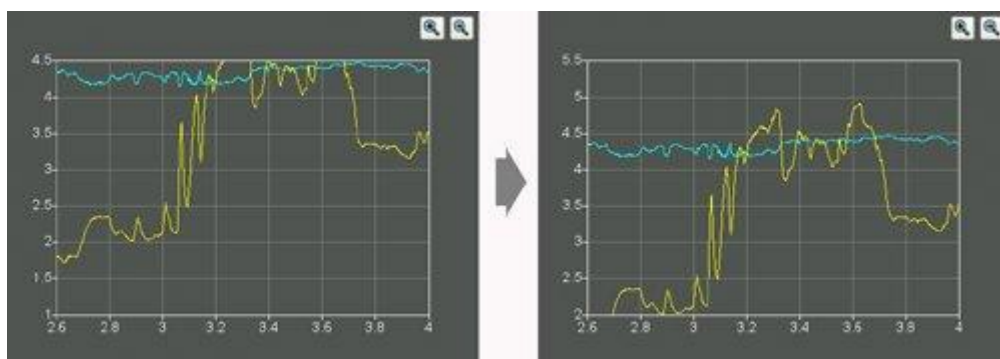
    setPlotSettings(PlotSettings());
}
```

在构造函数中，调用 **setBackgroundRole(QPalette::Dark)**，当对控件进行放大需要重新绘制时，提供给 Qt 一个缺省的颜色填充新的区域，为了能够使用这个机制，还调用了 **setAutoFillBackground(true)**。

函数 `setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding)` 让控件在水平和垂直两个方向上都可以进行伸缩。如果控件需要占据屏幕上很大的控件，经常设置这个属性。缺省的设置是两个方向都是 `QSizePolicy::Preferred`，意思是控件的实际尺寸和它的 `sizeHint` 一致，控件最小只能缩小到它的最小的 `sizeHint`，并能够无限放大。

调用 `setFocusPolicy(Qt::StrongFocus)` 可以使控件通过鼠标点击或者 `Tab` 得到焦点。当 `Plotter` 控件得到焦点时，它可以接受键盘敲击事件。`Plotter` 控件能够理解一些键盘事件，如 `+` 放大，`-` 为缩小，可以向上下左右平移。

Figure 5.9. Scrolling the Plotter widget



在构造函数中，我们还创建了两个 `QToolButton`，每一个按钮都有一个图标。点击这些图标可以放大或者缩小显示的图像。图标保存在资源文件中，为了任何程序都可以使用 `Plotter` 控件，需要在 `.pro` 添加资源条目：

```
RESOURCES    = plotter.qrc
```

资源文件是一个 XML 格式的文本文件，和在 `Spreadsheet` 中使用的很像：

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/zoomin.png</file>
    <file>images/zoomout.png</file>
</qresource>
</RCC>
```

调用 `QToolButton::adjustSize()` 调整按钮的大小为它们的 `sizeHint`。在这里按钮不在布局中，在控件大小改变的时候，又程序计算它们的位置。由于没有了布局管理，因为我们必须在按钮的构造函数中确定按钮的父控件。

调用 `setPlotSettings()` 函数用来完成控件的初始化。函数代码如下：

```
void Plotter::setPlotSettings(const PlotSettings &settings)
{
    zoomStack.clear();
    zoomStack.append(settings);
    curZoom = 0;
    zoomInButton->hide();
    zoomOutButton->hide();
    refreshPixmap();
}
```

函数 `setPlotSettings()` 确定显示控件时的 `PlotSettings`。它在 `Plotter` 构造函数中调用，也可以被 `Plotter` 的用户调用。开始的时候，`Plotter` 使用的是缺省的放缩值。用户进行放大一次，就有一个新的 `PlotSettings` 对象加入到堆栈中。这个堆栈中有两个变量：

`zoomStack` 是保存 `PlotSettings` 对象的一个数组；

`curZoom` 是当前使用的 `PlotSettings` 的一个索引值。

调用 `setPlotSettings()` 后，`zoomStack` 中只有一项，`zoomIn` 和 `zoomOut` 按钮隐藏。如果我们调用函数 `zoomIn()` 和 `zoomOut()`，这两个函数中调用了按钮的 `show()` 函数，它们才能显示出来。（通常，调用父控件的 `show()` 函数就显示所有的子控件。但是如果显式调用了子控件的 `hide()`，必须要显示调用其 `show()` 函数显示它，否则就会一直隐藏）

调用 `refreshPixmap()` 来更新显示。通常，我们调用 `update()` 就可以，这里有些不一样，因为我们要保持 `QPixmap` 一直最新的状态。更新了图片后，`refreshPixmap()` 再调用 `update()` 把图片显示到控件上。

```
void Plotter::zoomOut()
{
    if (curZoom > 0) {
        --curZoom;
        zoomOutButton->setEnabled(curZoom > 0);
        zoomInButton->setEnabled(true);
        zoomInButton->show();
    }
```

```

        refreshPixmap();
    }
}

```

如果图片放大了，调用 `zoomOut()` 缩小它。它缩小比例系数，如果还能进一步缩小，`zoomOut` 按钮一直有效。显示 `zoomIn` 按钮使之按钮有效，调用 `refreshPixmap()` 刷新控件。

```

void Plotter::zoomIn()
{
    if (curZoom < zoomStack.count() - 1) {
        ++curZoom;

        zoomInButton->setEnabled(curZoom < zoomStack.count() - 1);
        zoomOutButton->setEnabled(true);
        zoomOutButton->show();
        refreshPixmap();
    }
}

```

如果用户放大后又缩小控件，下一个放缩系数的 `PlotSettings` 就进入 `zoomStack`。我们就可以再放大控件。

函数 `zoomIn` 增加放缩系数，`zoomIn` 按钮显示出来，只要能够放大，按钮会一直有效。同时显示 `zoomOut` 按钮使之有效状态。

```

void Plotter::setCurveData(int id, const QVector<QPointF> &data)
{
    curveMap[id] = data;
    refreshPixmap();
}

```

函数 `setCurveData()` 设置一个指定 `id` 的曲线数据。如果曲线中有一个同样的 `id`，那么就用新的数据替代旧数据。如果没有指定的 `id`，则增加一个新的曲线。曲线的数据类型为 `QMap<int, QVector<QPointF> >`

```

void Plotter::clearCurve(int id)
{

```

```

        curveMap.remove(id);
        refreshPixmap();
    }

```

函数 `clearCurve()` 删除一个指定 `id` 的曲线。

```

QSize Plotter::minimumSizeHint() const
{
    return QSize(6 * Margin, 4 * Margin);
}

```

函数 `minimumSizeHint()` 和 `sizeHint()` 很像，确定控件的理想尺寸。

`minimumSizeHint()` 确定控件的最大尺寸。布局管理器排列控件时不会超过控件的最大尺寸。

由于 `Margin` 值为 50，所以我们返回的值为 300×200 ，包括四个边界的宽度和 `Plot` 本身。如果再小，尺寸太小 `Plot` 就不能正常显示了。

```

QSize Plotter::sizeHint() const
{
    return QSize(12 * Margin, 8 * Margin);
}

```

在 `sizeHint()` 中，我们返回控件的理想尺寸，用 `Margin` 常数作为倍数，长宽的比例为 3: 2，与 `minimumSizeHint()` 中比例一致。

以上是 `Plotter` 的公有函数和槽函数。

5-6 双缓冲技术 (Double **Buffering**) (3、事件处理函数)

以下是 `Plotter` 控件的事件处理函数部分

```

void Plotter::paintEvent(QPaintEvent * /* event */)
{
    QStylePainter painter(this);
    painter.drawPixmap(0, 0, pixmap);
    if (rubberBandIsShown) {

```

```

        painter.setPen(palette().light().color());
        painter.drawRect(rubberBandRect.normalized()
                        .adjusted(0, 0, -1, -1));
    }
    if (hasFocus()) {
        QStyleOptionFocusRect option;
        option.initFrom(this);
        option.backgroundColor = palette().dark().color();
        painter.drawPrimitive(QStyle::PE_FrameFocusRect, option);
    }
}

```

通常情况下，`paintEvent()`是我们处理控件的所有绘制的地方。这里 **Plotter** 控件的绘制是在 `refreshPixmap()`中完成的，因此在 `paintEvent()`函数中只是把图片显示在控件的 (0, 0) 位置上。

如果能看见橡皮线，我们把它画到控件的上面。使用控件当前颜色组的“轻”的颜色橡皮线的颜色，和“黑”的背景形成对比。需要注意的是这个线是直接画在控件上的，对图片没有任何影响。使用 `QRect::normalized()`确保橡皮线的矩形有着正数的宽和高，`adjusted()`减掉一个像素宽的矩形，显示它的轮廓。

如果 **Plotter** 有焦点，用控件样式的 `drawPrimitive()`绘制一个焦点矩形，第一个参数为 `QStyle::PE_FrameFocusRect`，第二个参数为一个 `QStyleOptionFocusRect` 对象。焦点矩形的绘制选项用 `initFrom()`函数设置，继承自 **Plotter**，但是背景颜色必须明确设置。如果我们想使用当前的样式，我们可以直接调用 `QStyle` 的函数，比如：

```
style()->drawPrimitive(QStyle::PE_FrameFocusRect, &option, &painter, this);
```

或者我们使用 `QStylePainter`，能绘制更加方便。

`QWidget::Style()`函数返回绘制控件使用的样式。在 **Qt** 中，一个控件的样式是 `QStyle` 的基类。**Qt** 提供的样式有 `QWindowStyle`，`QWindowXpStyle`，`QMotifStyle`，`QCDStyle`，`QMacStyle` 和 `QPlastiqueStyle`。这些样式类都是重新实现了 `QStyle` 的虚函数来模拟特定平台的样式。`QStylePainter::drawPrimitive()`函数调用 `QStyle` 的同名函数，绘制控件的一些基本原色，如面板，按钮，焦点矩形等。在一个应用程序中，所有

控件的样式都是一样的，可用通过 `QApplication::style()` 得到，也可以用 `QWidget::setStyle()` 设置某一个控件的样式。

把 `QStyle` 作为基类，可以定义一个用户样式。可以让一个应用程序看起来与众不同。通常的建议是使用和目标平台一致的样式。只要你有想法，`Qt` 提供了很多的灵活性。`Qt` 提供的控件都是用 `QStyle` 绘制自己，所以在所有 `Qt` 支持的平台上，它们看起来都和平台的风格一致。

用户空间可以使用 `QStyle` 绘制自己或者使用 `Qt` 提供的控件作为子控件。对于 `Plotter`，我们使用两个方式的组合，焦点矩形用 `QStyle` 样式绘制，`zoomIn` 和 `zoomOut` 按钮为 `Qt` 提供的控件。

```
void Plotter::resizeEvent(QResizeEvent * /* event */)
{
    int x = width() - (zoomInButton->width()+ zoomOutButton->width() + 10);
    zoomInButton->move(x, 5);
    zoomOutButton->move(x + zoomInButton->width() + 5, 5);
    refreshPixmap();
}
```

控件大小改变时，`Qt` 都会产生一个“resize”事件。这里，我们重写了 `resizeEvent()` 把 `zoomIn` 和 `zoomOut` 按钮放在 `Plotter` 控件的右上角。

我们把 `zoomIn` 和 `zoomOut` 按钮并排摆放，中间有 5 个像素的空隙，距离控件的上边距和右边距也为 5 个像素宽。

如果我们希望按钮放在控件的左上角（坐标点为（0，0））上，直接可以在 `Plotter` 构造函数中它们移动到左上角。如果我们想跟踪控件的右上角，它的坐标取决与控件的大小。因此需要重写 `resizeEvent()` 设置按钮位置。

在 `Plotter` 构造函数中，我们没有确定按钮的位置。但是不要紧，在控件第一次显示之前，`Qt` 就会产生一个 `resize` 事件。

如果不重写 `resizeEvent()` 手工排列子控件，还可以使用布局管理器，如 `QGridLayout`。使用一个布局会有些复杂，也会消耗更多资源。另一方面，它能够把左右排列的布局安排的更好，对 `Arabic` 和 `Hebrew` 语言尤其适用。

最后，调用 `refreshPixmap()` 绘制新的尺寸下的图片。

```

void Plotter::mousePressEvent(QMouseEvent *event)
{
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);
    if (event->button() == Qt::LeftButton) {
        if (rect.contains(event->pos())) {
            rubberBandIsShown = true;
            rubberBandRect.setTopLeft(event->pos());
            rubberBandRect.setBottomRight(event->pos());
            updateRubberBandRegion();
            setCursor(Qt::CrossCursor);
        }
    }
}

```

当用户点击了鼠标左键，在控件上显示出一个橡皮线，显示的条件是 `rubberBandIsShown` 为 `true`。把变量 `rubberBandRect` 的左上角和右下角都为当前的鼠标点，然后发出一个绘制事件绘制橡皮线，同时把光标改为十字型。

变量 `rubberBandRect` 为 `QRect` 类型。一个 `QRect` 可以由四个量 (`x,y,width,height`) 定义。其中 (`x,y`) 为矩形左上角的坐标，`width*height` 为矩形的面积。或者由左上角和右下角的坐标对定义。在这里使用了坐标对定义的方法，把矩形的左上角和右下角的坐标都设置为鼠标点击的位置。然后调用 `updateRubberBandRegion()` 把橡皮线内的区域绘制出来。

Qt 有两种设置光标形状的方法：

`QWidget::setCursor()`，当鼠标移动到一个控件上时，使用这个函数设置光标的形状。如果子控件上没有设置光标形状，则使用父控件的光标。通常使用的光标是一个箭头式光标。`QApplication::setOverrideCursor()` 设置应用程序的光标形状，取代控件中设定的光标，调用 `restoreOverrideCursor()` 后光标回到原来的状态。

在第四章中，我们调用了 `QApplication::setOverrideCursor()` 把光标设置为 `Qt::WaitCursor`，把应用程序光标设置为等待式光标。

```

void Plotter::mouseMoveEvent(QMouseEvent *event)

```

```

{
    if (rubberBandIsShown) {
        updateRubberBandRegion();
        rubberBandRect.setBottomRight(event->pos());
        updateRubberBandRegion();
    }
}

```

当用户点击鼠标左键移动鼠标时，调用 `updateRubberBandRegion()` 重新绘制橡皮线所在区域。然后根据鼠标移动的位置重新计算橡皮线区域的大小，最后在调用 `updateRubberBandRegion()` 绘制新的橡皮线区域。这样就可以删除原来的橡皮线，在新的位置绘制新的橡皮线。

如果用户向上或者向下移动鼠标，`rubberBandRect` 的右下角可能会到达它的左上角的上面或者左面，`QRect` 的 `width` 和 `height` 会出现负值，在 `paintEvent()` 函数中调用了 `QRect::normalized()` 函数，它可以重新计算矩形的左上角和右下角的坐标值，保证得到一个非负的宽和高。

```

void Plotter::mouseReleaseEvent(QMouseEvent *event)
{
    if ((event->button() == Qt::LeftButton) && rubberBandIsShown) {
        rubberBandIsShown = false;
        updateRubberBandRegion();
        unsetCursor();
        QRect rect = rubberBandRect.normalized();
        if (rect.width() < 4 || rect.height() < 4)
            return;
        rect.translate(-Margin, -Margin);
        PlotSettings prevSettings = zoomStack[curZoom];
        PlotSettings settings;
        double dx = prevSettings.spanX() / (width() - 2 * Margin);
        double dy = prevSettings.spanY() / (height() - 2 * Margin);
        settings.minX = prevSettings.minX + dx * rect.left();
    }
}

```



```

settings.maxX = prevSettings.minX + dx * rect.right();
settings.minY = prevSettings.maxY - dy * rect.bottom();
settings.maxY = prevSettings.maxY - dy * rect.top();
settings.adjust();

zoomStack.resize(curZoom + 1);
zoomStack.append(settings);

zoomIn();
}
}

```

用户释放鼠标左键时，我们删除橡皮线，恢复到正常的箭头式光标。如果橡皮线区域大于 4×4 ，则把这个区域放大。如果小于这个值，则很可能是用户的一个误操作，也许只是想给控件一个焦点罢了，程序返回，什么都不做了。

进行放大的这部分代码有点复杂，因为我们需要同时处理控件坐标和 **plotter** 的坐标。大部分代码都是把 **rubberBandRect** 从控件坐标转到 **plotter** 坐标。完成转换以后，调用 **PlotSettings::adjust()** 进行四舍五入，找到一个合理的坐标刻度。图 5-10 和图 5-11 示意了这个坐标的转换：

Figure 5.10. Converting the rubber band from widget to plotter coordinates

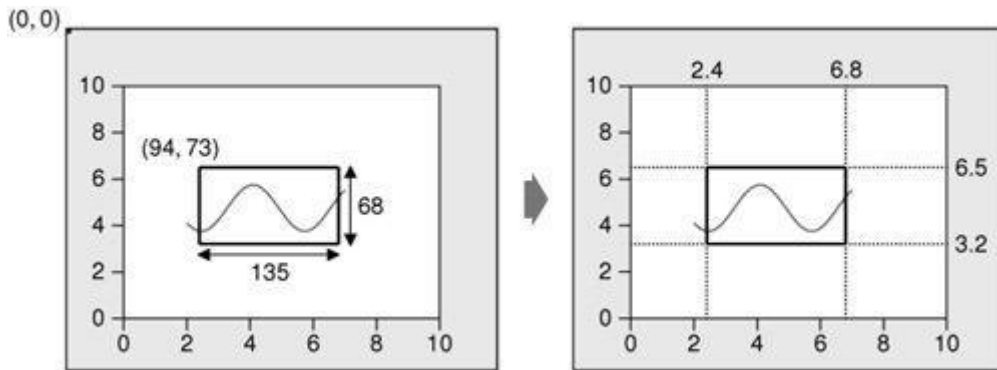
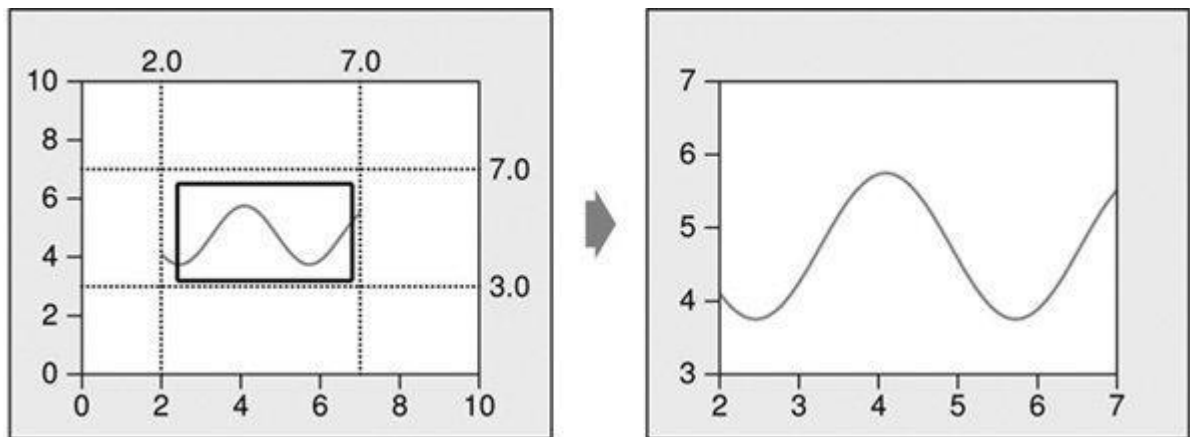


Figure 5.11. Adjusting plotter coordinates and zooming in on the rubber band



坐标转换以后，我们进行放大。同时把放大系数等设置形成一个新的 `PlotSettings` 对象，然后把它放到 `zoomStack` 的最上面。

```
void Plotter::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
        case Qt::Key_Plus:
            zoomIn();
            break;
        case Qt::Key_Minus:
            zoomOut();
            break;
        case Qt::Key_Left:
            zoomStack[curZoom].scroll(-1, 0);
            refreshPixmap();
            break;
        case Qt::Key_Right:
            zoomStack[curZoom].scroll(+1, 0);
            refreshPixmap();
            break;
        case Qt::Key_Down:
            zoomStack[curZoom].scroll(0, -1);
```

```

        refreshPixmap();
        break;
    case Qt::Key_Up:
        zoomStack[curZoom].scroll(0, +1);
        refreshPixmap();
        break;
    default:
        QWidget::keyPressEvent(event);
    }
}

```

当当前的焦点在 **Plotter** 控件上时，用户敲击了键盘的某一个键值，**keyPressEvent()** 就会调用。这里我们重写了这个函数，响应用户对 6 个键的相应：+，-，Up，Down，Left 和 Right。如果用户敲击的键不在这六个之中，则调用基类的函数进行处理。为了简便，我们这里忽略了 Shift，Ctrl，和 Alt 键，这些键可以通过 **QKeyEvent::modifiers()** 得到。

```

void Plotter::wheelEvent(QWheelEvent *event)
{
    int numDegrees = event->delta() / 8;
    int numTicks = numDegrees / 15;
    if (event->orientation() == Qt::Horizontal) {
        zoomStack[curZoom].scroll(numTicks, 0);
    } else {
        zoomStack[curZoom].scroll(0, numTicks);
    }
    refreshPixmap();
}

```

鼠标滚轮转动时，Qt 产生一个滚轮事件（**Wheel event**）。很多鼠标只有一个垂直的滚轮，但是考虑到一些鼠标也有水平滚轮，Qt 对这两种方式的滚轮都支持。滚轮事件只是发生在有焦点的控件上。函数 **delta()** 返回的是滚轮滚动了 8° 时移动的距离。一般鼠标都是以 15° 事件发生后，我们修改 **zoomStack** 最上面的设置，然后刷新图片。

滚轮鼠标一般用来处理滚动条。如果我们使用了 `QScrollArea` 提供一个可以滚动的区域，`QScrollBar` 自动处理滚轮事件，我们不用自己重写 `wheelEvent()` 函数。

5-7 双缓冲技术（**Double Buffering**）（4、私有函数的实现）

以下是私有函数的实现：

```
void Plotter::updateRubberBandRegion()

{

    QRect rect = rubberBandRect.normalized();
    update(rect.left(), rect.top(), rect.width(), 1);
    update(rect.left(), rect.top(), 1, rect.height());
    update(rect.left(), rect.bottom(), rect.width(), 1);
    update(rect.right(), rect.top(), 1, rect.height());
}
```

函数 `updateRubberBand()` 在 `mousePressEvent()`，`mouseMoveEvent()` 和 `mouseReleaseEvent()` 中被调用，用来删除或者从新绘制橡皮线。函数中调用了四次 `update()`，用四个绘制事件完成由橡皮线组成的四个小矩形的绘制。`Qt` 也提供了一个类 `QRubberBand` 用来绘制橡皮线，但是控件自己提供的绘制函数会更好

```
void Plotter::refreshPixmap()
{
    pixmap = QPixmap(size());
    pixmap.fill(this, 0, 0);
    QPainter painter(&pixmap);
    painter.initFrom(this);
    drawGrid(&painter);
    drawCurves(&painter);
    update();
}
```

函数 `refreshPixmap()` 把 `plot` 绘制到图片上，并且更新控件。首先我们把图片的大小调整为和当前控件大小相同，用控件的背景颜色填充整个图片。这个颜色是当前调色版的“dark”部分。如果背景用的刷子不是固体的（`solid brush`，刷子的样式，只有颜色，没有花纹的那种最简单的），`QPixmap::fill()` 需要知道控件中刷子的偏移量，以便图片和控件保持一致。因为我们保存的是整个控件，那么因此偏移位置为（0，0）。

在这个函数中，我们使用了一个 `QPainter` 绘制图片，`QPainter::initFrom()` 设置绘制图片所需画笔，背景和字体，参数 `this` 表示这些设置和 `Plotter` 控件的相应设置是一致的。然后我们调用 `drawGrid()`，`drawCurves()` 绘制网格和曲线。最后，`update()` 函数更新全部控件，在 `paintEvent()` 函数中把图片拷贝到控件上。

```
void Plotter::drawGrid(QPainter *painter)
{
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);
    if (!rect.isValid())
        return;

    PlotSettings settings = zoomStack[curZoom];
    QPen quiteDark = palette().dark().color().light();
    QPen light = palette().light().color();
    for (int i = 0; i <= settings.numXTicks; ++i) {
        int x = rect.left() + (i * (rect.width() - 1)
                               / settings.numXTicks);
        double label = settings.minX + (i * settings.spanX()
                                         / settings.numXTicks);
        painter->setPen(quiteDark);
        painter->drawLine(x, rect.top(), x, rect.bottom());
        painter->setPen(light);
        painter->drawLine(x, rect.bottom(), x, rect.bottom() + 5);
        painter->drawText(x - 50, rect.bottom() + 5, 100, 15,
                         Qt::AlignHCenter | Qt::AlignTop,
                         QString::number(label));
    }
}
```

```

}
for (int j = 0; j <= settings.numYTicks; ++j) {
    int y = rect.bottom() - (j * (rect.height() - 1)
                               / settings.numYTicks);

    double label = settings.minY + (j * settings.spanY()
                                     / settings.numYTicks);

    painter->setPen(quiteDark);
    painter->drawLine(rect.left(), y, rect.right(), y);
    painter->setPen(light);
    painter->drawLine(rect.left() - 5, y, rect.left(), y);
    painter->drawText(rect.left() - Margin, y - 10, Margin - 5, 20,
                     Qt::AlignRight | Qt::AlignVCenter,
                     QString::number(label));
}
painter->drawRect(rect.adjusted(0, 0, -1, -1));
}

```

函数 **drawGrid()** 在坐标轴和曲线的下面绘制网格。这个区域由一个矩形确定，如果控件太小，则不绘制。第一个循环绘制网格的垂直线，个数为 **x** 坐标轴的刻度个数。第二个循环绘制网格的水平线，共 **y** 坐标轴的刻度个数。最后，沿边界绘制一个矩形。**drawText()** 绘制两个坐标轴上刻度的个数。

函数 **painter->drawText()** 语法如下：

```
painter->drawText(x, y, width, height, alignment, text);
```

其中 **(x,y,width,height)** 所确定的矩形确定文字的大小和位置 **alignment** 为文字的对齐方式。

```
void Plotter::drawCurves(QPainter *painter)
```

```

{
    static const QColor colorForIds[6] = {
        Qt::red, Qt::green, Qt::blue, Qt::cyan, Qt::magenta, Qt::yellow
    };

    PlotSettings settings = zoomStack[curZoom];

```

```

QRect rect(Margin, Margin,
           width() - 2 * Margin, height() - 2 * Margin);
if (!rect.isValid())
    return;
painter->setClipRect(rect.adjusted(+1, +1, -1, -1));
QMapIterator<int, QVector<QPointF> > i(curveMap);
while (i.hasNext()) {
    i.next();
    int id = i.key();
    const QVector<QPointF> &data = i.value();
    QPolygonF polyline(data.count());
    for (int j = 0; j < data.count(); ++j) {
        double dx = data[j].x() - settings.minX;
        double dy = data[j].y() - settings.minY;
        double x = rect.left() + (dx * (rect.width() - 1)
                                   / settings.spanX());
        double y = rect.bottom() - (dy * (rect.height() - 1)
                                     / settings.spanY());
        polyline[j] = QPointF(x, y);
    }
    painter->setPen(colorForIds[uint(id) % 6]);
    painter->drawPolyline(polyline);
}
}

```

函数 `drawCurves()` 在网格上绘制出曲线。调用了 `QPainter::setClipRect()` 函数设置绘制曲线的矩形区域（不包括四周的间隙和框架）。`QPainter` 会忽略画到这个区域外的像素。

然后我们遍历所有的曲线，在每一条曲线，遍历它所有的 `QPointF` 点。函数 `key()` 得到曲线的 `id`，`value()` 函数得到曲线的 `QVector<QPointF>` 类型的数据。内层循环把 `QPointF` 记录的 `plotter` 坐标转换为控件坐标，把它们保存在多段线变量中。

转换坐标后，我们设置画笔的颜色（使用函数前面预定义的颜色），调用 `drawPolyline()` 绘制出所有的曲线的点。

5-8 双缓冲技术（**Double Buffering**）（5、类 **PlotSettings** 实现）

下面是 `PlotSettings` 的实现：

```
PlotSettings::PlotSettings()
```

```
{
    minX = 0.0;
    maxX = 10.0;
    numXTicks = 5;
    minY = 0.0;
    maxY = 10.0;
    numYTicks = 5;
}
```

在构造函数中，把两个坐标轴的初始化为从 0 到 10，分为 5 个刻度。

```
void PlotSettings::scroll(int dx, int dy)
```

```
{
    double stepX = spanX() / numXTicks;
    minX += dx * stepX;
    maxX += dx * stepX;
    double stepY = spanY() / numYTicks;
    minY += dy * stepY;
    maxY += dy * stepY;
}
```

函数 `scroll()` 增加或者减少 `minX`, `maxX`, `minY`, `maxY` 的值，放大或缩小控件的尺寸为给定的偏移值乘以坐标刻度的两倍。这个函数在 `Plotter::keyPressEvent()` 函数中调用。

```
void PlotSettings::adjust()
```

```
{
    adjustAxis(minX, maxX, numXTicks);
}
```



```

    adjustAxis(minY, maxY, numYTicks);
}

```

函数 `adjust()` 在 `Plotter::mouseReleaseEvent()` 中调用。重新计算 `minX`, `maxX`, `minY`, `maxY` 的值, 重新得到坐标轴刻度的个数。私有函数 `adjustAxis()` 一次计算一个坐标轴。

```

void PlotSettings::adjustAxis(double &min, double &max,
                              int &numTicks)
{
    const int MinTicks = 4;
    double grossStep = (max - min) / MinTicks;
    double step = pow(10.0, floor(log10(grossStep)));
    if (5 * step < grossStep) {
        step *= 5;
    } else if (2 * step < grossStep) {
        step *= 2;
    }
    numTicks = int(ceil(max / step) - floor(min / step));
    if (numTicks < MinTicks)
        numTicks = MinTicks;
    min = floor(min / step) * step;
    max = ceil(max / step) * step;
}

```

函数 `adjustAxis()` 修正 `minX`, `maxX`, `minY`, `maxY` 的值, 根据给定的最大最小范围值计算刻度的个数。函数修改了参数的值 (成员变量的值), 所以没有使用常引用。

前部分代码主要是确定坐标轴上单位刻度的值 (**step**)。为了得到合理的刻度数, 必须得到准确的步长值。例如, 一个坐标轴步长为 **3.8**, 坐标轴上其他的刻度值都是 **3.8** 的倍数, 在用户很不习惯, 对于一个整数坐标值, 合理的步长应给为 10^n , $2 \cdot 10^n$, 或者 $5 \cdot 10^n$ 。

首先我们计算最大步长 (**gross step**), 然后计算小于或者等于这个步长的 10^n , 通过计算这个步长的以十为底的对数, 然后计算这个值的 10 次方。例如, 如果最大步长为 **236**, $\log(236)$ 为 2.37291..., 四舍五入为 2, 得到 $10^2 = 100$ 作为候选的步长值。

有了第一个值以后，我们再继续计算其他的候选值 $2 \cdot 10^n$ 和 $5 \cdot 10^n$ 。如上例中，另外两个可能的值为 200 和 500。500 大于最大的步长值不能使用，200 小于 236，使用 200 作为步长的值。

接着计算刻度数，min 和 max 就很容易了。新的 min 值为原来的 min 值和步长乘积的较小整数值，新的 max 为原来的 max 值和步长乘积的较大整数值。新的 numTicks 为新的 min 和 max 的间隔数。例如，输入的 min 值为 240，max 为 1184，新的值就会变成 200，1200，200 为步长，就有 numTicks 值为 5；

有时这个算法并不是最优的。一个更加复杂的算法是 Paul S. Heckbert 在 Graphics Gem 上发表的一篇名为“Nice Numbers for Graph Labels”(ISBN 0-12-286166-3) 这一章是第一部分的最后一章。介绍了怎样从现有的 Qt 控件基础上得到一个新的控件，和以 QWidget 作为基类得到一个新的控件。在第二章我们看到了怎么在一个控件中对其他控件进行组合，在第六章中我们将会继续介绍。

到此为止，我们已经介绍了很多 Qt GUI 编程方面的知识。在第二部分和第三部分中，我们将会深入介绍 Qt 编程的其他方面。

第六章序-布局管理 (Chapter 6. Layout Management)

窗体上的所有的控件必须有一个合适的尺寸和位置。Qt 提供了一些类负责排列窗体上的控件，主要有：QHBoxLayout, QVBoxLayout, QGridLayout, QStackLayout。（有时在译文中我会把这些类叫做布局管理类）这些类简单易用，无论在代码中还是用 Qt Designer 开发程序都能用到。

使用这些 Qt 布局管理类的另一个原因是，在程序改变字体，语言或者在不同的平台上运行时，布局管理器能够自动调整窗体里所有控件的大小和尺寸。如果用户改变了系统的字体设置，窗体就会根据需要，自动调整控件。如果需要把程序的用户界面翻译成另外一种语言，布局管理器也会自动调整控件适应新的新的文本，避免窗体中的文字被覆盖或者剪切掉。

其他能够进行布局管理的类还有 QSplitter, QScrollArea, QMainWindow, QWorkspace。这些类的共同特点是提供了更加灵活的布局管理，在一定程度上用户能够控制窗体内控件的大小。例如，QSplitter 类显示一个分隔条（splitter bar），用户拖动分隔条时就可以改变控件的大小。QWorkspace 提供了对多文档（MDI, multiple document interface）的支持，在一个程序的主窗口内，可以同时显示多个文档。这些类也经常做为布局管理类的使用，在这一章中也会进行介绍。

6-1 排列窗体上的控件 (Laying Out Widgets on a Form)

中英文对照：

form（窗体），layout（布局或者排列，意思是进行窗体上控件的排列的过程，如大小位置等）

absolute positioning（绝对位置定位）, manual layout（手工布局）, layout managers（布局管理器）

Qt 中有三种方式对窗体上的控件进行布局管理：绝对位置定位（absolute positioning），手工布局（manual layout），布局管理器（layout managers）。我们使用图 6.1 中的对话框为例对这三种方式分别进行说明。

Figure 6.1. The Find File dialog



绝对位置定位的方法是最原始的排列控件的方法。这个方法是在程序中调用控件的函数设定它的位置和相对窗体它的大小。下面是用着个方法实现的 `FindFileDialog` 的构造函数。

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    namedLabel->setGeometry(9, 9, 50, 25);
    namedLineEdit->setGeometry(65, 9, 200, 25);
    lookInLabel->setGeometry(9, 40, 50, 25);
    lookInLineEdit->setGeometry(65, 40, 200, 25);
    subfoldersCheckBox->setGeometry(9, 71, 256, 23);
    tableWidget->setGeometry(9, 100, 256, 100);
    messageLabel->setGeometry(9, 206, 256, 25);
    findButton->setGeometry(271, 9, 85, 32);
    stopButton->setGeometry(271, 47, 85, 32);
    closeButton->setGeometry(271, 84, 85, 32);
    helpButton->setGeometry(271, 199, 85, 32);
    setWindowTitle(tr("Find Files or Folders"));
    setFixedSize(365, 240);
}
```

```
}
```

这种方法缺点很多：

1. 用户不能改变窗体的大小
2. 如果改变字体或者翻译到另一种语言，控件上的文本可能不能完全显示
3. 在一些样式下，控件的尺寸会不合适

另一种方法为手工布局。给出控件的绝对位置，但是他们的尺寸根据窗口的大小确定，可以通过重写窗体的 `resizeEvent()` 实现对子控件的大小设置：

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    setMinimumSize(265, 190);
    resize(365, 240);
}

void FindFileDialog::resizeEvent(QResizeEvent * /* event */)
{
    int extraWidth = width() - minimumWidth();
    int extraHeight = height() - minimumHeight();
    namedLabel->setGeometry(9, 9, 50, 25);
    namedLineEdit->setGeometry(65, 9, 100 + extraWidth, 25);
    lookInLabel->setGeometry(9, 40, 50, 25);
    lookInLineEdit->setGeometry(65, 40, 100 + extraWidth, 25);
    subfoldersCheckBox->setGeometry(9, 71, 156 + extraWidth, 23);
    tableWidget->setGeometry(9, 100, 156 + extraWidth,
                             50 + extraHeight);
    messageLabel->setGeometry(9, 156 + extraHeight, 156 + extraWidth,
                             25);
    findButton->setGeometry(171 + extraWidth, 9, 85, 32);
    stopButton->setGeometry(171 + extraWidth, 47, 85, 32);
    closeButton->setGeometry(171 + extraWidth, 84, 85, 32);
}
```

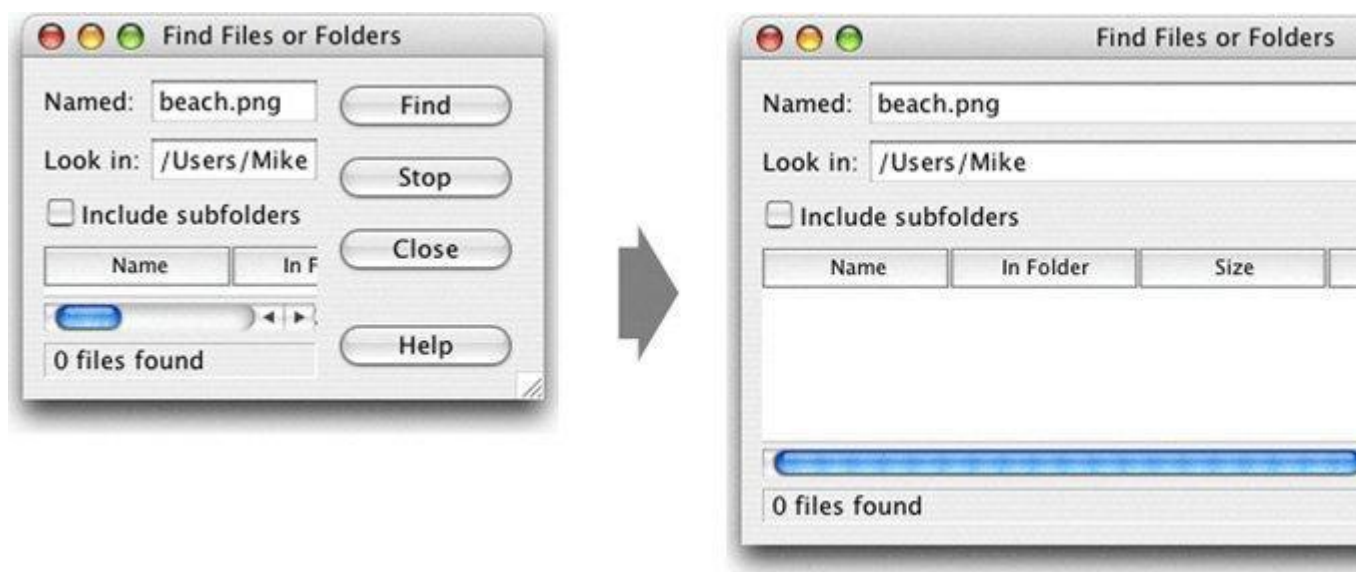
```

helpButton->setGeometry(171 + extraWidth, 149 + extraHeight, 85,
                        32);
}

```

在 `FindFileDialog` 构造函数中，设置窗体的最小尺寸为 `265×190`，初始大小为 `365×240`。在 `resizeEvent()` 中，变量 `extraWidth` 和 `extraHeight` 为控件相对最小尺寸的差值，根据差值计算子控件的大小，这个在改变窗体大小时控件能够跟着改变其大小。

Figure 6.2. Resizing a resizable dialog



绝对位置定位和手工布局管理都是需要更多的代码，也需要更多的常量参与计算。这样编写代码非常令人讨厌，如果设计改变了，所有的值都要重新计算一遍。虽然手工布局能改变空间大小，但是有时仍然会无法显示全部文字，为了避免这个错误，可以考虑控件的 `sizeHint`，但是这样的代码会更加复杂了。

管理窗体上控件最简单的方法就是使用 `Qt` 的布局管理类。这些类能够给出所有类型控件的默认值，能够根据控件的字体，样式，内容得到不同的控件的 `sizeHint`。布局管理类能够得到控件的最大，最小尺寸，在字体，内容或者窗口改变时自动调整布局。

`QHBoxLayout`，`QVBoxLayout`，`QGridLayout` 是三个最重要的布局管理器，这些类从 `QLayout` 继承，`QLayout` 提供布局最基本的框架。这三个类可以在代码中使用，也可以在 `Qt Designer` 中使用，下面是 `FindFileDialog` 使用布局管理器的代码

```
FindFileDialog::FindFileDialog(QWidget *parent)
```

```

: QDialog(parent)
{
    ...
    QGridLayout *leftLayout = new QGridLayout;
    leftLayout->addWidget(namedLabel, 0, 0);
    leftLayout->addWidget(namedLineEdit, 0, 1);
    leftLayout->addWidget(lookInLabel, 1, 0);
    leftLayout->addWidget(lookInLineEdit, 1, 1);
    leftLayout->addWidget(subfoldersCheckBox, 2, 0, 1, 2);
    leftLayout->addWidget(tableWidget, 3, 0, 1, 2);
    leftLayout->addWidget(messageLabel, 4, 0, 1, 2);
    QVBoxLayout *rightLayout = new QVBoxLayout;
    rightLayout->addWidget(findButton);
    rightLayout->addWidget(stopButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch();
    rightLayout->addWidget(helpButton);
    QHBoxLayout *mainLayout = new QHBoxLayout;
    mainLayout->addLayout(leftLayout);
    mainLayout->addLayout(rightLayout);
    setLayout(mainLayout);
    setWindowTitle(tr("Find Files or Folders"));
}

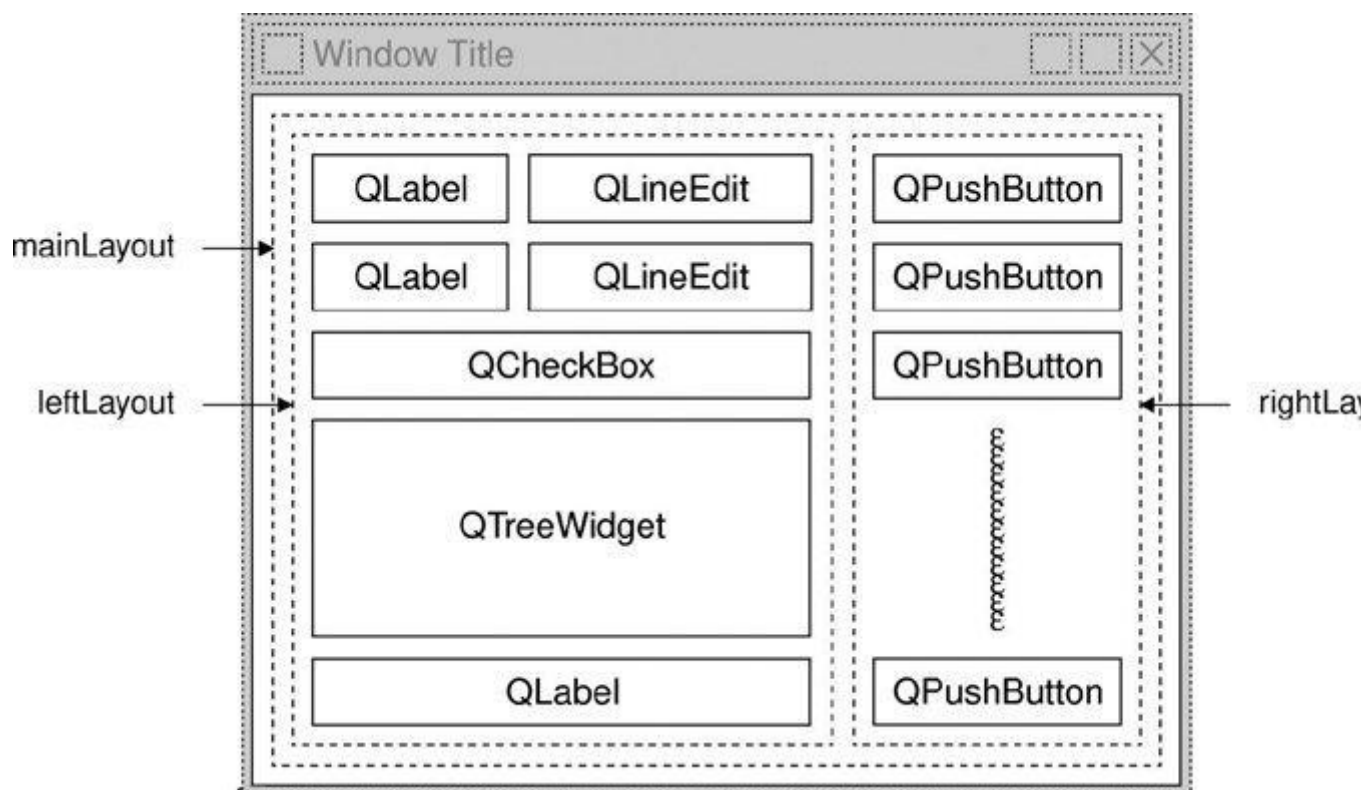
```

代码中用到了 `QHBoxLayout`、`QGridLayout` 和 `QVBoxLayout`。窗体的左边的子控件由 `QGridLayout` 负责，右边的子控件由 `QVBoxLayout` 负责。这两个布局由 `QHBoxLayout` 进行控制。对话框四周的边缘大小和控件之间的间隔设置为当前空间样式的缺省值，函数 `QLayout::setMargin()` 和 `QLayout::setSpacing()` 能够对这两个值进行修改。

这个对话框也可以使用 **Qt Designer** 实现，首先把所有的子控件放置在近似适当的位置，选择需要布局管理器一同管理的控件，点击 **Form|Layout Horizontally**，

Form|Layout Vertically 或者 Form|Layout in a Grid。在第二章我们这样创建了 Spreadsheet 程序的 Go-to-Cell 对话框和 Sort 对话框。

Figure 6.3. The Find File dialog's layout



`QHBoxLayout` 和 `QVBoxLayout` 的使用很简单, `QGridLayout` 有点复杂。

`QGridLayout` 工作的基础是一个二维的单元格。左上角的 `QLabel` 在布局中的位置为 (0, 0), 旁边的 `QLineEdit` 位置为 (0, 1)。 `QCheckBox` 占用了 (2, 0) 和 (2, 1) 两个列的空间, 下面的 `QTreeWidget` 和 `QLabel` 也是如此。 `QGridLayout::addWidget()` 语法如下:

```
layout->addWidget(widget, row, column, rowSpan, columnSpan);
```

参数 `widget` 为插入到这个布局的子控件, (row, column) 为控件占据的左上角单元格位置, `rowSpan` 是控件占据的行数, `columnSpan` 是控件占据的列的个数。 `rowSpan` 和 `columnSpan` 默认值为 1。

函数 `addStretch()` 使布局管理器在指定的位置留出一块空间。上面的代码中, 布局管理器在 `Close` 按钮和 `Help` 按钮之间留出一个额外的空隙。在 `Qt Designer` 中, 我们可以加入一个 `spacer` 实现这一功能, 在 `Qt Designer` 中, `spacer` 表现为蓝色的弹簧式折线。

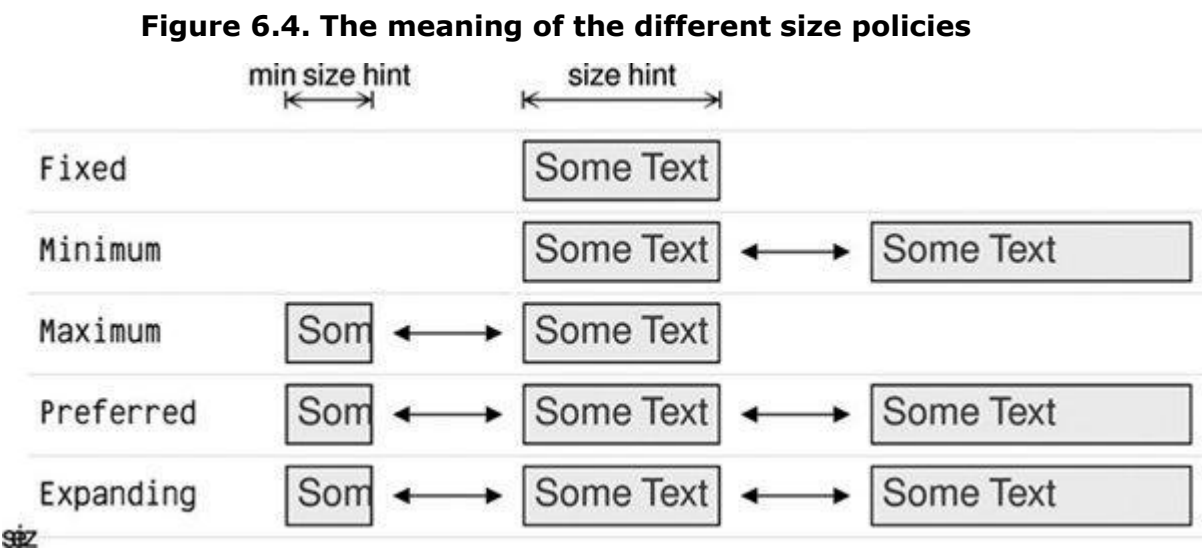
使用布局管理类还能获得其他多的功能。如果把一个控件加到一个布局中，或者从布局中删除一个控件，布局管理器会自动适应变化，调整控件大小。调用子控件的 `hide()` 或者 `show()` 函数时，布局管理器同样也会自动进行调整。如果子控件的 `sizeHint` 改变了，布局管理器就会根据控件新的 `sizeHint` 进行调整。根据所有子控件的最小尺寸和 `sizeHint`，布局管理器还会计算出整个窗体最小尺寸。

在上例中，我们只是把控件放到布局中，使用 `spacer` (`stretches`) 填满余下的空间。有时，光是这些还是不够的，我们还可以改变控件的 `sizePolicy`，或者 `sizeHint`，使窗体的布局更加符合我们的需要。

一个控件的 `sizePolicy` 说明控件在布局管理中的缩放方式。`Qt` 提供的控件都有一个合理的缺省 `sizePolicy`，但是这个缺省值有时不能适合所有的布局，开发人员经常需要改变窗体上的某些控件的 `sizePolicy`。一个 `QSizePolicy` 的所有变量对水平方向和垂直方向都适用。下面列举了一些最长用的值：

- 1. **Fixed**: 控件不能放大或者缩小，控件的大小就是它的 `sizeHint`。
- 2. **Minimum**: 控件的 `sizeHint` 为控件的最小尺寸。控件不能小于这个 `sizeHint`，但是可以放大。
- 3. **Maximum**: 控件的 `sizeHint` 为控件的最大尺寸，控件不能放大，但是可以缩小到它的最小允许尺寸。
- 4. **Preferred**: 控件的 `sizeHint` 是它的 `sizeHint`，但是可以放大或者缩小
- 5. **Expanding**: 控件可以自行增大或者缩小

图 6.4 以文本为“Some Text”的 `QLabel` 显示了这些不同的 `sizePolicy` 的含义，



在图中，Preferred 和 Expanding 的表现是一样的，二者的区别何在那？如果一个窗体中既有 Preferred 控件也有 Expanding 控件，在改变大小时，由 Expanding 控件填满其余的控件，而 Preferred 控件不变，认为它的 sizeHint。

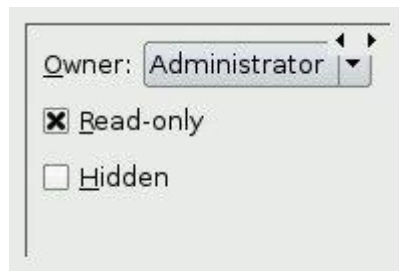
还有两个 sizePolicy 值为 MinimumExpanding 和 Ignored。MinimumExpanding 在老的 Qt 版本中有时会用到，但是现在已经不用了。替代的方法是使用 Expanding 值和重写合适的 minimumSizeHint() 函数。Ignored 和 Expanding 很像，只是它忽略控件的 sizeHint 和最小的 sizeHint。

除了水平和垂直方向的值，QSizePolicy 还包含了一个水平和垂直方向的放缩倍数（stretch factor）。当窗体放大时，这两个值决定不同控件放大的程度。例如，如果 QTreeWidget 和 QTextEdit 上下排列，如果我们希望 QTextEdit 高度为 QTreeWidget 的两倍，就可以设置 QTextEdit 的垂直放缩倍数为 2，QTreeWidget 的垂直放缩倍数为 1。控件的最小尺寸，最大尺寸和固定尺寸也是影响布局的因素。布局管理器排列控件时会考虑这些限制。如果这些还不够，可以创建新类重写 sizeHint()。

6-2 分组布局（Stacked Layouts）

QStackedLayout 类把子控件进行分组或者分页，一次只显示一组或者一页，隐藏其他组或者页上的控件。QStackedLayout 本身并不可见，对换页也不提供本质的支持。图 6.5 中的建头和黑灰色的框架是 Qt Designer 提供为了方便设计。为了方便起见，Qt 还提供了类 QStackedWidget，这个类的布局管理器为 QStackedLayout。

Figure 6.5. QStackedLayout

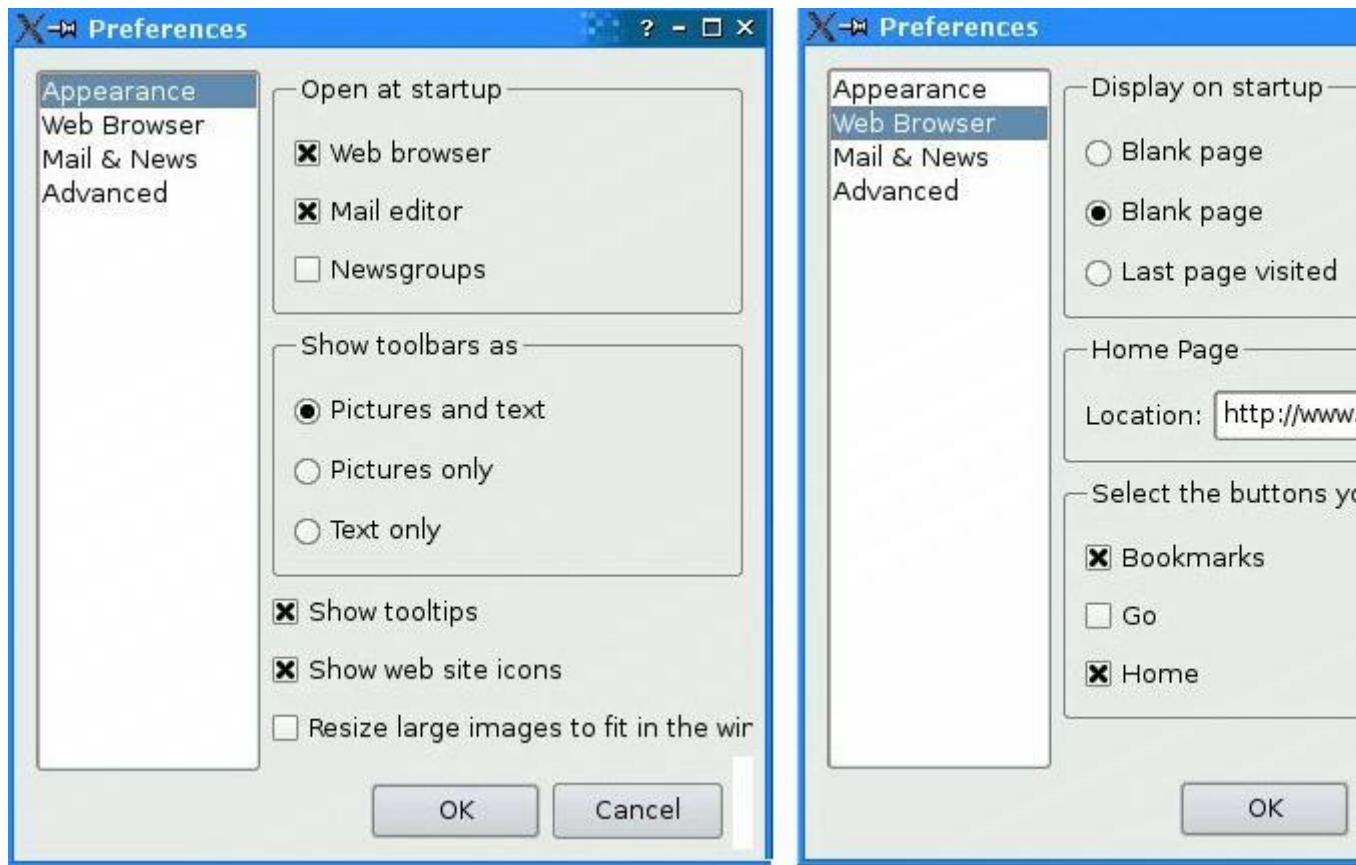


起始页为 0。为使某一个子控件可见，指定一个页号，调用 setCurrentIndex()。一个子控件的页号有函数 indexOf() 得到。

图 6-6 是一个 Preferences 对话框，以它为例说明 QStackedLayout 的用法。这个对话框的左边是一个 QListWidget 控件，右边为一个 QStackedLayout。QListWidget 中的每一项对应 QStackedLayout 中的一页。下面是这个类的构造函数的代码：

```
PreferenceDialog::PreferenceDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    listWidget = new QListWidget;
    listWidget->addItem(tr("Appearance"));
    listWidget->addItem(tr("Web Browser"));
    listWidget->addItem(tr("Mail & News"));
    listWidget->addItem(tr("Advanced"));
    stackedLayout = new QStackedLayout;
    stackedLayout->addWidget(appearancePage);
    stackedLayout->addWidget(webBrowserPage);
    stackedLayout->addWidget(mailAndNewsPage);
    stackedLayout->addWidget(advancedPage);
    connect(listWidget, SIGNAL(currentRowChanged(int)),
           stackedLayout, SLOT(setCurrentIndex(int)));
    ...
    listWidget->setCurrentRow(0);
}
```

Figure 6.6. Two pages of the Preferences dialog



我们创建一个 `QListWidget`，它的每一项为一个控件页的名字。然后我们创建一个 `QStackedLayout`，调用 `addWidget()` 把每一页的控件加入到布局中。连接 `QListWidget` 的信号 `currentRowChanged(int)` 和 `QStackedLayout` 的函数 `setCurrentIndex(int)` 连接，改变 `QListWidget` 的当前项时换页。最后调用 `QListWidget` 把开始页设置为 0。

这样的对话框使用 Qt Designer 会更加简单：

1. 用“Dialog”或者“Widget”模板创建一个窗体、
2. 在窗体上增加一个 `QListWidget` 和一个 `QStackedWidget` 控件。
3. 给每一页添加子控件，子控件按布局排列好（右击 `QStackedWidget` 控件，选择 `Insert Page` 菜单便可创建一个新页。点击右上角的左右键头，可以在页和页之间切换）。
4. 把 `QListWidget` 和 `QStackedWidget` 用水平布局管理。
5. 连接 `QListWidget` 控件的 `currentRowChanged(int)` 信号和 `QStackedWidget` 控件的槽 `setCurrentIndex(int)`。
6. 设置当前 `QListWidget` 控件的 `currentRow` 属性为 0。

在 Qt Designer 中预览时，点击列表控件中不同的项目，窗体就会换为不同的页。

6-3 分隔控件 (Splitters)

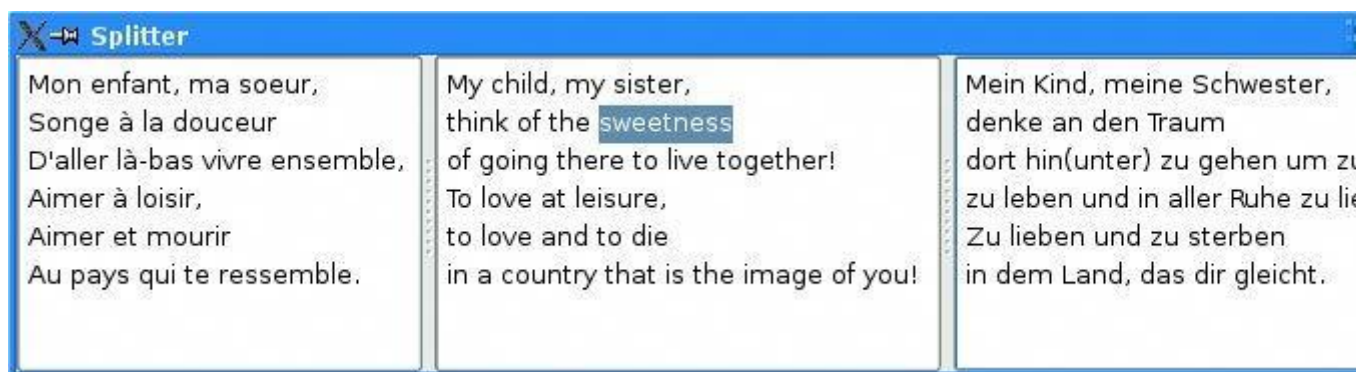
中英文对照 Splitter handles (分隔条)

一个 `QSplitter` 控件中可以包含其他控件，这些控件被一个分隔条隔开，托拽这个分隔条，里面的控件的大小可以改变。`QSplitter` 控件经常做为布局管理器使用，给用户提供更多的界面控制。

`QSplitter` 控件中的子控件总是按顺序自动并肩排列（或者上下排列）。相邻的控件之间有一个分隔条。下面是创建图 6.7 的窗体的代码：

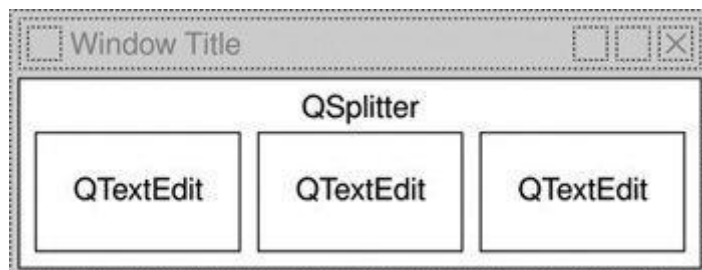
```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextEdit *editor1 = new QTextEdit;
    QTextEdit *editor2 = new QTextEdit;
    QTextEdit *editor3 = new QTextEdit;
    QSplitter splitter(Qt::Horizontal);
    splitter.addWidget(editor1);
    splitter.addWidget(editor2);
    splitter.addWidget(editor3);
    ...
    splitter.show();
    return app.exec();
}
```

Figure 6.7. The Splitter application



这个例子中的窗体有一个 `QSplitter` 控件，其中有三个水平排列的 `QTextEdit` 控件，和布局管理器不同，`QSplitter` 不但可以排列子控件，还有一个可视的外观，`QSplitter` 控件从 `QWidget` 继承，拥有 `QWidget` 所有的功能。

Figure 6.8. The Splitter application's widgets



第一次是调用 `rightSplitter` 的 `setStretchFactor`，把索引值为 1 的控件（`textEdit`）的拉伸系数设置为 1，第二次是调用 `mainSplitter` 的 `setStretchFactor()`，设置控件 `rightSplitter` 的拉伸系数为 1。这样，`textEdit` 控件就能够得到尽可能多余的空间。当应用程序启动时，`QSplitter` 根据子控件的初始尺寸或者 `sizeHint` 合理分配每一个子控件的大小。程序中，我们可以调用 `QSplitter::setSizes()` 改变分隔条的位置。`QSplitter` 还可以保存当前的状态，在程序下一次运行时恢复以前的状态。下面是 `writeSettings()` 函数，保存当前分隔条的状态：

```
void MailClient::writeSettings()
{
    QSettings settings("Software Inc.", "Mail Client");
    settings.beginGroup("mainWindow");
    settings.setValue("size", size());
    settings.setValue("mainSplitter", mainSplitter->saveState());
    settings.setValue("rightSplitter", rightSplitter->saveState());
    settings.endGroup();
}
```

下面是相应的 `readSettings()` 函数：

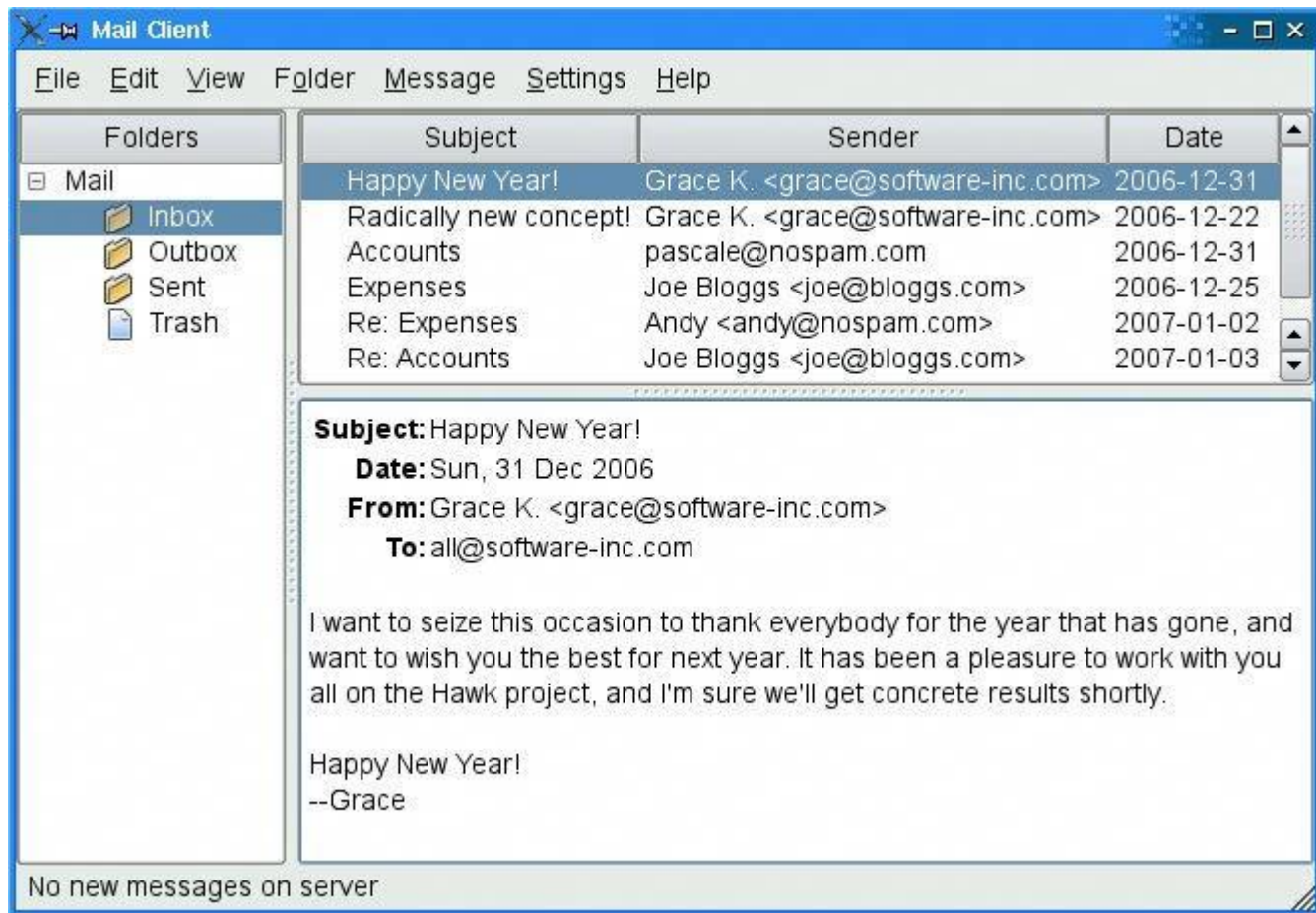
```
void MailClient::readSettings()
{
    QSettings settings("Software Inc.", "Mail Client");
    settings.beginGroup("mainWindow");
```

```
resize(settings.value("size", QSize(480, 360)).toSize());
mainSplitter->restoreState(
    settings.value("mainSplitter").toByteArray());
rightSplitter->restoreState(
    settings.value("rightSplitter").toByteArray());
settings.endGroup();
}
```

Qt Designer 也支持 QSplitter。把子控件放到合适的位置，把他们选中，选择菜单 Form|Lay out Horizontally in Splitter 或者 Form|Lay out Verticallly in Splitter，所选择的子控件就被加入到 QSplitter 控件中。

对 QSplitter 进行水平或者垂直嵌套可以实现更加复杂的布局。例如，图 6-9 所示的 MailClient 程序中，就是一个垂直方向的 QSplitter 控件中嵌套了一个水平方向的 QSplitter 控件。

Figure 6.9. The Mail Client application on Mac OS X



下面的代码是 **MailClient** 程序的主窗口类构造函数代码:

```
MailClient::MailClient()
```

```
{
```

```
...
```

```
    rightSplitter = new QSplitter(Qt::Vertical);
```

```
    rightSplitter->addWidget(messagesTreeWidget);
```

```
    rightSplitter->addWidget(textEdit);
```

```
    rightSplitter->setStretchFactor(1, 1);
```

```
    mainSplitter = new QSplitter(Qt::Horizontal);
```

```
    mainSplitter->addWidget(foldersTreeWidget);
```

```
    mainSplitter->addWidget(rightSplitter);
```

```
    mainSplitter->setStretchFactor(1, 1);
```



```

setCentralWidget(mainSplitter);
setWindowTitle(tr("Mail Client"));
readSettings();
}

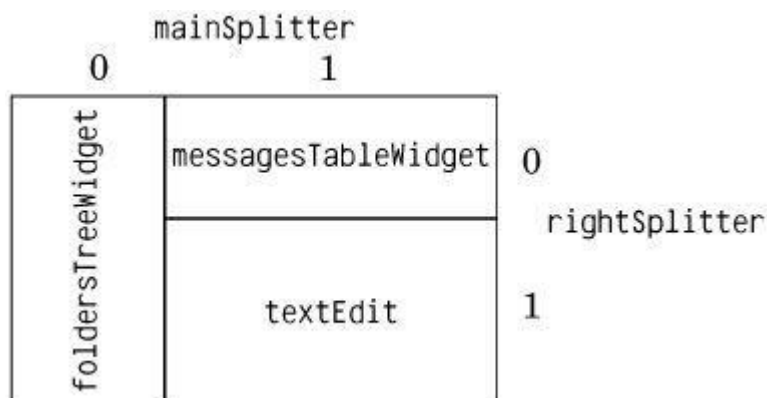
```

创建完我们需要显示三个控件（messageTreeWidget, textEdit, foldersTreeWidget）后，我们创建一个垂直的 QSplitter，rightSplitter 控件，把 messageTreeWidget 和 textEdit 控件加到 rightSplitter 中。然后创建一个水平的 QSplitter，mainSplitter 控件，把 rightSplitter 和 foldersTreeWidget 加入到 mainSplitter 中。把 mainSplitter 做为 QMainWindow 的中央控件。

当用户改变窗口的大小时，QSplitter 通常给所有的子控件一样的空间。在 MailClient 程序中，我们希望左边的文件树控件（foldersTreeWidget）和消息树控件

（messageTreeWidget）保持它们的大小，把其他的空间都分配给 QTextEdit。这由两个 setStretchFactor()调用实现。第一个参数是 0 开始的子控件的索引值，第二个参数为我们设置的拉伸系数，缺省值为 0。

Figure 6.10. The Mail Client's splitter indexing

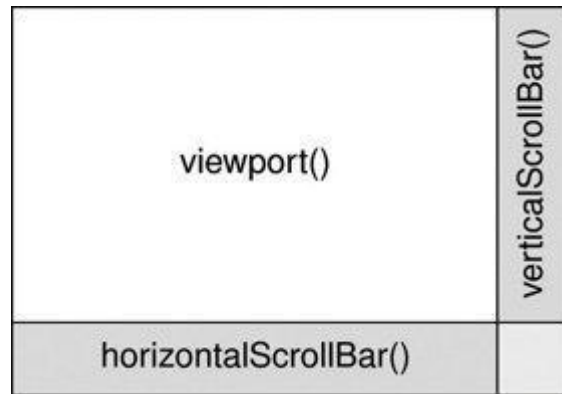


6-4 滚动区域（Scrolling Areas）

英汉对照：viewport（视图）

QScrollArea 类提供了一个可以滚动的可视窗口和两个滚动条。如果我们想给一个控件加上一个滚动条，从 QScrollArea 继承会比设计我们自己的 QScrollBar 类实现滚动函数更简单。

Figure 6.11. QScrollArea's constituent widgets



调用 `QScrollArea` 的 `setWidget()` 函数就能给控件加上滚动条。`QScrollArea` 自动把控件设置为视图（`viewport`，`QScrollArea::viewport()` 得到）的一个子控件。例如，如果我们想给第五章的 `IconEditor` 加上滚动条，我们可以这样写代码：

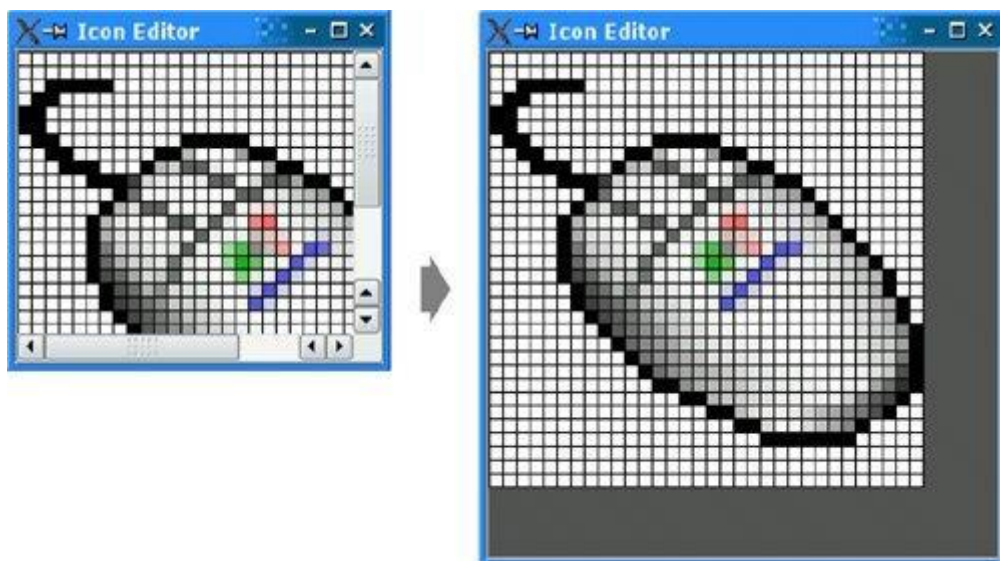
```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    IconEditor *iconEditor = new IconEditor;
    iconEditor->setIconImage(QImage(":/images/mouse.png"));
    QScrollArea scrollArea;
    scrollArea.setWidget(iconEditor);
    scrollArea.viewport()->setBackgroundRole(QPalette::Dark);
    scrollArea.viewport()->setAutoFillBackground(true);
    scrollArea.setWindowTitle(QObject::tr("Icon Editor"));
    scrollArea.show();
    return app.exec();
}
```

在 `QScrollArea` 控件上显示控件的当前尺寸或者使用控件的 `sizeHint`。调用 `setWidgetResizable(true)`，`QScrollArea` 自动改变控件的大小。

当视图小于控件大小时，滚动条会自动出现。也可以设置滚动条一直显示：

```
scrollArea.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
scrollArea.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
```

Figure 6.12. Resizing a QScrollArea



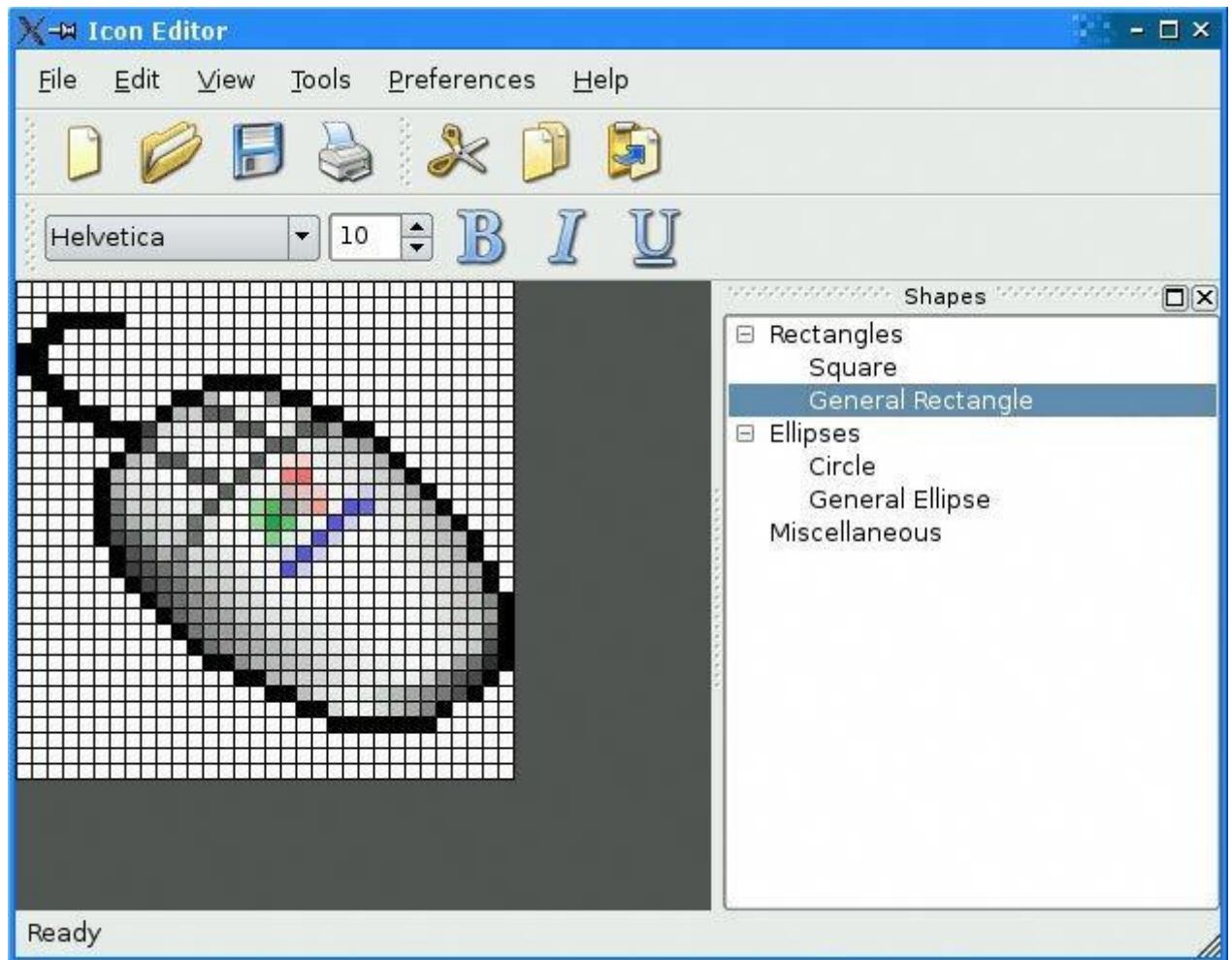
QScrollArea 继承了 QAbstractScrollArea 所有功能。QTextEdit 和 QAbstractItemView 也是继承自 QAbstractScrollArea，这些类不需要使用 QScrollArea。

6-5 可停靠控件和工具栏（Dock Widgets and Toolbars）

可停靠控件能够停靠在 QMainWindow 中或者作为一个独立窗口浮动。

QMainWindow 提供了四个可停靠控件的地方：上方，下方，左方，右方。Microsoft Visual Studio 程序和 Qt Linguist 程序使用大量的可停靠窗口实现更为复杂的用户界面。在 Qt 中，可停靠窗口是 QDockWidget 的一个实例。

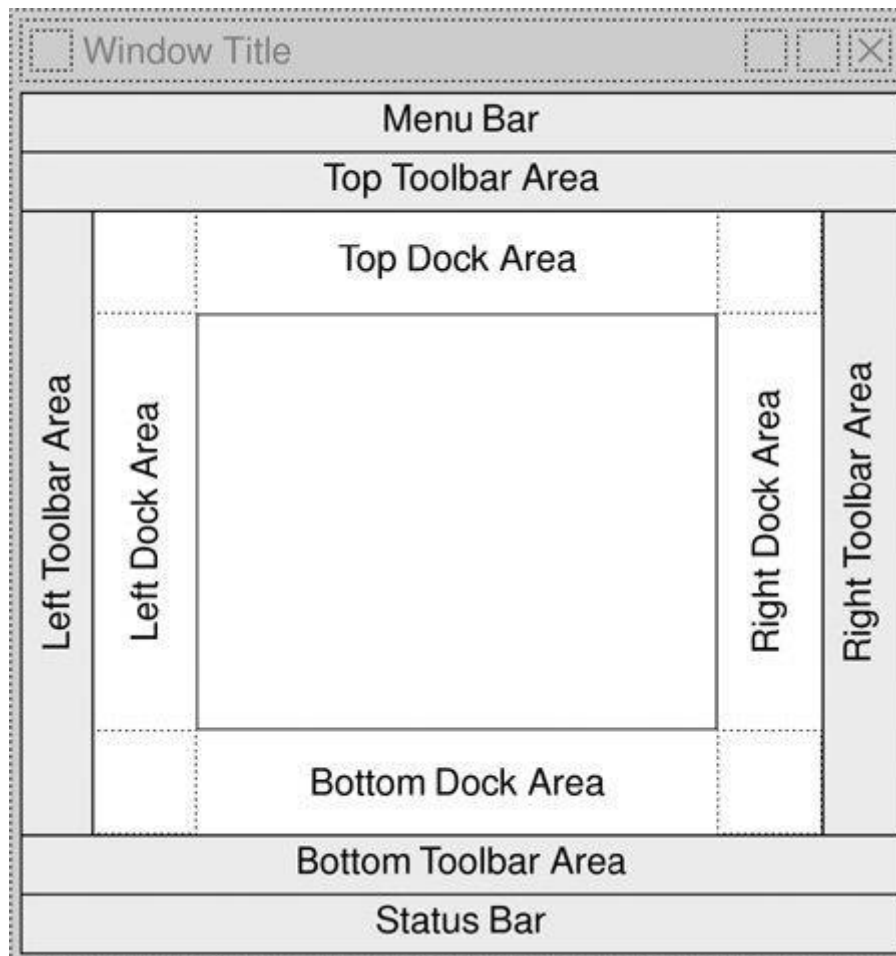
Figure 6.13. A QMainWindow with a dock widget



每一个可停靠控件都有一个标题条。用户可以拖动标题栏把窗口从一个可停靠区域移动到另一个可停靠区域。拖动标题栏把窗口移动到不能停靠的区域，窗口就浮动为一个独立的窗口。自由浮动的窗口总是在主窗口的上面。用户点击标题栏上的关闭按钮可以关闭 QDockWidget。调用 `QDockWidget::setFeatures()` 能够设置以上这些属性。

在 Qt 的早期版本中，工具条也是作为可停靠控件，可以放置在任何可停靠区域中。从 Qt4 开始，工具条有了自己的位置，不能再浮动了，如果需要一个可停靠的工具条，我们可以把它放到 QDockWindow 里面。

Figure 6.14. QMainWindow's dock and toolbar areas



用点线表示的四个角落可以属于任何一个相邻的可停靠区域。例如，调用 `QMainWindow::setCorner(Qt::TopLeftCorner, Qt::LeftDockWidgetArea)` 把左上角作为左侧的停靠区域。

下面的代码将一个现有的控件（如 `QTreeWidget`）放置到 `QDockWidget` 中，停靠再右边的停靠区域。

```
QDockWidget *shapesDockWidget = new QDockWidget(tr("Shapes"));
shapesDockWidget->setWidget(treeWidget);
shapesDockWidget->setAllowedAreas(Qt::LeftDockWidgetArea
    | Qt::RightDockWidgetArea);
addDockWidget(Qt::RightDockWidgetArea, shapesDockWidget);
```

函数 `setAllowAreas()` 确定控件可停靠的位置。在上面的代码允许用户把窗口拖动到左边或者右边的可停靠区域，这两个地方垂直控件足够显示一个树型控件。如果没有指定停靠区域，用户可以拖动控件到任何四个可停靠的区域。

下面的代码创建一个工具栏，包含一个 `QComboBox`，`QSpinBox` 和一些 `QToolButton`，代码为 `QMainWindow` 子类的构造函数的一部分：

```
QToolBar *fontToolBar = new QToolBar(tr("Font"));
fontToolBar->addWidget(familyComboBox);
fontToolBar->addWidget(sizeSpinBox);
fontToolBar->addAction(boldAction);
fontToolBar->addAction(italicAction);
fontToolBar->addAction(underlineAction);
fontToolBar->setAllowedAreas(Qt::TopToolBarArea
                           | Qt::BottomToolBarArea);
addToolBar(fontToolBar);
```

如果我们需要在下一次运行程序时恢复所有可停靠控件和工具栏的位置，可以使用和保存 `QSplitter` 状态相似的代码：

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");
    settings.beginGroup("mainWindow");
    settings.setValue("size", size());
    settings.setValue("state", saveState());
    settings.endGroup();
}

void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");
```

```
settings.beginGroup("mainWindow");  
resize(settings.value("size").toSize());  
restoreState(settings.value("state").toByteArray());  
settings.endGroup();  
}
```

`QMainWindow` 提供了一个上下文菜单，列出了所有可停靠的窗口和工具栏。用户可以通过这个菜单关闭显示停靠窗口和工具栏。

Figure 6.15. A QMainWindow context menu



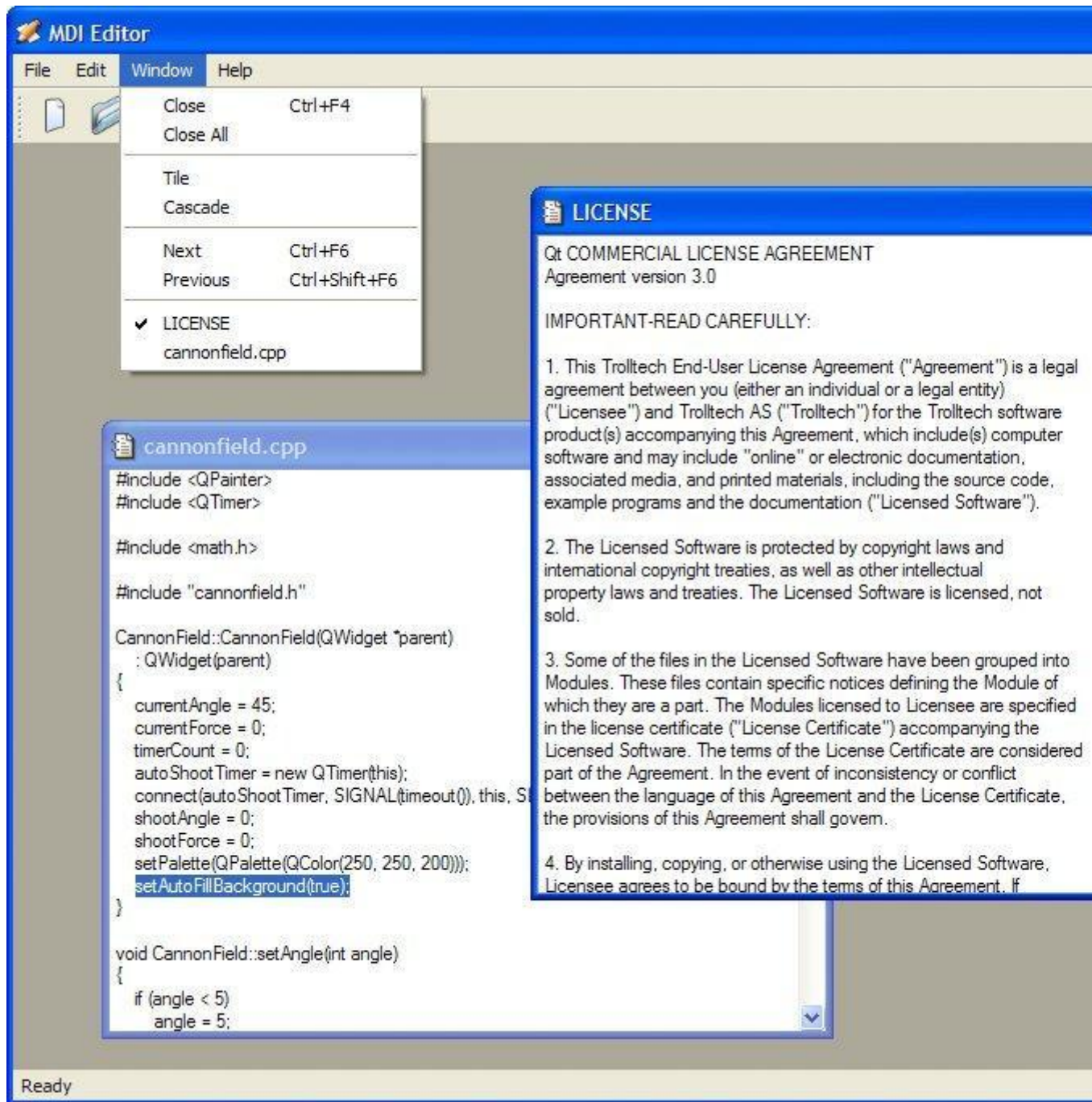
6-6 多文档界面（Multiple Document Interface）

一个主窗口区域内能够提供多个文档的程序称之为多文档程序，或者 MDI 程序。在 Qt 中，一个 MDI 程序是由 `QWorkspace` 类实现的，把 `QWorkspace` 做为中央控件，每一个文档窗口做为 `QWorkspace` 的子控件。

MDI 程序的惯例是提供一个 `window` 菜单，管理窗口的显示方式和当前打开的窗口列表。正在活动的窗口由选中记号标示。用户可以点击 `window` 菜单中窗口列表中的一个窗口把它激活。

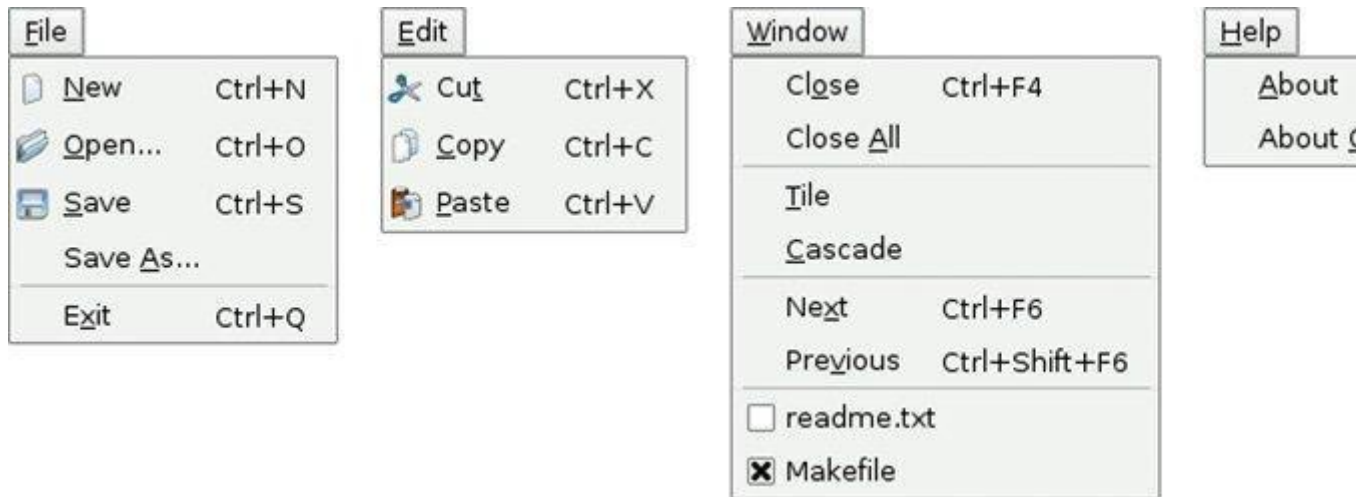
在这一节中，我们实现一个图 6.16 所示的 MDI 编辑程序，介绍如何创建 MDI 程序，如何实现 `window` 菜单。

Figure 6.16. The MDI Editor application



这个应用程序包含两个类：MainWindow 和 Editor 类。程序中大部分代码和第一部分的 Spreadsheet 程序相似，这里我们只介绍新增的代码。

Figure 6.17. The MDI Editor application's menus



首先看一下 MainWindow 类:

```
MainWindow::MainWindow()
{
    workspace = new QWorkspace;
    setCentralWidget(workspace);
    connect(workspace, SIGNAL(windowActivated(QWidget *)),
            this, SLOT(updateMenus()));
    createActions();
    createMenus();
    createToolBars();
    createStatusBar();
    setWindowTitle(tr("MDI Editor"));
    setWindowIcon(QPixmap(":/images/icon.png"));
}
```

在 MainWindow 的构造函数中，我们创建了一个 QWorkspace 控件，并把这个控件做为一个中央控件。连接 QWorkspace 的 windowActivated() 信号和 updateMenus() 函数，对 window 菜单进行更新。

```
void MainWindow::newFile()
{
    Editor *editor = createEditor();
    editor->newFile();
}
```

```

    editor->show();
}

```

函数 `newFile()` 用来相应 `File|New` 菜单, 调用 `createEditor` 创建一个子控件 `Editor`。

```

Editor *MainWindow::createEditor()
{
    Editor *editor = new Editor;
    connect(editor, SIGNAL(copyAvailable(bool)),
            cutAction, SLOT(setEnabled(bool)));
    connect(editor, SIGNAL(copyAvailable(bool)),
            copyAction, SLOT(setEnabled(bool)));
    workspace->addWindow(editor);
    windowMenu->addAction(editor->windowMenuAction());
    windowActionGroup->addAction(editor->windowMenuAction());
    return editor;
}

```

函数 `createEditor()` 创建一个 `Editor` 控件, 连接两个信号和槽, 如果由选中的文本, `Edit|Cut` 菜单和 `Edit|Copy` 菜单能够改变状态。

因为是 MDI 程序, 主窗口中可能有多个 `Editor` 控件。问题是当前活动的窗口发出的 `copyAvailable(bool)` 信号才能改变菜单的状态。实际上也只有当前活动的窗口能够发出信号, 所以这个问题也不用考虑。

一旦新加了一个 `Editor` 控件, 我们在 `Window` 菜单中增加一个 `QAction` 激活这个窗口。这个 `QAction` 是 `Editor` 类提供的, 稍后会介绍。我们增加了一个 `QActionGroup` 对象, 这样窗口菜单中只有一项是选中的, 即只有一个窗口是激活的。

```

void MainWindow::open()
{
    Editor *editor = createEditor();
    if (editor->open()) {
        editor->show();
    } else {
        editor->close();
    }
}

```

```

    }
}

```

函数 `open()` 相应菜单 `File|Open`。创建一个 `Editor` 调用 `open()` 函数。`Open()` 函数由 `Editor` 类实现，这样 `MainWindow` 类就不用维护 `Editor` 类的状态。

如果 `open()` 失败，关闭 `Editor`，错误的原因由 `Editor` 类告诉用户。我们也没有显式的删除 `Editor` 对象，在 `Editor` 的构造函数中，设置了 `Qt::WA_DeleteOn_Close` 属性，在关闭的同时 `Editor` 会自动删除自己。

```

void MainWindow::save()
{
    if (activeEditor())
        activeEditor()->save();
}

```

函数 `save()` 调用当前活动的 `Editor::save()`。具体的保存操作也是在 `Editor` 中实现。

```

Editor *MainWindow::activeEditor()
{
    return qobject_cast<Editor *>(workspace->activeWindow());
}

```

函数 `activeEditor()` 返回当前活动的 `Editor` 类型的子窗口指针，如果没有活动窗口，则返回一个空指针。

```

void MainWindow::cut()
{
    if (activeEditor())
        activeEditor()->cut();
}

```

函数 `cut()` 调用当前 `Editor::cut()`，`copy()`，`paste()` 函数和 `cut()` 函数相同，在此略去不谈。

```

void MainWindow::updateMenus()
{
    bool hasEditor = (activeEditor() != 0);
    bool hasSelection = activeEditor()

```

```

        && activeEditor()->textCursor().hasSelection());

saveAction->setEnabled(hasEditor);
saveAsAction->setEnabled(hasEditor);
pasteAction->setEnabled(hasEditor);
cutAction->setEnabled(hasSelection);
copyAction->setEnabled(hasSelection);
closeAction->setEnabled(hasEditor);
closeAllAction->setEnabled(hasEditor);
tileAction->setEnabled(hasEditor);
cascadeAction->setEnabled(hasEditor);
nextAction->setEnabled(hasEditor);
previousAction->setEnabled(hasEditor);
separatorAction->setVisible(hasEditor);
if (activeEditor())
    activeEditor()->>windowMenuAction()->setChecked(true);
}

```

当激活一个窗口或者关闭最后一个窗口时，调用 `updateMenus()` 更新菜单，`updateMenus()` 是槽函数，在 `MainWindow` 的构造函数调用了这个函数，使程序启动时也能更新菜单。

只要有一个活动窗口，大部分菜单都是有意义的，如果没有活动窗口，这些菜单都被禁止。最后，调用 `QAction::setChecked()` 标示活动窗口。由于使用了 `QActionGroup`，以前标示的活动窗口自动取消。

```

void MainWindow::createMenus()
{
    ...

    windowMenu = menuBar()->addMenu(tr("&Window"));
    windowMenu->addAction(closeAction);
    windowMenu->addAction(closeAllAction);
    windowMenu->addSeparator();
    windowMenu->addAction(tileAction);
}

```

```

windowMenu->addAction(cascadeAction);
windowMenu->addSeparator();
windowMenu->addAction(nextAction);
windowMenu->addAction(previousAction);
windowMenu->addAction(separatorAction);
...
}

```

列出的这部分的 `createMenu()` 这段代码实现了 `window` 菜单。这些 `QAction` 能够很容易通过 `QWorkspace` 的成员函数实现, 如, `closeActiveWindow()`, `closeAllWindows()`, `tile()`, `cascade()`, 只要打开一个新的子窗口, 就在 `window` 菜单中加一个 `Action`。当用户关闭一个窗口时, 相应的 `window` 菜单项就会删除, 这个 `Action` 会自动从 `Window` 菜单中删除。

```

void MainWindow::closeEvent(QCloseEvent *event)
{
    workspace->closeAllWindows();
    if (activeEditor()) {
        event->ignore();
    } else {
        event->accept();
    }
}

```

虚函数 `closeEvent()` 给每一个子窗口发送关闭事件, 关闭子窗口。如果还有一个子窗口, 这很可能是因为用户在“`unsaved changes`”消息对话框中选择了 `cancel` 按钮, 因此忽略这个事件。如果没有活动窗口, `Qt` 关闭所有的窗口。如果我们不在 `MainWindow` 类中重写 `closeEvent()`, 用户就没有机会保存文档的改变。

以上是 `MainWindow` 部分的代码。`Editor` 类代表的是一个子窗口。它继承自 `QTextEdit`, 基类中提供了文本编辑函数。`Qt` 中的所有控件都可以做为一个独立的窗口, 因此也能做为 `MDI` 中的一个子窗口。

类定义如下:

```

class Editor : public QTextEdit

```

```

{
    Q_OBJECT
public:
    Editor(QWidget *parent = 0);
    void newFile();
    bool open();
    bool openFile(const QString &fileName);
    bool save();
    bool saveAs();
    QSize sizeHint() const;
    QAction *windowMenuAction() const { return action; }
protected:
    void closeEvent(QCloseEvent *event);
private slots:
    void documentWasModified();
private:
    bool okToContinue();
    bool saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    QString strippedName(const QString &fullFileName);
    QString curFile;
    bool isUntitled;
    QString fileFilters;
    QAction *action;
};

```

在 **Spreadsheet** 程序中的四个私有函数，也同样出现在 **Editor** 类中：
okToContinue(), **saveFile()**, **setCurrentFile()**, **strippedName()**。
Editor::Editor(QWidget *parent)

```

: QTextEdit(parent)
{
    action = new QAction(this);
    action->setCheckable(true);
    connect(action, SIGNAL(triggered()), this, SLOT(show()));
    connect(action, SIGNAL(triggered()), this, SLOT(setFocus()));
    isUntitled = true;
    fileFilters = tr("Text files (*.txt)\n"
                     "All files (*)");
    connect(document(), SIGNAL(contentsChanged()),
            this, SLOT(documentWasModified()));
    setWindowIcon(QPixmap(":/images/document.png"));
    setAttribute(Qt::WA_DeleteOnClose);
}

```

在构造函数中，我们首先创建一个 `QAction`，把它添加大 `Window` 菜单中，并把这个 `QAction` 发出的消息 `triggered()`和窗口的 `show()`，`setFocus()`连接起来。

这个 `MDI` 程序允许用户创建任意数量的 `Editor` 窗口，因此我们必须在新建时给文档一个默认的名字，这样在保存时才能把不同的文档区分开。通常的做法是用一个包含一个数字的名字，例如，`document1.txt`。变量 `isUntitled` 区分文档的名字是用户输入的还是程序自动生成的。

我们连接文档的 `contentsChanged()`信号和 `documentWasModified()`函数，这个函数只是调用 `setWindowModified(true)`。

最后设置属性 `Qt::WA_DeleteOnClose`，在关闭 `Editor` 窗口时自动删除它，避免内存泄漏。

```

void Editor::newFile()
{
    static int documentNumber = 1;
    curFile = tr("document%1.txt").arg(documentNumber);
    setWindowTitle(curFile + "[*]");
    action->setText(curFile);
}

```

```

    isUntitled = true;
    ++documentNumber;
}

```

在 `newFile()` 函数中给新建的文档一个类似 `document1.txt` 的名字。这段代码放在了 `newFile()` 中而不是在构造函数中，是因为 `documentNumber` 是一个静态的变量，在所有的 `Editor` 类型的对象中只有一个实例，我们不想在 `open()` 函数中也调用这段代码，增加 `documentNumber` 的值。

主窗口标题中的[*]是一个位置标识号。在非 Mac OS X 平台上表示文档有需要保存。这个标识号在第三章也出现过。

```

bool Editor::open()
{
    QString fileName =
        QFileDialog::getOpenFileName(this, tr("Open"), ".",
                                     fileFilters);

    if (fileName.isEmpty())
        return false;
    return openFile(fileName);
}

```

函数 `open()` 打开一个已经存在的文件。

```

bool Editor::save()
{
    if (isUntitled) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

```

如果 `isUntitled` 为 `true`，则调用函数 `saveAs()` 让用户给文档输入一个名字，如果 `isUntitled` 为 `false`，调用 `saveFile()` 函数。

```

void Editor::closeEvent(QCloseEvent *event)

```



```

{
    if (okToContinue()) {
        event->accept();
    } else {
        event->ignore();
    }
}
}

```

`closeEvent()`函数是重写实现的，允许用户保存文档的改变。用户是否保存在 `okToContinue()`函数中实现，弹出对话框“Do you want to save your changes?“,如果 `okToContinue()`返回为 `true`，接受这个关闭事件，否则，忽略这个事件，不关闭窗口。

```

void Editor::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    isUntitled = false;
    action->setText(strippedName(curFile));

    document()->setModified(false);
    setWindowTitle(strippedName(curFile) + "[*]");
    setWindowModified(false);
}

```

函数 `setCurrentFile()`在 `openFile()`和 `saveFile()`中调用。改变 `curFile` 和 `isUntitled` 变量的值，设置窗口标题和子窗口对应的 `QAction` 的名称，设置 `document()->setModified(false)`。如果用户改变了文档中的文本，`QTextDocument` 会发出 `contentsChanged()`信号，把“modified”值为 `true`。

```

QSize Editor::sizeHint() const
{
    return QSize(72 * fontMetrics().width('x'),

```

```

        25 * fontMetrics().lineSpacing());
    }

```

用字母“X”的宽度和一行字符的高度做参考，`sizeHint()`函数返回一个尺寸，`QWorkspace` 用这个尺寸给窗口一个初始值。

一下是这个函数的 `main.cpp` 文件：

```

#include <QApplication>
#include "mainwindow.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringList args = app.arguments();
    MainWindow mainWin;
    if (args.count() > 1) {
        for (int i = 1; i < args.count(); ++i)
            mainWin.openFile(args[i]);
    } else {
        mainWin.newFile();
    }
    mainWin.show();
    return app.exec();
}

```

如果用户在命令行上指定了一个文件，则打开这个文件，否则新建一个空文档。`Qt` 指定的选项如 `-style` 和 `-font` 自动由 `QApplication` 的构造函数从参数列表中删除，因此如果在命令行上这样写：

```
mdieditor -style motif readme.txt
```

`QApplication::arguments()` 返回 `QStringList` 中有两个字符串（`mdieditor` 和 `readme.txt`），程序打开文档 `readme.txt`。

MDI 是同时处理多个文档的一种方法。在 **Mac OS X** 上，较好的方法是使用多个顶层的窗口，在第三章“多文档”中有介绍。

第七章（序）事件处理-（Event Processingn）

事件是视窗系统或者 Qt 本身在各种不同的情况下产生的。当用户点击或者释放鼠标，键盘时，一个鼠标事件或者键盘事件就产生了。当窗口第一次显示时，一个绘制事件会产生告诉新可见的窗口绘制自己。很多事件是为了相应用户动作产生的，也有一些事件是由系统独立产生的。

在用 Qt 编程时，我们很少要考虑事件，当一些事件发生时，Qt 控件会发出相应的信号。只有当实现用户控件或者需要修改现有控件的行为时，我们才需要考虑事件。事件不能和信号混淆。一般来讲，在使用控件时需要处理的是信号，在实现一个控件时需要处理事件。例如，我们使用 QPushButton 时，我们只要 clicked() 信号就可以了，而不用管鼠标点击事件。但是如果我们实现一个像 QPushButton 这样的类，我们就需要处理鼠标或者键盘事件，发出 clicked() 信号。

7-1 重写事件处理函数（Reimplementing Event Handlers）

在 Qt 中，一个事件是 QEvent 的子类的对象。Qt 能够处理上百种类型的事件，每一类型的事件由一个枚举值确定。例如，对鼠标点击事件，QEvent::type() 返回的值为 QEvent::MouseButtonPress。

很多情况下，一个 QEvent 对象不能保存有关事件的所有信息，例如，鼠标点击事件需要保存是左键还是右键触发了这个信息，还要知道事件发生时鼠标指针的位置，这些额外的信息储存在 QEvent 的子类 QMouseEvent 中。

Qt 的对象通过 QObject::event() 得到有关事件的信息。QWidget::event() 提供了很多普通类型的信息，实现了很多事件处理函数，例如 mousePressEvent(), keyPressedEvent(), paintEvent() 等等。

在前面的章节中，我们已经在 MainWindow 类，IconEditor 类，Plotter 类中看到了很多事件处理函数，在 QEvent 参考文档中，还列举了很多类型的事件。我们还可以定义自己的事件，把事件分派出去。这里，我们讨论一下两种最常用的事件：键盘事件和时间事件。

重写函数 keyPressedEvent() 和 keyReleaseEvent() 可以处理键盘事件。Plotter 控件就重写了 keyPressedEvent() 函数。通常，我们只需要重写 keyPressedEvent()，需要处理键盘释放事件的只有修改键（Ctrl, Shift, Alt），而这些键的信息可以通过

`QKeyEvent::modifiers()`得到。例如，如果我们重写了控件 `CodeEditor` 控件的 `KeyPressEvent()`函数，区分 `Home` 键和 `Ctrl+Home` 键：

```
void CodeEditor::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
    case Qt::Key_Home:
        if (event->modifiers() & Qt::ControlModifier) {
            goToBeginningOfDocument();
        } else {
            goToBeginningOfLine();
        }
        break;
    case Qt::Key_End:
        ...
    default:
        QWidget::keyPressEvent(event);
    }
}
```

`Tab` 键和 `Backtab(Shift+Tab)`键很特殊，它们是在控件调用 `keyPressEvent()`之前，由 `QWidget::event()`处理的，这两个键的作用是把输入焦点转到前一控件或者下一个控件上，在 `CodeEditor` 中，希望 `Tab` 键的作用是缩进，可以这样重写 `event()`：

```
bool CodeEditor::event(QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if (keyEvent->key() == Qt::Key_Tab) {
            insertAtCurrentPosition('\t');
            return true;
        }
    }
}
```

```

    return QWidget::event(event);
}

```

如果这个事件是一个键盘敲击事件，我们把 `QEvent` 对象转换成 `QKeyEvent`，然后确定是那个键敲击了，如果是 `Tab` 键，进行处理后返回 `true`，通知 `Qt` 我们已经对事件进行了处理。如果返回 `false`，`Qt` 还会把这个事件交给基类控件处理。

响应键盘事件的更好的方法是使用 `QAction`。例如，`goToBeginningOfLine()`和 `goToBeginningOfDocument()`是 `CodeEditor` 的两个公有槽函数，`CodeEditor` 是 `MainWindow` 的中央控件，下面的代码实现了键盘和槽函数的绑定：

```

MainWindow::MainWindow()
{
    editor = new CodeEditor;
    setCentralWidget(editor);
    goToBeginningOfLineAction =
        new QAction(tr("Go to Beginning of Line"), this);
    goToBeginningOfLineAction->setShortcut(tr("Home"));
    connect(goToBeginningOfLineAction, SIGNAL(activated()),
        editor, SLOT(goToBeginningOfLine()));
    goToBeginningOfDocumentAction =
        new QAction(tr("Go to Beginning of Document"), this);
    goToBeginningOfDocumentAction->setShortcut(tr("Ctrl+Home"));
    connect(goToBeginningOfDocumentAction, SIGNAL(activated()),
        editor, SLOT(goToBeginningOfDocument()));
    ...
}

```

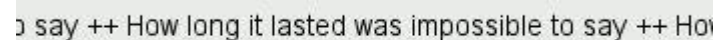
这样可以很容易把一个键盘敲击的命令加入到菜单或者工具条中。如果命令没有出现在用户界面中，可用 `QShortcut` 对象代替 `QAction` 对象，在 `QAction` 内部就是使用这个类实现键盘的绑定。

通常情况下，只要窗口中有激活的控件，控件上用 `QAction` 和 `QShortcut` 设置的键盘绑定都是可用的。绑定的键可用 `QAction::setShortcutContext()` 或者 `QShortcut::setContext()` 进行修改。

另一个常用的事件类型是时间事件。其他事件都是由用户的某种活动引发的，而时间事件则使程序按照一定的时间间隔执行特定的任务。时间事件一般用来使光标闪烁，或者播放动画，或者只是绘制显示界面或者控件。

为了介绍时间事件，我们将实现一个 `Ticker` 控件。这个控件显示一条标语，每隔 30 毫秒向左移动一个像素。如果控件比标语要宽，标语的文本重复的显示在控件上，填满整个控件。

Figure 7.1. The Ticker widget



say ++ How long it lasted was impossible to say ++ Ho

头文件如下：

```
#ifndef TICKER_H
#define TICKER_H
#include <QWidget>
class Ticker : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)
public:
    Ticker(QWidget *parent = 0);
    void setText(const QString &newText);
    QString text() const { return myText; }
    QSize sizeHint() const;
protected:
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);
```

```

    void showEvent(QShowEvent *event);
    void hideEvent(QHideEvent *event);
private:
    QString myText;
    int offset;
    int myTimerId;
};
#endif

```

在头文件中，我们实现了 **Ticker** 的四个事件处理函数，其中三个 **timeEvent()**，**showEvent()**和 **hideEvent()**是我们以前没有见过的。

下面是实现文件：

```

#include <QtGui>
#include "ticker.h"
Ticker::Ticker(QWidget *parent)
    : QWidget(parent)
{
    offset = 0;
    myTimerId = 0;
}

```

在构造函数中，设置 **offset** 为 0，这个变量是文本要显示的 **x** 坐标值。时间 **ID** 总是非 0 的，这里设置 **myTimerId** 为 0 说明我们还没有启动任何时间。

```

void Ticker::setText(const QString &newText)
{
    myText = newText;
    update();
    updateGeometry();
}

```

函数 `setText()` 设置要显示的文本。调用 `update()` 引发绘制事件重新显示文本，`updateGeometry()` 通知布局管理器改变控件的大小。

```
QSize Ticker::sizeHint() const
{
    return fontMetrics().size(0, text());
}
```

函数 `sizeHint()` 返回的是控件在不同文本时完整显示所需的尺寸。

`QWidget::fontMetrics()` 返回一个 `QFontMetrics` 对象，得到控件所用的字体的信息。在这里我们需要得到的是文本的大小。（在 `QFontMetrics::size()` 中，第一个参数是一个标识，对字符串来讲并不需要，所有赋了 0 值）。

```
void Ticker::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    int textWidth = fontMetrics().width(text());
    if (textWidth < 1)
        return;
    int x = -offset;
    while (x < width()) {
        painter.drawText(x, 0, textWidth, height(),
                        Qt::AlignLeft | Qt::AlignVCenter, text());
        x += textWidth;
    }
}
```

函数 `paintEvent()` 使用 `QPainter::drawText()` 绘制文本。调用 `fontMetrics()` 得到文本所需要的水平空间，然后多次绘制文本，直至填满整个控件。

```
void Ticker::showEvent(QShowEvent * /* event */)
```



```
{
    myTimerId = startTimer(30);
}
```

`showEvent()`启动了一个计时器。调用 `QObject::startTimer()` 返回一个 ID 值，这个 ID 值可以帮助我们识别这个计时器。`QObject` 能够支持多个独立的不同的时间间隔的计时器。调用 `startTimer()` 以后，Qt 大约每 30 毫秒产生一个事件，时间的准确与否取决于不同的操作系统。

我们也可以在 `Ticker` 的构造函数中调用 `startTimer()`。但是在控件可见以后再启动，能够节省一些资源。

```
void Ticker::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
        ++offset;
        if (offset >= fontMetrics().width(text()))
            offset = 0;
        scroll(-1, 0);
    } else {
        QWidget::timerEvent(event);
    }
}
```

函数 `timerEvent()` 由系统以一定间隔进行调用的。把 `offset` 增加 1 来模仿文字的移动，增加到标语的宽度时文字的宽度是重新设置为 0。然后调用 `scroll()` 把控件向左滚动一个像素。也可以调用 `update()`，但是 `scroll()` 更加高效，它对可见的像素进行移动，只是对需要新绘制的地方调用绘制事件（在这个例子中，只是一个像素宽的区域）。

如果计时器不是我们需要处理的，则把它传递给基类。

```
void Ticker::hideEvent(QHideEvent * /* event */)
{
    killTimer(myTimerId);
}
```

```
}
```

在 `hideEvent()` 中，调用 `QObject::killTimer()` 停止计时器。

时间事件的优先级很低，如果需要多个计时器，那么跟踪每一个计时器的 ID 是很费时的。这种情况下，较好的方法是为每一个计时器创建一个 `QTimer` 对象。在每一个时间间隔内，`QTimer` 发出一个 `timeout()` 信号。`QTimer` 还支持一次性计时器（只发出一次 `timeout()` 信号的计时器）。

7-2 安装事件过滤器（Installing Event Filters）

Qt 的事件模型一个强大的功能是一个 `QObject` 对象能够监视发送其他 `QObject` 对象的事件，在事件到达之前对其进行处理。

假设我们有一个 `CustomerInfoDialog` 控件，由一些 `QLineEdit` 控件组成。我们希望使用 `Space` 键得到下一个 `QLineEdit` 的输入焦点。一个最直接的方法是继承 `QLineEdit` 重写 `keyPressEvent()` 函数，当点击了 `Space` 键时，调用 `focusNextChild()`：

```
void MyLineEdit::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Space) {
        focusNextChild();
    } else {
        QLineEdit::keyPressEvent(event);
    }
}
```

这个方法有一个最大的缺点：如果我们在窗体中使用了很多不同类型的控件（`QComboBox`，`QSpinBox` 等等），我们也要继承这些控件，重写它们的 `keyPressEvent()`。一个更好的解决方法是让 `CustomerInfoDialog` 监视其子控件的键盘事件，在监视代码处实现以上功能。这就是事件过滤的方法。实现一个事件过滤包括两个步骤：

1. 在目标对象上调用 `installEventFilter()`，注册监视对象。
2. 在监视对象的 `eventFilter()` 函数中处理目标对象的事件。

注册监视对象的位置是在 `CustomerInfoDialog` 的构造函数中：

```

CustomerInfoDialog::CustomerInfoDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    firstNameEdit->installEventFilter(this);
    lastNameEdit->installEventFilter(this);
    cityEdit->installEventFilter(this);
    phoneNumberEdit->installEventFilter(this);
}

```

事件过滤器注册后，发送到 `firstNameEdit`, `lastNameEdit`, `cityEdit`, `phoneNumberEdit` 控件的事件首先到达 `CustomerInfoDialog::eventFilter()` 函数，然后在到达最终的目的地。

下面是 `eventFilter()` 函数的代码：

```

bool CustomerInfoDialog::eventFilter(QObject *target, QEvent *event)
{
    if (target == firstNameEdit || target == lastNameEdit
        || target == cityEdit || target == phoneNumberEdit) {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
            if (keyEvent->key() == Qt::Key_Space) {
                focusNextChild();
                return true;
            }
        }
    }
    return QDialog::eventFilter(target, event);
}

```

首先，我们看是目标控件是否为 `QLineEdit`，如果事件为键盘事件，把 `QEvent` 转换为 `QKeyEvent`，确定被敲击的键。如果为 `Space` 键，调用 `focusNextChild()`，把焦点交

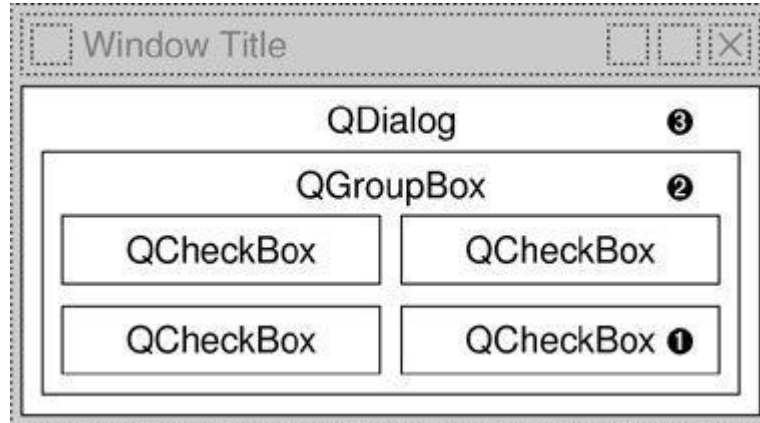
给下一个控件，返回 **true** 通知 Qt 已经处理了这个事件，如果返回 **false**，Qt 将会把事件传递给目标控件，把一个空格字符插入到 `QLineEdit` 中。

如果目标控件不是 `QLineEdit`，或者事件不是 `Space` 敲击事件，把控制权交给基类 `QDialog` 的 `eventFilter()`。目标控件也可以是基类 `QDialog` 正在监视的控件。（在 Qt4.1 中，`QDialog` 没有监视的控件，但是 Qt 的其他控件类，如 `QScrollArea`，监视一些它们的子控件）

Qt 的事件处理有 5 中级别：

1. 重写控件的事件处理函数：如重写 `keyPressEvent()`，`mousePressEvent()` 和 `paintEvent()`，这是最常用的事件处理方法，我们已经看到过很多这样的例子了。
2. 重写 `QObject::event()`，在事件到达事件处理函数时处理它。在需要改变 `Tab` 键的惯用法时这样做。也可以处理那些没有特定事件处理函数的比较少见的事件类型（例如，`QEvent::HoverEnter`）。我们重写 `event()` 时，必须要调用基类的 `event()`，由基类处理我们不需要处理的那些情况。
3. 给 `QObject` 对象安装事件过滤器：对象用 `installEventFilter()` 后，所有达到目标控件的事件都首先到达监视对象的 `eventFilter()` 函数。如果一个对象有多个事件过滤器，过滤器按顺序激活，先到达最近安装的监视对象，最后到达最先安装的监视对象。
4. 给 `QApplication` 安装事件过滤器，如果 `qApp`（唯一的 `QApplication` 对象）安装了事件过滤器，程序中所有对象的事件都要送到 `eventFilter()` 函数中。这个方法在调试的时候非常有用，在处理非活动状态控件的鼠标事件时这个方法也很常用。
5. 继承 `QApplication`，重写 `notify()`。Qt 调用 `QApplication::notify()` 来发送事件。重写这个函数是在其他事件过滤器处理事件前得到所有事件的唯一方法。通常事件过滤器是最有用的，因为在同一时间，可以有任意数量的事件过滤器，但是 `notify()` 函数只有一个。许多事件类型，包括鼠标，键盘事件，是能够传播的。如果事件在到达目标对象的途中或者由目标对象处理掉，事件处理的过程会重新开始，不同的是这时的目标对象是原目标对象的父控件。这样从父控件再到父控件，知道有控件处理这个事件或者到达了最顶级的那个控件。图 7.2 显示了一个键盘事件在一个对话框中从子控件到父控件的传播过程。当用户敲击一个键盘，时间首先发送到有焦点的控件上（这个例子中是 `QCheckBox`）。如果 `QCheckBox` 没有处理这个事件，Qt 把事件发送到 `QGroupBox` 中，如果仍然没有处理，则最后发送到 `QDialog` 中。

Figure 7.2. Event propagation in a dialog



7-3 系统繁忙时的响应（**Staying Responsive During Intensive Processing**）

当我们调用 `QApplication::exec()` 时，Qt 就开始了事件循环。启动时，Qt 发出显示和绘制事件，把控件显示出来。然后，事件循环就开始了，不停检查是否有事件发生，然后把事件分派到程序中的 `QObject` 对象。

一个事件正在处理时，其他的事件已经产生并加入到 Qt 的事件队列中，如果我们在处理某一个事件时花费了很多事件，这期间用户界面就不会有任何响应。例如，在程序保存文件时，窗口产生的事件就不会处理，只有在保存结束后才能处理。在保存的过程中，应用程序也不会处理窗口的绘制事件。

解决这个问题的方法是多线程：一个线程处理用户界面，另一个线程进行文件保存或者其他耗时的操作。这样，程序的用户界面就会在文件保存期间保持响应。在第 18 章会介绍这种方法。

还有一个简单的方法是在保存文件的过程中多次调用 `QApplication::processEvents()`。调用时 Qt 就会处理暂停的事件，然后返回继续保存文件。其实，`QApplication::exec()` 也是一个调用 `processEvents()` 的 while 循环。下面的例子是 Spreadsheet 在保存文件时用 `processEvents()` 响应用户界面：

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
    for (int row = 0; row < RowCount; ++row) {
```

```

        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
        qApp->processEvents();
    }
    return true;
}

```

但是这样做有一个危险，如果用户在保存文件期间关闭了主窗口，或者又点击了一次 **File|Save** 菜单，很容易造成死循环。解决的方法是把代码 `qApp->processEvents()` 用 `qApp->processEvents(QEventLoop::ExcludeUserInputEvents);` 代替，这样，Qt 就会不处理键盘和鼠标事件。

应用程序在进行长时间的操作时，经常使用 `QProgressDialog`，提示用户正在进行的操作的完成情况。`QProgressDialog` 还提供了一个 **Cancel** 按钮，允许用户取消当前的操作。下面的代码是 **Spreadsheet** 保存文件时使用 `QProgressDialog` 的代码：

```

bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
    QProgressDialog progress(this);
    progress.setLabelText(tr("Saving %1").arg(fileName));
    progress.setRange(0, RowCount);
    progress.setModal(true);
    for (int row = 0; row < RowCount; ++row) {
        progress.setValue(row);
        qApp->processEvents();
        if (progress.wasCanceled()) {
            file.remove();
            return false;
        }
    }
}

```

```

    }
    for (int column = 0; column < ColumnCount; ++column) {
        QString str = formula(row, column);
        if (!str.isEmpty())
            out << quint16(row) << quint16(column) << str;
    }
}
return true;
}

```

首先，创建一个 `QProgressDialog`，设置 `NumRows` 做为步骤的总数。然后，保存一行以后，调用 `setValue()` 更新进度条的状态。`QProgressDialog` 根据当前步骤数和总步骤数自动计算完成的百分比。调用 `QApplication::processEvents()` 处理可能发生的绘制事件，用户点击事件，或者键盘事件，如果用户点击了 **Cancel** 按钮，则取消保存操作，删除正在保存的文件。

我们没有调用 `QProgressDialog` 的 `show()` 函数，因为 `QProgressDialog` 会自己处理。如果需要保存的文件很小，所需时间很短，`QProgressDialog` 能够发觉这个情况，不显示进度条。

除了使用多线程和 `QProgressDialog`，还有一种完全不同的方法处理这种耗时较长的操作：在程序空闲时进行这类操作，而不是等待用户的请求才做。但是程序空闲的时间无法预计，这种方法的条件是所进行的操作能够安全中止和继续。具体实现是，启动一个 0 毫秒的计时器。只要程序中没有其他须处理的事件，这个事件就会触发。下面的 `timerEvent()` 函数就是这个方法的实现：

```

void Spreadsheet::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
        while (step < MaxStep && !qApp->hasPendingEvents()) {
            performStep(step);
            ++step;
        }
    }
}

```

```
    }  
} else {  
    QTableWidgetItem::timerEvent(event);  
}  
}
```

如果 `hasPendingEvents()` 返回 `true`，暂停操作，让 Qt 控制程序运行。当 Qt 没有需要处理的事件时，操作继续。

第八章序 2D 和 3D 图形系统 (2D and 3D Graphics)

Qt 的 2D 图形系统的基础是类 **QPainter**。**QPainter** 能够绘制各种几何图形（点，线，矩形，椭圆，圆弧，弦，扇形，多段线，贝赛尔曲线），还能绘制位图，图像和文字。此外 **QPainter** 还提供了很多高级功能：如平滑（平滑文字和几何图形的边界），透明度，渐变色，和矢量路径。**QPainter** 还支持矩阵变换，使绘制 2D 图形和分辨率无关。

QPainter 能够在“绘图设备”上绘图，如 **QWidget**，**QPixmap**，**QImage** 等都是绘图设备。在我们实现用户控件或者改变控件的外观时经常使用它。**QPainter** 还能构和 **QPrinter** 一起使用进行打印，制作 PDF 文档。这样我们可以用同样的代码把数据显示在屏幕上或者打印出来。

OpenGL 能够代替 **QPainter**。**OpenGL** 是绘制 2D 和 3D 图形的一个标准库。**QtOpenGL** 模块能够方便的把 **OpenGL** 代码整合到 Qt 应用程序中

8-1 用 **QPainter** 绘图 (Painting with **QPainter**)

要在绘图设备（**paint device**，一般是一个控件）上开始绘制，我们只要创建一个 **QPainter**，把绘图设备指针传给 **QPainter** 对象。例如：

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    ...
}
```

使用 **QPainter** 的 **draw...()** 函数我们可以绘制各种图形。图 8.1 给出了主要的一些。绘制的方式由 **QPainter** 的设置决定。设置的一部分是从绘图设备得到的，其他是初始化时的默认值。三个主要的设置为：画笔，刷子和字体。

画笔用来绘制直线和图形的边框。包含颜色，宽度，线型，角设置和连接设置。刷子是用来填充几何图形的方式。包含颜色，方式设置，也可以是一个位图或者渐变色。字体用来绘制文本。字体的属性很多，如字体名，字号等。

这些设置随时可以改变，可用 QPen, QBrush, QFont 对象调用 setPen(), setBrush(), setFont()修改。

Figure 8.1. QPainter's most frequently used draw...() functions

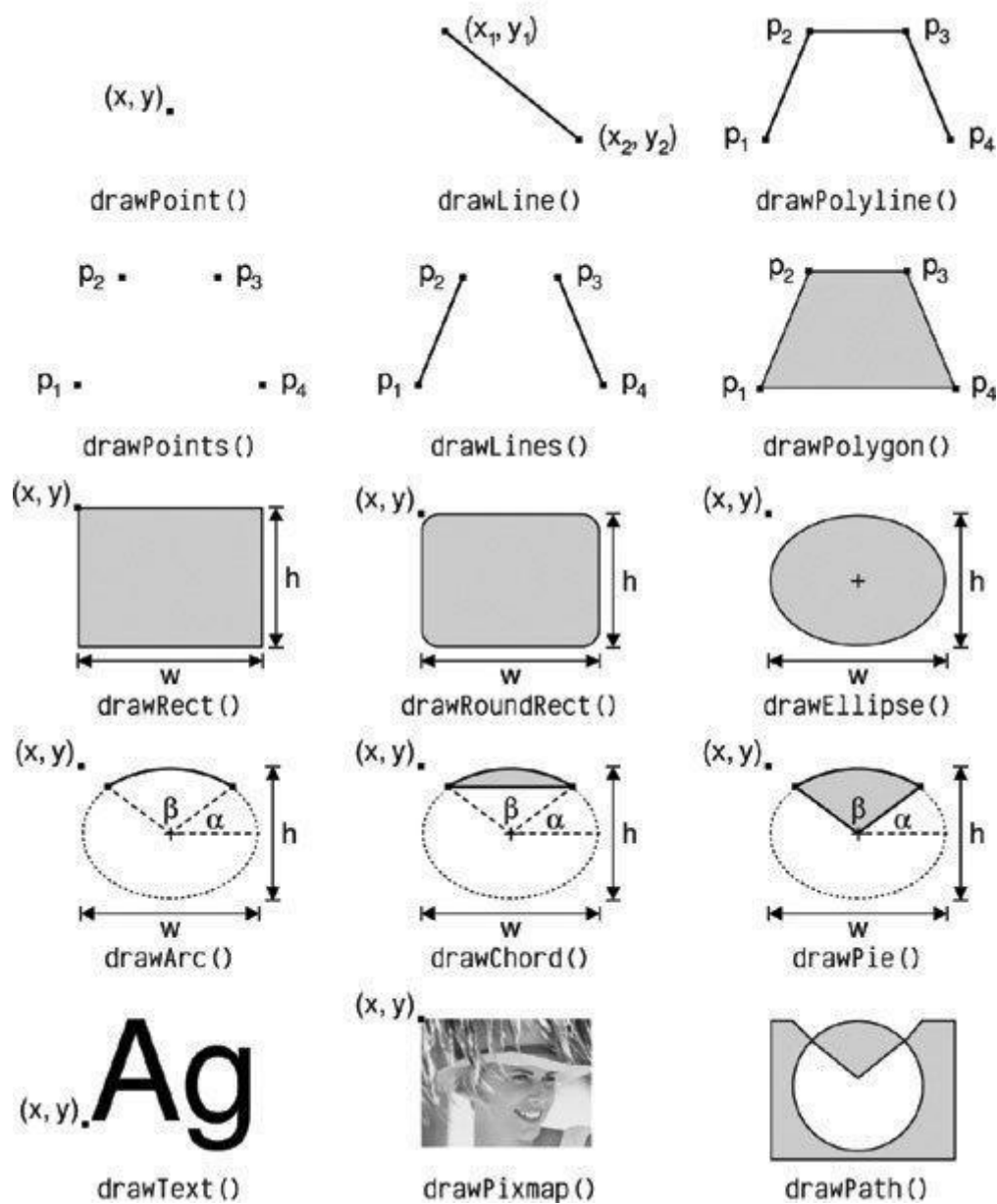


Figure 8.2. Cap and join styles

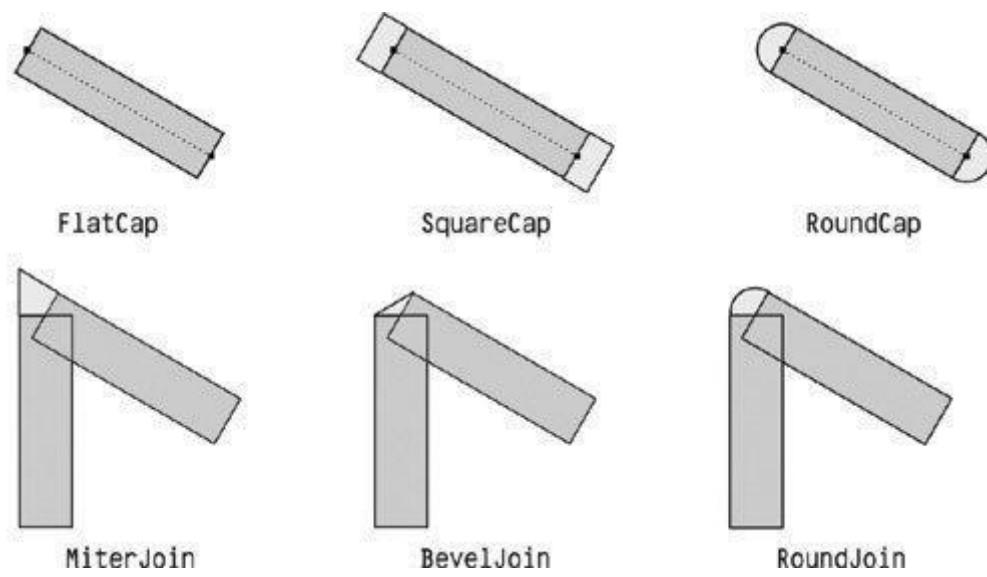
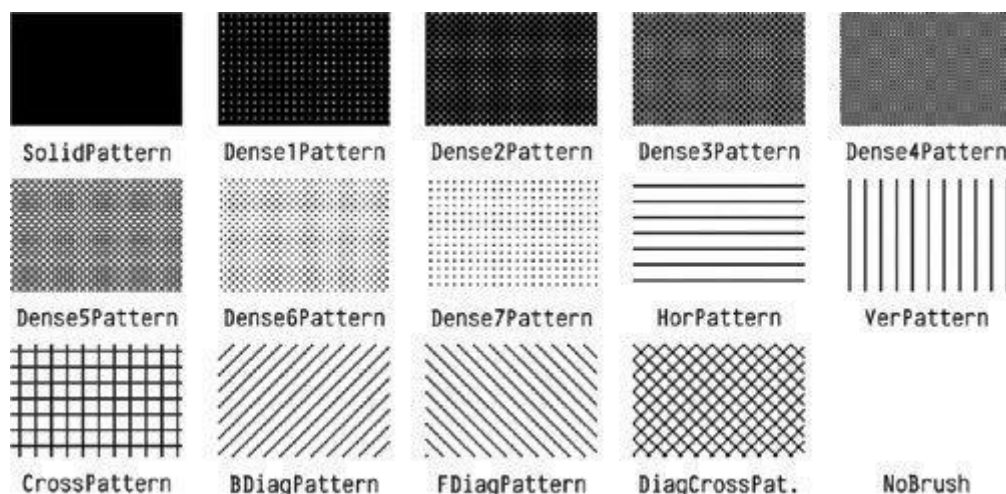


Figure 8.3. Pen styles

	line width			
	1	2	3	4
NoPen				
SolidLine				
DashLine				
DotLine				
DashDotLine				
DashDotDotLine				

Figure 8.4. Predefined brush styles

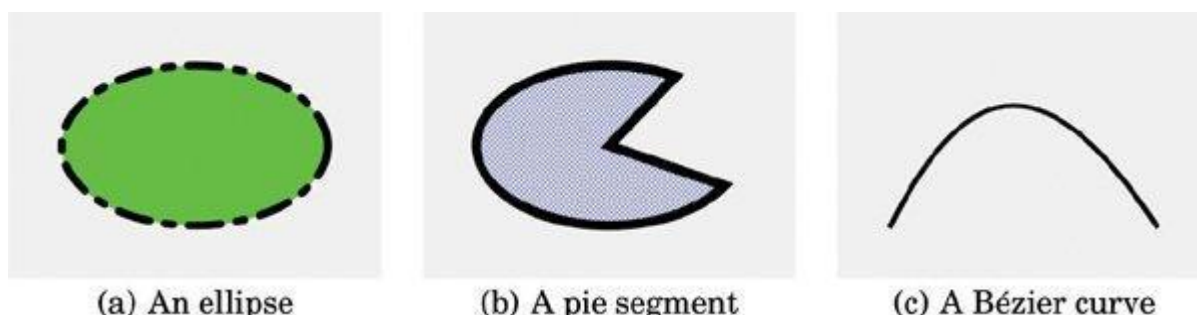


现在来看看具体的例子。下面的代码是绘制图 8.5(a)中椭圆的代码：

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 12, Qt::DashDotLine, Qt::RoundCap));
painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
painter.drawEllipse(80, 80, 400, 240);
```

调用函数 `setRenderHint(QPainter::Antialiasing, true)`，使绘制时边缘平滑，使用颜色浓度的变化，把图形的边缘转换为像素时引起的扭曲变形尽可能减少，在支持这一功能的平台或者绘图设备上得到一个平滑的边缘。

Figure 8.5. Geometric shape examples



(a) An ellipse

(b) A pie segment

(c) A Bézier curve

下面的代码是图 8.5(b)中绘制扇形的代码：

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
```

```

painter.setPen(QPen(Qt::black, 15, Qt::SolidLine, Qt::RoundCap,
                    Qt::MiterJoin));
painter.setBrush(QBrush(Qt::blue, Qt::DiagCrossPattern));
painter.drawPie(80, 80, 400, 240, 60 * 16, 270 * 16);

```

函数 `drawPie()` 的最后两个参数值的单位为一度的十六分之一。

下面的代码是图 8.5(c) 中绘制贝赛尔曲线的代码：

```

QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
QPainterPath path;
path.moveTo(80, 320);
path.cubicTo(200, 80, 320, 80, 480, 320);
painter.setPen(QPen(Qt::black, 8));
painter.drawPath(path);

```

通过连接基本图形元素，直线，椭圆，多段线，圆弧，二次和三次贝塞尔曲线等，`QPainterPath` 类能确定任何矢量图形。因此，绘图路径（**Painter paths**）是最基本的绘制元素，任何图形和图形的组合都可以同路径（**path**）表示。

一个路径能够确定一个轮廓，由这个轮廓确定的区域可以由刷子来填充。在图 8.5(c) 中我们没有设置刷子，因此只绘制了轮廓。

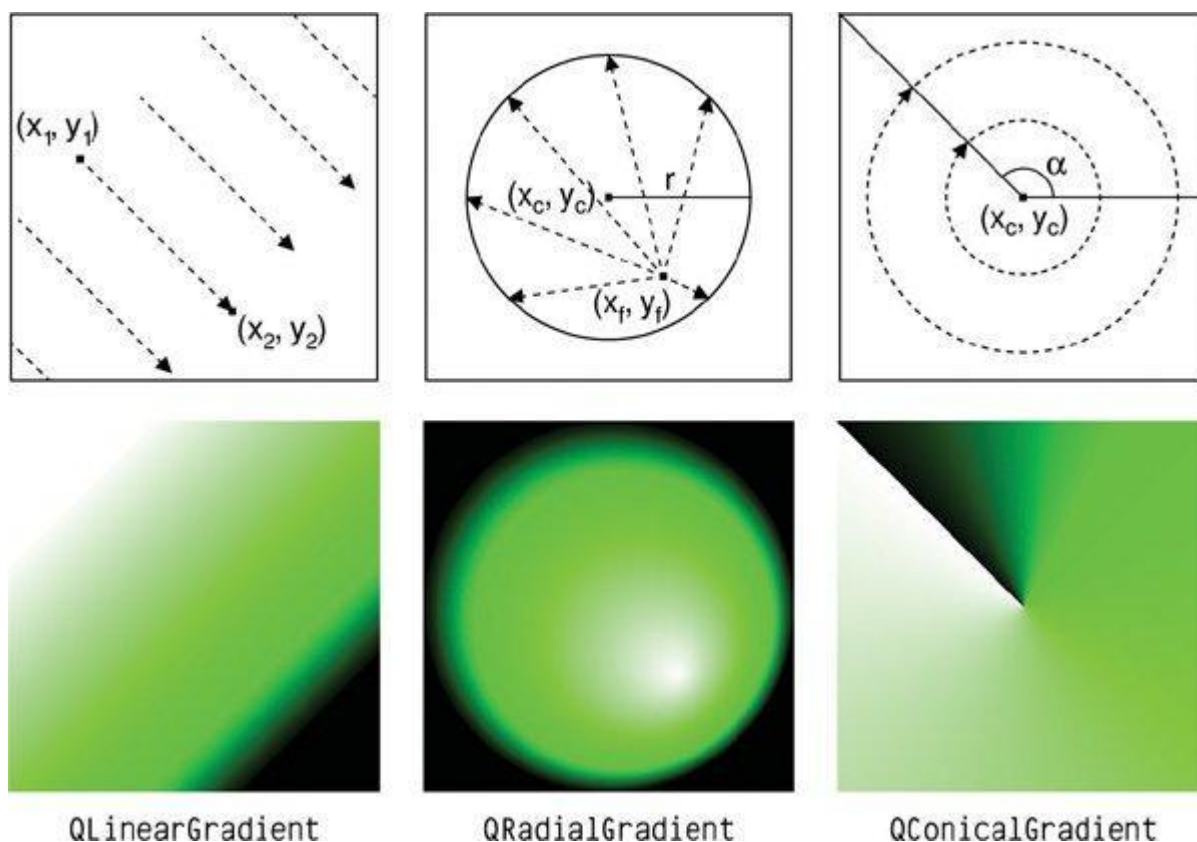
以上的三个例子都是使用了默认的刷子（`Qt::SolidPattern`，`Qt::DiagCrossPattern`，`Qt::NoBrush`）。在现在的应用程序中，单色填充已经很少使用，渐变色填充开始收到欢迎。渐变是依靠颜色的变化实现两种或者多种颜色之间平滑的过渡。渐变通常用来处理 3D 效果，如使用 **Plastique** 渐变方式来表现 `QPushButtons`。

Qt 支持三种类型的渐变：线形渐变，圆锥渐变和圆形渐变（**linear**, **conical**, and **radial**）。下一节的 `OvenTimer` 例子就是在一个控件中使用了所有这三种渐变。

线形渐变由两个控制点和直线上的一系列颜色点组成。图 8.6 由下面的代码得到：在两个控制点之间，在三个不同的位置确定了三个不同的颜色值。位置有 0 到 1 的浮点数得到，0 为第一个控制点，1 为第二个控制点。不同位置点之间的颜色由差值计算得到。

```
LinearGradient gradient(50, 100, 300, 350);
gradient.setColorAt(0.0, Qt::white);
gradient.setColorAt(0.2, Qt::green);
gradient.setColorAt(1.0, Qt::black);
```

Figure 8.6. QPainter's gradient brushes



圆形渐变由颜色组，圆心 (x_c, y_c) ，半径 r 和焦点 (x_f, y_f) 定义。圆心和半径定义一个圆，颜色从焦点开始扩散到周围，焦点可以是圆心也可以是圆内的任意一个点。

圆锥渐变由圆心 (x_c, y_c) 和一个角度 a 定义。颜色从圆心开始像表的秒针一样扩散。

我们已经提到了 **QPainter** 的画笔，刷子和字体设置。此外，**QPainter** 还有其他一些设置影响图形和文字的绘制：

1. 背景刷子，当背景模式为 **Qt::OpaqueMode**（缺省值为 **Qt::transparentMode**）时，背景刷子用来填充几何图形，文字，和位图的背景（在绘图刷子的下面）

2. 刷子的起点：刷子的起始绘制点，通常为控件的左上角。
3. 剪辑区域，剪辑区域为绘图设备上可以绘制的区域，在剪辑区域意外进行的绘制是无效的。
4. 视口，窗口，世界坐标：这三个决定了 `QPainter` 的逻辑坐标映射到物理坐标的方式。通常，逻辑坐标和物理坐标是重合的。坐标系统在下一节介绍。
5. 组合方式：组合方式决定绘制设备上新绘制的像素和已经存在的像素的影响方式。缺省方式为覆盖式（`source over`），新像素画在已有元素的上面。只是有限一个绘图设备支持组合方式的设置，将在本章后面介绍

在任何时候，我们可以调用 `save()` 把 `QPainter` 当前的设置保存在一个内部栈里，然后调用 `restore()` 进行恢复。我们能够临时改变 `QPainter` 的一些设置，然后恢复先前的值。

8-2 坐标变换（Painter Transformations）

在 `QPainter` 的初始坐标系统中，点 $(0,0)$ 位于绘图设备的左上角。X 轴坐标向右递增，y 轴向下递增，一个像素占据 1×1 的面积。

需要说明的一点是一个像素的中心位于坐标的一半处。例如，左上角位于点 $(0,0)$ 和点 $(1,1)$ 之间区域的像素，它的中心位于 $(0.5,0.5)$ 。如果我们使用 `QPainter` 绘制一个位置在 $(100,100)$ 的像素，`QPainter` 会在每个坐标值上增加 0.5，以坐标 $(100.5,100.5)$ 为中心绘制这个像素。

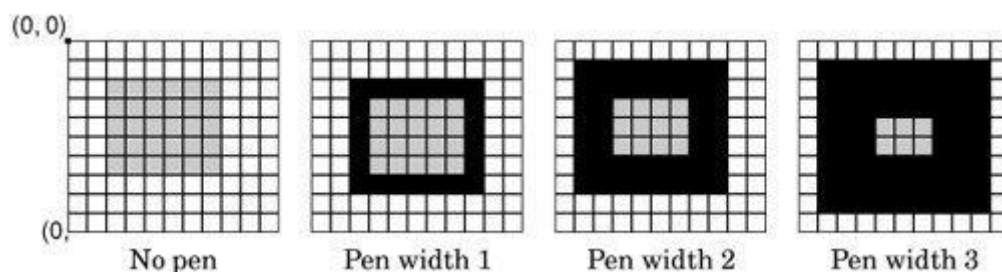
一个需要注意的事情是，一个像素的中心位于像素坐标的“半像素”坐标。例如，窗口左上角像素占据从点 $(0,0)$ 到 $(1,1)$ 的位置，它的中心位于 $(0.5,0.5)$ 。如果我们需要 `QPainter` 在点 $(100,100)$ 的坐标处绘制另一个像素，`QPainter` 将会在两个坐标轴方向偏移 0.5 个坐标点，即像素的中心点将会位于 $(100.5,100.5)$ 。

这个偏移看起来有些教条，但是实际上有这重要的作用。首先，在禁止消除锯齿功能（缺省设置）时才进行 0.5 的偏移。如果许可了消除锯齿功能，`QPainter` 会在 $(100,100)$ 的位置绘制一个黑色的像素。事实是 `QPainter` 在 $(99.5,99.5)$, $(99.5,100.5)$, $(100.5,99.0)$, $(100.5,100.5)$ 绘制亮灰色像素，这样产生的效果就是一个黑色像素位于四个像素的焦点 $(100,100)$ 处。如果我们不需要这个功能，可以把坐标偏移半个像素。

在绘制直线，矩形，椭圆时，上述规则都是适用的。图 8.7 表明了在不消除锯齿功能时，用不同的笔宽度绘制矩形 `drawRect(2,2,6,5)` 的不同结果。需要特别注意用 1 像素

的笔宽绘制 6*5 的矩形时实际的矩形面积为 7*6。这和以前的 Qt 版本不同，但是这个功能对绘制看缩放的，独立于分辨率的矢量图形很有帮助。

Figure 8.7. Drawing a 6 x 5 rectangle with no antialiasing



现在我们已经理解了 Qt 的默认坐标系，现在再来了解 QPainter 的视口(viewport)，窗口(window)和世界坐标系矩阵(world matrix)的变化。(在这一节中，窗口(window)不是控件的窗口，视口(viewport)也和 QScrollArea 的视口也没有联系)

窗口和视口是紧密联系在一起的。视口是由物理坐标确定的任意矩形。窗口是由逻辑坐标表示的视口大小。QPainter 在进行绘制时，我们给 QPainter 的是逻辑坐标，根据视口和窗口的设置，这些逻辑坐标通过线性变换，转换为物理坐标。

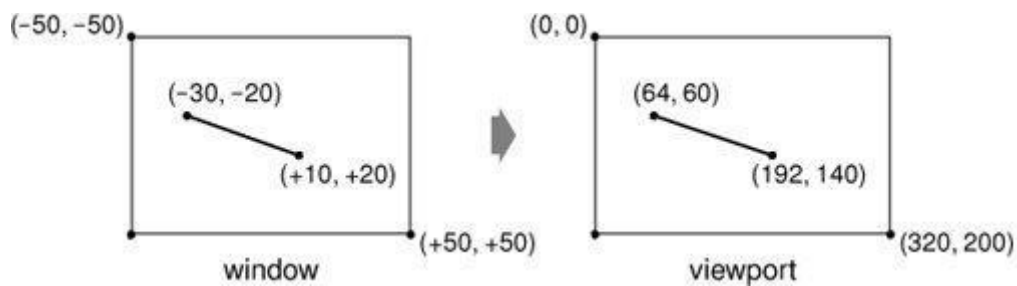
通常，窗口和视口的大小和绘图设备是一致的。例如，一个 320*200 的控件，视口和窗口都是一个 320*200 的矩形，起始点(0,0)位于左上角。这时，逻辑坐标和物理坐标是相同的。

视口窗口机制是为了绘制与绘图设备的大小和分辨率无关的图形。如果我们的逻辑坐标设置为从(-50,-50)到(+50,+50)的矩形，(0,0)点在中心。如下这样设置窗口：

```
painter.setWindow(-50, -50, 100, 100);
```

(-50,-50)确定了原点，(100,100)确定矩形的宽和高。在窗口中，逻辑坐标(-50,-50)相当于物理坐标中的原点(0,0)，(+50,+50)相当于物理坐标的点(320,320)。视口的设置没有改变。

Figure 8.8. Converting logical coordinates into physical coordinates



现在来说明世界坐标系矩阵。窗口视口可以转换变形，世界坐标系矩阵也是一个用来图形变换的转换矩阵。用来平移，缩放，旋转，剪切图形。例如，如果要绘制一行倾斜 45° 的文字，代码如下：

```
QMatrix matrix;
matrix.rotate(45.0);
painter.setMatrix(matrix);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

传给 `drawText()` 函数的逻辑坐标由世界矩阵进行旋转，然后根据窗口视口设置映射到物理坐标。

如果我们指定了多个坐标变换，按照设置顺序应用。例如，以 $(10, 20)$ 做为中心旋转 45° ，可以把原点移动到 $(10, 20)$ ，然后旋转，再把窗口原点平移到原来的位置：

```
QMatrix matrix;
matrix.translate(-10.0, -20.0);
matrix.rotate(45.0);
matrix.translate(+10.0, +20.0);
painter.setMatrix(matrix);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

一个简单的方法是使用 `QPainter` 的转换函数 `translate()`，`scale()`，`rotate()` 和 `shear()`。

```
painter.translate(-10.0, -20.0);
painter.rotate(45.0);
painter.translate(+10.0, +20.0);
```

```
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

但是，如果我们反复需要同一个矩阵，最好还是把它保存到 `QMatrix` 中，在需要的时候给 `QPainter` 设置。

为了更好的解释绘图的坐标变换，我们看一下图 8.9 所示 `OvenTimer` 控件的代码。`OvenTimer` 以厨房计时器为模型，在烤炉没有自带的计时器之前，这种定时器使用很广泛。用户点击定时器上面的一个刻度值，指针就从这个刻度值开始，自动逆时针旋转，到达刻度 0 的位置，这时，`OvenTimer` 发出 `timeout()` 信号。

Figure 8.9. The OvenTimer widget



头文件 `oventimer.h`，从 `QWidget` 继承，重写了 `paintEvent()` 和 `mousePressEvent()` 函数。

```
class OvenTimer : public QWidget
{
|   Q_OBJECT
| public:
|   OvenTimer(QWidget *parent = 0);
|   void setDuration(int secs);
|   int duration() const;
|   void draw(QPainter *painter);
| signals:
|   void timeout();
| protected:
```

```

| void paintEvent(QPaintEvent *event);
| void mousePressEvent(QMouseEvent *event);
| private:
|     QDateTime finishTime;
|     QTimer *updateTimer;
|     QTimer *finishTimer;
| };

```

源文件 oventimer.cpp，首先是一些常量的定义，确定定时器的外观。

在构造函数中，我们创建了两个 `QTimer` 对象：`updateTimer` 每一秒中更新控件的外观，`finishTimer` 在定时器到达 0 点时发出 `timeOut` 信号。`finishTimer` 只需要一次 `timeOut`，所以调用了 `setSingleShot(true)`。通常计时器 `QTimer` 自创建开始就计时，直到它们停止或者销毁。最后一个 `connect` 语句用来定时结束时停止计时器。

函数 `setDuration()` 设置计时器的时间周期，以秒为单位。结束时间由当前时间（由 `QDateTime::currentDateTime()` 得到）加上定时周期得到，保存在 `finishTime` 中。最后调用 `update()` 用新的计时周期重新绘制控件。

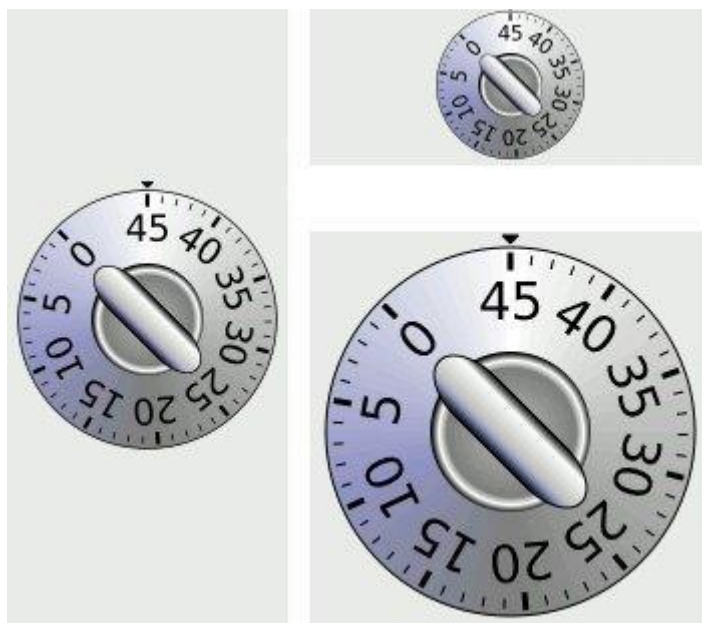
`finishTime` 变量为 `QDateTime` 类型，因此变量中包含当前的日期和时间。我们需要避免一个循环错误，例如当前时间为午夜以前而结束时间为午夜以后。

函数 `duration()` 函数返回在定时结束之前还剩下的时间。如果计时器没有启动，则返回 0。

如果用户点击了控件，我们就找到距离点击点最近的一个刻度值（当然有细微的误差）我们使用得到的刻度值设置新的定时周期。然后开始重新绘制控件。指针开始逆时针移动直到计时结束。

在 `paintEvent()` 中，设置视口与控件的尺寸一致，设置窗口为 `(50,50,100,100)`，即有点 `(-50,-50)` 到 `(50,50)` 的矩形。`qMin()` 模板函数得到两个参数中的最小值，调用 `draw()` 函数绘制。

Figure 8.10. The OvenTimer widget at three different sizes



现在我们看一下 `draw()` 函数，首先我们绘制一个小的倒三角形表示控件的 0 位置。这个三角形由三个坐标指定，使用函数 `drawPolygon()` 绘制它。

```
static const int triangle[3][2] = {
    { -2, -49 }, { +2, -49 }, { 0, -47 }
};

QPen thickPen(palette().foreground(), 1.5);
QPen thinPen(palette().foreground(), 0.5);
QColor niceBlue(150, 150, 200);
painter->setPen(thinPen);
painter->setBrush(palette().foreground());
painter->drawPolygon(QPolygon(3, &triangle[0][0]));
```

视口窗口机制的好处就在于我们可以直接在绘图函数中指定坐标值，根据自动坐标变换能适应控件的各种大小。

在绘制最外面的一个圆形我们使用了圆锥渐变。渐变的中心点位于(0,0)，角度为-90°。

```
QConicalGradient coneGradient(0, 0, -90.0);
coneGradient.setColorAt(0.0, Qt::darkGray);
coneGradient.setColorAt(0.2, niceBlue);
coneGradient.setColorAt(0.5, Qt::white);
coneGradient.setColorAt(1.0, Qt::darkGray);
painter->setBrush(coneGradient);
painter->drawEllipse(-46, -46, 92, 92);
```

绘制里面的圆形时使用了圆形渐变。圆心和渐变的中心点位于(0,0)，渐变半径为 20。

```
QRadialGradient haloGradient(0, 0, 20, 0, 0);
haloGradient.setColorAt(0.0, Qt::lightGray);
haloGradient.setColorAt(0.8, Qt::darkGray);
haloGradient.setColorAt(0.9, Qt::white);
haloGradient.setColorAt(1.0, Qt::black);
painter->setPen(Qt::NoPen);
painter->setBrush(haloGradient);
painter->drawEllipse(-20, -20, 40, 40);
```

在绘制刻度时，我们旋转控件的坐标系。在原来的坐标系中，0 分钟刻度在最上面，现在 0 刻度被移动到相当于剩余时间的位置。坐标旋转后我们绘制矩形的突起手柄，它的旋转角度和坐标旋转角度相同。

```
QLinearGradient knobGradient(-7, -25, 7, -25);
knobGradient.setColorAt(0.0, Qt::black);
knobGradient.setColorAt(0.2, niceBlue);
knobGradient.setColorAt(0.3, Qt::lightGray);
knobGradient.setColorAt(0.8, Qt::white);
knobGradient.setColorAt(1.0, Qt::black);
```

```

painter->rotate(duration() * DegreesPerSecond);
painter->setBrush(knobGradient);
painter->setPen(thinPen);
painter->drawRoundRect(-7, -25, 14, 50, 150, 50);
for (int i = 0; i <= MaxMinutes; ++i) {
    if (i % 5 == 0) {
        painter->setPen(thickPen);
        painter->drawLine(0, -41, 0, -44);
        painter->drawText(-15, -41, 30, 25,
                           Qt::AlignHCenter | Qt::AlignTop,
                           QString::number(i));
    } else {
        painter->setPen(thinPen);
        painter->drawLine(0, -42, 0, -44);
    }
    painter->rotate(-DegreesPerMinute);
}

```

在 `for` 循环中，我们沿着最外层圆形的边绘制时间记号，每隔 5 分钟一次。记号值画在刻度的下面。在每一次循环结束，坐标旋转 7° ，相当于 1 分钟。这样再次绘制标记时，虽然我们传给 `drawLine()` 和 `drawText()` 坐标值没有变，但是却能绘制在不同的地方。

这个代码中的 `for` 循环有一个小的缺陷，如果我们执行更多的循环就能很明显出现。我们每次调用 `rotate()`，当前世界坐标系矩阵乘以一个旋转矩阵，得到一个新的世界坐标系矩阵。由于浮点数运算时产生的四舍五入误差就会累加，世界坐标系矩阵就越发不准确。我们可以重新设计 `for` 循环避免这个问题，在每一次循环中，使用 `save()` 和 `restore()` 函数保存和重新加载原始的坐标系。

```

for (int i = 0; i <= MaxMinutes; ++i) {
    painter->save();
    painter->rotate(-i * DegreesPerMinute);
    if (i % 5 == 0) {
        painter->setPen(thickPen);

```

```

painter->drawLine(0, -41, 0, -44);
painter->drawText(-15, -41, 30, 25,
                  Qt::AlignHCenter | Qt::AlignTop,
                  QString::number(i));
} else {
    painter->setPen(thinPen);
    painter->drawLine(0, -42, 0, -44);
}
painter->restore();
}

```

另一种实现计时器的方法是不进行坐标变换，使用算术函数 `sin()` 和 `cos()` 计算刻度位置。但是如果想绘制文本，还是需要旋转坐标系。

```

const double DegreesPerMinute = 7.0;
const double DegreesPerSecond = DegreesPerMinute / 60;
const int MaxMinutes = 45;
const int MaxSeconds = MaxMinutes * 60;
const int UpdateInterval = 1;

OvenTimer::OvenTimer(QWidget *parent)
    : QWidget(parent)
{
    finishTime = QDateTime::currentDateTime();
    updateTimer = new QTimer(this);
    connect(updateTimer, SIGNAL(timeout()), this, SLOT(update()));
    finishTimer = new QTimer(this);
    finishTimer->setSingleShot(true);
    connect(finishTimer, SIGNAL(timeout()), this, SIGNAL(timeout()));
    connect(finishTimer, SIGNAL(timeout()), updateTimer, SLOT(stop()));
}

```

```

void OvenTimer::setDuration(int secs)
{
    if (secs > MaxSeconds) {
        secs = MaxSeconds;
    } else if (secs <= 0) {
        secs = 0;
    }
    finishTime = QDateTime::currentDateTime().addSecs(secs);
    if (secs > 0) {
        updateTimer->start(UpdateInterval * 1000);
        finishTimer->start(secs * 1000);
    } else {
        updateTimer->stop();
        finishTimer->stop();
    }
    update();
}

int OvenTimer::duration() const
{
    int secs = QDateTime::currentDateTime().secsTo(finishTime);
    if (secs < 0)
        secs = 0;
    return secs;
}

void OvenTimer::mousePressEvent(QMouseEvent *event)
{
    QPointF point = event->pos() - rect().center();
    double theta = atan2(-point.x(), -point.y()) * 180 / 3.14159265359;
    setDuration(duration() + int(theta / DegreesPerSecond));
    update();
}

```



```

}

void OvenTimer::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    int side = qMin(width(), height());
    painter.setViewport((width() - side) / 2, (height() - side) / 2,
                        side, side);
    painter.setWindow(-50, -50, 100, 100);
    draw(&painter);
}

void OvenTimer::draw(QPainter *painter)
{
    static const int triangle[3][2] = {
        { -2, -49 }, { +2, -49 }, { 0, -47 }
    };

    QPen thickPen(palette().foreground(), 1.5);
    QPen thinPen(palette().foreground(), 0.5);
    QColor niceBlue(150, 150, 200);
    painter->setPen(thinPen);
    painter->setBrush(palette().foreground());
    painter->drawPolygon(QPolygon(3, &triangle[0][0]));
    QConicalGradient coneGradient(0, 0, -90.0);
    coneGradient.setColorAt(0.0, Qt::darkGray);
    coneGradient.setColorAt(0.2, niceBlue);
    coneGradient.setColorAt(0.5, Qt::white);
    coneGradient.setColorAt(1.0, Qt::darkGray);
    painter->setBrush(coneGradient);
    painter->drawEllipse(-46, -46, 92, 92);
    QRadialGradient haloGradient(0, 0, 20, 0, 0);

```

```

haloGradient.setColorAt(0.0, Qt::lightGray);
haloGradient.setColorAt(0.8, Qt::darkGray);
haloGradient.setColorAt(0.9, Qt::white);
haloGradient.setColorAt(1.0, Qt::black);
painter->setPen(Qt::NoPen);
painter->setBrush(haloGradient);
painter->drawEllipse(-20, -20, 40, 40);
QLinearGradient knobGradient(-7, -25, 7, -25);
knobGradient.setColorAt(0.0, Qt::black);
knobGradient.setColorAt(0.2, niceBlue);
knobGradient.setColorAt(0.3, Qt::lightGray);
knobGradient.setColorAt(0.8, Qt::white);
knobGradient.setColorAt(1.0, Qt::black);
painter->rotate(duration() * DegreesPerSecond);
painter->setBrush(knobGradient);
painter->setPen(thinPen);
painter->drawRoundRect(-7, -25, 14, 50, 150, 50);
for (int i = 0; i <= MaxMinutes; ++i) {
    if (i % 5 == 0) {
        painter->setPen(thickPen);
        painter->drawLine(0, -41, 0, -44);
        painter->drawText(-15, -41, 30, 25,
                           Qt::AlignHCenter | Qt::AlignTop,
                           QString::number(i));
    } else {
        painter->setPen(thinPen);
        painter->drawLine(0, -42, 0, -44);
    }
    painter->rotate(-DegreesPerMinute);
}

```

```
}  
}
```

8-3 使用 QImage 进行高质量绘制 (High-Quality Rendering with QImage)

在进行绘图时,我们经常要面对速度和效率两者之间矛盾。在 X11 和 MacOSX 系统上,在 QWidget 和 QPixmap 绘图要依赖平台自身的绘图引擎。在 X11 上,与 X server 的通信很少,Qt 只是发送绘图命令而不是真正的绘图数据。这种画法的不足是 Qt 要收到平台自身绘图引擎的限制。

在 X11 上,消除锯齿和支持分数坐标这些功能只有在 Xserver 上安装了 XRender 扩展才能实现;

在 MacOSX 平台,它自己的绘图引擎在绘制多段线时使用了和 X11 和 Windows 不同的算法,因此得到的结果会有稍许差别。

当准确性比效率重要时,我们可以先绘制在 QImage 上,然后把结果拷贝到屏幕。在 QImage 绘图使用 Qt 自己的绘图引擎,因此在所有平台上都能得到一致的结果。使用这个方法的额外工作是用 QImage::Format_RGB32 或者

QImage::Format_ARGB32_Premultiplied 参数创建 QImage 对象。

QImage::Format_ARGB32_Premultiplied 和传统的 ARGB32 格式(0xaarrggb)格式完全一致,不同在于红,绿,蓝三个通道值都“乘以”了 alpha 通道值。这样,0x00 到 0xFF 的 RGB 颜色值范围变为 0x00 到 alpha 通道值。例如 50%透明度的蓝色用 ARGB 格式表示为 0x7F0000FF,在用 Format_ARGB32_Premultiplied 表示时为 0x7F00007F,同理,75%透明度的黑绿色在 ARGB 格式中表示为 0x3F008000,在 Format_ARGB32_Premultiplied 格式中表示为 0x3F002000。

如果我们想用消除锯齿的方式绘制一个控件,并希望在没有 XRender 扩展的 X11 平台上也得到很好的结果,在原来需要依靠 XRender 的 paintEvent()函数代码如下:

```
void MyWidget::paintEvent(QPaintEvent *event)  
{  
    QPainter painter(this);  
    painter.setRenderHint(QPainter::Antialiasing, true);  
    draw(&painter);  
}
```

下面的代码为重写的 `paintEvent()`，使用了 Qt 的平台独立的绘图引擎：

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QImage image(size(), QImage::Format_ARGB32_Premultiplied);
    QPainter imagePainter(&image);
    imagePainter.initFrom(this);
    imagePainter.setRenderHint(QPainter::Antialiasing, true);
    imagePainter.eraseRect(rect());
    draw(&imagePainter);
    imagePainter.end();
    QPainter widgetPainter(this);
    widgetPainter.drawImage(0, 0, image);
}
```

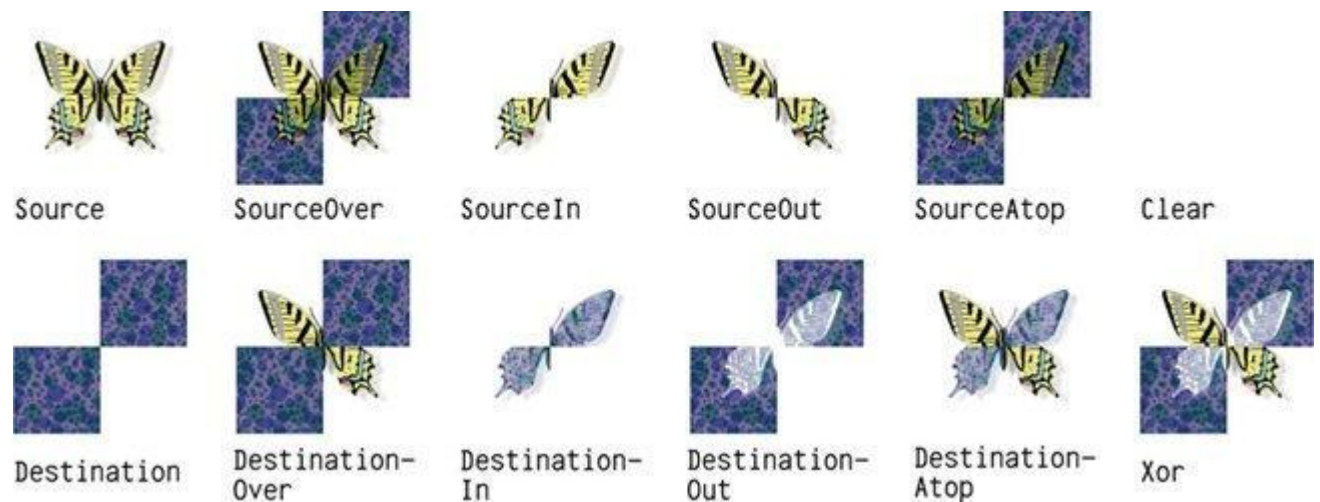
在上面的代码中，我们创建了一个 `Format_ARGB32_Premultiplied` 格式的 `QImage`，大小和控件相同，创建一个 `QPainter` 绘制这个图像。`QPainter::initFrom()` 调用用控件的设置初始化画笔，刷子和字体。然后想以前一样绘制。最后，创建控件的 `QPainter` 对象，把图片拷贝到控件上。

这种方式能够在不同的平台上得到效果一样的结果，但是绘制的文字除外，因为这取决于安装的字体。

Qt 绘图引擎的另外一个尤为有用的功能是它能支持混和模式。在进行绘图时，原图像和目标图像能够组合起来。这个混和支持多种绘图操作：如笔，刷子，渐变色，图像等。

缺省的组合模式为 `QImage::CompositionMode_SourceOver`，即原像素（正在绘制的像素）和目标像素（已经存在的像素）混和，原像素的 `alpha` 分量定义为最终的透明度。图 8.11 显示了不同的模式下绘制的半透明蝴蝶的效果。

Figure 8.11. QPainter's composition modes



函数 `QPainter::setCompositionMode()` 用来设置组合模式。下面的代码创建一个 `QImage`，包含一个蝴蝶和一个棋盘格子的混和：

```
QImage resultImage = checkerPatternImage;
QPainter painter(&resultImage);
painter.setCompositionMode(QPainter::CompositionMode_Xor);
painter.drawImage(0, 0, butterflyImage);
```

一个需要注意的问题是，`QImage::CompositionMode_Xor` 模式对 `alpha` 通道也同样适用。如果用白颜色 `0xFFFFFFFF` 同它自己混和，得到的将是透明色 `0x00000000`，而不是 `0xFF000000`。

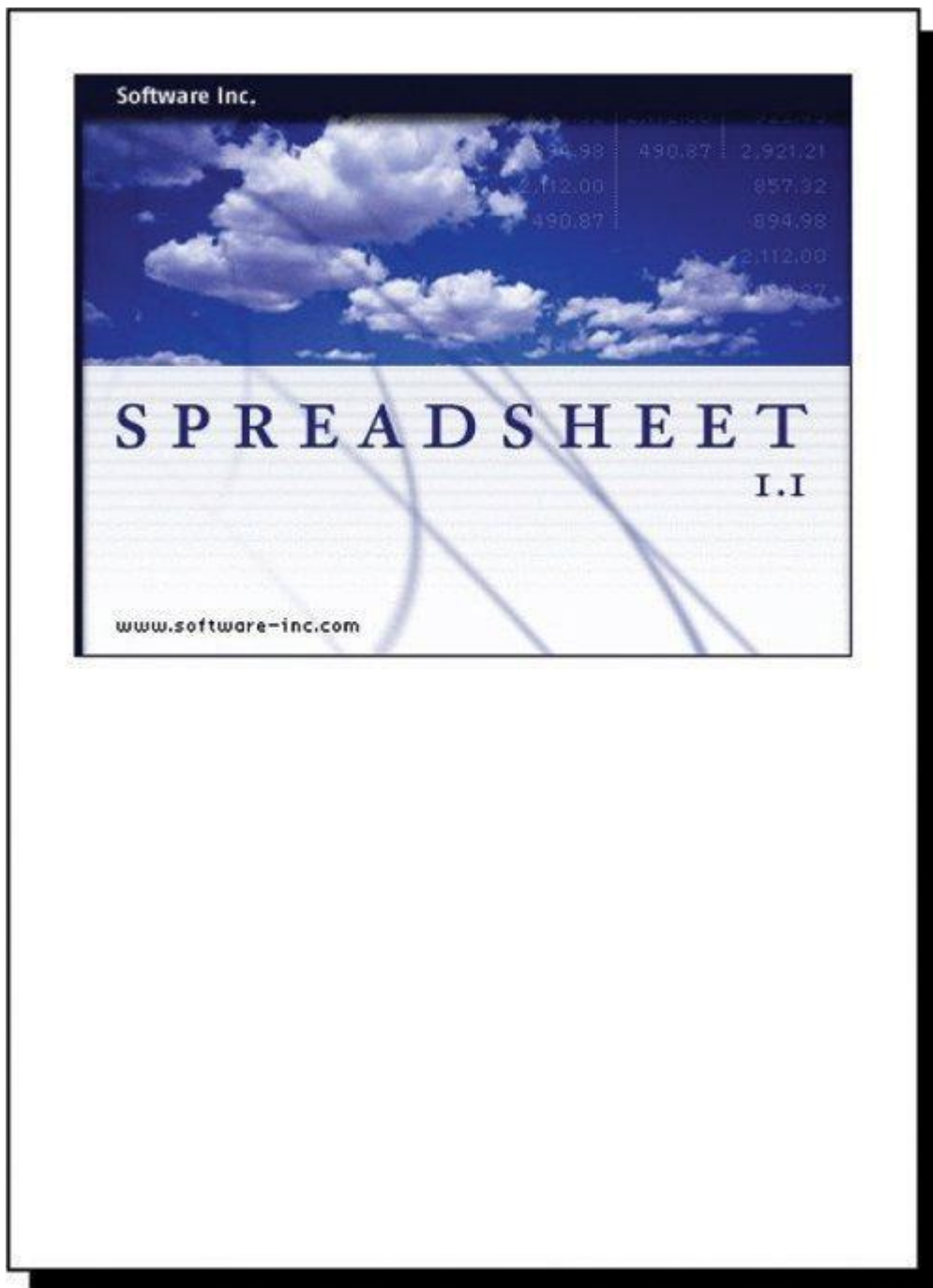
8-4 打印（Printing）

在 Qt 中，打印与在 `QWidget`，`QPixmap` 或者 `QImage` 绘图很相似，一般步骤如下：

- 1、创建绘图设备的 `QPrinter`；
- 2、弹出打印对话框，`QPrintDialog`，允许用户选择打印机，设置属性等；
- 3、创建一个 `QPrinter` 的 `QPainter`；
- 4、用 `QPainter` 绘制一页；
- 5、调用 `QPrinter::newPage()`，然后绘制下一页；
- 6、重复步骤 4，5，直到打印完所有页。

在 Windows 和 Mac OS X 平台，`QPrinter` 使用系统的打印驱动程序。在 Unix 上，`QPrinter` 生成脚本并把脚本发送给 `lp` 或者 `lpr`（或者发送给程序，打印程序有函数 `QPrinter::setPrintProgram()`）。调用 `QPrinter::setOutputFormat(QPrinter::PdfFormat)` `QPrinter` 也可以生成 PDF 文件。

Figure 8.12. Printing a QImage



首先看一个简单的例子，打印一个 QImage 到一页纸上。

```
void PrintWindow::printImage(const QImage &image)
{
    QPrintDialog printDialog(&printer, this);
    if (printDialog.exec()) {
```

```

    QPainter painter(&printer);
    QRect rect = painter.viewport();
    QSize size = image.size();
    size.scale(rect.size(), Qt::KeepAspectRatio);
    painter.setViewport(rect.x(), rect.y(),
                        size.width(), size.height());
    painter.setWindow(image.rect());
    painter.drawImage(0, 0, image);
}
}

```

这里，我们假设了在 `PrintWindow` 类有一个 `QPrinter` 类型的成员变量 `printer`。当然在 `printImage()` 函数的堆上我们也可以创建一个 `QPrinter`，但是这样不能记录用户进行打印时的设置

创建 `QPrintDialog`，调用 `exec()` 显示出来，如果用户点击了 `OK` 返回 `true`，否则返回 `false`。调用 `exec()` 后，`QPrinter` 对象就可以使用了。（也可以不显示 `QPrintDialog`，直接调用 `QPrinter` 的成员函数进行复制也可以）

然后，我们创建 `QPainter`，绘图设备为 `QPrinter`。设置窗口为所显示图形的矩形，视图口也同样比例，然后在 `(0, 0)` 绘制图像。

通常，`QPainter` 的窗口自动进行了初始化，打印机和屏幕有着一致的分辨率（一英寸有 72 到 100 个点），使控件的打印代码能够重用。在上面的函数中，我们自己设置来 `QPainter` 的窗口。

在一页中进行打印很简单，但是，很多应用程序需要打印多页。这时我们一次打印一页，然后调用 `newPage()` 打印另一页。这里需要解决一个问题是要确定一页打印多少内容。在 `Qt` 中有两种方法处理多页的打印文档：

- 1、我们可以把数据转换为 `HTML` 格式，使用 `QTextDocument` 描述他们，`QTextDocument` 是 `Qt` 的多文本引擎。
- 2、手动进行分页

下面我们来分别看一下这两种方法。第一个例子，我们想打印一个花卉的指导：一列为花的名字，另一列为文本描述。每一条的文本格式存储为：“名称：描述”。例如：**Miltonopsis santanae**: A most dangerous orchid species.

由于每一种花卉的数据都可以用一个字符串表示，我们可以用 `QStringList` 表示所有花卉的数据。下面的代码为使用 Qt 的多文本引擎进行打印的例子：

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
    QString html;
    foreach (QString entry, entries) {
        QStringList fields = entry.split(": ");
        QString title = Qt::escape(fields[0]);
        QString body = Qt::escape(fields[1]);
        html += "<table width=\"100%\" border=1 cellpadding=0>\n"
            "<tr><td bgcolor=\"lightgray\"><font size=\"+1\">"
            "<b><i>" + title + "</i></b></font>\n<tr><td>" + body
            + "\n</table>\n<br>\n";
    }
    printHtml(html);
}
```

首先把 `QStringList` 转换为 HTML。每一种花卉为 HTML 表格中的一行，调用 `Qt::escape()` 将特殊字符‘&’，‘>’，‘<’等用相应的 HTML 字符表示（‘amp’，‘>’，‘<’），然后调用 `printHtml()` 打印文本：

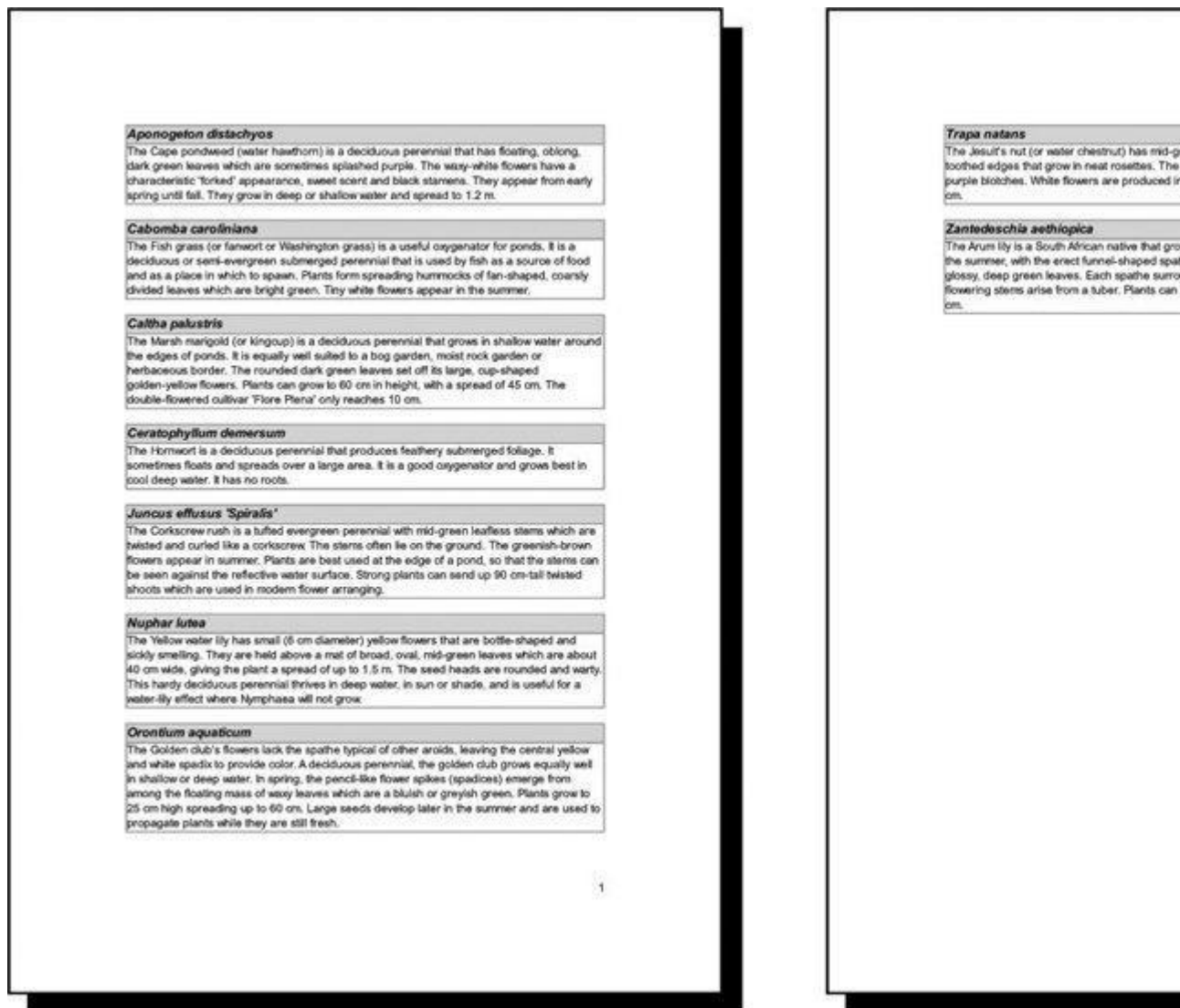
```
void PrintWindow::printHtml(const QString &html)
{
    QPrintDialog printDialog(&printer, this);
    if (printDialog.exec()) {
        QPainter painter(&printer);
        QTextDocument textDocument;
        textDocument.setHtml(html);
        textDocument.print(&printer);
    }
}
```



```
}
}
```

函数 `printHtml()` 弹出 `QPrintDialog` 对话框，负责打印一个 HTML 文档。这些代码可以在所有 Qt 的应用程序中打印任意 HTML 文档。

Figure 8.13. Printing a flower guide using QTextdocument



目前，把文本转换为 HTML 文档用 `QTextDocument` 打印是最方便的一个方法。如果需要更多的设置，就需要我们自己进行页面布局和绘制。下面的方法就是用人工干预的方式打印花卉指南。首先看一下 `printFlowerGuide()` 函数：

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
```

```

{
    QPrintDialog printDialog(&printer, this);
    if (printDialog.exec()) {

        QPainter painter(&printer);
        QList<QStringList> pages;
        paginate(&painter, &pages, entries);
        printPages(&painter, pages);
    }
}

```

在创建 `QPainter`，设置好打印机以后，调用函数 `paginate()` 确定那些项目在那一页。执行这个函数的结果是得到一个 `QStringList` 的列表，每一个 `QStringList` 在一页里显示，把这个结果传递给 `printPages()` 进行打印。

例如：需要打印的花卉指南有 6 个条目：A,B,C,D,E,F。其中 A 和 B 在第一页，C,D,E 打印在第二页，F 在第三页打印。

```

void PrintWindow::paginate(QPainter *painter, QList<QStringList> *pages,
                           const QStringList &entries)
{
    QStringList currentPage;
    int pageHeight = painter->window().height() - 2 * LargeGap;
    int y = 0;
    foreach (QString entry, entries) {
        int height = entryHeight(painter, entry);
        if (y + height > pageHeight && !currentPage.empty()) {
            pages->append(currentPage);
            currentPage.clear();
            y = 0;
        }
        currentPage.append(entry);
        y += height + MediumGap;
    }
}

```

```

    }
    if (!currentPage.empty())
        pages->append(currentPage);
}

```

函数 `paginate()` 把花会指南条目分页。根据 `entryHeight()` 计算每一个条目的高度。同时考虑页面顶端和底端的垂直距离 `LargeGap`。

遍历所有的条目, 如果这个条目可以放在当前页, 就把这个条目放到当前页的列表里面。当前页排满后, 把当前页放到页的列表中, 开始新的一页。

```

int PrintWindow::entryHeight(QPainter *painter, const QString &entry)
{
    QStringList fields = entry.split(": ");
    QString title = fields[0];
    QString body = fields[1];

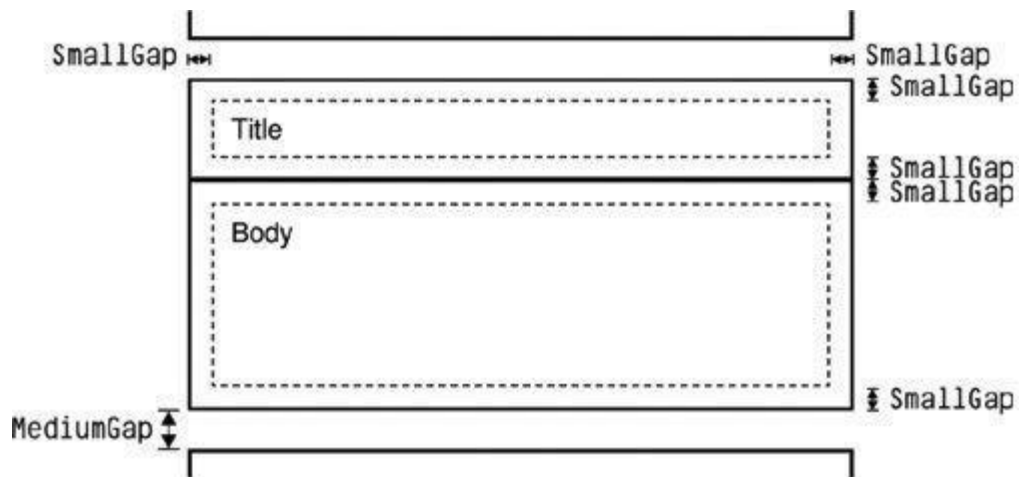
    int textWidth = painter->window().width() - 2 * SmallGap;
    int maxHeight = painter->window().height();
    painter->setFont(titleFont);
    QRect titleRect = painter->boundingRect(0, 0, textWidth, maxHeight,
                                           Qt::TextWordWrap, title);
    painter->setFont(bodyFont);
    QRect bodyRect = painter->boundingRect(0, 0, textWidth, maxHeight,
                                           Qt::TextWordWrap, body);
    return titleRect.height() + bodyRect.height() + 4 * SmallGap;
}

```

函数 `entryHeight()` 根据 `QPainter::boundingRect()` 计算每一个条目的垂直距离, 图 8.4 表明了条目的布局和 `SmallGap` 还 `MediumGap` 的含义:

The `entryHeight()` function uses `QPainter::boundingRect()` to compute the vertical space needed by one entry. [Figure 8.14](#) shows the layout of a flower entry and the meaning of the `SmallGap` and `MediumGap` constants.

Figure 8.14. A flower entry's layout



```
void PrintWindow::printPages(QPainter *painter,  
                             const QList<QStringList> &pages)  
{  
    int firstPage = printer.fromPage() - 1;  
    if (firstPage >= pages.size())  
        return;  
    if (firstPage == -1)  
        firstPage = 0;  
    int lastPage = printer.toPage() - 1;  
    if (lastPage == -1 || lastPage >= pages.size())  
        lastPage = pages.size() - 1;  
    int numPages = lastPage - firstPage + 1;  
    for (int i = 0; i < printer.numCopies(); ++i) {  
        for (int j = 0; j < numPages; ++j) {  
            if (i != 0 || j != 0)  
                printer.newPage();  
            int index;  
            if (printer.pageOrder() == QPainter::FirstPageFirst) {  
                index = firstPage + j;
```

```

    } else {

        index = lastPage - j;

    }

    printPage(painter, pages[index], index + 1);

}

}

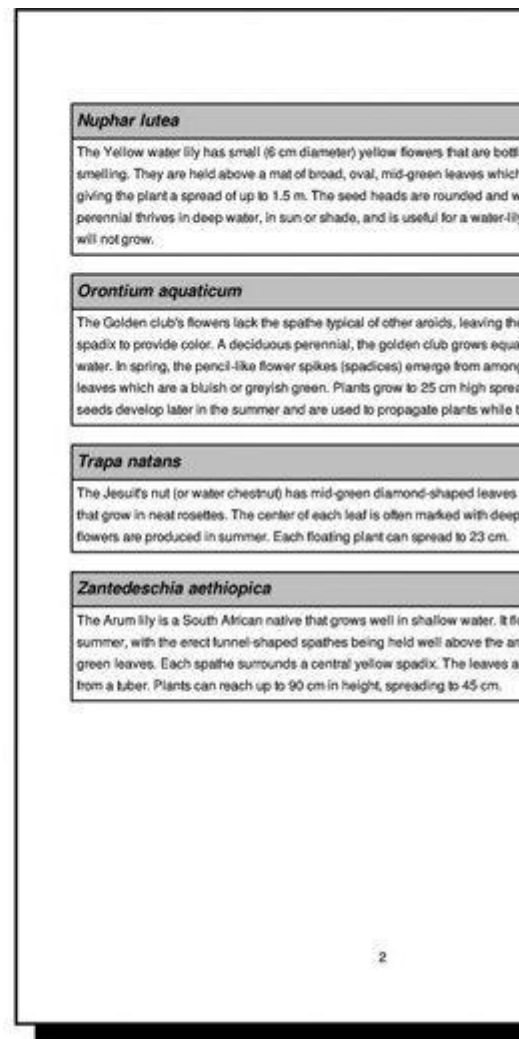
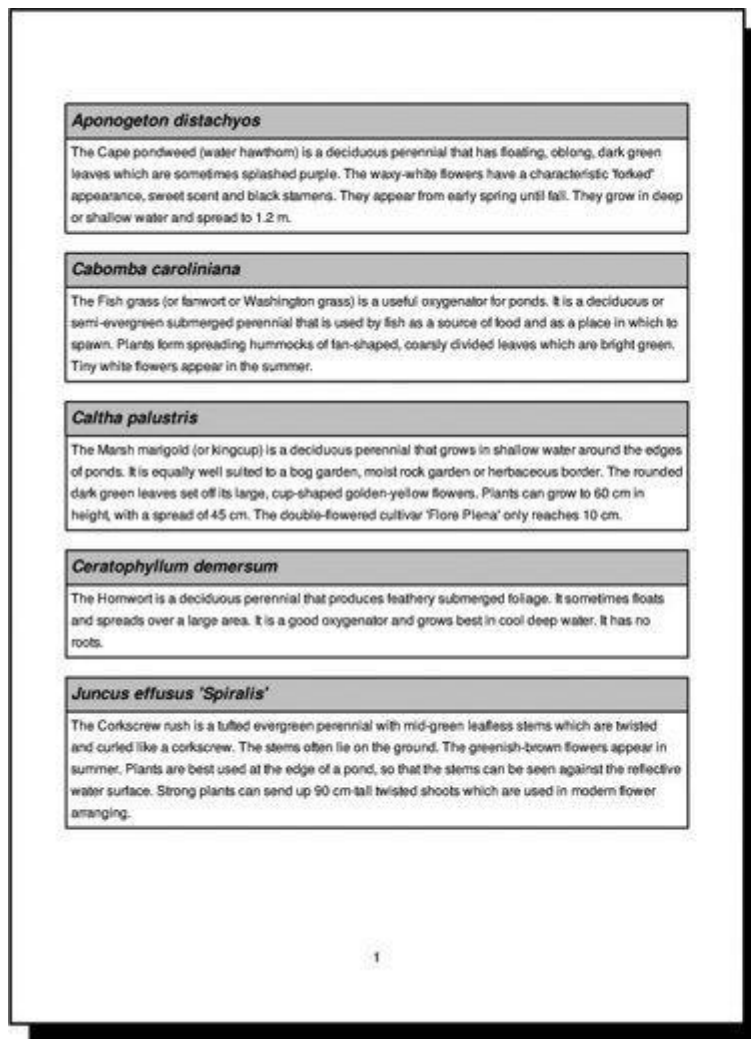
}

```

函数 `printPages()` 的作用是调用 `printPage()` 按照顺序和打印份数打印每一页。通过 `QPrintDialog`，用户可能需要打印多份，设置了打印范围，或者要求逆序打印。我们需要在程序中考考虑这些需求

首先确定打印范围。`QPrinter::fromPage()` 和 `toPage()` 返回用户选择的页面范围。如果没有选择，返回为 0。我们进行了减 1 操作是因为我们的页面索引是从 0 开始的。如果用户没有选定范围，则打印全部，`firstPage` 和 `lastPage` 包含量所有的页面。然后我们打印每一页。最外层循环为用户设定的打印的份数。对于那些支持多份打印的打印机，`QPrinter::numCopies()` 总是返回 1。如果打印机驱动程序不支持多份打印，`numCopies()` 返回到是用户指定的打印份数，有应用程序实现多份打印。（在这一节的 `QImage` 例子中，为了简单起见，我们没有考虑多份打印。）

Figure 8.15. Printing a flower guide using QPainter



内层循环遍历打印的页数。如果页数不是第一页，调用 `newPage()` 清楚原来的页面开始填充新页面。调用 `printPage()` 打印每一页。

```
void PrintWindow::printPage(QPainter *painter,
                           const QStringList &entries, int pageNumber)
{
    painter->save();
    painter->translate(0, LargeGap);
    foreach (QString entry, entries) {
        QStringList fields = entry.split(": ");
        QString title = fields[0];
        QString body = fields[1];
        printBox(painter, title, titleFont, Qt::lightGray);
        printBox(painter, body, bodyFont, Qt::white);
    }
}
```

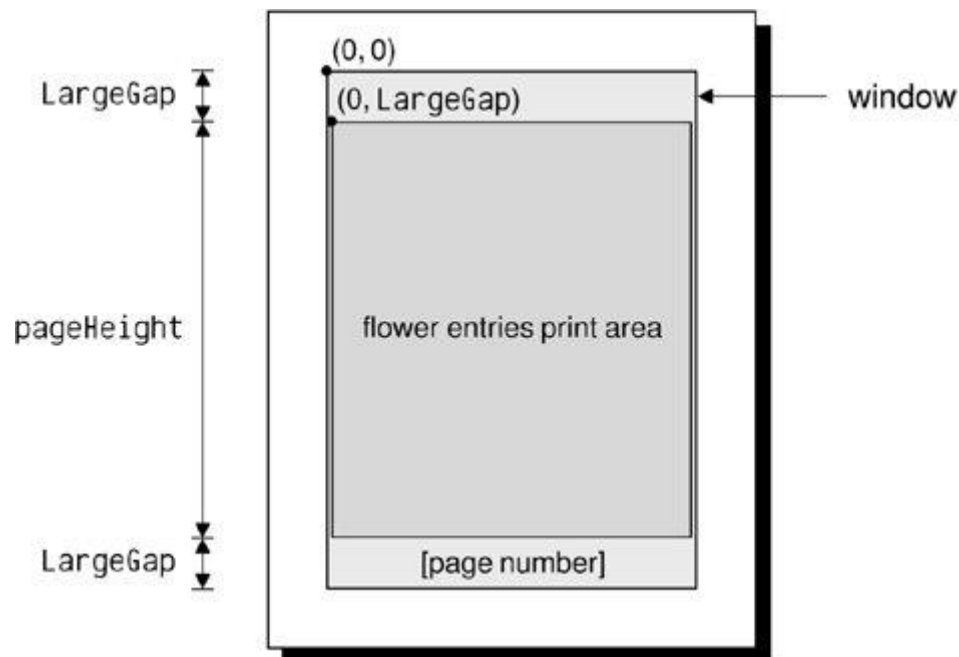
```

        painter->translate(0, MediumGap);
    }
    painter->restore();
    painter->setFont(footerFont);
    painter->drawText(painter->window(),
        Qt::AlignHCenter | Qt::AlignBottom,
        QString::number(pageNumber));
}

```

函数 `printPage()` 打印页面中的每一个条目。首先用 `printBox()` 打印标题，然后用 `printBox()` 打印描述。在每一页的底端打印页码。

Figure 8.16. The flower guide's page layout



```

void PrintWindow::printBox(QPainter *painter, const QString &str,
    const QFont &font, const QBrush &brush)
{
    painter->setFont(font);
    int boxWidth = painter->window().width();
    int textWidth = boxWidth - 2 * SmallGap;
    int maxHeight = painter->window().height();

```

```

QRect textRect = painter->boundingRect(SmallGap, SmallGap,
                                       textWidth, maxHeight,
                                       Qt::TextWordWrap, str);

int boxHeight = textRect.height() + 2 * SmallGap;
painter->setPen(QPen(Qt::black, 2, Qt::SolidLine));
painter->setBrush(brush);
painter->drawRect(0, 0, boxWidth, boxHeight);
painter->drawText(textRect, Qt::TextWordWrap, str);
painter->translate(0, boxHeight);
}

```

`printBox()`首先绘制一个矩形框，然后在矩形框中绘制文本。

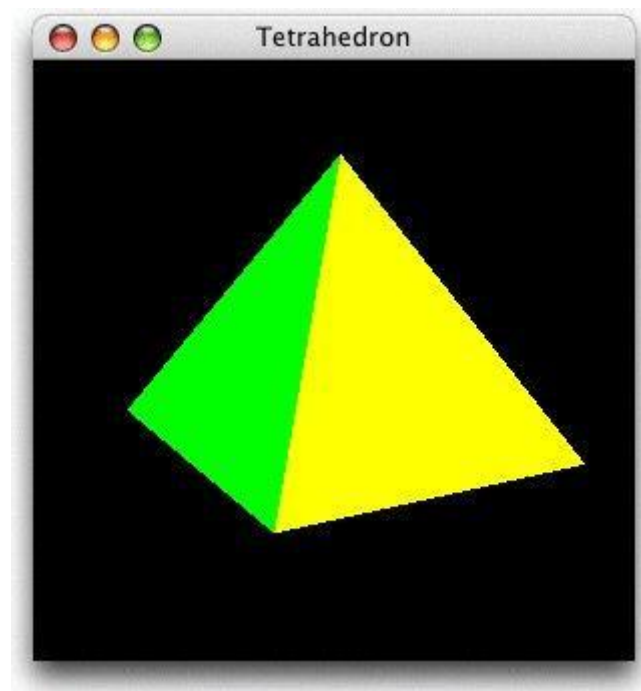
8-5 用 OpenGL 绘图（Graphics with OpenGL）

OpenGL 是绘制 2D 和 3D 模型的标准 API。在 OpenGL 基础上，Qt 可以使用 QtOpenGL 模块绘制 3D 图形。本节假设您已经熟悉 OpenGL。如果对 OpenGL 不了解，可以浏览 <http://www.opengl.org/>。

在 Qt 应用程序中使用 OpenGL 绘图非常简单：我们需要从 QGLWidget 继承自己的控件类，实现一些虚函数，连接到 QtOpenGL 和 OpenGL 库。因为 QGLWidget 从 QWidget 继承，我们以前学习的控件内容仍然适用。主要区别是我们使用 OpenGL 函数绘图而不是使用 QPainter。

为了说明 OpenGL 的工作方式，我们查看图 8.17 所示的四面体程序。这个程序显示了一个 3D 的四面体，每一个面都由不同的颜色显示。用户可以通过鼠标点击或者托拽进行旋转。双击一个面，会弹出 QColorDialog，选择一个其他的颜色。

Figure 8.17. The Tetrahedron application



```
class Tetrahedron : public QGLWidget
{
    Q_OBJECT

public:
    Tetrahedron(QWidget *parent = 0);

protected:
    void initializeGL();

    void resizeGL(int width, int height);

    void paintGL();

    void mousePressEvent(QMouseEvent *event);

    void mouseMoveEvent(QMouseEvent *event);
```

```

        void mouseDoubleClickEvent(QMouseEvent *event);

private:

        void draw();

        int faceAtPosition(const QPoint &pos);

        GLfloat rotationX;

        GLfloat rotationY;

        GLfloat rotationZ;

        QColor faceColors[4];

        QPoint lastPos;

};

```

类 Tetrahedron 继承自 QGLWidget, 函数 initializeGL(), resizeGL() 和 paintGL() 是从 QGLWidget 继承的虚函数。鼠标事件的处理函数是从 QWidget 继承。

```

Tetrahedron::Tetrahedron(QWidget *parent)

    : QGLWidget(parent)

{

    setFormat(QGLFormat(QGL::DoubleBuffer | QGL::DepthBuffer));

    rotationX = -21.0;

    rotationY = -57.0;

    rotationZ = 0.0;

```

```

        faceColors[0] = Qt::red;

        faceColors[1] = Qt::green;

        faceColors[2] = Qt::blue;

        faceColors[3] = Qt::yellow;

    }

```

在构造函数中，调用 `QGLWidget::setFormat()` 确定 OpenGL 的显示方式。然后初始化类的私有函数。

```

void Tetrahedron::initializeGL()

{

    qglClearColor(Qt::black);

    glShadeModel(GL_FLAT);

    glEnable(GL_DEPTH_TEST);

    glEnable(GL_CULL_FACE);

}

```

函数 `initializeGL()` 在 `paintGL()` 之前调用，且只调用一次，在这里可以设置 OpenGL 的显示内容，定义显示列表或者其他初始化操作。

其中 `qglClearColor()` 是 `QGLWidget` 的函数，其他函数都是 OpenGL 标准函数。如果全部遵循 OpenGL 库，可以调用 RGBA 格式的 `glClearColor()` 函数和颜色索引函数 `glClearIndex()`。

```

void Tetrahedron::resizeGL(int width, int height)

{

```

```

glViewport(0, 0, width, height);

glMatrixMode(GL_PROJECTION);

glLoadIdentity();

GLfloat x = GLfloat(width) / height;

glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);

glMatrixMode(GL_MODELVIEW);

}

```

函数 `resizeGL()` 在 `paintGL()` 之前开始调用，在任何时候只要控件大小改变，都会调用这个函数。在这个函数中可以设置 OpenGL 的视口，投影和其他与控件大小有关的设置。

```

void Tetrahedron::paintGL()

{

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    draw();

}

```

函数 `paintGL()` 在控件需要重新绘制时调用，和 `QWidget` 控件的 `paintEvent()` 函数相似，不同的地方是绘制 `OpenGL` 控件时，使用 `OpenGL` 函数。实际的绘制由私有函数 `draw()` 实现。

```

void Tetrahedron::draw()

{

    static const GLfloat P1[3] = { 0.0, -1.0, +2.0 };

```

```

static const GLfloat P2[3] = { +1.73205081, -1.0, -1.0 };

static const GLfloat P3[3] = { -1.73205081, -1.0, -1.0 };

static const GLfloat P4[3] = { 0.0, +2.0, 0.0 };

static const GLfloat * const coords[4][3] = {

    { P1, P2, P3 }, { P1, P3, P4 }, { P1, P4, P2 }, { P2, P4, P3 }

};

glMatrixMode(GL_MODELVIEW);

glLoadIdentity();

glTranslatef(0.0, 0.0, -10.0);

glRotatef(rotationX, 1.0, 0.0, 0.0);

glRotatef(rotationY, 0.0, 1.0, 0.0);

glRotatef(rotationZ, 0.0, 0.0, 1.0);

for (int i = 0; i < 4; ++i) {

    glLoadName(i);

    glBegin(GL_TRIANGLES);

    glColor(faceColors[i]);

    for (int j = 0; j < 3; ++j) {

        glVertex3f(coords[i][j][0], coords[i][j][1],

            coords[i][j][2]);

    }

```

```

        glEnd();

    }

}

```

在函数 `draw()` 中，我们参照 `x`，`y`，`z` 的坐标和 `faceColor` 中的颜色，绘制了这个四面体。除了 `glColor()`，其他所有的函数都是调用 OpenGL 库。我们可以根据 OpenGL 模式使用 `glColor3d()` 或者 `glIndex()` 代替

```

void Tetrahedron::mousePressEvent(QMouseEvent *event)

{

    lastPos = event->pos();

}

void Tetrahedron::mouseMoveEvent(QMouseEvent *event)

{

    GLfloat dx = GLfloat(event->x() - lastPos.x()) / width();

    GLfloat dy = GLfloat(event->y() - lastPos.y()) / height();

    if (event->buttons() & Qt::LeftButton) {

        rotationX += 180 * dy;

        rotationY += 180 * dx;

        updateGL();

    } else if (event->buttons() & Qt::RightButton) {

        rotationX += 180 * dy;

```

```

        rotationZ += 180 * dx;

        updateGL();

    }

    lastPos = event->pos();

}

```

函数 `mousePressEvent()` 和 `mouseMoveEvent()` 是对 `QWidget` 类的重写, 使用户通过鼠标点击或者拖动实现旋转。点击鼠标左键则沿 `x` 轴和 `y` 轴方向旋转, 点击右键沿 `x` 轴和 `z` 轴旋转。

在修改了 `rotationX` 变量, `rotationY` 变量或者 `rotationZ` 变量后, 调用 `updateGL()` 重新绘制控件。

```

void Tetrahedron::mouseDoubleClickEvent(QMouseEvent *event)
{
    int face = faceAtPosition(event->pos());

    if (face != -1) {
        QColor color = QColorDialog::getColor(faceColors[face], this);

        if (color.isValid()) {
            faceColors[face] = color;

            updateGL();
        }
    }
}

```

```
}
```

函数 `mouseDoubleClickEvent()` 重写了 `QWidget` 的同名函数，允许用户双击控件设置四面体的一个面的颜色。私有函数 `faceAtPosition()` 得到鼠标双击位置所在四面体的那个面，如果双击了某一个面，调用 `QColorDialog::getColor()` 得到一个面的新的颜色。然后更新变量 `faceColors` 数组，调用 `updateGL()` 重新绘制控件。

```
int Tetrahedron::faceAtPosition(const QPoint &pos)

{

    const int MaxSize = 512;

    GLuint buffer[MaxSize];

    GLint viewport[4];

    glGetIntegerv(GL_VIEWPORT, viewport);

    glSelectBuffer(MaxSize, buffer);

    glRenderMode(GL_SELECT);

    glInitNames();

    glPushName(0);

    glMatrixMode(GL_PROJECTION);

    glPushMatrix();

    glLoadIdentity();

    gluPickMatrix(GLdouble(pos.x()), GLdouble(viewport[3] - pos.y()),

                  5.0, 5.0, viewport);

    GLfloat x = GLfloat(width()) / height();
```



```

        glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);

        draw();

        glMatrixMode(GL_PROJECTION);

        glPopMatrix();

        if (!glRenderMode(GL_RENDER))

            return -1;

        return buffer[3];

    }

```

函数 `faceAtPosition()` 返回控件上某一个位置所在的平面号，如果没有在平面上则返回 -1。使用 OpenGL 实现代码有些复杂。实际上，我们用 `GL_SELECT` 模式绘制四面体，利用 OpenGL 的点获取功能，然后得到平面号。

下面是 `main.cpp` 的实现代码：

```

#include <QApplication>

#include <iostream>

#include "tetrahedron.h"

using namespace std;

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

{

    QApplication app(argc, argv);

```

```

if (!QGLFormat::hasOpenGL()) {

    cerr << "This system has no OpenGL support" << endl;

    return 1;

}

Tetrahedron tetrahedron;

tetrahedron.setWindowTitle(QObject::tr("Tetrahedron"));

tetrahedron.resize(300, 300);

tetrahedron.show();

return app.exec();

}

```

如果用户的系统不支持 OpenGL，在控制台上打印一条错误消息然后退出。

在.pro 文件中，需要应用程序连接到 QtOpenGL 库：

```

QT += opengl

```

到现在，这个四面体的程序就全部完成了。如果了解更多的 QtOpenGL，可以查看参考文档中 QGLWidget, QGLFormat, QGLContext 和 QGLPixelBuffer。

9-1 支持托拽功能（Enabling Drag and Drop）

名词解释：

Drag: 拖动，拉动，计算机中就是用鼠标拖动的过程

Drop: 滴下，落下，松开，计算机中就是鼠标拖动到某一个位置后放开左键的过程，Drag 后总会 Drop 的。

Drag 和 Drop 是两个完全不同的动作。Qt 中的控件可以作为拖动（drag）的地点，也可以作为松开（drop）的地点，或者同时作为拖动和松开的地点。

第一个例子用来说明一个 Qt 应用程序接受 另一个程序触发的拖动事件。Qt 应用程序为一个 QTextEdit 为中央控件的窗口。当用户从桌面或者一个文件浏览器中拖动一个文本文件到 Qt 程序时，程序把文件显示在 QTextEdit 控件中。

下面是主窗口的定义

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow();
protected:
    void dragEnterEvent(QDragEnterEvent *event);
    void dropEvent(QDropEvent *event);
private:
    bool readFile(const QString &fileName);
    QTextEdit *textEdit;
};
```

在 MainWindow 类中，重新实现了 QWidget 的函数 dragEnterEvent() 和 dropEvent()。由于这个例子主要用来显示托拽，主窗口的很多其他功能都省略了。

```
MainWindow::MainWindow()
{
```

```

    textEdit = new QTextEdit;
    setCentralWidget(textEdit);
    textEdit->setAcceptDrops(false);
    setAcceptDrops(true);
    setWindowTitle(tr("Text Editor"));
}

```

在构造函数中，我们创建了一个 `QTextEdit` 控件，并设置为中央控件。缺省情况下，`QTextEdit` 接受来自其他应用程序拖拽来的文本，把文件名显示出来。由于 `drop` 事件是由子控件向父控件传播的，通过禁止 `QTextEdit` 控件的 `drop`，允许主窗口得到 `drop` 事件，我们就得到了 `MainWindow` 的 `drop` 事件。

```

void MainWindow::dragEnterEvent(QDragEnterEvent *event)
{
    if (event->mimeTypeData()->hasFormat("text/uri-list"))
        event->acceptProposedAction();
}

```

任何时候用户拖动一个对象到一个控件上，函数 `dragEnterEvent()` 都会被调用。如果在这个事件处理函数中调用函数 `acceptProposeAction()`，说明我们允许用户这个对象拖拽到这个控件上。通常，控件不允许 `drag`。Qt 会自动改变光标状态指示用户当前的控件是否是一个合法的 `drop` 地点。

在这里我们只允许用户 `drag` 一个文本文件，因此，我们检查这个这个 `dravg` 的 `MIME` 类型。`MIME` 类型 `text/uri-list` 用来保存 URL 的一个地址列表，可以使一个文本文件，也可以是其他 URL（HTTP 或者 FTP 路径），也可以是其他的全局资源标识。标准的 `MIME` 类型由 IANA（Internet Assigned Numbers Authority）定义，由一个类型名/子类型名组成。`MIME` 类型用于在剪贴板和拖拽时区别不同的数据类型，`MIME` 类型列表可以点击访问 <http://www.iana.org/assignments/media-types/> 得到

```

void MainWindow::dropEvent(QDropEvent *event)
{
    QList<QUrl> urls = event->mimeTypeData()->urls();
}

```

```

if (urls.isEmpty())
    return;

QString fileName = urls.first().toLocalFile();

if (fileName.isEmpty())
    return;

if (readFile(fileName))
    setWindowTitle(tr("%1 - %2").arg(fileName)
                    .arg(tr("Drag File"))));
}

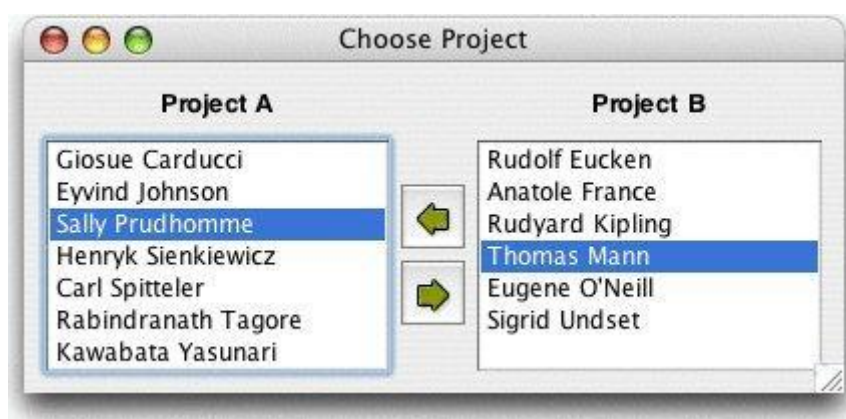
```

当用户 **drop** 一个对象到控件上，函数 `dropEvent()` 就会调用。`QMimeType::urls()` 得到一个 `QUrls` 列表。通常用户一次只会托拽一个文件，但是托拽多个文件也是允许的。如果用户拖动了多个文件或者多个 `URLs`，程序立即返回。

`QWidget` 还提供了 `dragMoveEvent()` 和 `dragLeaveEvent()`，但是对于大多数应用程序，这两个函数都不需要重写。

第二个例子来说明怎样进行 **drag**，怎样接受 **drop**。我们将会创建一个 `QListWidget` 子类，这个控件可以接受 **drag** 和 **drop**。并把它作为 `Project Chooser` 程序的一个组件，如 9.1 所示：

Figure 9.1. The Project Chooser application



`Project Chooser` 程序由两个名字列表控件组成。每一个列表控件表示一个项目。用户可以 **drag** 和 **drop** 列表中的名字，把一个名字从一个项目移到另一个项目中。

在列表控件的子类中实现了 **drag** 和 **drop** 部分的代码。下面是类的定义：

```

class ProjectListWidget : public QListWidget

```

```

{
    Q_OBJECT
public:
    ProjectListWidget(QWidget *parent = 0);
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void dragEnterEvent(QDragEnterEvent *event);
    void dragMoveEvent(QDragMoveEvent *event);
    void dropEvent(QDropEvent *event);
private:
    void startDrag();
    QPoint startPos;
};

ProjectListWidget::ProjectListWidget(QWidget *parent)
    : QListWidget(parent)
{
    setAcceptDrops(true);
}

```

在构造函数中，我们让列表控件允许 **drop**。

```

void ProjectListWidget::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton)
        startPos = event->pos();
    QListWidget::mousePressEvent(event);
}

```

当用户点击了鼠标左键时，我们把鼠标位置保存在 **startPos** 变量中。然后调用基类的 **mousePressEvent()**，使列表控件的鼠标点击事件进入程序的消息循环。

```

void ProjectListWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance())
            startDrag();
    }
    QListWidget::mouseMoveEvent(event);
}

```

如果用户点中鼠标左边同时移动鼠标，把这个过程认为是一个 **drag**。计算当前鼠标位置和鼠标第一次点击的位置之间的距离，如果这个距离大于 **QApplication** 认定的拖动的最小距离（通常为四个像素），调用私有函数 **startDrag()** 开始拖动。通过判断距离可以避免因为用户手抖动引起的误操作。

```

void ProjectListWidget::startDrag()
{
    QListWidgetItem *item = currentItem();
    if (item) {
        QMimeData *mimeData = new QMimeData;
        mimeData->setText(item->text());
        QDrag *drag = new QDrag(this);
        drag->setMimeData(mimeData);
        drag->setPixmap(QPixmap(":/images/person.png"));
        if (drag->start(Qt::MoveAction) == Qt::MoveAction)
            delete item;
    }
}

```

在 **startDrag()** 中，我们创建一个 **QDrag** 对象，父类为当前的列表控件 **ProjectListWidget**。**QDrag** 对象保存量一个 **QMimeData** 对象中的数据。在这个例子中，

我们只是使用 `QMimeData::setText()` 保存了一个文本。`QMimeData` 提供了很多函数处理经常用到的数据类型（如图像，URLs，颜色等等），对于用 `QByteArrays` 表示的任意 MIME 类型都可以处理。当 `drag` 发生时，函数 `QDrag::setPixmap()` 设置了跟随鼠标的图标。

调用 `QDrag::start()` 开始拖动，直到用户开始 `drop` 或者取消了拖动。函数的参数为多个“drag actions”的组合（`Qt::CopyAction`, `Qt::MoveAction`, `Qt::LinkAction`）。返回值为执行拖动的“drag action”，如果没有执行拖动的操作，则返回

`Qt::IgnoreAction`。具体执行那个 action 取决于原控件允许的操作，目标控件允许的操作和是否有附加键同时按下。调用 `start()` 后，Qt 拥有被拖动的对象，在不需要时将其删除。

```
void ProjectListWidget::dragEnterEvent(QDragEnterEvent *event)
{
    ProjectListWidget *source =
        qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

`ProjectListWidget` 不但可以产生 `drag`，还可以接受来自程序中其他 `ProjectListWidget` 控件的 `drag`。如果 `drag` 发生在同一个应用程序中，`QDragEnterEvent::source()` 得到产生 `drag` 控件的指针。如果不是一个程序，则返回空指针。`qobject_cast<T>()` 可以确保拖动来自与一个 `ProjectListWidget` 控件。如果一切正常，则调用 `accept()` 通知 Qt 接受这个 action 作为一个移动操作

```
void ProjectListWidget::dragMoveEvent(QDragMoveEvent *event)
{
    ProjectListWidget *source =
        qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
    }
}
```



```

        event->accept();
    }
}

```

函数 `dragMoveEvent()` 和 `dragEnterEvent()` 很相似。

```

void ProjectListWidget::dropEvent(QDropEvent *event)
{
    ProjectListWidget *source =
        qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        addItem(event->mimeType()->text());
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}

```

在 `dropEvent()` 中，用 `QMimeType::text()` 得到拖动的文本，并用这个文本创建一个列表项目。我们还需要调用 `event->setDropAction(Qt::MoveAction)`，用参数 `Qt::MoveAction` 通知源控件可以删除原来拖动的项目。

在程序间需要转移数据时，托拽是一个非常有用的机制。但是有时候不用托拽机制也可以实现托拽同样的操作。如果我们只是想在同一个控件中移动数据，只需要重写 `mousePressEvent()` 和 `mouseReleaseEvent()` 就可以。

9-2 支持自定义数据类型的托拽（Supporting Custom Drag Types）

在上一节的例子中，我们使用类 `QMimeType` 表示普通的 MIME 类型。调用 `QMimeType::setText()` 进行对文本的托拽，调用 `QMimeType::urls()` 得到托拽到 url 文本列表。如果我们想托拽普通的文本，HTML 文本，图片，URLs，颜色等，都可以使用 `QMimeType` 类。但是 如果我们需要对自定义的类型进行托拽，就需要使用下面的方法：

1. 用 `QByteArray` 表示任意数据，调用函数 `QMimeType::setData()`，通过 `QMimeType::data()` 得到数据。

2. 定义 `QMimeData` 子类, 重新实现函数 `formats()` 和 `retrieveData()` 处理自定义的数据类型。

3. 对于只发生在一个应用程序内部的托拽, 定义 `QMimeData` 子类, 在这个子类中保存任意的数据类型。

第一个方法不需要继承类, 但是有些缺点: 即使托拽最终不被接受, 我们也需要把数据类型转换为 `QByteArray`, 如果我们希望提供的 **MIME** 类型能够和大量的应用程序进行交互, 我们需要把交互的数据保存多份 (每一类 **MIME** 类型保存一份)。要是交互的数据量很大, 那么程序的执行速度肯定会慢下来。第二种和第三种方法能够避免或者尽量减少这些问题。

我们用一个例子说明这些方法, 让 `QTableWidget` 控件支持托拽。能够进行拖动的数据类型为: `text/plain, text/html, text/csv`。第一种方法实现如下:

```
void MyTableWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance())
            startDrag();
    }

    QTableWidget::mouseMoveEvent(event);
}

void MyTableWidget::startDrag()
{
    QString plainText = selectionAsPlainText();
    if (plainText.isEmpty())
        return;

    QMimeData *mimeData = new QMimeData;
    mimeData->setText(plainText);
    mimeData->setHtml(toHtml(plainText));
    mimeData->setData("text/csv", toCsv(plainText).toUtf8());

    QDrag *drag = new QDrag(this);
```

```

drag->setMimeData(mimeData);

if (drag->start(Qt::CopyAction | Qt::MoveAction) == Qt::MoveAction)
    deleteSelection();
}

```

函数 `mouseMoveEvent()` 中调用 `startDrag()` 开始进行拖动。我们调用函数 `setText()`, `setHtml()` 设置 MIME 类型为 `text/plain`, `text/html` 的数据, 函数 `setData()` 保存 `text/csv` 类型的数据, `setData()` 把任意的 MIME 数据类型保存为一个 `QByteArray` 类型的数据。函数 `selectionAsString()` 的实现与第四章中 `Spreadsheet::copy()` 相似。

```

QString MyTableWidget::toCsv(const QString &plainText)

```

```

{
    QString result = plainText;
    result.replace("\\", "\\");
    result.replace("\"", "\\");
    result.replace("\t", "\\t");
    result.replace("\n", "\\n");
    result.prepend("\"");
    result.append("\"");
    return result;
}

```

```

QString MyTableWidget::toHtml(const QString &plainText)

```

```

{
    QString result = Qt::escape(plainText);
    result.replace("\t", "<td>");

    result.replace("\n", "\n<tr><td>");
    result.prepend("<table>\n<tr><td>");
    result.append("\n</table>");
    return result;
}

```

函数 `toCsv()` 和 `toHtml()` 将 `tabs` 和 `newlines` 字符转换为 `vcsv` 格式的字符（用逗号分割数据）或者 `HTML` 字符。例如

Red	Green	Blue
Cyan	Yellow	Magenta

转换为 (**CSV**) :

```
"Red", "Green", "Blue"
"Cyan", "Yellow", "Magenta"
```

或者 (**HTML**)

```
<table>
<tr><td>Red<td>Green<td>Blue
<tr><td>Cyan<td>Yellow<td>Magenta
</table>
```

使用函数 `QString::replace()`，使转换尽可能简单。函数 `Qt::escape()` 用来解释 `HTML` 中的特殊字符。

```
void MyTableWidget::dropEvent(QDropEvent *event)
{
    if (event->mimeTypeData()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeTypeData()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        ...
        event->acceptProposedAction();
    } else if (event->mimeTypeData()->hasFormat("text/plain")) {
        QString plainText = event->mimeTypeData()->text();
        ...
        event->acceptProposedAction();
    }
}
```

虽然我们提供了三种不同的数据格式，但在 `dropEvent()` 中我们只接受了其中的两种。如果用户从 `QTableWidget` 的一个网格中拖动一串 HTML 字符到一个 HTML 编译器，则把网格中的数据转换为 HTML。如果用户拖动一个 HTML 到 `QTableWidget` 控件中，我们不想接受它。

要让这个例子顺利实现拖拽，在 `MyTableWidget` 构造函数中还要调用 `setAcceptDrops(true)` 和 `setSelectionMode(ContiguousSelection)`。现在我们重新实现这个例子，这次，我们用 `QMimeData` 作为基类，用 `QMimeData` 的子类实现数据转换，避免了 `QTableWidgetItem` 和 `QByteArray` 之间的转换。下面是子类的定义：

```
class TableMimeData : public QMimeData
{
    Q_OBJECT

public:
    TableMimeData(const QTableWidget *tableWidget,
                  const QTableWidgetSelectionRange &range);

    const QTableWidget *tableWidget() const { return myTableWidget; }
    QTableWidgetSelectionRange range() const { return myRange; }
    QStringList formats() const;

protected:
    QVariant retrieveData(const QString &format,
                          QVariant::Type preferredType)
const;

private:
    static QString toHtml(const QString &plainText);
    static QString toCsv(const QString &plainText);
    QString text(int row, int column) const;
    QString rangeAsPlainText() const;
    const QTableWidget *myTableWidget;
    QTableWidgetSelectionRange myRange;
    QStringList myFormats;
```

```
};
```

在类中，我们保存了一个 `QTableWidgetSelectionRange` 对象和一个 `QTableWidget` 指针，用来得到托拽的数据。函数 `formats()` 和 `retrieveData()` 是对 `QMimeType` 函数的重写。在构造函数中实现对私有变量的初始化。

```
TableMimeType::TableMimeType(const QTableWidget *tableWidget,
                              const
                              QTableWidgetSelectionRange &range)
{
    myTableWidget = tableWidget;
    myRange = range;
    myFormats << "text/csv" << "text/html" << "text/plain";
}

QStringList TableMimeType::formats() const
{
    return myFormats;
}
```

函数 `formats()` 返回一个能够支持的 **MIME** 类型的列表。类型之间的顺序对程序没有影响，但是把“最好”的格式放在第一个是很好的习惯。支持多种格式的程序一般使用第一个能够匹配的格式。

```
QVariant TableMimeType::retrieveData(const QString &format,
                                      QMimeType::Type preferredType) const
{
    if (format == "text/plain") {
        return rangeAsPlainText();
    } else if (format == "text/csv") {
        return toCsv(rangeAsPlainText());
    } else if (format == "text/html") {
        return toHtml(rangeAsPlainText());
    } else {
```

```

        return QMimeData::retrieveData(format, preferredType);
    }
}

```

指定一个 **MIME** 类型，函数 **retrieveData()** 把拖动的数据作为一个 **QVariant** 数据返回。参数 **format** 的值一般是 **formats()** 返回列表中之一。我们并没有做这样的假设，应用程序也不会在拖动时检查 **MIME** 类型。函数 **text()**，**html()**，**urls()**，**image-Data()**，**colorData()** 和 **data()** 都是由 **QMimeData** 提供的。

参数 **preferredType** 用来给出一个 **QVariant** 类型的建议，这里我们忽略这个参数由 **QMimeData** 处理。

```

void MyTableWidget::dropEvent(QDropEvent *event)
{
    const TableMimeData *tableData =
        qobject_cast<const TableMimeData *>(event->mimeType());
    if (tableData) {
        const QTableWidgetItem *otherTable = tableData->tableWidget();
        QTableWidgetItemSelectionRange otherRange = tableData->range();
        ...
        event->acceptProposedAction();
    } else if (event->mimeType()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeType()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        ...
        event->acceptProposedAction();
    } else if (event->mimeType()->hasFormat("text/plain")) {
        QString plainText = event->mimeType()->text();
        ...
        event->acceptProposedAction();
    }
    QTableWidgetItem::mousePressEvent(event);
}

```

函数 `dropEvent()` 和本节前面写的 `dropEvent()` 相似。在这个函数中，我们首先把 `QMimeData` 安全转换为 `TableMimeData`。如果 `qobject_cast<T>()` 返回了正确指针，说明拖拽发生在同一个应用程序中的 `MyTableWidget` 控件，我们根据 `QMimeData` 的 API 直接得到表格中拖动数据。如果没有得到正确的指针，则按照标准方式处理。

9-3 处理剪贴板 (Clipboard Handling)

很多应用程序使用 Qt 提供的剪贴板。例如：`QTextEdit` 类提供了 `cut()`、`copy()` 和 `paste()` 槽函数，也能相应键盘的快捷键。客户程序只要编写很少的代码，甚至不写代码。

如果我们正在开发自己的类，可以使用函数 `QApplication::clipboard()` 得到 Qt 的剪贴板，该函数返回的是一个 `QClipboard` 类型对象的指针。使用这个指针很容易对剪贴板进行读写访问，调用 `setText()`、`setImage()`、`setPixmap()` 把数据写到剪贴板。调用 `text()`、`image()`、`pixmap()` 得到剪贴板里的数据。在第 4 章中的 `Spreadsheet` 程序中就是一个使用剪贴板的例子。

对于有些应用程序来说，Qt 提供的剪贴板是不够用的。除了文本和图像等类型的数据，我们还想让程序支持更多类型的数据，让自己的程序和其他应用程序进行交互。这个问题和拖拽很相似，解决方法也是一样：继承类 `QMimeData`，重新实现几个虚函数。

如果我们的程序用一个 `QMimeData` 子类支持拖拽，那么这个子类可以用在剪贴板中。用函数 `setMimeData()` 把数据写到剪贴板，函数 `mimeTypeData()` 得到剪贴板的数据。

在 X11 系统，通常会点击三键鼠标的中间键完成对选择数据的粘贴操作。在 Qt 中用一个单独的“选择”剪贴板。如果一个控件要支持这种剪贴板，同时也要支持标准的剪贴板，需要在 `QClipboard` 函数调用中使用 `QClipboard::Selection` 参数，下面的函数 `mousePressEvent()` 是一个文本编辑器的鼠标相应函数，支持鼠标中键粘贴。

```
void MyTextEditor::mousePressEvent(QMouseEvent *event)
{
    QClipboard *clipboard = QApplication::clipboard();
    if (event->button() == Qt::MidButton
```



```
        && clipboard->supportsSelection()) {  
            QString text = clipboard->text(QClipboard::Selection);  
            pasteText(text);  
        }  
    }  
}
```

在 X11 中，`supportsSelection()` 返回 `true`。在其他平台上返回 `false`。

剪贴板中的数据改变时，`QClipboard` 会发出 `QClipboard::dataChanged()` 信号。

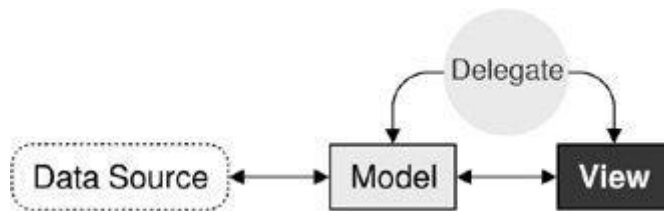
第十章 数据视图类 (Item View Classes)

很多 Qt 应用程序都允许用户查找，查看，编辑一个数据集中的一个具体数据。这些数据可能存在在一个文件中，或者在数据库中，也可能来自于网络。Qt 提供的数据视图类能很好的处理这些数据。

在 Qt 早先的版本中，数据视图控件中保存了一个数据集中的所有数据。用户在这个控件中对数据进行查找，编辑等操作，有时候还要把数据写回到数据源中。这种处理方式很容易使用，但是当数据量很大时就会影响程序性能，也不能将一个数据源用多个不同的控件表示出来。

Smalltalk 语言在处理大数据集时使用了一个更加灵活的方式：模型视图控制器 (modelviewcontroler, MVC)。在 MVC 方式中，模型 (model) 代表数据集，负责数据的获取，查看及保存。尽管每种数据集的数据模型都不同，但是模型提供的 API 对视图都是一致的。视图 (view) 把得到的数据呈现给用户，如果数据量比较大时，用户能够查看的只是全部数据的一部分，即只是视图请求查看得那部分。控制器在用户和视图之间进行协调，把用户的动作转换为对数据的查看或者编辑等操作，然后视图在把数据的变化通知模型。

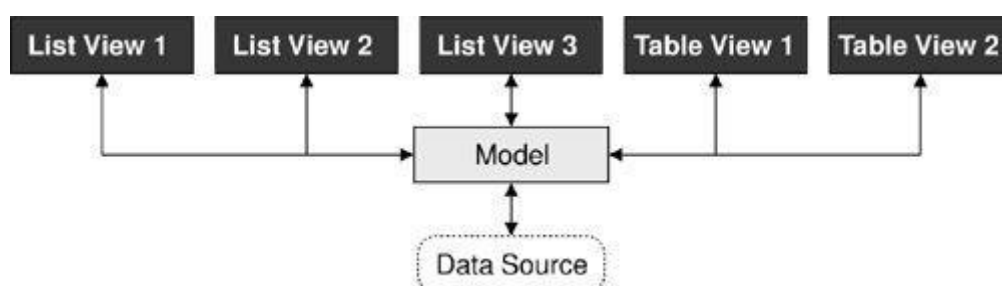
Figure 10.1. Qt's model/view architecture



Qt 提供了一个类似 MVC 的模型/视图 (model/view) 模式。在 Qt 中，模型 (model) 和 MVC 中的模型功能一样。Qt 没有使用 MVC 中的控制器 (controller)，而是使用了代理 (delegate)。代理控制对数据项的显示和编辑。Qt 为每一种视图提供了一个缺省的代理。这个代理对于大多数应用程序已经足够了，一般我们不需要太多关注它。

使用 Qt 的模型/视图结构，我们只要用模型获取需要显示的数据，再把这些数据提供给视图就可以。在处理大量数据时，这种结构能够处理的更快，内存消耗也比较小，因为不再一次显示出所有的数据了。同时，一个模型能够用一个或者多个视图显示出来，用户能够用多种方式和数据进行交换。Qt 能够自动对多个视图进行同步，将一个视图中的变化同步到其他视图中。使用模型视图结构的另一个好处是，如果我们想改变数据集，那么只需要改变模型就可以了，而不需要改变视图。

Figure 10.2. One model can serve multiple views



在很多情况下，应用程序只需要显示一小部分数据这时，可以使用 Qt 提供的方便的视图类（`QListWidget`, `QTableWidget`, `QTreeWidget`），直接在这些视图中填充数据。这些类和 Qt 的早前版本提供的类是一样的。把数据保存在数据项（item）中，如 `QTableWidget` 中包含了多个 `QTableWidgetItem`。在内部，这些类使用自定义的模型，使视图能够显示这些项目

在数据量比较多时，复制所有的数据是不明智的。这种情况下，我们可以使用 Qt 提供的视图类（`QListView`, `QTableView`, `QTreeView`）与一个数据模型共同实现。例如，如果数据集在一个数据库中，我们可以使用 `QTableView` 和 `QSqlTableModel` 显示这些数据。

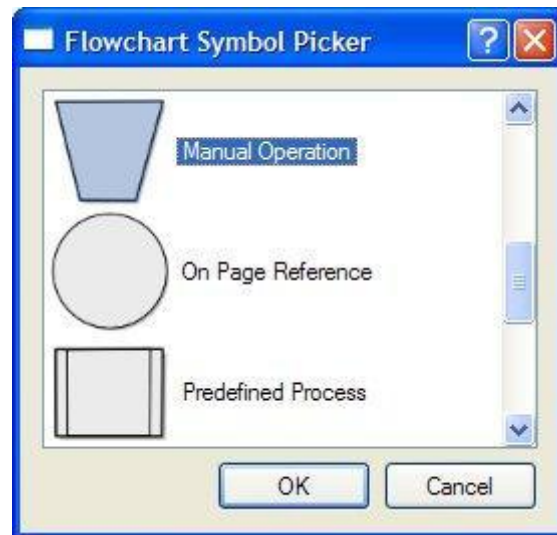
10-1 使用数据视图便捷类（Using the Item View Convenience Classes）

通常使用 Qt 提供的数据视图便捷类（item view convenience class）要比定义一个模型简单的多，适合无需分离模型和视图的操作。在第四章中我们使用了 `QTableWidget` 和 `QTableWidgetItem` 实现了表格的功能。

在这一节中，我们将介绍这些便捷类的使用。第一个例子是一个只读的 `QListWidget`，第二个例子是一个可编辑的 `QTableWidget`，第三个例子显示的是一个只读的 `QTreeWidget`。

首先我们显示一个简单的对话框，用户通过鼠标点击选中列表中的一个流程图符号，每一个项目包含一个图标，一个文字说明和一个唯一 ID。

Figure 10.3. The Flowchart Symbol Picker application



下面是头文件中声明的类：

```
class FlowChartSymbolPicker : public QDialog
{
    Q_OBJECT
public:
    FlowChartSymbolPicker(const QMap<int, QString> &symbolMap,
                           QWidget *parent = 0);

    int selectedId() const { return id; }

    void done(int result);

    ...
};
```

当我们构造对话框时，要传递一个 `QMap<int, QString>` 参数，这个参数保存了一个项目的 ID。调用函数 `selectedId()` 能够得到选中的 ID，如果用户没有选择，返回的 ID 为 -1。

```
FlowChartSymbolPicker::FlowChartSymbolPicker(
    const QMap<int, QString> &symbolMap, QWidget *parent)
    : QDialog(parent)
{
    id = -1;

    listWidget = new QListWidget;
    listWidget->setIconSize(QSize(60, 60));
    QMapIterator<int, QString> i(symbolMap);
    while (i.hasNext()) {
        i.next();

        QListWidgetItem *item = new QListWidgetItem(i.value(),
                                                    listWidget);

        item->setIcon(iconForSymbol(i.value()));
        item->setData(Qt::UserRole, i.key());
    }
    ...
}
```

我们把用户的最后一次选择的 ID 放在成员变量 `id` 中，初始化为 -1。然后创建一个 `QListWidget`，一个便捷类。我们遍历流程图符号映射中的所有项目，每一个项目用一个 `QListWidgetItem` 类对象表示。新建一个 `QListWidgetItem` 所需的第一个参数为显示的文本，另一个参数为 `QListWidget` 作为其他父容器。

然后我们设置了这个项目的图标，调用 `setData()` 函数把每一个流程符号的 ID 保存在 `QListWidgetItem` 中。函数 `iconForSymbol()` 返回表示当前流程符号的图标。

QListWidgetItem 有很多角色(role)，每一个角色由一个 QVariant 类型的数据进行读取和设置。最常用的角色是 Qt::DisplayRole, Qt::EditRole 和 Qt::IconRole。这些角色的数值可以通过 setter 和 getter 函数获得。如(setText(), setIcon()), 也有其他一些角色。我们也可以使用 Qt::UserRole 或者比这个值更大的数值定义用户自己的角色。在这个例子中，我们使用 Qt::UserRole 保存每一项的 ID。

构造函数中省略的部分主要用于创建按钮，布局和设置窗口标题。

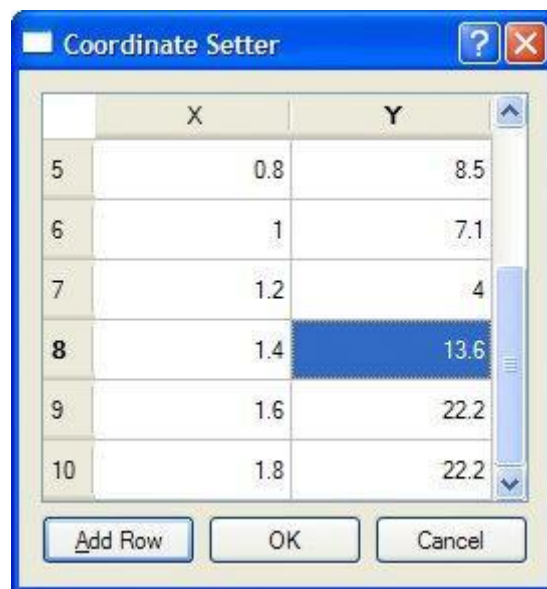
```
void FlowChartSymbolPicker::done(int result)
{
    id = -1;
    if (result == QDialog::Accepted) {
        QListWidgetItem *item = listWidget->currentItem();
        if (item)
            id = item->data(Qt::UserRole).toInt();
    }
    QDialog::done(result);
}
```

函数 done() 是对 QDialog::done() 的重写。用户点击了对话框上的确定或者取消按钮后调用这个函数。如果用户点击了确定按钮，我们调用 data() 函数得到选中项目的 ID。如果需要得到的是项目显示的文本，可以用函数 item->data(Qt::DisplayRole).toString(), 或者 item->text() 更方便。

缺省状态下，QListWidget 是只读的。如果我们希望用户能够编辑这个项目，可以调用 QAbstractItemView::setEditTriggers()。例如，设置为 QAbstractItemView::AnyKeyPressed 说明用户敲击键盘就可以进行编辑。另外我们也可以提供一个编辑按钮（也许是增加或者删除按钮），把他们的信号和槽函数关联起来，在程序中实现编辑。

现在我们已经知道怎么使用一个便捷类查看和选择数据。我们将要实现一个可以进行编辑的例子。也是用到了对话框，这个例子中要表现的是一对 (x, y) 坐标，用户对这些坐标可以编辑。

Figure 10.4. The Coordinate Setter application



和前一个例子一样，我们主要关注和视图有关的代码，首先从构造函数开始。

```
CoordinateSetter::CoordinateSetter(QList<QPointF> *coords,
                                   QWidget *parent)
    : QDialog(parent)
{
    coordinates = coords;
    tableWidget = new QTableWidgetItem(0, 2);
    tableWidget->setHorizontalHeaderLabels(
        QStringList() << tr("X") << tr("Y"));
    for (int row = 0; row < coordinates->count(); ++row) {
        QPointF point = coordinates->at(row);
        addRow();
    }
}
```

```

        tableWidget->item(row, 0)->setText(QString::number(point.x()));
        tableWidget->item(row, 1)->setText(QString::number(point.y()));
    }
    ...
}

```

QTableWidget 的构造函数中的两个初始化参数是表格的行数和列数。QTableWidget 中的每一个小格为一个 QTableWidgetItem 实例，垂直和水平表头也是如此。函数 setHorizontalHeaderLabels() 用来设置列标题。缺省情况下，QTableWidget 的每一行标题由序列号 1 开始向下排列，这已经能够满足我们的要求了，所以没有设置行标题。

设置好列标题以后，我们开始遍历所有传递来的坐标数据。对每一个 (x, y) 数据对，我们创建两个 QTableWidgetItem 实例，分别表示 x 坐标和 y 坐标值。调用函数 QTableWidgetItem::setItem() 把 QTableWidgetItem 实例添加到表格中去。

在缺省情况下，QTableWidget 是可以进行编辑的。用户可以编辑任意一个小格子中的内容，通过鼠标定位到小格子然后按 F2 或者直接输入就可以。用户进行的任何修改都会自动更新到对应的 QTableWidgetItem。如果想不允许任何编辑，可以调用函数 setEditTriggers(QAbstractItemView::NoEditTriggers)。

```

void CoordinateSetter::addRow()
{
    int row = tableWidget->rowCount();
    tableWidget->insertRow(row);

    QTableWidgetItem *item0 = new QTableWidgetItem;
    item0->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
    tableWidget->setItem(row, 0, item0);

    QTableWidgetItem *item1 = new QTableWidgetItem;
    item1->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
    tableWidget->setItem(row, 1, item1);
    tableWidget->setCurrentItem(item0);
}

```



```
}
```

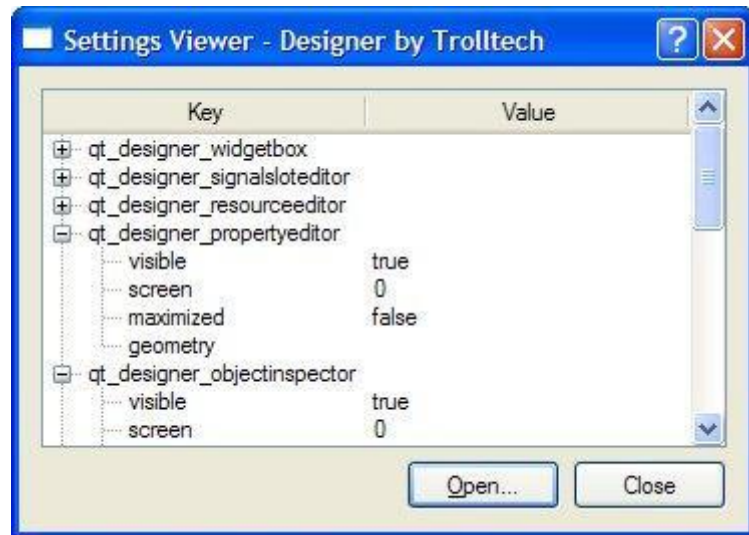
当用户点击 AddRow 按钮时，函数 `addRow()` 就会被触发。函数 `insertRow()` 往表格中加入一行。如果用户想编辑新加入行的某一个列，`QTableWidget` 会自动创建一个 `QTableWidgetItem` 实例。

```
void CoordinateSetter::done(int result)
{
    if (result == QDialog::Accepted) {
        coordinates->clear();
        for (int row = 0; row < tableWidget->rowCount(); ++row) {
            double x = tableWidget->item(row, 0)->text().toDouble();
            double y = tableWidget->item(row, 1)->text().toDouble();
            coordinates->append(QPointF(x, y));
        }
    }
    QDialog::done(result);
}
```

最后，当用户点击了 OK 按钮，我们清除对话框中的坐标值，根据 `QTableWidget` 中的数据形成一个新的坐标集合。

下面是最后一个例子，我们使用 `QTreeWidget` 查看一下一个 Qt 应用程序的设置文件。缺省情况下，`QTreeWidget` 中的项目是只读的

Figure 10.5. The Settings Viewer application



下面是构造函数的一部分。

```

SettingsViewer::SettingsViewer(QWidget *parent)
    : QDialog(parent)
{
    organization = "Trolltech";
    application = "Designer";
    treeWidget = new QTreeWidget;
    treeWidget->setColumnCount(2);
    treeWidget->setHeaderLabels(
        QStringList() << tr("Key") << tr("Value"));
    treeWidget->header()->setResizeMode(0, QHeaderView::Stretch);
    treeWidget->header()->setResizeMode(1, QHeaderView::Stretch);
    ...
    setWindowTitle(tr("Settings Viewer"));
    readSettings();
}

```

为了读取到一个应用程序的设置文件，必须创建一个 QSettings 对象，并且把组织名称和程序名称作为参数。我们使用了一个缺省名称（"Desinger" by "Trolltech"），然后创建一个 QTreeWidget。最后我们调用 readSettings() 函数。

```
void SettingsViewer::readSettings()
{
    QSettings settings(organization, application);
    treeWidget->clear();
    addChildSettings(settings, 0, "");
    treeWidget->sortByColumn(0);
    treeWidget->setFocus();
    setWindowTitle(tr("Settings Viewer - %1 by %2")
                   .arg(application).arg(organization));
}
```

程序设置保存在一个有层次关系的属性关键字和对应值当的文件中（windows 平台下也可能保存在注册表中）。函数 addChildSettings() 读取一个属性组的信息，需要的参数分别是一个 QSettings 对象，父项目 QTreeWidgetItem 和当前的属性组名，在 QSettings 中组相当于文件中的目录。函数 addChildSettings() 递归调用自己遍历一个完整的树结构。在 readSettings() 函数中开始调用它，父项目为 0 代表根节点。

```
void SettingsViewer::addChildSettings(QSettings &settings,
                                     QTreeWidgetItem *parent, const QString &group)
{
    QTreeWidgetItem *item;
    settings.beginGroup(group);
    foreach (QString key, settings.childKeys()) {
        if (parent) {
            item = new QTreeWidgetItem(parent);
        } else {
            item = new QTreeWidgetItem(treeWidget);
        }
    }
}
```

```

    }

    item->setText(0, key);

    item->setText(1, settings.value(key).toString());
}

foreach (QString group, settings.childGroups()) {
    if (parent) {
        item = new QTreeWidgetItem(parent);
    } else {
        item = new QTreeWidgetItem(treeWidget);
    }

    item->setText(0, group);

    addChildSettings(settings, item, group);
}

settings.endGroup();
}

```

函数 `addChildSettings()` 创建了所有的 `QTreeWidgetItem`。它遍历应用设置层次树当前节点的所有属性，为每一个属性创建一个 `QTreeWidgetItem`。如果传递的父项目为 0，新创建的 `QTreeWidgetItem` 作为 `QTreeWidget` 的子项目（树的根节点），如果父项目不为 0，在父项目下面创建一个子项目。控件 `QTreeWidget` 的第一列为属性关键字名称，第二列为属性值。

然后函数开始遍历当前节点下的属性组，为每一个组创建一个 `QTreeWidgetItem` 对象，第一列为属性的名称。函数递归调用自己得到所有的属性组，把每一个组的子项目放到 `QTreeWidget` 控件中。

在本节所介绍的控件中，这些控件的编程方式和 Qt 的早期版本很像。把所有数据读到控件中，使用 `item` 代表一个数据，如果 `item` 是可编辑的，就把更新的数据写回到数据源。接下来的几节将改变这种简单的方式，充分利用 Qt 的 `model/view` 结构。

10-2 使用已有的模型类 (Using Predefined Models)

Qt 已经提供了一些可以和视图类配合使用的模型：

QStringListModel	保存一系列字符串
QStandardItemModel	保存有任意继承关系的数据
QDirModel	对本地文件系统进行的封装
QSqlQueryModel	对 SQL 查询结果进行的封装
QSqlTableModel	封装一个 SQL 表
QSqlRelationalTableModel	封装带有外键的 SQL 表
QSortFilterProxyModel	排序或者顾虑一个模型的封装

在本节中,我们将会讨论 QStringListModel, QDirModel 和 QSortFilterProxyModel 的使用。SQL 有关的模型将在第 13 章介绍。

首先我们看一个简单的对话框，用户可以增加爱，删除编辑一个 QStringList，每一个字符串代表一个项目领导。

Figure 10.6. The Team Leaders application



下面是其构造函数的一部分：

```
TeamLeadersDialog::TeamLeadersDialog(const QStringList &leaders,  
                                     QWidget *parent)  
    : QDialog(parent)  
{
```

```

    model = new QStringListModel(this);
    model->setStringList(leaders);
    listView = new QListView;
    listView->setModel(model);
    listView->setEditTriggers(QAbstractItemView::AnyKeyPressed
                             | QAbstractItemView::DoubleClicked);
    ...
}

```

首先我们创建一个 `QStringListModel`，然后用传递来的参数 `leaders` 作为模型的数据。然后我们创建一个 `QListView`，把刚才创建的模型作为这个视图的模型。调用 `setEditTriggers()` 允许用户通过敲击键盘或者双击它。缺省情况下，`QListView` 没有编辑触发器，是只读的。

```

void TeamLeadersDialog::insert()
{
    int row = listView->currentIndex().row();
    model->insertRows(row, 1);
    QModelIndex index = model->index(row);
    listView->setCurrentIndex(index);
    listView->edit(index);
}

```

当用户点击了 `insert` 按钮，槽函数 `insert()` 就会被调用。这个函数首先得到列表中当前项目的所在行数，模型中每一个数据项都有一个对应的模型索引（`model index`），有一个 `QModelIndex` 对象表示。在下一节我们再详细讨论这个类。现在我们需要知道的是一个索引有三个主要组成对象：行，列，一个指向实际数据的指针。对一个一维列表，索引得列数总是为 0。

一旦确定了行数，我们就在这个位置插入一新行。插入操作时是在模型上实现的。模型会自动刷新列表视图。然后，我们把新插入的行作为当前行。并把当前行设为编辑状态，就像用户点击了键盘或者双击了项目一样。

```

void TeamLeadersDialog::del()
{

```

```

model->removeRows(listView->currentIndex().row(), 1);
}

```

在构造函数中，Delete 按钮发出的 clicked()信号和 del()函数连接。因为我们只是删除当前行，调用函数 removeRows()，当前索引的行号和要删除的行数作为参数。和插入操作一样，模型类负责通知视图的更新。

```

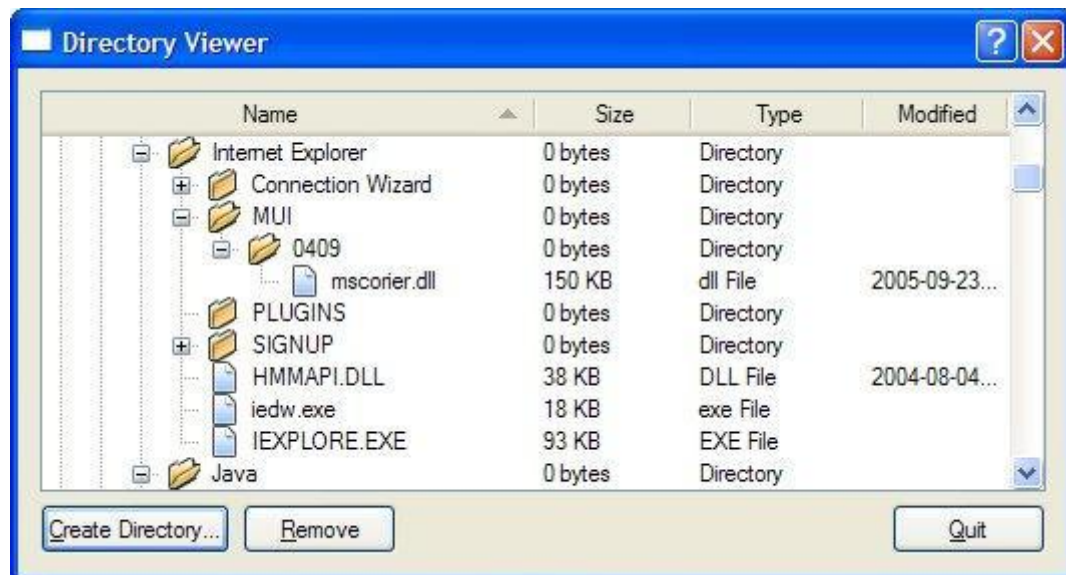
QStringList TeamLeadersDialog::leaders() const
{
    return model->stringList();
}

```

当对话框关闭时，函数 leaders()能够得到编辑过的字符列表。

TeadLeaderDialog 可以作为一个通用的字符列表编辑对话框，只要改变窗口标题就可以了。另一个经常使用的是显示文件列表或者目录。下一个例子我们使用了 QDirModel，这个类封装了计算机的文件系统，能够显示（或者隐藏）各种文件属性。这个模型类能够根据文件种类进行过滤，按照不同的方式进行排序等。

Figure 10.7. The Directory Viewer application



首先看类的构造函数，首先了 model 和 view 的创建和其他需要的设置。

```

DirectoryViewer::DirectoryViewer(QWidget *parent)
    : QDialog(parent)
{
    model = new QDirModel;
}

```

```

model->setReadOnly(false);
model->setSorting(QDir::DirsFirst | QDir::IgnoreCase | QDir::Name);
treeView = new QTreeView;
treeView->setModel(model);
treeView->header()->setStretchLastSection(true);
treeView->header()->setSortIndicator(0, Qt::AscendingOrder);
treeView->header()->setSortIndicatorShown(true);
treeView->header()->setClickable(true);
QModelIndex index = model->index(QDir::currentPath());
treeView->expand(index);
treeView->scrollTo(index);
treeView->resizeColumnToContents(0);
...
}

```

一旦创建了 **model**，我们设置它可编辑，并且设置排序方式。然后创建一个 **QTreeView** 显示 **model** 的数据。设置用户能够控制 **QTreeView** 的标题列顺序。设置标题列可点击，用户能够根据点击的任何一列排序，升序或者降序。然后得到当前目录的 **index**，调用 **expand()** 和 **scrollTo()** 使它可见。调用函数 **resizeColumnToContents()** 让第一列宽度足够显示所有的文字，而不是使用(...)

在构造函数的其他部分，我们把 **CreateDirectory** 按钮和 **Remove** 按钮的点击信号和各自的槽函数连接起来。注意我们没有 **Rename** 按钮，因为不需要，用户可以敲击 **F2** 后键入新的名称，**model** 会自动为我们保存。

```

void DirectoryViewer::createDirectory()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;
    QString dirName = QInputDialog::getText(this,
                                              tr("Create Directory"),
                                              tr("Directory name"));
}

```



```

if (!dirName.isEmpty()) {
    if (!model->mkdir(index, dirName).isValid())
        QMessageBox::information(this, tr("Create Directory"),
            tr("Failed to create the directory"));
}
}

```

用户点击 **CreateDirectory** 后，显示输入对话框，在对话框中输入目录名称。我们打算把新目录建在当前目录的下面。**QDirModel::mkdir()**函数创建一个新目录，参数为父目录的 **index** 和要创建的目录名称。如果创建成功，就返回新创建目录的 **index**。如果失败，则返回一个不合法的 **index**。

```

void DirectoryViewer::remove()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;
    bool ok;
    if (model->fileInfo(index).isDir()) {
        ok = model->rmdir(index);
    } else {
        ok = model->remove(index);
    }
    if (!ok)
        QMessageBox::information(this, tr("Remove"),
            tr("Failed to remove %1").arg(model->fileName(index)));
}

```

如果用户点击了 **Remove** 按钮，我们就删除当前 **index** 关联的文件或者目录。**vQDir** 可以实现删除操作，**QDirModel** 提供了这个操作的方便函数。

本节的最后一个例子讨论 **QSortFilterProxyModel** 的使用。与其他 Qt 提供的 **model** 不同，这个 **model** 封装了一个现有的 **model**，对已有的 **model** 和 **view** 之间传递的数据进行控制。在我们的这个例子中，已有的 **model** 为一个 **QStringListModel**，用一系列

QColor::colorNames()对 model 进行初始化。用户可以在一个 QLineEdit 控件中键入要过滤从字符串，用一个组合框确定这个字符串的解析方式（正则表达式，通配符等）

Figure 10.8. The Color Names application



下面是 ColorNameDialog 对话框的一部分：

```
ColorNamesDialog::ColorNamesDialog(QWidget *parent)
    : QDialog(parent)
{
    sourceModel = new QStringListModel(this);
    sourceModel->setStringList(QColor::colorNames());
    proxyModel = new QSortFilterProxyModel(this);
    proxyModel->setSourceModel(sourceModel);
    proxyModel->setFilterKeyColumn(0);
    listView = new QListView;
    listView->setModel(proxyModel);
    ...
    syntaxComboBox = new QComboBox;
    syntaxComboBox->addItem(tr("Regular expression"), QRegExp::RegExp);
    syntaxComboBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
    syntaxComboBox->addItem(tr("Fixed string"), QRegExp::FixedString);
    ...
}
```

The `QStringListModel` is created and populated in the usual way. This is followed by the construction of the `QSortFilterProxyModel`. We pass the underlying model using `setSourceModel()` and tell the proxy to filter based on column 0 of the original model. The `QComboBox::addItem()` function accepts an optional "data" argument of type `QVariant`; we use this to store the `QRegExp::PatternSyntax` value that corresponds to each item's text.

```
void ColorNamesDialog::reapplyFilter()
{
    QRegExp::PatternSyntax syntax =
        QRegExp::PatternSyntax(syntaxComboBox->itemData(
            syntaxComboBox->currentIndex()).toInt());
    QRegExp regExp(filterLineEdit->text(), Qt::CaseInsensitive, syntax);
    proxyModel->setFilterRegExp(regExp);
}
```

The `reapplyFilter()` slot is invoked whenever the user changes the filter string or the pattern syntax combobox. We create a `QRegExp` using the text in the line edit. Then we set its pattern syntax to the one stored in the syntax combobox's current item's data. When we call `setFilterRegExp()`, the new filter becomes active and the view is automatically updated.