



- 基本对话框
- 布局管理
- 对话框进阶
- QMainWindow
- 图形与图画
- Graphics View
- Model / View
- 网络与通信
- 磁盘文件
- 事件

- 全书所有实例的
源代码
- 开源工具软件包
- Windows动态链接库
- Linux动态链接库

Linux窗口程序设计

——Qt4精彩实例分析



成洁 卢紫毅 编著

清华大学出版社

[软考官方指定教材及同步辅导书下载](#) | [软考历年真题解析与答案](#)

[软考视频](#) | [考试机构](#) | [考试时间安排](#)

[Java一览无余：Java视频教程](#) | [Java SE](#) | [Java EE](#)

[.Net技术精品资料下载汇总：ASP.NET篇](#)

[.Net技术精品资料下载汇总：C#语言篇](#)

[.Net技术精品资料下载汇总：VB.NET篇](#)

[撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程](#)

[Visual C++\(VC/MFC\)学习电子书及开发工具下载](#)

[Perl/CGI脚本语言编程学习资源下载地址大全](#)

[Python语言编程学习资料\(电子书+视频教程\)下载汇总](#)

[最新最全Ruby、Ruby on Rails精品电子书等学习资料下载](#)

[数据库管理系统\(DBMS\)精品学习资源汇总：MySQL篇](#) | [SQL Server篇](#) | [Oracle篇](#)

[平面设计优秀资源学习下载](#) | [Flash优秀资源学习下载](#) | [3D动画优秀资源学习下载](#)

[最强HTML/xhtml、CSS精品学习资料下载汇总](#)

[最新JavaScript、Ajax典藏级学习资料下载分类汇总](#)

[网络最强PHP开发工具+电子书+视频教程等资料下载汇总](#)

[UML学习电子资源下载汇总 软件设计与开发人员必备](#)

[经典LinuxCBT视频教程系列 Linux快速学习视频教程一帖通](#)

[天罗地网：精品Linux学习资料大收集\(电子书+视频教程\)](#) [Linux参考资源大系](#)

[Linux系统管理员必备参考资料下载汇总](#)

[Linux shell、内核及系统编程精品资料下载汇总](#)

[UNIX操作系统精品学习资料<电子书+视频>分类总汇](#)

[FreeBSD/OpenBSD/NetBSD精品学习资源索引 含书籍+视频](#)

[Solaris/OpenSolaris电子书、视频等精华资料下载索引](#)

[>> 更多精品资料请访问大家论坛计算机区...](#)



Linux 窗口程序设计

——Qt4 精彩实例分析

成洁 卢紫毅 编著

清华大学出版社

北京

内 容 简 介

Qt 作为 Linux 下 GUI 的强大编程工具，能给用户提供精美的图形界面所需要的所有元素，已经得到了越来越广泛的应用。本书共分 11 章，以循序渐进的方式对 Qt 应用开发进行了介绍，涵盖了界面外观、图像处理、磁盘文件、网络与通信、事件等程序设计中经常涉及的内容。作者希望通过本书为想学习 Qt 编程的读者提供入门的指导，也为从事 Qt 开发应用的读者提供帮助。

本书内容全面，针对性强，叙述言简意赅、清晰流畅，讲解透彻，通俗易懂，图例丰富，所有实例均在 Linux 和 Windows 操作系统下进行了验证。

本书适合于从事或准备从事 Qt 开发的技术人员，也可作为 Linux 窗口应用开发者的参考书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目 (CIP) 数据

Linux 窗口程序设计——Qt4 精彩实例分析/成洁，卢紫毅编著. —北京：清华大学出版社，2008.11

ISBN 978-7-302-18158-3

I. L… II. ①成… ②卢… III. Linux 操作系统-程序设计 IV. TP316.89

中国版本图书馆 CIP 数据核字 (2008) 第 104638 号

责任编辑：涂 荣 张丽萍

封面设计：张 岩

版式设计：赵丽娜

责任校对：王 云

责任印制：王秀菊

出版发行：清华大学出版社

地 址：北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编：100084

社 总 机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969,c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015,zhilang@tup.tsinghua.edu.cn

印 刷 者：北京市清华园胶印厂

装 订 者：三河市李旗庄少明装订厂

经 销：全国新华书店

开 本：185×230 印 张：20.25 字 数：396 千字

(附光盘 1 张)

版 次：2008 年 11 月第 1 版 印 次：2008 年 11 月第 1 次印刷

印 数：1~5000

定 价：38.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系
调换。联系电话：(010)62770177 转 3103 产品编号：027649-01



前　　言

Qt 作为 Linux 下图形用户界面的强大编程工具，能给用户提供精美的图形界面所需的所有元素，已经得到了越来越广泛的应用，并且当前多数高端嵌入式设备生产商都选择了 Qt 作为开发工具。

目前，市场上关于 Qt 编程的书籍与其他编程开发工具的参考资料相比，可谓少之又少，这与快速发展的 Qt 不相符合。现有的有关 Qt 的书籍，内容基本上都是面向 Qt 理论，涵盖的内容很多，对 Qt 的发展发挥着强有力的作用。本书主要以 Qt 编程实例为基点，将程序设计中经常使用的编程方法和技巧介绍给大家，针对性强，对于初学者来说是一本非常实用的参考书。作者希望通过本书为想学习 Qt 编程的读者提供入门的指导，也为从事 Qt 开发应用的读者提供帮助。

对于应用程序开发者而言，最快捷的学习方式就是对具体实例的分析，本书的编写正是基于这一理念，为从事或准备从事 Qt 开发应用的人员提供一种便捷的学习途径。本书包含 63 个实例，涵盖了界面外观、图像处理、磁盘文件、网络与通信、事件等程序设计中经常涉及的方面，所选实例都是作者结合平时研发工作中的经验精心挑选出来的，内容丰富，具有很强的针对性和实用性。

本书共分 11 章，以循序渐进的方式对 Qt 应用开发进行了介绍，包括：

- 第 1 章 基本对话框
- 第 2 章 布局管理
- 第 3 章 对话框进阶
- 第 4 章 QMainWindow
- 第 5 章 图形与图画
- 第 6 章 Graphics View
- 第 7 章 Model/View
- 第 8 章 网络与通信
- 第 9 章 磁盘文件
- 第 10 章 事件
- 第 11 章 其他

本书结构没有按照 Qt 类的划分进行介绍，因为 Qt 的在线帮助已经提供了详尽的参



考，而是通过具体实例对涉及的 Qt 类进行了介绍和分析，突出实用性。本书是面向 Qt 程序开发人员的一本入门级参考书，面向实际应用开发，旨在为 Qt 开发人员提供直观的实例参考，针对性强，对 Qt 开发人员具有很好的参考价值。

本书读者对象为从事或准备从事 Qt 开发应用的人员，阅读本书的读者需具备 C++ 面向对象编程方面的知识。

由于时间仓促，作者水平有限，本书无法覆盖 Qt 的全部内容，难免存在缺点和不足之处，敬请读者批评指正。

本书在写作的过程中得到了清华大学出版社的大力支持与帮助，在此，作者向其表示诚挚的谢意！

作 者
2008 年 3 月

目 录

第 1 章 基本对话框	1
实例 1 Hello World!	2
实例 2 标准对话框的使用	5
实例 3 各类位置信息	10
实例 4 使用标准输入框	15
实例 5 各种消息框的使用	20
实例 6 实现 QQ 抽屉效果	27
实例 7 表格的使用	30
实例 8 使用进度条	31
实例 9 利用 Qt Designer 设计一个对话框	35
实例 10 在程序中使用 ui	40
实例 11 动态加载 ui	43
第 2 章 布局管理	45
实例 12 基本布局管理	46
实例 13 多文档	52
实例 14 分割窗口	56
实例 15 停靠窗口	58
实例 16 堆栈窗体	61
实例 17 综合布局实例	63
第 3 章 对话框进阶	69
实例 18 可扩展对话框	70
实例 19 利用 QPalette 改变控件颜色	73
实例 20 窗体的淡入淡出效果	79
实例 21 不规则窗体	84
实例 22 电子钟	87
实例 23 程序启动画面	92



第 4 章 QMainWindows.....	95
实例 24 基本 QMainWindow 主窗口程序	96
实例 25 打印文本	103
实例 26 打印图像	106
实例 27 图片的缩放与旋转	108
实例 28 在工具栏中嵌入控件	112
实例 29 设置字体、字号等格式属性	115
实例 30 设置文本排序及对齐	122
第 5 章 图形与图画	130
实例 31 利用 QPainter 绘制各种图形	132
实例 32 利用 QPainterPath 进行画图	143
实例 33 渐变效果	149
实例 34 QPainter 坐标系的变形	159
实例 35 SVG 格式图片的显示	162
实例 36 一个简单的绘图工具	169
实例 37 改变图片的透明度	177
实例 38 橡皮筋线	182
第 6 章 Graphics View.....	185
实例 39 地图浏览器	187
实例 40 各种 Graphics Item.....	193
实例 41 Graphics Item 的各种变形.....	202
实例 42 飞舞的蝴蝶	208
第 7 章 Model/View	212
实例 43 文件目录浏览器	215
实例 44 利用特定控件进行表项编辑	217
实例 45 自定义 Model.....	222
实例 46 柱状统计图	230
第 8 章 网络与通信	241
实例 47 获取本机网络信息	242
实例 48 基于 UDP 的网络广播程序	244

实例 49 基于 TCP 的网络聊天室程序	247
实例 50 实现 HTTP 文件下载	256
实例 51 实现 FTP 上传和下载	261
第 9 章 磁盘文件	266
实例 52 获取文件属性	267
实例 53 文件浏览器	269
第 10 章 事件	273
实例 54 获得鼠标事件	274
实例 55 使用键盘控制移动	276
实例 56 事件过滤器实现动态图片按钮	281
第 11 章 其他	284
实例 57 利用 QSettings 保存程序窗口状态	285
实例 58 利用 QDataStream 对文件进行存取	290
实例 59 改变鼠标指针形状	293
实例 60 改变窗体显示风格	295
实例 61 拖拽图标	299
实例 62 拖拽文字	307
实例 63 字符串编码格式转换	313

第1章 基本对话框

本章的实例对 Qt 编程的基本流程、标准对话框的使用方法以及 Qt Designer 的使用方法等进行了分析，包括 11 个实例：

- Hello World!
- 标准对话框的使用
- 各类位置信息
- 使用标准输入框
- 各种消息框的使用
- 实现 QQ 抽屉效果
- 表格的使用
- 使用进度条
- 利用 Qt Designer 设计一个对话框
- 在程序中使用 ui
- 动态加载 ui



实例 1 Hello World!

知识点：

- 开发 Qt 程序的基本流程和编译运行方式
- 信号和槽机制 (Signal&Slot)

本实例实现一个“Hello World！”例子，简单介绍 Qt 编程的基本流程，以及 Qt 程序的编译运行方式。实例效果图如图 1-1 所示。



图 1-1 Hello World！

这是一个简单的例子，整个对话框只有一个按钮，单击该按钮，对话框关闭，退出程序。

实现代码如下：

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

第 1 行包括<QApplication>，所有 Qt 图形化应用程序都必须包含此文件，它包含了 Qt 图形化应用程序的各种资源、基本设置、控制流以及事件处理等，若是 Qt 的非图形化应用程序，则需包含<QCoreApplication>。

小贴士：Qt 最初的框架只有关于图形化应用的类，随着它的一步步发展，Qt 已独立发展出了许多非图形化的类库，如数据库应用、XML 解析等。



第 2 行包含了程序中要应用到的按钮控件的头文件。

 小贴士：在 Qt4 中，头文件的包含可以采用类似< QApplication >和< QPushButton >的形式，也可以写成< qapplication.h>和< qpushbutton.h>的形式。

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

第 4 行新创建了一个 QApplication 对象，每个 Qt 应用程序都必须有且只一个 QApplication 对象，采用 argc、argv 作为参数，便于程序处理命令行参数。

第 5 行创建了一个 QPushButton 对象，并设置它的显示文本为“Hello World！”，由于此处并没有指定按钮的父窗体，因此以自己作为主窗口。

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

第 7 行的 QObject::connect() 方法是 Qt 最重要的特征，即信号与槽的机制。当按钮被按下则触发 clicked() 信号发射，与之相连的 QApplication 对象的槽 quit() 响应按钮信号，执行退出应用程序的操作。关于信号与槽机制在本实例最后将进行详细的分析。

最后调用 QApplication 的 exec() 方法，程序进入消息循环，等待可能输入进行响应。Qt 完成事件处理及显示的工作，并在应用程序退出时返回 exec() 的值。

Qt 程序的编译运行很简单，利用 Qt 提供的 qmake 工具能够很方便地对程序进行编译，编译流程如下：

```
qmake -project  
qmake  
make  
./hello
```

其中，qmake -project 命令用于生成程序的项目文件 (*.pro)；qmake 用于生成程序的 Makefile 文件；make 编译 Makefile 文件得到可执行文件；最后执行程序即可出现图 1-1 所示的对话框，一个简单的 Hello World！例子完成。

 小贴士：确保 Qt 的环境变量路径设置正确，程序编译时若出现连接错误，请首先检查有关 Qt 的环境变量，保证调用的 qmake 为相应的 Qt 版本。

信号与槽机制（signal&slot）作为 Qt 最重要的特性，提供了任意两个 Qt 对象之间的通信机制。其中，信号会在某个特定情况或动作下被触发，槽是用于接收并处理信号的函数。例如，要将一个窗口中的变化情况通知给另一个窗口，则一个窗口发送信号，另一个窗口的槽接收此信号并进行相应的操作，即可实现两个窗口之间的通信。这比传统



的图形化程序采用回调函数的方式实现对象间通信要简单灵活得多。每个 Qt 对象都包含预定的信号和槽，当某一特定事件发生时，一个信号被发射，与信号相关联的槽则会响应信号完成相应的处理。

信号与槽机制常用的连接方式为：

```
connect( Object1, SIGNAL(signal), Object2, SLOT(slot) );
```

signal 为对象 Object1 的信号，slot 为对象 Object2 的槽，Qt 的窗口部件都包含若干个预定义的信号和若干个预定义的槽。当一个类被继承时，该类的信号和槽也同时被继承。开发人员也可以根据需要定义自己的信号和槽。

信号与槽机制可以有多种连接方式，图 1-2 中描述了信号与槽的多种可能连接方式。

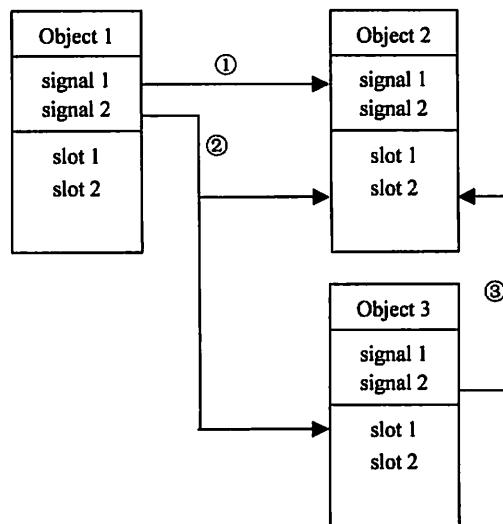


图 1-2 信号与槽的连接方式

① 一个信号可以与另一个信号相连。

```
connect(Object1, SIGNAL(signal 1), Object2, SIGNAL(signal 1));
```

即表示 Object1 的信号 1 发射可以触发 Object2 的信号 1 发射。

② 表示同一个信号可以与多个槽相连。

```
connect(Object1, SIGNAL(signal 2), Object2, SIGNAL(slot 2));  
connect(Object1, SIGNAL(signal 2), Object3, SIGNAL(slot 1));
```

③ 表示同一个槽可以响应多个信号。

```
connect(Object1, SIGNAL(signal 2), Object2, SIGNAL(slot 2));
connect(Object3, SIGNAL(signal 2), Object2, SIGNAL(slot 2));
```

实例 2 标准对话框的使用

知识点：

- 标准通用文件对话框的使用
- 标准通用颜色对话框的使用
- 标准通用字体对话框的使用

和大多数操作系统一样，Linux 也提供了一系列的标准对话框，如文件选择、字体选择、颜色选择等，这些标准对话框为应用程序提供了一致的观感。Qt 对这些标准对话框都定义了相关的类，这些类让使用者能够很方便地使用标准对话框进行文件、颜色以及字体的选择。标准对话框在软件设计过程中是经常需要使用的。

 小贴士：Qt 提供的标准对话框除了本实例提到的，还有 QErrorMessage、QInputDialog、QMessageBox、QPrintDialog、QProgressDialog 等，这些标准对话框的使用在本书的后续部分将会陆续介绍。

本实例主要演示上面几种标准对话框的使用，如图 1-3 所示。



图 1-3 标准对话框

在图 1-3 中，单击“文件对话框”按钮，会弹出文件选择对话框，选中的文件名将显示在右边；单击“颜色对话框”按钮，会弹出颜色选择对话框，选中的颜色将显示在右边；单击“字体对话框”按钮，会弹出字体选择对话框，选中的字体将更新右边显示的字符串。

具体实现代码如下所示。

头文件 standarddialogs.h：

```
class StandardDialogs : public QDialog
```



```
{  
    Q_OBJECT  
public:  
    StandardDialogs( QWidget *parent=0, Qt::WindowFlags f=0 );  
    ~StandardDialogs();  
public:  
    QGridLayout *layout;  
    QPushButton *filePushButton;  
    QPushButton *colorPushButton;  
    QPushButton *fontPushButton;  
    QLineEdit *fileLineEdit;  
    QLineEdit *fontLineEdit;  
    QFrame *colorFrame;  
private slots:  
    void slotOpenFileDialog();  
    void slotOpenColorDlg();  
    void slotOpenFontDlg();  
};
```

头文件定义了实例中需要用到的各种窗体控件以及各种操作的槽函数。

实现文件 standarddialogs.cpp:

```
StandardDialogs::StandardDialogs( QWidget *parent, Qt::WindowFlags f )  
    : QDialog( parent, f )  
{  
    1    setWindowTitle(tr("Standard Dialogs"));  
  
    2    layout = new QGridLayout( this );  
  
    3    filePushButton = new QPushButton;  
    4    filePushButton->setText(tr("File Dialog"));  
  
    5    colorPushButton = new QPushButton;  
    6    colorPushButton->setText(tr("Color Dialog"));  
  
    7    fontPushButton = new QPushButton;  
    8    fontPushButton->setText(tr("Font Dialog"));  
  
    9    fileLineEdit = new QLineEdit;  
  
   10    colorFrame = new QFrame;  
   11    colorFrame->setFrameShape( QFrame::Box );  
   12    colorFrame->setAutoFillBackground(true);
```

```

13 fontLineEdit = new QLineEdit;
14 fontLineEdit->setText(tr("Hello World"));

15 layout->addWidget( filePushButton, 0, 0 );
16 layout->addWidget( fileLineEdit, 0, 1 );
17 layout->addWidget( colorPushButton, 1, 0 );
18 layout->addWidget( colorFrame, 1, 1 );
19 layout->addWidget( fontPushButton, 2, 0 );
20 layout->addWidget( fontLineEdit, 2, 1 );
21 layout->setMargin(15);
22 layout->setSpacing(10);

23 connect(filePushButton,SIGNAL(clicked()),this,SLOT(slotOpenFileDialog()));
24 connect(colorPushButton,SIGNAL(clicked()),this,SLOT(slotOpenColorDlg()));
25 connect(fontPushButton,SIGNAL(clicked()),this,SLOT(slotOpenFontDlg()));
}

```

第 1 行设置主窗体的标题。

第 2~8 行分别创建各个按钮控件。

第 9 行创建一个 QLineEdit 类实例 fileLineEdit，用来显示选择的文件名。

第 10~12 行创建一个 QFrame 类实例 colorFrame，当用户选择不同的颜色时，colorFrame 会根据用户选择的颜色更新其背景。

第 13、14 行创建一个 QLineEdit 类实例 fontLineEdit，当用户选择不同的字体时，fontLineEdit 会根据用户选择的字体更新其内容。

第 15~22 行将各个控件进行布局。

第 23~25 行将各个按钮的 clicked 事件与相应的槽进行连接。

```

void StandardDialogs::slotOpenFileDialog()
{
    QString s = QFileDialog::getOpenFileName(
        this,
        "open file dialog",
        "/",
        "C++ files (*.cpp);;C files (*.c);;Head files (*.h)");

   lineEditFile->setText( s.toAscii() );
}

```

getOpenFileName()是 QFileDialog 类的一个静态函数，返回用户选择的文件名，如果用户选择取消（Cancel），则返回一个空串。函数形式如下：



```
QString QFileDialog::getOpenFileName ( QWidget * parent = 0, const QString & caption = QString(),
const QString & dir = QString(), const QString & filter = QString(), QString * selectedFilter = 0, Options options = 0 )
```

调用 `getOpenFileName()` 函数将创建一个模态的文件对话框，如图 1-4 所示。dir 参数指定了默认的目录，如果 dir 参数带有文件名，则该文件将是默认选中的文件；filter 参数对文件类型进行过滤，只有与过滤器匹配的文件类型才显示，filter 可以同时指定多种过滤方式供用户选择，多种过滤器之间用“::”隔开，用户选择的过滤器通过参数 `selectedFilter` 返回。

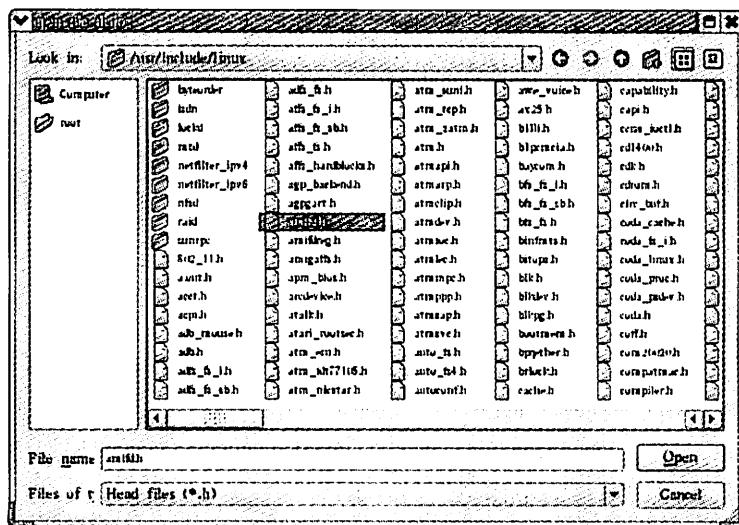


图 1-4 标准文件对话框

文件对话框的文件类型选择如图 1-5 所示。

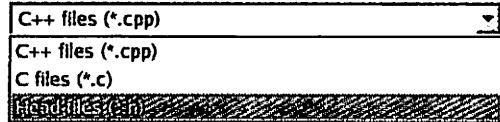


图 1-5 文件对话框的文件类型选择

`QFileDialog` 类还提供了类似的其他静态函数，如表 1-1 所示，通过这些函数，用户可以很方便地定制自己的文件对话框。

表 1-1 QFileDialog 的静态函数

静态函数	说 明
getOpenFileName	获得用户选择的文件名
getSaveFileName	获得用户保存的文件名
getExistingDirectory	获得用户选择的已存在的目录名
getOpenFileNames	获得用户选择的文件名列表

```
void StandardDialogs::slotOpenColorDlg()
{
    QColor color = QColorDialog::getColor(Qt::blue);

    if(color.isValid())
    {
        frameColor->setPalette(QPalette(color));
    }
}
```

getColor()是 QColorDialog 的一个静态函数，返回用户选择的颜色值，函数形式如下：

```
QColor getColor( const QColor & initial = Qt::white, QWidget * parent = 0 )
```

调用 getColor() 函数将创建一个模态的颜色对话框，如图 1-6 所示。initial 参数指定了默认选中的颜色，默认为白色。通过 QColor::isValid() 可以判断用户选择颜色是否有效，若用户选择取消（Cancel），QColor::isValid() 将返回 false。

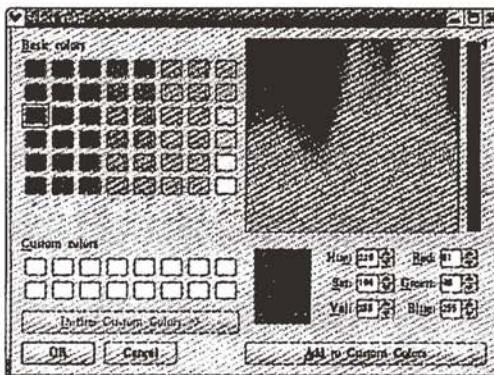


图 1-6 标准颜色对话框

```
void StandardDialogs::slotOpenFontDlg()
```

```

{
    bool ok;
    QFont font = QFontDialog::getFont( &ok );
    if( ok )
    {
       lineEditFont->setFont( font );
    }
}

```

getFont()是 QFontDialog 的一个静态函数，返回用户选择的字体，函数形式如下：

```
QFont getFont( bool * ok, QWidget * parent = 0 )
```

调用 getFont() 函数将创建一个模态的字体对话框，如图 1-7 所示。用户选择 OK，参数*ok 将为 true，函数返回用户选择的字体，否则为 false，此时函数返回默认字体。

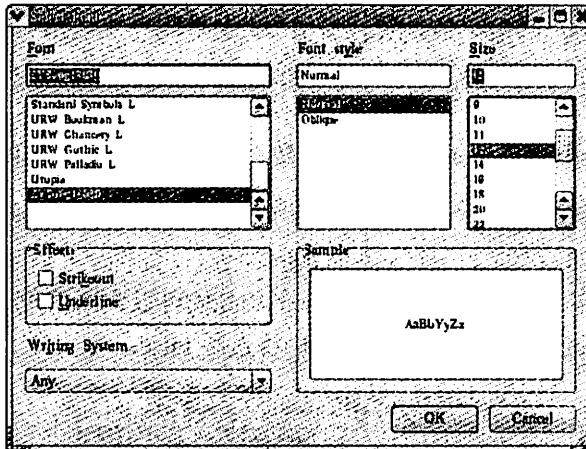


图 1-7 标准字体对话框

实例 3 各类位置信息

知识点：

- 各种与位置相关函数的区别
- 各种与位置相关函数的使用场合

Qt 提供了很多关于获取窗体位置及显示区域大小的函数，本实例利用一个简单的对

话框显示窗体的各种位置信息，包括窗体的所在点位置、长、宽信息等。本实例的目的是分析各个有关位置信息的函数之间的区别，如 `x()`、`y()`、`pos()`、`rect()`、`size()`、`geometry()` 等，以及在不同的情况下应使用哪个函数来获取位置信息。实现的效果图如图 1-8 所示。

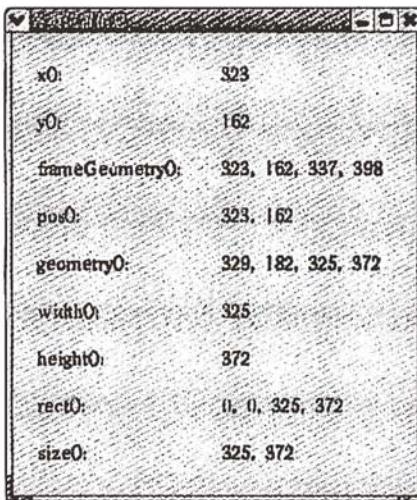


图 1-8 各类位置信息

在实例中，分别调用了 `x()`、`y()`、`frameGeometry()`、`pos()`、`geometry()`、`width()`、`height()`、`rect()`、`size()` 几个函数，这几个函数均是 `QWidget` 提供的。当改变对话框的大小，或移动对话框时，调用各个函数所获得的信息显示也相应地发生变化，从变化中可得知各函数之间的区别。

由于本实例的目的是为了分析各函数之间的区别并获得结论，而程序本身的实现较为简单，因此只简单介绍实现过程。

头文件 `geometry.h`:

```
class Geometry : public QDialog
{
    Q_OBJECT
public:
    Geometry();
    /* 声明所需的控件 主要为 QLabel 类 */
    .....
    void updateLabel();
protected:
    void moveEvent(QMoveEvent *);
```

```
    void resizeEvent(QResizeEvent *);  
};
```

实现文件 geometry.cpp:

```
Geometry::Geometry()  
{  
    setWindowTitle("Geometry");  
    /* 创建程序所需的各控件 */  
    .....  
    /* 布局 */  
    .....  
    updateLabel();  
}
```

程序初始化时调用 updateLabel() 函数，以获得各位置函数的信息并显示。具体代码如下：

```
void Geometry::updateLabel()  
{  
    QString temp;  
    //获得 x() 函数的结果并显示  
    QString str_x;  
    xLabel->setText(str_x.setNum(x()));  
  
    //获得 y() 函数的结果并显示  
    QString str_y;  
    yLabel->setText(str_y.setNum(y()));  
  
    //获得 frameGeometry() 函数的结果并显示  
    QString frameGeo;  
    frameGeo = temp.setNum(frameGeometry().x()) + ", " +  
              temp.setNum(frameGeometry().y()) + ", " +  
              temp.setNum(frameGeometry().width()) + ", " +  
              temp.setNum(frameGeometry().height());  
    frameGeoLabel->setText(frameGeo);  
  
    //获得 pos() 函数的结果并显示  
    QString position ;  
    position = temp.setNum(pos().x()) + ", " + temp.setNum(pos().y());  
    posLabel->setText(position);  
  
    //获得 geometry() 函数的结果并显示  
    QString geo;
```

```

geo = temp.setNum(geometry().x()) + ", " +
      temp.setNum(geometry().y()) + ", " +
      temp.setNum(geometry().width()) + ", " +
      temp.setNum(geometry().height());
geoLabel->setText(geo);

//获得 width()、height()函数的结果并显示
QString w;
widthLabel->setText(w.setNum(width()));
QString h;
heightLabel->setText(h.setNum(height()));

//获得 rect()函数的结果并显示
QString r;
r = temp.setNum(rect().x()) + ", " + temp.setNum(rect().y()) + ", " +
    temp.setNum(rect().width()) + ", " + temp.setNum(rect().height());
rectLabel->setText(r);

//获得 size()函数的结果并显示
QString s;
s = temp.setNum(size().width()) + ", " + temp.setNum(size().height());
sizeLabel->setText(s);
}

```

updateLabel()函数负责调用各个位置函数获得结果并显示。

重定义 QWidget 的 moveEvent()和 resizeEvent()函数，分别响应对话框的移动事件和大小调整事件，使得窗体在被移动或窗体大小发生改变时，能同步更新各函数结果的显示。具体代码如下：

```

void Geometry::moveEvent(QMoveEvent *)
{
    updateLabel();
}

void Geometry::resizeEvent(QResizeEvent *)
{
    updateLabel();
}

```

通过这个例子可以获得如图 1-9 所示的结论。

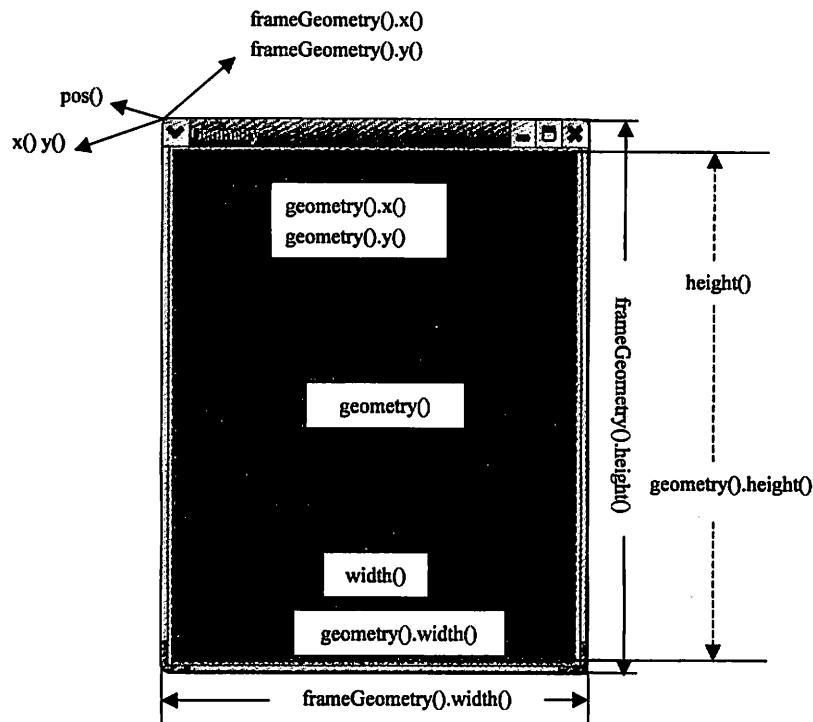


图 1-9 各位置函数获得结果的区别示意图

`x()`、`y()`和 `pos()` 函数都是获得整个窗体左上角的坐标位置。而 `frameGeometry()` 与 `geometry()` 相对应, `frameGeometry()` 是获得整个窗体的左上顶点和长、宽值, 而 `geometry()` 函数获得的是窗体内中央区域的左上顶点坐标以及长、宽值。直接调用 `width()` 和 `height()` 函数获得的是中央区域的长和宽的值。

还有两个函数 `rect()`、`size()`, 调用它们获得的结果也都是对于窗体的中央区域而言的, `size()` 获得的是窗体中央区域的长、宽值, `rect()` 与 `geometry()` 一样返回一个 `QRect` 对象。其中, 两个函数获得的长、宽值是一样的, 都是窗体中央区域的长、宽值, 只是左上顶点的坐标值不一样, `geometry()` 获得的左上顶点坐标是相对于父窗体而言的坐标, 而 `rect()` 获得的左上顶点坐标始终为(0,0)。

因此, 在实际应用中需根据情况使用正确的位置信息函数以获得准确的位置尺寸信息, 尤其是在编写对位置精度要求较高的程序时, 如地图浏览程序, 更应注意函数的选择, 避免产生不必要的误差。

 小贴士：在编写程序时，初始化窗体时最好不要使用 `setGeometry()` 函数，而用 `resize()` 和 `move()` 函数代替，因为使用 `setGeometry()` 会导致窗体 `show()` 之后在错误的位置上停留很短暂的一段时间，带来闪烁现象。

实例 4 使用标准输入框

知识点：

各种标准输入框的使用方法

本实例演示如何使用标准输入框，Qt 提供了一个 `QInputDialog` 类，`QInputDialog` 类提供了一种简单方便的对话框来获得用户的单个输入信息，目前提供了 4 种数据类型的输入，可以是一个字符串、一个 Int 类型数据、一个 double 类型数据或是一个下拉列表框的条目。一个标准输入对话框的基本结构如图 1-10 所示。其中包括一个提示标签，一个输入控件。若是调用字符串输入框，则为一个 `QlineEdit`；若是调用 Int 类型或 double 类型输入框，则为一个 `QspinBox`；若是调用列表条目输入框，则为一个 `QComboBox`；还包括一个确定输入（OK）按钮和一个取消输入（Cancel）按钮。

本实例的实现效果图如图 1-11 所示。

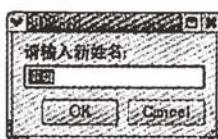


图 1-10 标准输入对话框的基本结构



图 1-11 使用标准输入框

实例中列举了以上 4 种输入类型，右侧的按钮用于弹出标准输入对话框修改各条信息的值。具体实现代码如下所示。

头文件 `inputdialog.h`:

```
class InputDlg : public QDialog
{
    Q_OBJECT
public:
    InputDlg();
```

```
public:  
    QPushButton *nameButton;  
    QPushButton *sexButton;  
    QPushButton *ageButton;  
    QPushButton *statureButton;  
  
    QLabel *label1;  
    QLabel *label2;  
    QLabel *label3;  
    QLabel *label4;  
    QLabel *nameLabel;  
    QLabel *sexLabel;  
    QLabel *ageLabel;  
    QLabel *statureLabel;  
  
private slots:  
    void slotName();  
    void slotSex();  
    void slotAge();  
    void slotStature();  
};
```

头文件中声明了对话框中用到的控件以及各按钮触发的槽函数。

实现文件 inputdialog.cpp:

```
InputDialog::InputDialog()  
{  
    setWindowTitle(tr("Input Dialog"));  
  
    /* 创建各标签对象 */  
    label1 = new QLabel(tr("Name : "));  
    .....  
  
    /* 创建各显示标签 */  
    nameLabel = new QLabel(tr("LiMing"));  
    nameLabel->setFrameStyle(QFrame::Panel|QFrame::Sunken);  
    .....  
  
    /* 创建各修改按钮 */  
    nameButton = new QPushButton;  
    nameButton->setIcon(QIcon("btn.png"));  
    .....  
  
    /* 布局管理 */
```

```

QGridLayout *layout = new QGridLayout( this );
.....
/* 连接信号与槽函数 */
connect(nameButton,SIGNAL(clicked()),this,SLOT(slotName()));
connect(sexButton,SIGNAL(clicked()),this,SLOT(slotSex()));
connect(ageButton,SIGNAL(clicked()),this,SLOT(slotAge()));
connect(statureButton,SIGNAL(clicked()),this,SLOT(slotStature()));
}

```

构造函数中设置对话框的标题、创建各类控件并连接信号和槽。

slotName()函数的代码如下：

```

void InputDlg::slotName()
{
    1   bool ok;
    2   QString name = QInputDialog::getText(this,tr("User Name"),
        tr("Please input new name."), QLineEdit::Normal,nameLabel->text(),&ok);
    3   if(ok && !name.isEmpty())
    4       nameLabel->setText(name);
}

```

单击姓名修改按钮触发此函数，弹出标准字符串输入对话框，如图 1-12 所示。

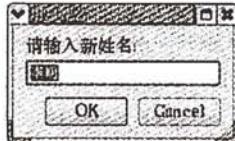


图 1-12 标准字符串输入对话框

第 2 行调用 QInputDialog 的 getText() 函数弹出标准字符串输入对话框，getText() 函数原型如下：

```

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

```

此函数的第一个参数 parent 为标准输入对话框的父窗口；第二个参数 title 为标准输入对话框的标题名；第三个参数 label 为标准输入对话框的标签提示；第四个参数 mode 指定标准输入对话框中 QLineEdit 控件的输入模式；第五个参数 text 为标准字符串输入对话框弹出时 QLineEdit 控件中默认出现的文字；第六个参数 ok 用于指示标准输入对话框的哪个按钮被触发，若 ok 为 true，则表示用户单击了 OK（确定）按钮；若 ok 为 false，

则表示用户单击了 Cancel (取消) 按钮; 最后一个参数 f 指明标准输入对话框的窗体标识。

第 3 行判断 ok 的值, 若用户单击了“确定”按钮, 则把新输入的姓名更新至显示标签。

slotSex()函数的代码如下:

```
void InputDlg::slotSex()
{
    QStringList list;
    list << tr("male") << tr("female");
    bool ok;
    QString sex = QInputDialog::getItem(this, tr("Sex"),
                                         "Please select sex.", list, 0, &ok);
    if(ok)
        sexLabel->setText(sex);
}
```

单击性别修改按钮触发此函数, 弹出标准条目选择对话框, 如图 1-13 所示。

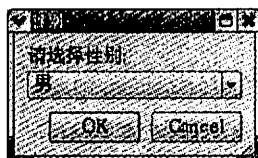


图 1-13 标准条目选择对话框

第 1、2 行创建一个 QStringList 对象, 包含两个 QString 项, 用于标准输入对话框中下拉列表框的条目显示。

第 4 行调用 QInputDialog 的 getItem() 函数弹出标准条目选择对话框, getItem() 函数原型如下:

```
QString getItem( QWidget * parent, const QString & title, const QString & label, const QStringList &
                  list, int current = 0, bool editable = true, bool * ok = 0, Qt::WindowFlags f = 0 )
```

此函数的第一个参数 parent 为标准输入对话框的父窗口; 第二个参数 title 为标准输入对话框的标题名; 第三个参数 label 为标准输入对话框的标签提示; 第四个参数 list 指定标准输入对话框中 QComboBox 控件显示的可选条目, 为一个 QStringList 对象; 第五个参数 current 为标准条目选择对话框弹出时 QComboBox 控件中默认显示的条目序号; 第六个参数 editable 指定 QComboBox 控件中显示的文字是否可编辑; 第七个参数 ok 用于指示标准输入对话框的哪个按钮被触发, 若 ok 为 true, 则表示用户单击了 OK (确定)

按钮；若 ok 为 false，则表示用户单击了 Cancel（取消）按钮；最后一个参数 f 指明标准输入对话框的窗体标识。

第 5 行判断 ok 的值，若用户单击了“确定”按钮，则把新输入的性别更新至显示标签。

slotAge()函数的代码如下：

```
void InputDlg::slotAge()
{
    1   bool ok;
    2   int age = QInputDialog::getInteger(this,tr("User Age"),
                                         tr("Please input age:"),ageLabel->text(),0,150,1,&ok);
    3   if(ok)
    4       ageLabel->setText(QString(tr("%1")).arg(age));
}
```

单击年龄修改按钮触发此函数，弹出标准 int 类型输入对话框，如图 1-14 所示。

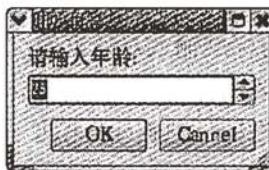


图 1-14 标准 int 类型输入对话框

调用 QInputDialog 的 getInteger() 函数弹出标准 int 类型输入对话框，getInteger() 函数原型如下：

```
int getInteger ( QWidget * parent, const QString & title, const QString & label, int value = 0, int minValue =
-2147483647, int maxValue = 2147483647, int step = 1, bool * ok = 0, Qt::WindowFlags f = 0 )
```

此函数的第一个参数 parent 为标准输入对话框的父窗口；第二个参数 title 为标准输入对话框的标题名；第三个参数 label 为标准输入对话框的标签提示；第四个参数 value 指定标准输入对话框中 QSpinBox 控件默认显示值；第五、第六个参数 minValue、maxValue 指定 QSpinBox 控件的数值范围；第七个参数 step 指定 QSpinBox 控件的步进值；第八个参数 ok 用于指示标准输入对话框的哪个按钮被触发，若 ok 为 true，则表示用户单击了 OK（确定）按钮；若 ok 为 false，则表示用户单击了 Cancel（取消）按钮；最后一个参数 f 指明标准输入对话框的窗体标识。

第 3 行判断 ok 的值，若用户单击了“确定”按钮，则把新输入的年龄值更新至显示标签。

slotStature()函数的代码如下：

```
void InputDlg::slotStature()
{
    1   bool ok;
    2   double d = QInputDialog::getDouble(this,tr("Stature"),
                                         tr("Please input stature:"),175.00,0,230.00,1,&ok);
    3   if(ok)
    4       statureLabel->setText(QString(tr("%1")).arg(d));
}
```

单击身高修改按钮触发此函数，弹出标准 double 类型输入对话框，如图 1-15 所示。

调用 QInputDialog 的 getDouble() 函数弹出标准 double 类型输入对话框，getDouble() 函数原型如下：

```
double getDouble ( QWidget * parent, const QString & title, const QString & label, double value = 0,
                  double minValue = -2147483647, double maxValue = 2147483647, int decimals = 1, bool * ok = 0,
                  Qt::WindowFlags f = 0 )
```

此函数的第一个参数 parent 为标准输入对话框的父窗口；第二个参数 title 为标准输入对话框的标题名；第三个参数 label 为标准输入对话框的标签提示；第四个参数 value 指定标准输入对话框中 QSpinBox 控件默认显示值；第五、第六个参数 minValue、maxValue 指定 QSpinBox 控件的数值范围；第七个参数 decimals 指定 QSpinBox 控件的步进值；第八个参数 ok 用于指示标准输入对话框的哪个按钮被触发，若 ok 为 true，则表示用户单击了 OK（确定）按钮；若 ok 为 false，则表示用户单击了 Cancel（取消）按钮；最后一个参数 f 指明标准输入对话框的窗体标识。

第 3 行判断 ok 的值，若用户单击了“确定”按钮，则把新输入的身高值更新至显示标签。

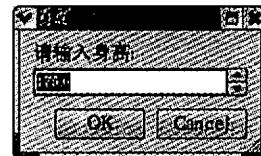


图 1-15 标准 double 类型输入对话框

实例 5 各种消息框的使用

知识点：

□ 各种标准消息框的使用

实现自定义消息框

在实际的程序开发中，经常会用到各种各样的消息框来给用户一些提示或提醒，Qt 提供了 QMessageBox 类来实现此项功能。在本实例中，分析了各种消息框的使用方式及之间的区别。各种消息框的使用如图 1-16 所示。



图 1-16 各种消息框的使用

本实例主要分析了 7 种类型的消息框，包括 Question 消息框、Information 消息框、Warning 消息框、Critical 消息框、About（关于）消息框、About（关于）Qt 消息框以及 Custom（自定义）消息框。

Question 消息框、Information 消息框、Warning 消息框和 Critical 消息框的用法大同小异，这些消息框一般都包含一条提示信息、一个图标以及若干个按钮，它们的作用都是给用户提供一些提醒或一些简单的询问。按图标的不同可区分为以下 4 个级别。

- ⓘ (Question)：为正常的操作提供一个简单的询问。
- ⓘ (Information)：为正常的操作提供一个提示。
- ⚠ (Warning)：提醒用户发生了一个错误。
- ⚡ (Critical)：警告用户发生了一个严重错误。

下面分别对各种不同消息框的使用方法进行分析。如图 1-17 所示为 Question 消息框。

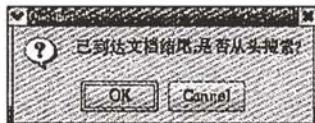


图 1-17 Question 消息框

```
void MessageBox::slotQuestion()
{
```

```

switch(QMessageBox::question(this, "Question",
    tr("It's end of document, search from begin?"),
    QMessageBox::Ok|QMessageBox::Cancel,QMessageBox::Ok))
{
    case QMessageBox::Ok:
        label->setText(" Question button / Ok ");
        break;
    case QMessageBox::Cancel:
        label->setText(" Question button / Cancel ");
        break;
    default:
        break;
}
return;
}

```

对于 Question 消息框，调用时直接使用 `QMessageBox::question()` 即可。

第一个参数为消息框的父窗口指针。

第二个参数为消息框的标题栏。

第三个参数为消息框的文字提示信息，前 3 个参数对于其他几种消息框基本是一样的。

后两个参数都是对消息框按钮的设定，`QMessageBox` 类提供了许多标准按钮，如 `QMessageBox::Ok`、`QMessageBox::Close`、`QMessageBox::Discard` 等，具体可查阅 Qt 帮助。

第四个参数即填写希望在消息框中出现的按钮，可根据需要在标准按钮中选择，用“|”连写，默认为 `QMessageBox::Ok`。

第五个参数为默认按钮，即消息框出现时，焦点默认处于那个按钮上。

函数的返回值为按下的按钮，当用户按 Escape 键时，相当于返回 `QMessageBox::Cancel`。

如图 1-18 所示为 Information 消息框。



图 1-18 Information 消息框

```

void MessageBox::slotInformation()
{
    QMessageBox::information(this,"Information",tr("anything you want tell user"));
    return;
}

```

Information 消息框使用频率最高也最简单，直接调用 QMessageBox::information() 即可。

第一个参数为消息框的父窗口指针。

第二个参数为消息框的标题栏。

第三个参数为消息框的文字提示信息。

后面的两个参数与 Question 消息框的用法一样，但在使用的过程中，经常会省略后两个参数，直接使用默认的 QMessageBox::Ok 按钮。

Information 消息框和 Question 消息框可以通用，使用 Question 消息框的地方都可以用 Information 消息框替换。

如图 1-19 所示为 Warning 消息框。



图 1-19 Warning 消息框

```
void MessageBox::slotWarning()
{
    switch(QMessageBox::warning(this, "Warning",
        tr("Save changes to document?"),
        QMessageBox::Save|QMessageBox::Discard|QMessageBox::Cancel,
        QMessageBox::Save))
    {
        case QMessageBox::Save:
            label->setText(" Warning button / Save ");
            break;
        case QMessageBox::Discard:
            label->setText(" Warning button / Discard ");
            break;
        case QMessageBox::Cancel:
            label->setText(" Warning button / Cancel ");
            break;
        default:
            break;
    }
    return;
}
```



Warning 消息框的最常用法为当用户进行了一个非正常操作时，提醒用户并询问是否进行某项操作，如关闭文档时，提醒并询问用户是否保存对文档的修改。实例中实现的即是此操作。

函数调用的方式与前面 Question 消息框的调用方式大致相同。

第一个参数为消息框的父窗口指针。

第二个参数为消息框的标题栏。

第三个参数为消息框的文字提示信息。

第四个参数为希望在消息框中出现的按钮，可根据需要在标准按钮中选择，用“|”连写，默认为 QMessageBox::Ok。

第五个参数为默认按钮，即消息框出现时，焦点默认处于那个按钮上。

小贴士：消息框的第 4 个参数的选择，虽然是在 QMessageBox::standardButtons 中任意选择，但并不是随意选择，应注意按常规成对出现。如选择了 QMessageBox::Save 按钮则最好成对地选择 QMessageBox::Discard 按钮；而 Abort、Retry、Ignore 一般是一起出现的。

如图 1-20 所示为 Critical 消息框。

```
void MessageBox::slotCritical()
{
    QMessageBox::critical(this,"Information",tr("tell user a critical error"));
    label->setText(" Critical MessageBox ");
    return;
}
```

Critical 消息框是在系统出现严重错误时对用户进行提醒的。它的用法也相对简单，通常情况下和 Information 消息框一样，在调用时只填写前 3 个参数即可。如图 1-21 所示为 About 消息框。

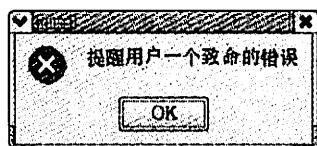


图 1-20 Critical 消息框

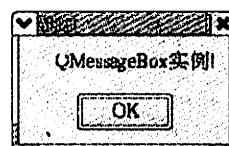


图 1-21 About 消息框

```
void MessageBox::slotAbout()
{
    QMessageBox::about(this,"About",tr("Message box example!"));
}
```

```

label->setText(" About MessageBox ");
return;
}

```

About 消息框一般用于提供系统的版本等信息。只需提供信息而并不需要用户反馈信息，因此它的用法相对简单，直接调用 QMessageBox::about()，并只用指定消息框父窗口、标题栏以及信息的内容即可。

在介绍完以上几种基本消息框的用法后，还有两种特殊的消息框类型，分别是“About Qt 消息框”（如图 1-22 所示）以及自定义消息框（如图 1-23 所示）。

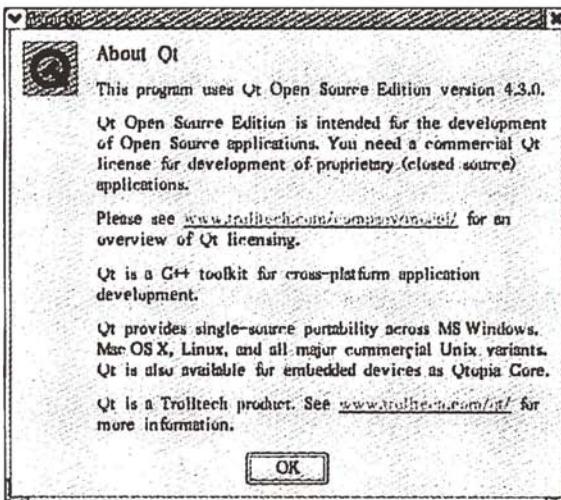


图 1-22 About Qt 消息框

```

void MessageBox::slotAboutQt()
{
    QMessageBox::aboutQt(this, "About Qt");
    label->setText(" About Qt MessageBox ");
    return;
}

```

“About Qt 消息框”是 Qt 预定好的一种消息框，用于提供 Qt 的相关信息，只需直接调用 QMessageBox::aboutQt()，并指定父窗口和标题栏即可，其中显示的内容是 Qt 预定好的。

最后，当以上所有的消息框都不能满足开发的需求时，Qt 还允许 Custom（自定义）消息框。包括消息框的图标、按钮、内容等都可根据需要进行设定。本实例中即实现了一个如图 1-23 所示的自定义消息框。



图 1-23 Custom 消息框

具体实现代码如下：

```
void MessageBox::slotCustom()
{
    1   QMessageBox customMsgBox;
    2   customMsgBox.setWindowTitle("Custom message box");
    3   QPushButton *lockButton = customMsgBox.addButton(tr("Lock"),
    4                                                 QMessageBox::ActionRole);
    5   QPushButton *unlockButton = customMsgBox.addButton(tr("Unlock"),
    6                                                 QMessageBox::ActionRole);
    7   QPushButton *cancelButton = customMsgBox.addButton(
    8                                                 QMessageBox::Cancel);
    9   customMsgBox.setIconPixmap(QPixmap("./images/linuxredhat.png"));
    10  customMsgBox.setText(tr("This is a custom message box"));
    11  customMsgBox.exec();
    12
    13  if(customMsgBox.clickedButton() == lockButton)
    14      label->setText(" Custom MessageBox / Lock ");
    15  if(customMsgBox.clickedButton() == unlockButton)
    16      label->setText(" Custom MessageBox / Unlock ");
    17  if(customMsgBox.clickedButton() == cancelButton)
    18      label->setText(" Custom MessageBox / Cancel ");
    19
    20  return;
}
```

第 1 行首先创建一个 QMessageBox 对象 customMsgBox。

第 2 行设置此消息框的标题栏为 Custom message box。

第 3~5 行定义消息框所需的按钮，由于 QMessageBox::standardButtons 只提供了常用的一些按钮，并不能满足所有应用的需求，因此 QMessageBox 类提供了一个 addButton()函数来为消息框增加自定义的按钮，addButton()函数的第一个参数为按钮显示的文字，第二个参数为按钮类型的描述，具体可查阅 QMessageBox::ButtonRole，当然也可使用 addButton()

函数来加入一个标准按钮。如第 5 行在消息框中加入了一个 QMessageBox::Cancel 按钮，消息框将会按调用 addButton()的先后次序在消息框中由左至右依次插入按钮。

第 6 行调用 setIconPixmap()设置自定义消息框的图标。

第 7 行调用 setText()设置自定义消息框中显示的提示信息内容。

第 8 行调用 exec()显示此自定义消息框。

后面几行代码完成的都是实例中一些显示的功能，此处不再赘述。

通过本实例的分析可见，Qt 提供的消息框类型基本涵盖了开发应用中使用的各种情况，并且提供了自定义消息框的方式，满足各种特殊的需求，在实际应用中关键是分析实际的应用需求，根据不同的应用环境选择最合适的消息框，以使程序简洁而合理。

实例 6 实现 QQ 抽屉效果

知识点：

利用 QToolBox 实现抽屉效果

抽屉效果是软件界面设计中的一种常用形式，目前很多流行软件都采用了抽屉效果，如腾讯公司的 QQ 软件。抽屉效果可以以一种动态直观的方式在有限大小的界面上扩展出更多的功能。本实例在 Qt 下实现抽屉效果，如图 1-24 所示。

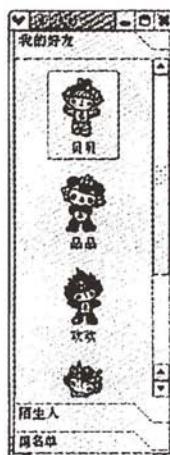


图 1-24 实现 QQ 抽屉效果实例图



具体实现代码如下：

```
class Drawer : public QToolBox
{
    Q_OBJECT
public:
    Drawer( QWidget *parent=0, Qt::WindowFlags f=0 );
    ~Drawer();

    QToolButton *toolButton1_1;
    QToolButton *toolButton1_2;
    QToolButton *toolButton1_3;
    QToolButton *toolButton1_4;
    QToolButton *toolButton1_5;
    QToolButton *toolButton2_1;
    QToolButton *toolButton2_2;
    QToolButton *toolButton3_1;
    QToolButton *toolButton3_2;
};
```

头文件定义了实例中需要用到的各种窗体控件。Drawer 类继承自 QToolBox，QToolBox 提供了一种列状的层叠窗体，本实例通过 QToolBox 来实现一种抽屉效果，QToolButton 提供了一种快速访问命令或选择项的按钮，通常在工具条中使用。

```
Drawer::Drawer( QWidget *parent, Qt::WindowFlags f )
    : QToolBox( parent, f )
{
    1   setWindowTitle(tr("My QQ"));

    2   QGroupBox *groupBox1 = new QGroupBox;

    3   toolButton1_1 = new QToolButton;
    4   toolButton1_1->setText( tr( "beibei" ) );
    5   toolButton1_1->setIcon( QPixmap( "./images /bb.png" ) );
    6   toolButton1_1->setIconSize( QPixmap( "./images /bb.png" ).size() );
    7   toolButton1_1->setAutoRaise( TRUE );
    8   toolButton1_1->setToolButtonStyle( Qt::ToolButtonTextUnderIcon );

    /*其余控件的初始化*/
    .....
    9   QVBoxLayout *layout1 = new QVBoxLayout(groupBox1);
    10  layout1->setMargin(10);
    11  layout1->.setAlignment(Qt::AlignHCenter);
```

```

12 layout1->addWidget(toolButton1_1);
13 layout1->addWidget(toolButton1_2);
14 layout1->addWidget(toolButton1_3);
15 layout1->addWidget(toolButton1_4);
16 layout1->addWidget(toolButton1_5);
17 layout1->addStretch();

18 QGroupBox *groupBox2 = new QGroupBox;
.....
19 QGroupBox *groupBox3 = new QGroupBox;
.....
20 this->addItem( ( QWidget* )groupBox1 , tr("my friends" ) );
21 this->addItem( ( QWidget* )groupBox2 , tr("stranger" ) );
22 this->addItem( ( QWidget* )groupBox3 , tr("blacklist" ) );
}

```

第 1 行设置主窗体的标题。

第 2 行创建了一个 QGroupBox 类实例，在本例中对应每一个抽屉。

第 3 行创建了一个 QToolButton 类实例，在这里 QToolButton 分别对应于抽屉中的每一个按钮。

第 4~6 行对按钮的文字、图标以及大小等进行设置。

第 7 行设置按钮的 AutoRaise 属性为 TRUE，即当鼠标离开时，按钮自动恢复成弹起状态。

第 8 行设置按钮的 ToolButtonStyle 属性，ToolButtonStyle 属性主要用来描述按钮的文字和图标的显示方式。Qt 定义了 4 种 ToolButtonStyle 类型，分别介绍如下。

- Qt::ToolButtonIconOnly：只显示图标。
- Qt::ToolButtonTextOnly：只显示文字。
- Qt::ToolButtonTextBesideIcon：文字显示在图标旁边。
- Qt::ToolButtonTextUnderIcon：文字显示在图标下面。

程序员可以根据显示需要调整显示方式。

第 9 行创建一个 QVBoxLayout 类实例，用来设置抽屉内各按钮的布局。

第 10、11 行设置布局中各窗体的显示间距和显示位置。

第 12~16 行将抽屉内的各个按钮加入。

第 17 行调用 addStretch() 方法在按钮之后插入一个占位符，使得所有按钮能靠上对齐。并且在整个抽屉大小发生改变时，保证按钮的大小不发生变化。

第 18、19 行创建其余两栏抽屉。

第 20~22 行把准备好的抽屉插入至 ToolBox 中。



实例 7 表格的使用

知识点：

- 表格的使用方法
- 在表格中嵌入控件

制作统计软件时经常会使用表格将资料列出，或是通过表格进行资料的设置，在 Qt 中可以使用 QTableWidget 实现一个表格。本实例演示如何使用表格，并在表格中嵌入控件。如图 1-25 所示为“表格的使用”对话框。

1	2	3	4	5
1 男	姓名	出生日期	职业	收入
2	Tom	04/4/1985	医生	3500
3	Jack	20/8/1973	工人	2100
4	Alice	15/1/1985	律师	4000
5	John			

周日 周一 周二 周三 周四 周五 周六

30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

图 1-25 “表格的使用”对话框

QTableWidget 类提供了一个灵活的和可编辑的表格控件，包含很多 API，可以处理标题、行列、单元格和选中区域，QTableWidget 可以嵌入编辑框或显示控件，并可通过拖动控制柄调节各单元格的大小。表格中的每一项可以定义成不同的属性，可以显示文本，也可以插入控件，这样就给表格的使用带来了很好的扩展性。

相关实现代码如下：

```
1 QLabel *LabelPixmap1 = new QLabel();
2 LabelPixmap1->setPixmap(QPixmap("./images/Male.png"));
3 setCellWidget (1,0,LabelPixmap1);
```

```

4   QTableWidgetItem *TableWidgetItem1 = new QTableWidgetItem(tr("Tom"));
5   setItem(1, 1, QTableWidgetItem1);

6   QDateTimeEdit *DateTimeEdit1= new QDateTimeEdit();
7   DateTimeEdit1->setDateTime(QDateTime::currentDateTime());
8   DateTimeEdit1->setDisplayFormat("dd/M/yyyy");
9   DateTimeEdit1->setCalendarPopup(true);
10  setCellWidget (1,2,DateTimeEdit1);

11  QComboBox *ComboBoxWork1 = new QComboBox();
12  ComboBoxWork1->addItem(tr("Worker"));
13  ComboBoxWork1->addItem(tr("Farmer"));
14  ComboBoxWork1->addItem(tr("Doctor"));
15  ComboBoxWork1->addItem(tr("Lawyer"));
16  ComboBoxWork1->addItem(tr("Soldier"));
17  setCellWidget(1,3,ComboBoxWork1);

18  QSpinBox *SpiBoxIncome1 = new QSpinBox();
19  SpiBoxIncome1->setRange (1000,10000);
20  setCellWidget(1,4,SpiBoxIncome1);

```

第 1~3 行在表格中插入一个 QLabel 控件，并设置 QLabel 的图形属性。

第 4、5 行设置表格单元的属性为文本显示。

第 6~10 行在表格中插入一个 QDateTimeEdit 控件，该控件可以编辑日期时间，
setCalendarPopup()方法设置是否弹出日历编辑器。

第 11~17 行在表格中插入一个 QComboBox 控件，调用 QTableWidgetItem 的 setCellWidget()
函数可在某个指定的表格单元格中插入一个控件，函数的前两个参数用于指定单元格的
行、列号。

第 18~20 行在表格中插入一个 QSpinBox 控件。

实例 8 使用进度条

知识点：

- ❑ QProgressBar 的使用
- ❑ QProgressDialog 的使用

通常在处理长时间任务时需要提供进度条的显示，告诉用户当前任务的进展情况。



本实例演示如何使用进度条，如图 1-26 和图 1-27 所示。

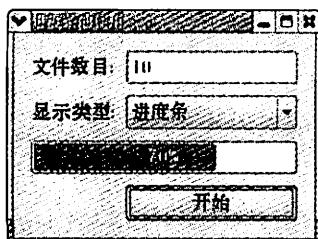


图 1-26 使用进度条显示进度

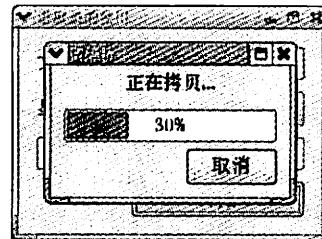


图 1-27 使用进度对话框显示进度

Qt 提供了两种显示进度条的方式，一种是 `QProgressBar`，另一种是 `QProgressDialog`。`QProgressBar` 类提供了一种横向或纵向显示进度的控件表示方式，用来描述任务的完成情况。`QProgressDialog` 类提供了一种针对慢速过程的进度对话框表示方式，用于描述任务完成的进度情况。标准的进度条对话框包括一个进度显示条、一个取消按钮以及一个标签。

`QProgressBar` 有几个重要的属性值，`minimum`、`maximum` 决定进度条指示的最大值和最小值，`format` 决定进度条显示文字的格式，可以有 3 种显示格式：`%p%`、`%v`、`%m`。`%p%` 显示完成的百分比，这是默认显示方式，例如，`██████████`；`%v` 显示当前的进度值，例如，`██████`；`%m` 显示总的步进值，例如，`██████████`。`invertedAppearance` 属性可以让进度条以反方向显示进度。

`QProgressDialog` 也有几个重要的属性值，决定了进度条对话框何时出现，出现多长时间，分别是 `minimum`、`maximum` 和 `minimumDuration`。`minimum` 和 `maximum` 分别表示进度条的最小值和最大值，决定了进度条的变化范围；`minimumDuration` 为进度条对话框出现前的等待时间。系统根据所需完成的工作量估算一个预计花费的时间，若大于设定的等待时间 `minimumDuration`，则出现进度条对话框；若小于设定的等待时间，则不出现进度条对话框。

进度条使用了一个步进值的概念，即一旦设置好了进度条的最大值和最小值，进度条将会显示完成的步进值占总的步进值的百分比，百分比的计算公式为：

$$\text{百分比} = (\text{value}() - \text{minimum}()) / (\text{maximum}() - \text{minimum}())$$

具体实现代码如下：

```
class Progress : public QDialog
{
    Q_OBJECT
public:
```



```
Progress( QWidget *parent=0, Qt::WindowFlags f=0 );
~Progress();
public:
    QLabel *numLabel;
    QLineEdit *numLineEdit;
    QLabel *typeLabel;
    QComboBox *typeComboBox;

    QProgressBar *progressBar;
    QPushButton *startPushButton;

private slots:
    void slotStart();
};

Progress::Progress( QWidget *parent, Qt::WindowFlags f)
    : QDialog( parent, f )
{
    QFont font("ZYSong18030",12);
    setFont(font);

    setWindowTitle(tr("Progress"));

    numLabel = new QLabel(tr("File Num:"));
    numLineEdit = new QLineEdit;
    numLineEdit->setText(tr("10"));

    typeLabel = new QLabel(tr("Progress Type:"));
    typeComboBox = new QComboBox;
    typeComboBox->addItem(tr("ProgressBar"));
    typeComboBox->addItem(tr("ProgressDialog"));

    progressBar = new QProgressBar;

    startPushButton = new QPushButton(tr("Start"));

    QGridLayout *layout = new QGridLayout( this );
    layout->addWidget( numLabel, 0, 0 );
    layout->addWidget( numLineEdit, 0, 1 );
    layout->addWidget( typeLabel, 1, 0 );
    layout->addWidget( typeComboBox, 1, 1 );
    layout->addWidget( progressBar, 2, 0, 1, 2 );
    layout->addWidget( startPushButton, 3, 1 );
}
```



```
    layout->setMargin(15);
    layout->setSpacing(10);

    connect(startPushButton,SIGNAL(clicked()),this,SLOT(slotStart()));
}
```

构造函数主要完成主界面的初始化工作，包括各控件的创建、布局以及信号/槽的连接。

```
void Progress::slotStart()
{
    1   int num=numLineEdit->text().toInt();

    2   if(typeComboBox->currentIndex() == 0)/*ProgressBar*/
    {
        3       progressBar->setRange(0,num);
        4       for (int i=1; i<num+1; i++)
        {
            5           progressBar->setValue(i);
            /*此处模拟文件复制过程 */
            6           sleep(1);
        }
    }
    7   else if(typeComboBox->currentIndex() == 1)/*modal ProgressDialog*/
    {
        8       QProgressDialog *progressDialog = new QProgressDialog(this);
        9       QFont font("ZYSong18030",12);
        10      progressDialog->setFont(font);
        11      progressDialog->setWindowModality(Qt::WindowModal);
        12      progressDialog->setMinimumDuration(5);
        13      progressDialog->setWindowTitle(tr("Please Wait"));
        14      progressDialog->setLabelText(tr("Copying..."));
        15      progressDialog->setCancelButtonText(tr("Cancel"));
        16      progressDialog->setRange(0,num);

        17      for (int i=1; i<num+1; i++)
        {
            18          progressDialog->setValue(i);
            19          qApp->processEvents();
            /*此处模拟文件复制过程 */
            20          sleep(1);
            21          if (progressDialog->wasCanceled())
            22              return;
        }
    }
}
```

```
    }  
}  
}
```

第 1 行获得当前需要复制的文件数目，这里对应进度条的总的步进值。

第 3~6 行采用进度条的方式显示进度。

第 3 行设置进度条的步进范围从 0 到需要复制的文件数目。

第 5、6 行模拟每一个文件的复制过程，这里通过 sleep(1) 进行模拟，在实际中使用文件复制过程来替换 sleep(1)，进度条的总的步进值为需要复制的文件数目，当复制完一个文件后，步进值增加 1。

第 8~22 行采用进度对话框的方式显示进度。

第 8 行创建一个进度对话框。

第 11 行设置进度对话框采用模态方式进行显示，即显示进度的同时，其他窗口将不响应输入信号。

第 12 行设置进度对话框出现需等待的时间，此处设定为 5 秒，默认为 4 秒。

第 13~15 行设置进度对话框的窗体标题、显示文字信息以及取消按钮的显示文字。

第 16 行设置进度对话框的步进范围。

第 18~20 行模拟每一个文件的复制过程，这里通过 sleep(1) 进行模拟，在实际中使用文件复制过程来替换 sleep(1)，进度条的总的步进值为需要复制的文件数目，当复制完一个文件后，步进值增加 1，这里需要使用 qApp->processEvents(); 来正常响应事件循环，以确保应用程序不会出现阻塞。

第 21、22 行检测“取消”按钮是否被触发，若触发则退出循环并关闭进度对话框，在实际应用中，此处还需进行相关的清理工作。

进度对话框的使用有两种方法，即模态方式与非模态方式。本实例中使用的是模态方式，模态方式的使用比较简单方便，但必须使用 QApplication::processEvents() 来使事件循环保持正常进行状态，从而确保应用不会阻塞。若使用非模态方式，则需要通过 QTimer 来实现定时设置进度条的值。

实例 9 利用 Qt Designer 设计一个对话框

知识点：

使用 Qt Designer 设计对话框的一般步骤



在 Qt 编程中，程序员通常都是使用手动编写 C++ 源代码来进行 Qt 程序开发，但有些程序员也喜欢使用可视化的方法进行对话框设计，因此，Qt 为习惯利用可视化方式进行窗口程序设计的程序员提供了 Designer，它可以让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，可以用作一个常用的工具，并不对编辑产生影响。使用 Qt Designer 可以方便快速地对对话框进行修改，在对话框经常需要变化的情况下，这是一种很好的方式。本书利用第 1 章的最后几个实例分析使用 Qt Designer 设计对话框的相关内容。使用 Qt Designer 设计对话框一般都有如下几个步骤：

- (1) 创建窗体并在窗体中放置各种控件。
- (2) 对窗体进行布局设计。
- (3) 设置各控件的标签顺序。
- (4) 创建信号和槽。
- (5) 连接信号和槽。

Qt Designer 的启动可以通过命令行运行 `designer` 完成，启动后界面如图 1-28 所示。

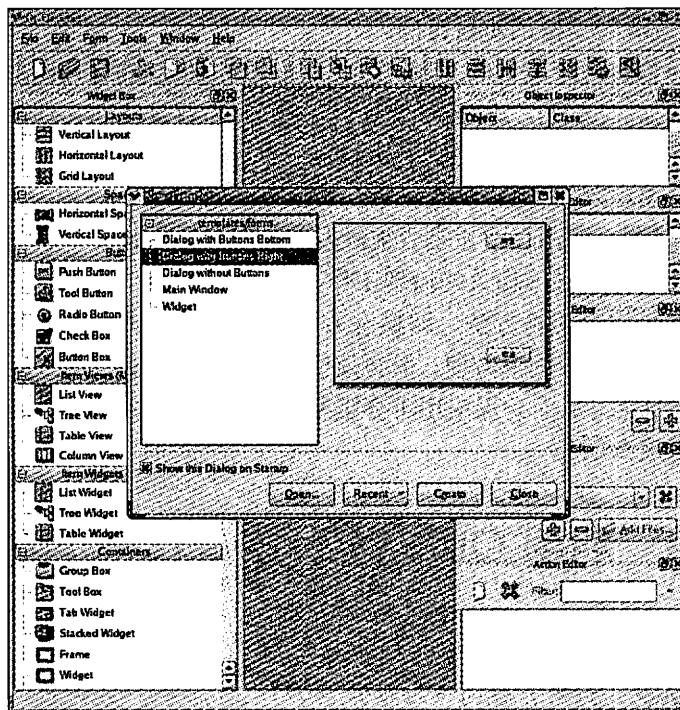


图 1-28 Qt Designer

Qt Designer 提供如下 5 种表单模板可供选择：

- 底部带“确定”、“取消”按钮的对话框窗体。
- 右侧带“确定”、“取消”按钮的对话框窗体。
- 不带按钮的对话框窗体。
- Main Window 类型窗体。
- 通用窗体。

这里选择创建一种不带按钮的对话框窗体，接下来需要做的就是在窗体中放置各种需要的控件，Qt Designer 的设计空间列出了所有控件以及各控件的属性设置窗体。在窗体中放置一个 Label 和 LineEdit、两个 PushButton 和一个 Horizontal Spacer 控件，并设置各控件的 text 属性，如图 1-29 所示。在开始向窗体中放置控件时，不用太在意控件对齐与否，只用放置大概位置即可。

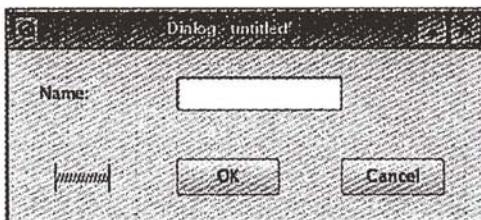


图 1-29 使用 Qt Designer 放置控件

接下来对窗体中的各个控件进行布局设计，选择位于同一行的所有控件，选择 Qt Designer 菜单中的 Form→Layout Horizontally 命令，完成所选控件的水平布局。完成布局设计后适当调整整个窗体的大小，以适合控件的大小，如图 1-30 所示。

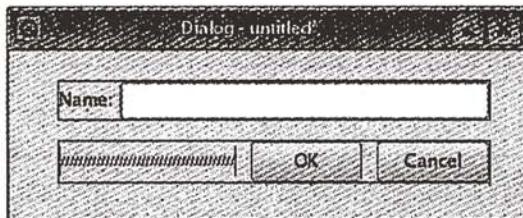


图 1-30 完成布局的对话框

然后对各控件的标签顺序进行设置，选择 Qt Designer 菜单中的 Edit→Edit Tab Order 命令，进入标签设置模式，窗体中各个控件上出现一个蓝色的小框，框内的数字表示该控件的标签顺序，即焦点顺序，如图 1-31 所示，可以单击蓝色小框修改标签顺序。完成



标签顺序设置后选择 Qt Designer 菜单中的 Edit→Edit Widgets 命令离开标签设置模式。

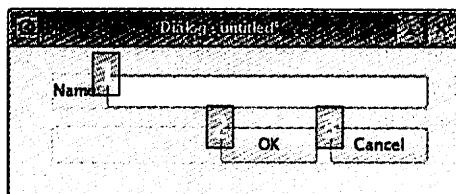


图 1-31 设置标签顺序

接下来进行信号和槽的连接,选择 Qt Designer 菜单中的 Edit→Edit Signals/Slots 命令,进入信号/槽连接模式,如图 1-32 所示。此时单击 OK 按钮,然后拖动鼠标,可以发现有一根红色的类似接地线的标志线被拖出,松开鼠标,弹出信号/槽的连接配置窗口,如图 1-33 所示。

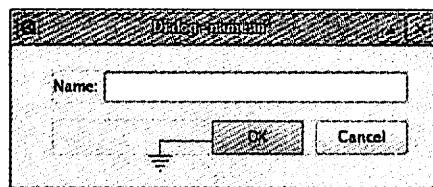


图 1-32 配置信号/槽

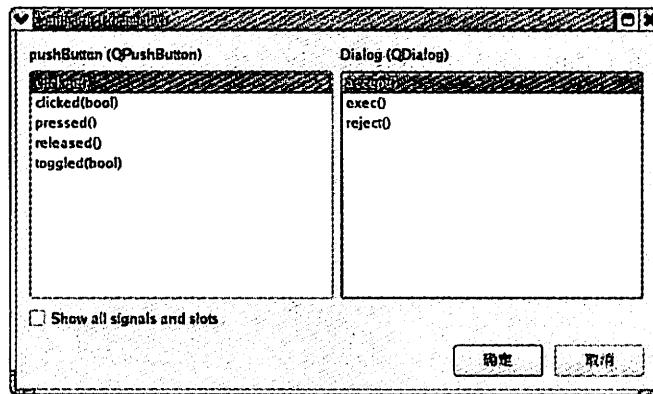


图 1-33 连接信号/槽

连接配置窗口左侧列出了按钮 OK 的所有信号,右侧列出了对话框的所有槽,选择 OK 按钮的 clicked()信号和 accept()信号,单击“确定”按钮,此时完成按钮 OK 的信号/槽的连接,用同样的方式配置按钮 Cancel 的信号/槽,如图 1-34 所示。

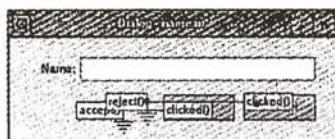


图 1-34 完成信号/槽的连接

至此，关于 Qt Designer 的操作就结束了，生成一个.ui 文件。接下来编写 main.cpp 文件，代码如下：

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

main.cpp 的内容比较简单，这里不再赘述，下面进行编译链接。启动 Linux 控制台，在终端下依次运行：

```
qmake -project           //生成工程文件.pro
qmake                   //生成 Makefile 文件
make                     //编译链接成可执行文件
```

在执行完 qmake -project 生成工程文件后，打开.pro 文件可看到在工程中加入了.ui 文件：

```
FORMS += name.ui
```

这样，在 make 之后会自动生成 ui_name.h 文件，这个文件包含了在 Designer 中所做的所有工作。运行可执行程序，分别单击 OK 和 Cancel 按钮，执行对话框的 accept() 和 reject() 函数，如图 1-35 所示。

使用 Qt Designer 设计对话框是一种简单有效的方法，可以节省设计对话框的时间，而且修改方便、直观，对于初学者来说，这是一种入门的好方法。但随着程序越来越复杂，Qt Designer 也有不利的地方：首先，使用 Qt Designer 生成的代码比较庞大，很多代码是自动生成的，不利于开发者阅读；其次对于初学者而言，使用 Qt Designer 不利于掌



握 Qt 编程的本质。因此，笔者还是建议尽量使用手动的方式编写源代码，这样能更好地理解 Qt 编程的本质，更多地体验 Qt 编程的乐趣。本书的绝大部分实例都是采用手动编写代码的方式进行实现的。

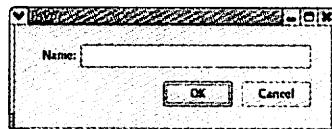


图 1-35 运行可执行程序

实例 10 在程序中使用 ui

知识点：

- 利用 Designer 设计生成 ui 的 3 种使用方式
- 单继承法
- 多继承法
- 两种继承方式的比较

本实例利用一个简单的例子说明如何在程序开发中使用 Designer 生成.ui 文件。

本实例利用 Qt Designer 生成了 3 个简单的 ui，在使用时，两个 ui 插入到主程序的 QTabWidget 中，另一个 ui 由按钮触发弹出，如图 1-36 所示。主程序窗口 MyWidget 采用的是手动编写代码的方式实现。

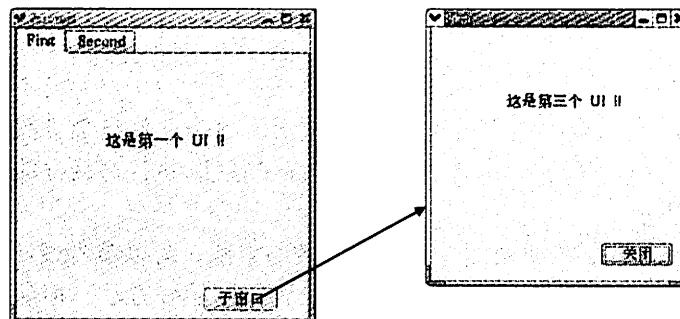


图 1-36 生产 3 个 ui

利用 Qt Designer 设计生成的.ui 文件，在使用时可利用 Qt 自带的 uic 工具生成 ui_xxx.h 文件进行使用。在具体实现时有多种方法，包括：

- 直接使用法（实例 9 使用的即是直接使用法）。
- 单继承法。
- 多继承法。

本实例中的实现采用的是单继承法，因此首先分析单继承法的实现方式，同时也会穿插分析多继承法的实现方式。

具体实现代码如下：

```
#include "ui_first.h"
#include "ui_second.h"
#include "ui_third.h"

class MyWidget : public QWidget
{
    Q_OBJECT
public:
    MyWidget(QWidget *parent=0);

public slots:
    void slotChild();

private:
    Ui::First firstUi;
    Ui::Second secondUi;
    Ui::Third thirdUi;
};
```

MyWidget 类是自定义的一个对话框类，首先需包含 3 个 ui 的头文件，即 ui_first.h、ui_second.h 和 ui_third.h，这 3 个头文件是 Qt 自动生成的。

slotChild()槽函数用于响应弹出子窗口的按钮事件。

在 MyWidget 类中声明了 3 个私有变量 firstUi、secondUi 和 thirdUi，分别对应 3 个 ui。

MyWidget 类的构造函数：

```
MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    QTabWidget *tabWidget = new QTabWidget(this);
```

```
2 QWidget *w1 = new QWidget;
3 firstUi.setupUi(w1);
4
5 QWidget *w2 = new QWidget;
6 secondUi.setupUi(w2);
7
8 tabWidget->addTab(w1, "First");
9 tabWidget->addTab(w2, "Second");
10 tabWidget->resize(300,300);
11
12 connect(firstUi.childPushButton,SIGNAL(clicked()),this,SLOT(slotChild()));
13 connect(secondUi.closePushButton,SIGNAL(clicked()),this,SLOT(close())));
14 }
```

第1行首先创建一个 QTabWidget 对象。

第 2、3 行创建第一个 ui，首先新建一个 QWidget 对象，以此 QWidget 对象为参数调用第一个 ui 的 setupUi() 函数，生成第一个 ui 页面。

第4、5行以同样的方式创建第二个ui页面。

第 6~8 行在 QTabWidget 对象中插入两个准备好的 ui 页面。

第 9 行连接第一个 ui 页面上 childPushButton 的 clicked() 信号与 slotChild() 槽函数。

第 10 行连接第二个 ui 页面上 closePushButton 的 clicked() 信号与 close() 槽函数，关闭窗口程序。



 小贴士：在使用 ui 页面上的控件时，一定要记得加上 ui 前缀。

实现弹出对话框的槽函数，首先新建一个 QDialog 对象，以此 QDialog 对象为参数调用第三个 ui 对象的 setupUi() 函数，最后调用 exec() 显示此对话框。具体代码如下：

```
void MyWidget::slotChild()
{
    QDialog *dlg = new QDialog;
    thirdUi.setupUi(dlg);
    dlg->exec();
}
```

同样是这个实例的实现，若采用多继承法实现，则要复杂得多，针对每个 ui 页面都需要实现一个类，此处以第三个 ui 对话框为例，以多继承的方式实现它。

新定义一个类 ThirdDialog 继承自 QDialog。在声明时，同时还继承自动生成的 ui 类 Ui::Third，此处是以 private 方式继承，当然也可根据需要以 public 或 protected 方式继承。具体代码如下：

```
#include "ui_third.h"
class ThirdDialog: public QDialog, private Ui::Third
{
    ThirdDialog (QWidget *parent=0);
    .....
}
```

由于 ThirdDialog 是 Ui::Third 的子类，因此可直接在构造函数中调用 setupUi() 函数实现第三个对话框的显示。具体代码如下：

```
ThirdDialog::ThirdDialog (QWidget *parent)
    : ThirdDialog (parent)
{
    setupUi(this);
}
```

在 MyWidget 类的 slotChild() 函数中也要作相应的修改，新建 ThirdDialog 对象并调用 exec() 显示。具体代码如下：

```
void MyWidget::slotChild()
{
    ThirdDialog *dlg = new ThirdDialog;
    dlg->exec();
}
```

从以上对两种继承方式的分析可见，多继承方式可直接对 ui 页面上的控件或函数进行操作调用，代码编写更加简洁；而使用单继承方式，在操作 ui 页面上的控件时需加上 ui 对象前缀，编写代码较为麻烦。但对于程序中所需用到的 ui 页面较多时，使用单继承法则要简单灵活得多，如本实例的实现。因此在实现较为复杂的窗口程序时，建议使用单继承法进行实现。

实例 11 动态加载 ui

知识点：

- QtUiTools 模块
- 动态加载 ui 的条件
- 利用 QUiLoader 实现动态加载.ui 文件

Qt 提供了一个 QtUiTools 模块，包含了与 ui 相关的类，如 QUiLoader，可使程序在



运行中动态加载 Designer 设计生成的.ui 文件，本实例即利用 QUiLoader 类实现实例 10 中的弹出子窗口显示部分。

实现动态加载 ui，首先需在程序中包含 QtUiTools 模块的头文件：

```
#include <QtUiTools>
```

并且新建一个.qrc 文件，描述.ui 文件的路径：

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>forms/third.ui</file>
</qresource>
</RCC>
```

将弹出子窗口的.ui 文件放在 forms 目录下。

在程序的.pro 文件中加入以下两行代码：

```
CONFIG += uitoools
RESOURCES += uiloader.qrc
```

最后修改 slotChild() 槽函数的实现代码如下：

```
void MyWidget::slotChild()
{
    1   QUiLoader loader;
    2   QFile file("./forms/third.ui");
    3   file.open(QFile::ReadOnly);
    4   QWidget *third = loader.load(&file);
    5   file.close();
    6   third->show();
}
```

第 1 行新建一个 QUiLoader 对象。

第 2 行指定所需.ui 文件的路径，新建一个 QFile 对象。

第 3 行以只读方式打开此文件。

第 4 行调用 QUiLoader 对象的 load() 函数将.ui 文件装载到一个 QWidget 对象中，并将此 QWidget 对象返回。

第 5 行关闭文件。

第 6 行调用 show() 显示此子窗口。

这种动态加载的方式不用生成 ui_third.h 文件，在程序运行时才会被加载。采用这种方式最大的好处是可以在不重新编译程序的情况下，改变窗口的布局和显示。但也存在不方便的地方，即在主程序中无法对子窗口的控件进行操作。

第 2 章 布局管理

Qt 为程序开发提供了灵活、强大的布局管理方法，包括基本布局类 QLayout、多文档、分割窗、依靠窗、堆栈窗等。

布局类 QLayout 是为手动编写代码而设计的，因此非常容易理解，使用起来也很方便，初学者刚接触布局可能会觉得太麻烦，不如使用 setGeometry() 直接设置控件的位置简洁方便，这种直接使用 setGeometry 手动布局的方式在窗口程序简单、控件较少的情况下可能更方便，但对于复杂窗口布局则就相当麻烦，大量的时间、精力花在控件位置的计算上，并且完成后再想进行调整就会变得十分麻烦，哪怕只是移动一个控件，也要相关地移动其他许多控件的位置。而使用布局类 QLayout 进行窗口布局则要简单得多，并且窗口布局需要调整时，也非常方便。

本章将通过 6 个实例对 Qt 与布局相关的类进行分析：

- 基本布局管理
- 多文档
- 分割窗口
- 停靠窗口
- 堆栈窗体
- 综合布局实例



实例 12 基本布局管理

知识点：

- 各种布局类的使用
- QHBoxLayout
- QVBoxLayout
- QGridLayout

本实例利用基本布局管理（QHBoxLayout、QVBoxLayout、QGridLayout）实现一个类似 QQ 的用户资料修改页面。实现效果图如图 2-1 所示。

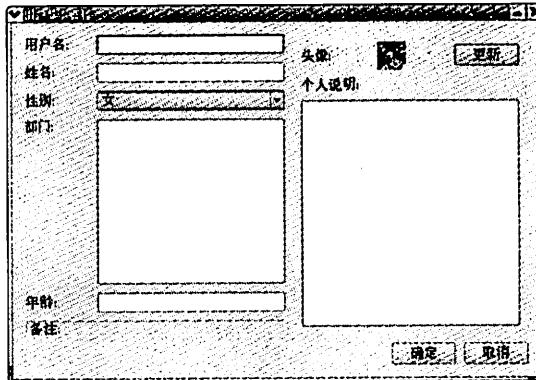


图 2-1 实例效果图

Qt 提供的布局类以及它们之间的继承关系如图 2-2 所示。

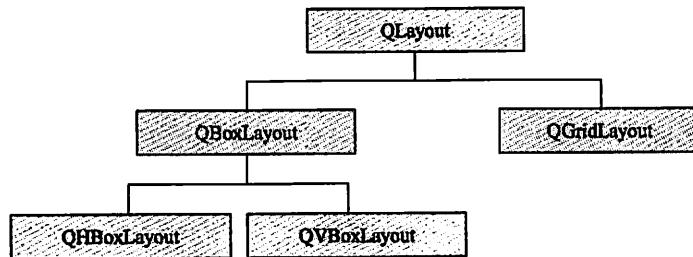


图 2-2 各布局类之间的结构关系图

常用到的布局类有 QBoxLayout、QVBoxLayout、QGridLayout 3 种，分别是水平排列布局、垂直排列布局和表格排列布局。Qt3 中的 QHBox 和 QVBox 到 Qt4 以后被废弃。布局中最常用的方法有 addWidget() 和 addLayout()，addWidget() 方法用于在布局中插入控件，addLayout() 用于在布局中插入子布局。

下面通过实例的实现过程了解布局管理的使用方法。首先通过一个示意图了解此对话框的布局结构，如图 2-3 所示。

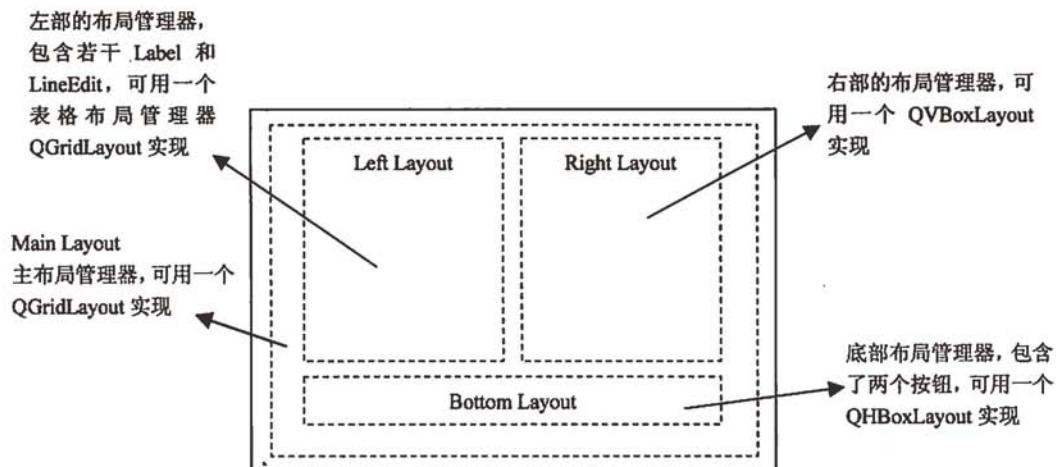


图 2-3 实例布局结构图

从图 2-3 中可知，本实例共用到 4 个布局管理器，分别是 LeftLayout、RightLayout、BottomLayout 和 MainLayout。

下面是具体的实现，首先在头文件 layoutDlg.h 中声明对话框中的各个控件。

```
class LayoutDlg : public QDialog
{
    Q_OBJECT
public:
    LayoutDlg(QWidget *parent = 0, Qt::WindowFlags f = 0);

    QLabel *label1;
    QLabel *label2;
    QLabel *label3;
    QLabel *label4;
    QLabel *label5;
    QLabel *label6;
```



```

    QLabel * label7;
    QLabel * labelOther;
    QLabel * labelIcon;
    QLineEdit * lineEditUser;
    QLineEdit * lineEditName;
    QComboBox * comboBoxSex;
    QTextEdit * textEditDepartment;
    QLineEdit * lineEditAge;
    QTextEdit * textEditDisc;
    QPushButton * pushButtonIcon;
    QPushButton * pushButtonOK;
    QPushButton * pushButtonExit;
};


```

自定义一个对话框类 LayoutDlg，继承自 QDialog，定义对话框中包含的控件。

```

LayoutDlg::LayoutDlg(QWidget *parent, Qt::WindowFlags f)
    : QDialog(parent,f)
{
    setWindowTitle(tr("User Infomation"));
}

```

第 1 行设置对话框的标题。

```

2     label1 = new QLabel(tr("User Name:"));
3     label2 = new QLabel(tr("Name:"));
4     label3 = new QLabel(tr("Sex"));
5     label4 = new QLabel(tr("Department:"));
6     label5 = new QLabel(tr("Age:"));
7     labelOther = new QLabel(tr("Remark"));
8     labelOther->setFrameStyle(QFrame::Panel|QFrame::Sunken);
9     lineEditUser = new QLineEdit;
10    lineEditName = new QLineEdit;
11    comboBoxSex = new QComboBox;
12    comboBoxSex->insertItem(0,tr("Female"));
13    comboBoxSex->insertItem(1,tr("Male"));
14    textEditDepartment = new QTextEdit();
15    lineEditAge = new QLineEdit;

```

第 2~15 行定义对话框左侧的控件。其中，第 8 行设置控件的风格，setFrameStyle() 是 QFrame 的方法，参数以或的方式设定控件的面板风格，由形状 (QFrame::Shape) 和 阴影 (QFrame::Shadow) 两项配合设定。其中，形状有 NoFrame、Panel、Box、HLine、VLine 以及 WinPanel 6 种，阴影有 Plain、Raised 和 Sunken 3 种，具体的效果读者可自行搭配试验。

在定义代码中，可以不必为各个控件指定父窗口，使用布局管理会自动指定布局管理下的所有控件的父窗口。

下面对上段代码定义的控件进行布局，实现左部布局。

```
16   QGridLayout * leftLayout = new QGridLayout();  
  
17   int labelCol = 0;  
18   int contentCol = 1;  
19   leftLayout->addWidget(label1,0,labelCol);           //用户名行  
20   leftLayout->addWidget(lineEditUser,0,contentCol);  
21   leftLayout->addWidget(label2,1,labelCol);           //姓名行  
22   leftLayout->addWidget(lineEditName,1,contentCol);  
23   leftLayout->addWidget(label3,2,labelCol);           //性别行  
24   leftLayout->addWidget(comboBoxSex,2,contentCol);  
25   leftLayout->addWidget(label4,3,labelCol,Qt::AlignTop); //部门行  
26   leftLayout->addWidget(textEditDepartment,3,contentCol);  
27   leftLayout->addWidget(label5,4,labelCol);           //年龄行  
28   leftLayout->addWidget(lineEditAge,4,contentCol);  
29   leftLayout->addWidget(labelOther,5,labelCol,1,2);    //其他行  
  
30   leftLayout->setColumnStretch(0,1);  
31   leftLayout->setColumnStretch(1,3);
```

第 16 行定义一个 QGridLayout 对象 leftLayout，由于此布局管理器并不是主布局管理器，因此不用指定父窗口，最后由主布局管理器统一指定。

QGridLayout 类的 addWidget()方法用来向布局中加入需布局的控件，第 19~29 行调用此方法插入需布局的控件。addWidget()的函数原型如下：

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

widget 参数为需插入的控件对象，row、column 参数为插入的行和列，alignment 参数描述各控件的对齐方式。

第 30 行和第 31 行设定两列分别占用空间的比例，此处设定两列的空间比为 1:3。即使对话框框架大小改变了，两列之间的宽度比依然保持不变。

 小贴士：第 17 行和第 18 行分别定义了两个 int 变量 labelCol 和 ContentCol，分别表示两列，这样是为了在下面的编程过程中不容易写错。

在行数较多而列数较少的情况下，可把列数定义成有意义的名称，这样在后面多次插入控件时就不容易出错了，相反在行数较少列数较多的情况下也是一样处理。



```

32  label7 = new QLabel(tr("Head"));
33  labelIcon = new QLabel();
34  QPixmap icon("icon.png");
35  labelIcon->resize(icon.width(),icon.height());
36  labelIcon->setPixmap(icon);
37  pushButtonIcon = new QPushButton();
38  pushButtonIcon->setText(tr("Change"));
39  QBoxLayout *hLayout = new QBoxLayout;
40  hLayout->setSpacing(20);
41  hLayout->addWidget(label7);
42  hLayout->addWidget(labelIcon);
43  hLayout->addWidget(pushButtonIcon);

```

第 32~43 行实现对话框右上侧的头像选择区的布局，此处采用一个 `QHBoxLayout` 类进行布局管理。`QHBoxLayout` 默认采取自左向右的方式顺序排列插入的控件，也可通过调用 `setDirection()` 方法设定排列的顺序，例如：

```
hLayout->setDirection(QBoxLayout::RightToLeft);
```

第 40 行调用 `QLayout` 的 `setSpacing()` 方法设定各个控件之间的间距为 20。

```

44  label6 = new QLabel(tr("Individual:"));
45  textEditDisc = new QTextEdit();

46  QVBoxLayout *rightLayout = new QVBoxLayout();
47  rightLayout->setMargin(10);
48  rightLayout->addLayout(hLayout);
49  rightLayout->addWidget(label6);
50  rightLayout->addWidget(textEditDisc);

```

这段代码实现对话框右侧的布局。由一个 `QVBoxLayout` 实现布局，`QVBoxLayout` 默认自上而下顺序排列插入的控件或子布局，也可通过 `setDirection()` 方法改变排列的顺序。由于右侧上部的头像选择区已使用布局，因此第 48 行调用 `addLayout()` 方法在布局中插入子布局。

```

51  pushButtonOK = new QPushButton(tr("OK"));
52  pushButtonExit = new QPushButton(tr("Cancel"));
53  QBoxLayout *bottomLayout = new QBoxLayout();
54  bottomLayout->addStretch();
55  bottomLayout->addWidget(pushButtonOK);
56  bottomLayout->addWidget(pushButtonExit);

```

这段代码实现对话框下方两个按钮的布局，采用 `QHBoxLayout` 实现。

第 54 行调用 addStretch()方法在按钮之前插入一个占位符，使两个按钮能靠右对齐。并且在整个对话框的大小发生改变时，保证按钮的大小不发生变化。

 小贴士：合理使用 addStretch()能让界面的布局效果增色不少。

```

57   QGridLayout * mainLayout = new QGridLayout(this);
58   mainLayout->setMargin(15);
59   mainLayout->setSpacing(10);
60   mainLayout->addLayout(leftLayout,0,0);
61   mainLayout->addLayout(rightLayout,0,1);
62   mainLayout->addLayout(bottomLayout,1,0,1,2);
63   mainLayout->setSizeConstraint(QLayout::SetFixedSize);

```

最后实现主布局，用一个 QGridLayout 实现，并在定义主布局时指定父窗口 this，也可调用 this->setLayout(mainLayout)实现。

第 58 行设定对话框的边距为 15。

第 62 行插入的子布局占用了两列，使用的函数方法和前面的是一样的，也调用 addLayout()方法，只是参数不同，原型如下：

```
void addLayout ( QLayout * layout, int row, int column, int rowSpan, int columnSpan, Qt::Alignment alignment = 0 )
```

此处有 6 个参数，layout 参数表示要插入的子布局对象；row、column 参数分别表示插入的起始行、列；rowSpan、columnSpan 参数分别表示占用的行数和列数，插入占用多行或多列的控件调用 addWidget()方法，参数使用亦同此例；alignment 参数指定对齐方式。

例如：

```
gridLayout->addWidget(xxx,1,0,2,2);
```

表示在表格布局的第 1 行第 0 列处插入一个控件 xxx，此控件占用了两行的宽度、两列的高度。

最后，第 63 行设定对话框的控件总是最优化显示，并且用户无法改变对话框的大小。所谓最优化显示，即控件都按其 sizeHint()的大小显示。

 小贴士：在 Qt3 的布局中，插入占用多行或多列的方法为 addMultiCellWidget()和 addMultiCellLayout()，在 Qt4 中废弃了这两种方法，而是统一成 addWidget()和 addLayout()两种方法。

本实例分析了 Qt4 中布局管理常用到的类及其方法，如果读者觉得这种手动布局的方法比较麻烦，也可采用 Qt Designer 来进行布局，参考本书相关内容。



实例 13 多文档

知识点：

- QWorkspace 的使用
- 多文档的各种布局方式

在使用 QMainWindow 作为主窗口时，经常会用到多文档的方式对文件进行显示，本实例通过一个简单的例子分析如何实现多文档的布局方式。实例效果图如图 2-4 所示。

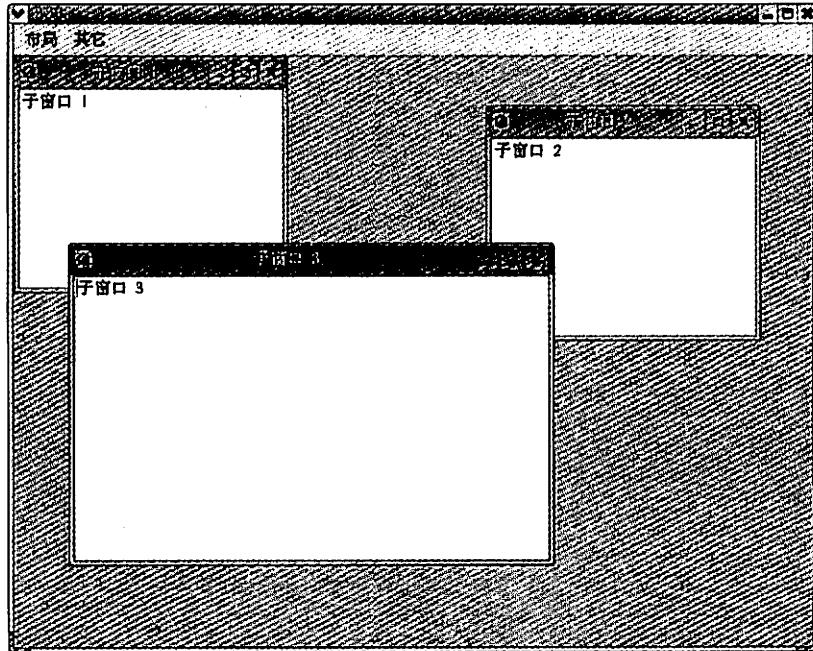


图 2-4 实例效果图

Qt 提供了一个 QWorkspace 类，利用 QWorkspace 类可以很方便地实现多文档的应用。QWorkspace 类继承自 QWidget 类，因此只需在 QMainWindow 主窗口中把 QWorkspace 对象设置为中央窗体即可。QWorkspace 类提供了许多对子窗口进行排序的函数接口，如 cascade()、arrangeIcon()、title()等。

主窗口 MainWidget 类的声明：

```
class MainWidget : public QMainWindow
{
    Q_OBJECT
public:
    MainWidget(QWidget *parent=0);
    void createMenu();
public slots:
    void slotScroll();
private:
    QWorkspace *workSpace;
};
```

第 1 行为 MainWidget 类的构造函数声明。

第 2 行为创建菜单栏的函数。

第 3 行声明的槽函数 slotScroll() 完成对多文档空间 QWorkspace 的滑动条进行设置。

第 4 行声明了一个 QWorkspace 对象。

MainWidget 类的构造函数：

```
MainWidget::MainWidget(QWidget *parent)
    : QMainWindow(parent)
{
    workSpace = new QWorkspace;
    setCentralWidget(workSpace);

    createMenu();

    QMainWIndow *window1 = new QMainWIndow;
    window1->setWindowTitle(tr("window 1"));
    QTextEdit *edit1 = new QTextEdit;
    edit1->setText(tr("Window 1"));
    window1->setCentralWidget(edit1);

    ...
    workSpace->addWindow(window1);
    workSpace->addWindow(window2);
    workSpace->addWindow(window3);
}
```

第 1 行创建一个 QWorkspace 对象 workSpace。

第 2 行设置主窗口的中央窗体为 QWorkspace 对象，以使主窗口能实现多文档的布局方式。



第 3 行调用 `createMenu()` 函数创建主窗口的菜单栏。

第 4~8 行新建一个子窗口用于在主窗口中显示，并重复创建多个子窗口，同时进行显示。

第 9~11 行在 `workSpace` 中插入这些子窗口，即实现了多文档的显示。

菜单栏的创建函数：

```
void MainWidget::createMenu()
{
    //布局菜单
    QMenu *layoutMenu = menuBar()->addMenu(tr("Layout"));

    QAction *arrange = new QAction(tr("Arrange Icons"),this);
    connect(arrange,SIGNAL(triggered()),workSpace,SLOT(arrangeIcons()));
    layoutMenu->addAction(arrange);
```

实现对子窗口的 `arrangeIcons` 布局，它的布局方式是将所有子窗口以标题栏的方式在主窗口的底部进行排列，如图 2-5 所示。可直接把菜单的 `triggered()` 信号与 `QWorkspace` 对象的 `arrangeIcons()` 函数相连。

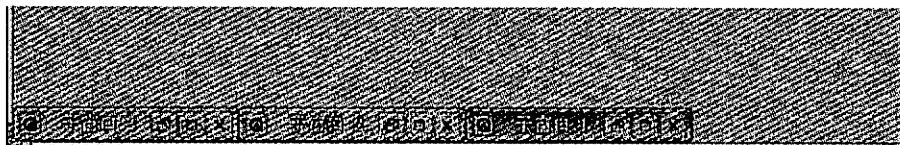


图 2-5 实例效果图



小贴士：注意此排列方式，仅对已经最小化的子窗口起作用。

```
QAction *tile = new QAction(tr("tile"),this);
connect(tile,SIGNAL(triggered()),workSpace,SLOT(tile()));
layoutMenu->addAction(tile);
```

实现对子窗口的 `tile` 布局，`tile` 的意思是用子窗口把主窗口像铺瓦片或贴瓷砖一样排满，如图 2-6 所示，可直接连接 `QWorkspace` 对象的 `tile()` 函数实现。

```
QAction *cascade = new QAction(tr("cascade"),this);
connect(cascade,SIGNAL(triggered()),workSpace,SLOT(cascade()));
layoutMenu->addAction(cascade);
```

实现对子窗口的 `cascade` 布局，即子窗口的层叠显示，如图 2-7 所示，可直接连接 `QWorkspace` 对象的 `cascade()` 函数实现。

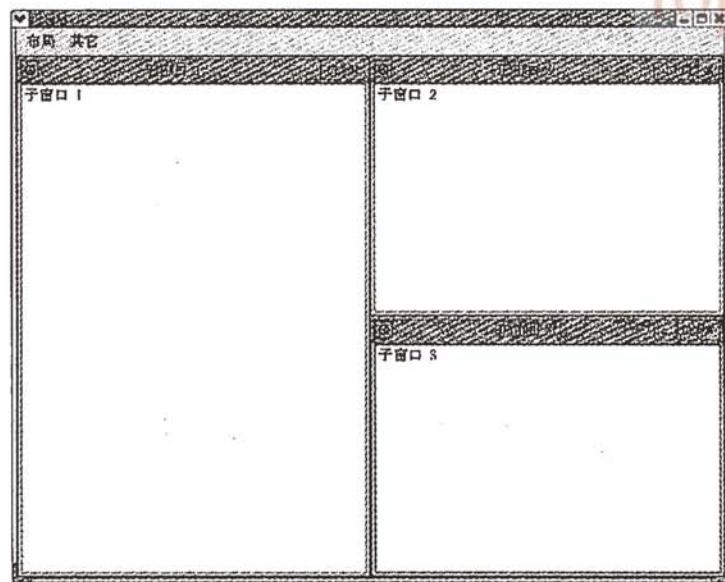


图 2-6 实例效果图

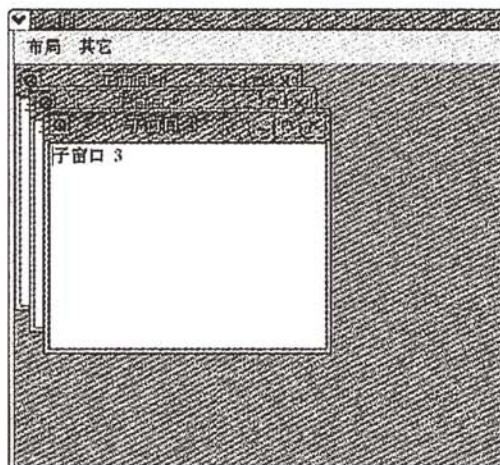


图 2-7 实例效果图

```
//其他菜单
QMenu *otherMenu = menuBar()->addMenu(tr("Other"));

 QAction *scrollAct = new QAction(tr("Scroll"),this);
 connect(scrollAct,SIGNAL(triggered()),this,SLOT(slotScroll()));
 otherMenu->addAction(scrollAct);
```



```

otherMenu->addSeparator();

QAction *nextAct = new QAction(tr("Next"),this);
connect(nextAct,SIGNAL(triggered()),workSpace,SLOT(activateNextWindow()));
otherMenu->addAction(nextAct);

```

使下一个子窗口获得焦点，成为当前应用子窗口，直接把菜单项与 QWorkspace 对象的 activateNextWindow() 函数连接实现。

```

QAction *previousAct = new QAction(tr("Previous"),this);
connect(previousAct,SIGNAL(triggered()),workSpace,
        SLOT(activatePreviousWindow()));
otherMenu->addAction(previousAct);
}

```

使前一个子窗口获得焦点，成为当前应用子窗口，直接把菜单项与 QWorkspace 对象的 activatePreviousWindow() 函数连接实现。

子窗口的顺序由 QWorkspace 的 WindowOrder 属性决定，有以下两种可能的顺序。

- QWorkspace::CreationOrder：子窗口创建的先后顺序。
- QWorkspace::StackingOrder：子窗口堆栈的顺序，即处于最上方的子窗口是最后一个子窗口。

其中， CreationOrder 是默认的子窗口顺序。

当调用 windowList() 函数获得主窗口的所有子窗口列表时，以 .WindowOrder 作为参数，返回的子窗口列表即以指明的顺序进行排列。

滑动条设置函数如下：

```

void MainWidget::slotScroll()
{
    workSpace->setScrollBarsEnabled(!workSpace->scrollBarsEnabled());
}

```

通过 QWorkspace 的 setScrollBarsEnabled() 函数，可对 workSpace 滑动条的可用性进行设置。

实例 14 分割窗口

知识点：

QSplitter 的使用

分割窗口是应用程序中经常用到的，它可以灵活分布窗口的布局，经常用于类似文件资源管理器的窗口设计中。本实例实现一个分割窗口使用的例子，实现的效果图如图 2-8 所示。

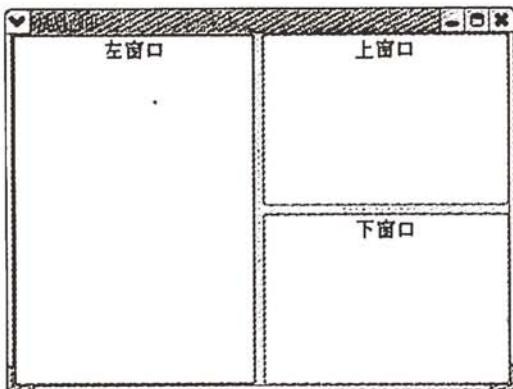


图 2-8 分割窗口效果图

整个对话框由 3 个窗口组成，各窗口之间的大小可随意拖动改变。此实例使用 QSplitter 类来实现，实现代码如下所示：

```
int main( int argc, char * argv[] )
{
    1   QFont font("ZYSong18030",12);
    2   QApplication::setFont(font);

    3   QApplication app(argc, argv);
    4   QTranslator translator();
    5   translator.load("splitter_zh","");
    6   app.installTranslator(&translator);
    //主分割窗口
    7   QSplitter *splitterMain = new QSplitter(Qt::Horizontal,0);
    8   QTextEdit *textLeft = new QTextEdit( QObject::tr("Left Widget"),splitterMain);
    9   textLeft->setAlignment(Qt::AlignCenter);
    //右部分割窗口
   10  QSplitter *splitterRight = new QSplitter(Qt::Vertical,splitterMain);
   11  splitterRight->setOpaqueResize(false);
   12  QTextEdit *textUp = new QTextEdit( QObject::tr("Top Widget"),splitterRight);
   13  textUp->setAlignment(Qt::AlignCenter);
   14  QTextEdit *textBottom = new QTextEdit( QObject::tr("Bottom Widget"),
                                         splitterRight);
```



```
15    textBottom->setAlignment(Qt::AlignCenter);  
16    splitterMain->setStretchFactor(1,1);  
17    splitterMain->setWindowTitle( QObject::tr("Splitter"));  
18    splitterMain->show();  
19    return app.exec();  
}
```

第 1~6 行指定显示字体、翻译文件等信息。

第 7 行定义一个 QSplitter 类对象，为主分割窗口，设定此分割窗为水平分割窗。

第 8 行定义一个 QTextEdit 类对象，并插入主分割窗口中。

第 9 行调用 setAlignment()方法，设定TextEdit 中文字的对齐方式，常用的有以下几种。

- Qt::AlignLeft: 左对齐。
- Qt::AlignRight: 右对齐。
- Qt::AlignCenter: 文字居中（Qt::AlignHCenter 为水平居中，Qt::AlignVCenter 为垂直居中）。
- Qt::AlignUp: 文字与顶端对齐。
- Qt::AlignBottom: 文字与底部对齐。

第 10 行定义一个右部的分割窗口，定义为垂直分割窗，并以主分割窗口为父窗口。

第 11 行调用的方法 setOpaqueResize(bool)用由设定分割窗的分割条在拖动时是否为实时更新显示，若设为 true 则实时更新显示，若设为 false 则在拖动时只显示一条灰色的粗线条，在拖动到位并弹起鼠标后再显示分割条。默认设为 true，这和 Qt3 正好相反，Qt3 中默认为 false。

第 16 行 setStretchFactor()方法用于设定可伸缩控件，它的第一个参数指定设置的控件序号，控件序号按插入的先后次序从 0 起依次编号；第二个参数为大于 0 的值表示此控件为可伸缩控件。此实例中设定右部分割窗为可伸缩控件，当整个对话框的宽度发生改变时，左部的文件编辑框宽度保持不变，右部的分割窗宽度随整个对话框大小的改变进行调整。

实例 15 停靠窗口

知识点：

- QDockWidget 的使用
- 各种停靠方式

□ 停靠窗体特性的设置方法

本实例实现停靠窗口的基本使用方法，实现的效果图如图 2-9 所示。

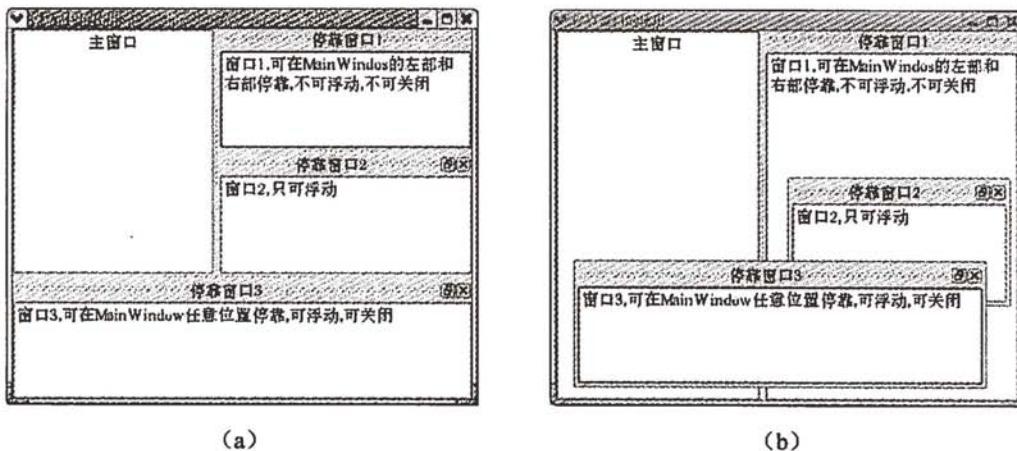


图 2-9 停靠窗口实例效果图

本实例实现的停靠窗口的可实现状态已在各窗口中进行了描述，停靠窗口 1 只可在主窗口的左边和右边停靠；停靠窗口 2 只可在浮动和在右部停靠两种状态间切换，并且不可移动；停靠窗口 3 可实现停靠窗口的各个状态。具体状态此处不再一一用图示的方式列出，读者可自行编译源代码进行试验。

源代码头文件 dockWindows.h:

```
class DockWindows : public QMainWindow
{
    Q_OBJECT
public:
    DockWindows(QWidget *parent=0);
};
```

自定义一个继承自 QMainWindow 的类。停靠窗只在 QMainWindow 中使用。

实现代码 dockWindows.cpp 如下：

```
DockWindows::DockWindows(QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle( tr( "DockWindows" ) );
```



```

2   QTextEdit *te = new QTextEdit( this );
3   te->setText(tr("Main Window"));
4   te->setAlignment(Qt::AlignCenter);
5   setCentralWidget( te );

6   //停靠窗口 1
7   QDockWidget *dock = new QDockWidget(tr("DockWindow 1"), this );
8   dock->setFeatures( QDockWidget::DockWidgetMovable );
9   dock->setAllowedAreas(Qt::LeftDockWidgetArea|Qt::RightDockWidgetArea);
10  QTextEdit *te1 = new QTextEdit();
11  te1->setText(tr("Window 1"));
12  dock->setWidget( te1 );
13  addDockWidget( Qt::RightDockWidgetArea, dock );

14  //停靠窗口 2
15  dock = new QDockWidget( tr("DockWindow 2"), this );
16  dock->setFeatures( QDockWidget::DockWidgetFloatable|QDockWidget::DockWidgetClosable );
17  QTextEdit *te2 = new QTextEdit();
18  te2->setText(tr("Window 2"));
19  dock->setWidget( te2 );
20  addDockWidget( Qt::RightDockWidgetArea, dock );

21  //停靠窗口 3
22  dock = new QDockWidget( tr("DockWindow 3"), this );
23  dock->setFeatures( QDockWidget::AllDockWidgetFeatures );
24  QTextEdit *te3 = new QTextEdit();
25  te3->setText(tr("Window 3"));
26  dock->setWidget( te3 );
27  addDockWidget( Qt::BottomDockWidgetArea, dock );
}

```

第 1 行设置主窗口的标题栏文字。

第 2~5 行定义一个 QTextEdit 对象作为主窗口，并把此编辑框设为 MainWindow 的中央窗体。

第 6~12 行设置停靠窗口 1。

第 13~18 行设置停靠窗口 2。

第 19~24 行设置停靠窗口 3。

设置停靠窗口的一般流程为：

- (1) 创建一个 QDockWidget 对象的停靠窗体。

- (2) 设置此停靠窗体的属性，通常调用 setFeatures() 及 setAllowedAreas() 两种方法。

(3) 新建一个要插入停靠窗体的控件，本实例中为 QTextEdit，也可为其他控件，常用的一般为 QListWidget 和 QTextEdit。

(4) 把控件插入停靠窗体，调用 QDockWidget 的 setWidget()方法。

(5) 使用 addDockWidget()方法在 MainWindow 中加入此停靠窗体。

本实例中的 3 个停靠窗体都是按照此流程实现的，此处需重点介绍的是设置停靠窗体状态的方法 setAllowedAreas()和 setFeatures()。

其中，setAllowedAreas()方法设置停靠窗体可停靠的区域，原型如下：

```
void setAllowedAreas ( Qt::DockWidgetAreas areas )
```

参数 Qt::DockWidgetAreas 指定了停靠窗体可停靠的区域，包括以下几种。

- Qt::LeftDockWidgetArea：可在主窗口的左侧停靠。
- Qt::RightDockWidgetArea：可在主窗口的右侧停靠。
- Qt::TopDockWidgetArea：可在主窗口的顶端停靠。
- Qt::BottomDockWidgetArea：可在主窗口的底部停靠。
- Qt::AllDockWidgetAreas：可在主窗口任意（以上四个）部位停靠。
- Qt::NoDockWidgetArea：只可停靠在插入处。

各区域设定可采用或 (|) 的方式进行设定，如本实例中的第 8 行。

setFeatures()方法设置停靠窗体的特性，原型如下：

```
void setFeatures ( DockWidgetFeatures features )
```

参数 QDockWidget::DockWidgetFeature 指定停靠窗体的特性，包括以下几种。

- QDockWidget::DockWidgetClosable：停靠窗可关闭，右上角的关闭按钮。
- QDockWidget::DockWidgetMovable：停靠窗可移动。
- QDockWidget::DockWidgetFloatable：停靠窗可浮动。
- QDockWidget::AllDockWidgetFeatures：此参数表示拥有停靠窗的所有特性。
- QDockWidget::NoDockWidgetFeature：不可移动、不可关闭、不可浮动。

此参数也可采用或 (|) 的方式进行特性的设定，如实例中的第 14 行所示。

实例 16 堆栈窗体

知识点：

QStackedWidget 的使用



本实例实现一个堆栈窗体的使用，实现的效果图如图 2-10 所示。

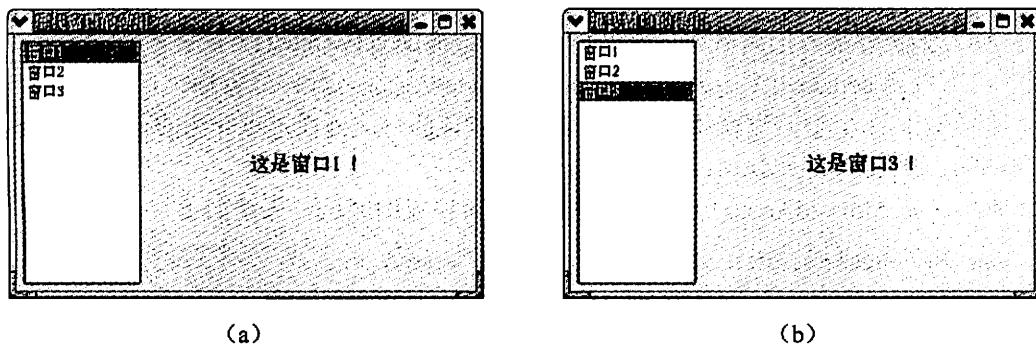


图 2-10 堆栈窗口的实例效果图

选择左侧列表框中不同的选项，右侧则显示所选的窗体。

实现头文件 stackdlg.h:

```
class StackDlg : public QDialog
{
    Q_OBJECT
public:
    StackDlg(QWidget *parent = 0, Qt::WindowFlags fl = 0);

    QLabel *label1;
    QLabel *label2;
    QLabel *label3;
    QListWidget *list;
    QStackedWidget *stack;
};
```

在头文件中定义一个对话框，并声明所用到的控件。

实现源文件 stackdlg.cpp:

```
StackDlg::StackDlg(QWidget *parent, Qt::WindowFlags fl)
    : QDialog(parent, fl)
{
    1   setWindowTitle(tr("Stacked Widgets"));

    2   list = new QListWidget(this);
    3   list->insertItem(0, tr("Window 1"));
    4   list->insertItem(1, tr("Window 2"));
    5   list->insertItem(2, tr("Window 3"));
}
```

```
6  label1 = new QLabel(tr("This is Window 1 !"));
7  label2 = new QLabel(tr("This is Window 2 !"));
8  label3 = new QLabel(tr("This is Window 3 !"));

9  stack = new QStackedWidget(this);
10 stack->addWidget(label1);
11 stack->addWidget(label2);
12 stack->addWidget(label3);

13 QHBoxLayout *mainLayout = new QHBoxLayout(this);
14 mainLayout->setMargin(5);
15 mainLayout->setSpacing(5);
16 mainLayout->addWidget(list);
17 mainLayout->addWidget(stack,0,Qt::AlignHCenter);
18 mainLayout->setStretchFactor(list,1);
19 mainLayout->setStretchFactor(stack,3);
20 connect(list,SIGNAL(currentRowChanged(int)),stack,
           SLOT setCurrentIndex(Int)));
}
```

第 2~5 行创建一个 QListWidget 控件，并在控件中插入 3 个条目，作为选择项。

第 6~8 行创建 3 个 QLabel 标签控件，作为堆栈窗口需显示的三层窗体。

第 9 行创建一个 QStackWidget 堆栈窗。

第 10~12 行调用 addWidget()方法把前面创建的 3 个标签控件依次插入堆栈窗中。

第 13~19 行使用 QHBoxLayout 对整个对话框进行布局。

第 20 行连接 QListWidget 的 currentRowChanged()信号与堆栈窗的 setCurrentIndex()槽，实现按选择显示窗体。此处的堆栈窗体 index 按插入的顺序从 0 起依次排序，与 QListWidget 的条目排序相一致。

本实例分析了堆栈窗的基本使用方法。在实际应用中，堆栈窗口多与列表框 QListWidget 及下拉列表框 QComboBox 配合使用。

实例 17 综合布局实例

知识点：

各种布局方式的综合使用



本实例综合应用前面介绍的布局方法实现一个复杂的窗口布局，实现效果图如图 2-11 所示。其中包括了基本布局、分割窗以及堆栈窗。

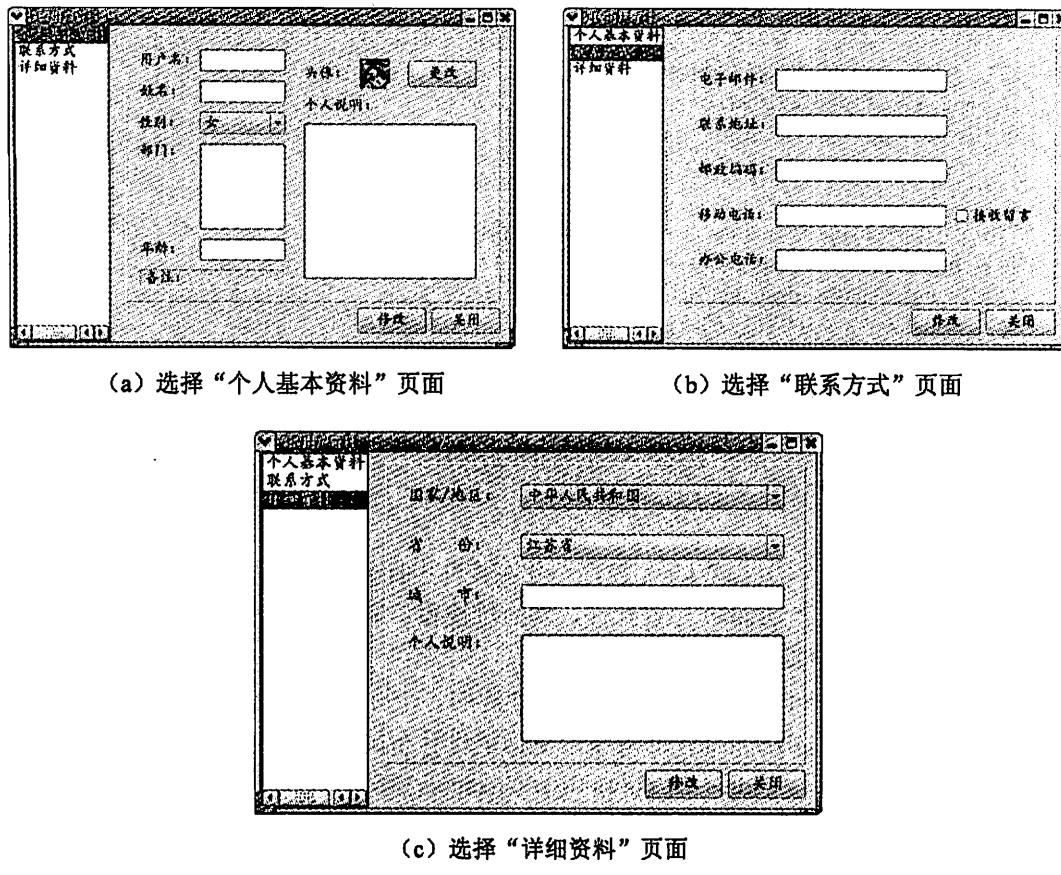


图 2-11 综合布局实例效果图

首先对整个窗体的构成进行一个整体的分析。最外层的是一个分割窗体 `QSplitter`，分割窗的左侧为一个 `QListWidget`，右侧为一个 `QVBoxLayout` 布局，包括一个堆栈窗 `QStackWidget` 和一个按钮布局，在堆栈窗中包含 3 个窗体，每个窗体采用基本布局方式进行布局管理。窗体的布局可用如图 2-12 所示的示意图表示。

堆栈窗中的 3 个页面分别定义了 3 个 `QWidget` 子类，包括“个人基本资料”页，由 `BaseInfo` 类实现；“联系方式”页，由 `Content` 类实现；“详细资料”页，由 `Detail` 类实现。

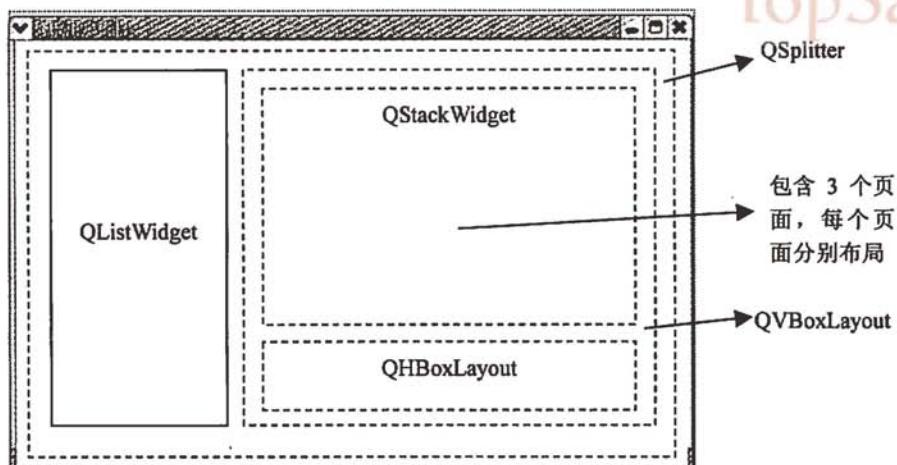


图 2-12 布局结构图

“个人基本资料”页与实例 12 中的内容一样，可直接引用。

“联系方式”页和“详细资料”页都是采用表格布局的方式进行布局管理，用法与前面所介绍的类似，此处不再重复进行分析。

本例重点分析对话框右侧采用 QVBoxLayout 的布局，实例中为右侧的页面定义了独立的类 Content，实现头文件如下：

```
#include "contact.h"
#include "baseinfo.h"
#include "detail.h"

class Content : public QFrame
{
    Q_OBJECT
public:
    Content(QWidget *parent=0, Qt::WindowFlags f1=0);

    QPushButton *pushButtonAmend;
    QPushButton *pushButtonClose;
    QStackedWidget *stack;
private:
    BaseInfo *baseInfo;
    Contact *contact;
    Detail *detail;
};
```



头文件中首先包含自定义的 3 个页面类的头文件，声明了两个按钮对象及一个堆栈窗体对象，并声明了 3 个页面类对象。

实现源代码如下：

```
Content::Content(QWidget *parent, Qt::WindowFlags f1)
    : QFrame(parent,f1)
{
    1 stack = new QStackedWidget();
    2 stack->setFrameStyle(QFrame::Panel|QFrame::Raised);
    3 baseInfo = new BaseInfo();
    4 contact = new Contact();
    5 detail = new Detail();
    6 stack->addWidget(baseInfo);
    7 stack->addWidget(contact);
    8 stack->addWidget(detail);

    9 pushButtonAmend = new QPushButton();
    10 pushButtonAmend->setText(tr("Demand"));
    11 pushButtonClose = new QPushButton();
    12 pushButtonClose->setText(tr("Close"));
    13 QHBoxLayout *buttonLayout = new QHBoxLayout;
    14 buttonLayout->addStretch(1);
    15 buttonLayout->addWidget(pushButtonAmend);
    16 buttonLayout->addWidget(pushButtonClose);

    17 QVBoxLayout *mainlayout = new QVBoxLayout(this);
    18 mainlayout->setMargin(10);
    19 mainlayout->setSpacing(6);
    20 mainlayout->addWidget(stack);
    21 mainlayout->addLayout(buttonLayout);

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

第 1~8 行创建一个 QStackWidget 对象，第 2 行调用 QFrame 的 setFrameStyle()方法，对堆栈窗的显示风格进行设置。

第 6、7、8 行在堆栈窗中顺序插入“个人基本资料”、“联系方式”、“详细资料”3 个页面。

第 9~16 行创建两个按钮，并用 QHBoxLayout 对其进行布局。

第 17~21 行采用 QVBoxLayout 生成主布局。

第 22 行完成退出按钮功能。

 小贴士：在程序中需要用到程序自身的指针时，可使用 qApp，它是一个 QApplication 对象。

最后的对话框布局在 main.cpp 文件中由 QSplitter 完成，main.cpp 文件如下：

```
#include "content.h"  
.....  
int main( int argc, char * argv[] )  
{  
1   QApplication app(argc, argv);  
2   QFont font("AR PL KaitiM GB",12);  
3   app.setFont(font);  
  
4   QTranslator translator;  
5   translator.load("combine_zh");  
6   app.installTranslator(&translator);  
  
7   QSplitter *splitterMain = new QSplitter(Qt::Horizontal,0);  
8   splitterMain->setOpaqueResize(true);  
9   QListWidget *list = new QListWidget(splitterMain);  
10  list->insertItem(0,QObject::tr("Base Info"));  
11  list->insertItem(1,QObject::tr("Contact"));  
12  list->insertItem(2,QObject::tr("Detail"));  
  
13  Content *content = new Content(splitterMain);  
14  QObject::connect(list,SIGNAL(currentRowChanged(int)),  
                   content->stack,SLOT(setcurrentIndex(int)));  
  
15  splitterMain->setWindowTitle(QObject::tr("Change User Info"));  
16  splitterMain->setMinimumSize(splitterMain->minimumSize());  
17  splitterMain->setMaximumSize(splitterMain->minimumSize());  
18  splitterMain->show();  
19  return app.exec();  
}
```

第 2、3 行设定整个程序采用的字体与字号，本实例使用楷体 12 号字。

第 7 行创建一个水平分割窗，作为主布局框。

第 8~12 行在水平分割窗的左侧窗体中插入一个 QListWidget 作为条目选择框，并在列表框中依次插入相应的条目。

第 13 行在水平分割窗的右侧插入 Content 类对象。



第 14 行把列表框的 currentRowChanged() 信号与堆栈窗的 setCurrentIndex() 槽相连接，达到按用户选择的条目显示页面的要求。

本实例综合应用了各种布局方式，完成一个较为复杂的界面显示。包括了各种基本布局类的应用、堆栈窗的应用和分割窗的应用。

要达到同样的显示效果，会有多种可能的布局方案，在实际应用中，应根据具体情况选择，使用最方便合理的布局方式。一般来说，QGridLayout 功能较为强大，能完成 QHBoxLayout 与 QVBoxLayout 的功能，但视具体情况，若只是简单的控件水平或竖直排列，使用 QHBoxLayout 和 QVBoxLayout 更加方便，QGridLayout 适合较为整齐方正的界面布局。

第3章 对话框进阶

本章在前面章节分析的基础上，通过 6 个实例对 Qt 程序开发中的一些应用效果及特殊功能需求进行了分析，如可扩展对话框、窗体的淡入淡出效果等。

- 可扩展对话框
- 利用 QPalette 改变控件颜色
- 窗体的淡入淡出效果
- 不规则窗体
- 电子钟
- 程序启动画面



实例 18 可扩展对话框

知识点：

利用 `setSizeConstraint(QLayout::SetFixedSize)` 方法使对话框尺寸保持相对固定

可扩展对话框一般用于使用用户有区分的场合。通常情况下，只出现基本的对话窗体；当有高级用户使用，或需要更多信息时，通过某种方式的切换显示完整的对话窗体，切换的工作通常由一个按钮来实现。本实例即实现了一个简单的填写资料的例子，通常情况下，只需填写姓名和性别，在有特殊需要时，还需填写更多信息则切换至完整对话窗体，如图 3-1 所示。

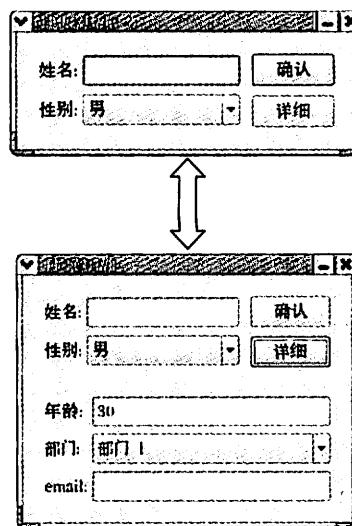


图 3-1 可扩展对话框实例效果图

当单击“详细”按钮时，对话框扩展，显示其他更详细的信息，再次单击“详细”按钮，扩展窗口又重新隐藏。

具体实现代码如下：

```
class Extension : public QDialog
{
    Q_OBJECT
```

```
public:  
    Extension(QWidget *parent=0);  
    void createBaseInfo();  
    void createDetailInfo();  
public slots:  
    void slotExtension();  
private:  
    QWidget *baseWidget;  
    QWidget *detailWidget;  
};
```

新建一个可扩展对话框类 Extension，继承自 QDialog。其中，私有变量 baseWidget 和 detailWidget 分别表示基本对话窗体部分和扩展窗体部分。与之对应的 createBaseInfo() 和 createDetailInfo() 函数分别用于实现这两部分窗体的内容。

Extension() 函数的实现代码如下：

```
Extension::Extension(QWidget *parent)  
    : QDialog(parent)  
{  
1   setWindowTitle(tr("Extension Dialog"));  
  
2   createBaseInfo();  
3   createDetailInfo();  
  
4   QVBoxLayout *layout = new QVBoxLayout;  
5   layout->addWidget(baseWidget);  
6   layout->addWidget(detailWidget);  
7   layout->setSizeConstraint(QLayout::SetFixedSize);  
8   layout->setSpacing(10);  
9   setLayout(layout);  
}
```

第 1 行设置对话框的标题栏信息。

第 2、3 行调用 createBaseInfo() 和 createDetailInfo() 方法分别构建两部分窗体的内容。

第 4~9 行对整个对话框进行布局，其中，调用 setSizeConstraint(QLayout::SetFixedSize) 设置窗体的大小固定，不能经过拖动改变大小，否则当再次单击“详细”按钮时，对话框不能恢复到初始状态。

createBaseInfo() 函数的实现代码如下：

```
void Extension::createBaseInfo()  
{
```



```
baseWidget = new QWidget;

QLabel *nameLabel = new QLabel(tr("Name:"));
QLineEdit *nameEdit = new QLineEdit;
QLabel *sexLabel = new QLabel(tr("Sex:"));
QComboBox *sexComboBox = new QComboBox;
sexComboBox->addItem(tr("male"));
sexComboBox->addItem(tr("female"));

QPushButton *okButton = new QPushButton(tr("OK"));
QPushButton *detailButton = new QPushButton(tr("Detail"));
connect(detailButton,SIGNAL(clicked()),this,SLOT(slotExtension()));

QDialogButtonBox *btnBox = new QDialogButtonBox(Qt::Vertical);
btnBox-> addButton(okButton,QDialogButtonBox::ActionRole);
btnBox-> addButton(detailButton,QDialogButtonBox::ActionRole);
.....
}
```

createBaseInfo()完成基本信息窗体部分的构建，基本控件的创建与布局此处不再赘述，关键的部分在于实现切换功能的“详细”按钮 detailButton 的实现，连接 detailButton 的 clicked 信号与 slotExtension()槽函数以实现对话框的可扩展。

createDetailInfo()函数的实现代码如下：

```
void Extension::createDetailInfo()
{
    detailWidget = new QWidget;

    QLabel *label1 = new QLabel(tr("Age:"));
    QLineEdit *ageEdit = new QLineEdit;
    ageEdit->setText("30");
    QLabel *label2 = new QLabel(tr("Department:"));
    QComboBox *deptComboBox = new QComboBox;
    deptComboBox->addItem(tr("dept 1"));
    deptComboBox->addItem(tr("dept 2"));
    deptComboBox->addItem(tr("dept 3"));
    deptComboBox->addItem(tr("dept 4"));
    QLabel *label3 = new QLabel(tr("email:"));
    QLineEdit *edit = new QLineEdit;
    .....
    detailWidget->hide();
}
```

此函数实现详细信息窗体部分 detailWidget 的构建，并在函数的最后调用 hide()隐藏此部分窗体。

slotExtension()函数的实现代码如下：

```
void Extension::slotExtension()
{
    if (detailWidget->isHidden())
        detailWidget->show();
    else
        detailWidget->hide();
}
```

此函数完成窗体扩展切换的工作，在用户单击 detailButton 时调用此函数，首先检测 detailWidget 窗体处于何种状态。若此时是隐藏状态，则应用 show()函数显示 detailWidget 窗体，否则调用 hide()隐藏 detailWidget 窗体。

通过本实例的分析，可了解可扩展对话框的基本实现方法，其中最关键的部分有以下两点：

(1) 在整个对话框的构造函数中调用。

```
layout->setSizeConstraint(QLayout::SetFixedSize);
```

这个设置保证了对话框的尺寸保持相对固定，始终是各控件组合的默认尺寸，在扩展部分显示时，对话框尺寸根据需显示的控件进行扩展调整，而在扩展部分隐藏时，对话框尺寸又恢复至初始状态。

(2) 切换按钮的实现。整个窗体可扩展的工作都是在此按钮所连接的槽函数中完成。

实例 19 利用 QPalette 改变控件颜色

知识点：

- 利用 QPalette 改变控件颜色的方法
- QPalette::ColorGroup 和 QPalette::ColorRole 的概念

在实际应用中，常常会需要改变某个控件的颜色外观，如背景、文字颜色等，Qt 提供的调色板类 QPalette 专门用于管理对话框的外观显示。本实例即通过一个具体的例子，分析如何利用 QPalette 来改变窗体中控件的颜色，如图 3-2 所示。

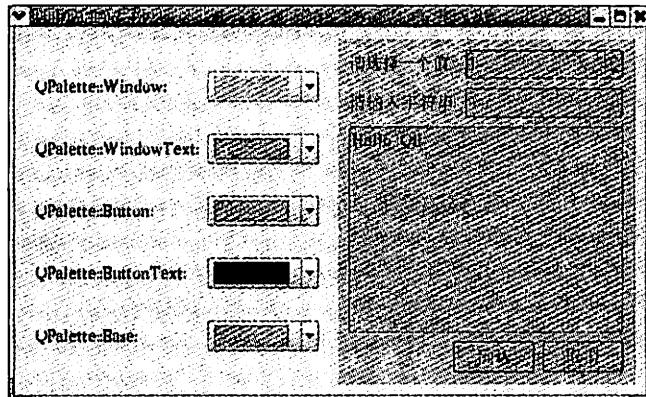


图 3-2 实例效果图

QPalette 类相当于对话框或是控件的调色板，它管理着控件或窗体的所有颜色信息，每个窗体或控件都包含一个 QPalette 对象，在显示时按照它的 QPalette 对象中对各部分各状态下的颜色的描述来进行绘制。就像油漆匠的油漆计划，当要刷墙时，到计划中去查一下墙需要刷成什么颜色；当要刷门时，到计划中去查一下门需要刷成什么颜色。采用这种方式可以很方便地对窗体的各种颜色信息进行管理。

QPalette 类有两个基本的概念，一个是 ColorGroup，另一个是 ColorRole。其中，ColorGroup 指的是 3 种不同的状态，包括以下几种。

- ❑ QPalette::Active：获得焦点的状态。
- ❑ QPalette::Inactive：未获得焦点的状态。
- ❑ QPalette::Disable：不可用状态。

通常情况下，Active 状态与 Inactive 状态下颜色显示是一致的，当然也可根据需要设置成不一样的颜色。

ColorRole 指的是颜色主题，即对窗体中不同部位颜色的分类，如 QPalette::Window 是指背景色，QPalette::WindowText 指的是前景色等，由于数量较多，此处不一一列举，具体使用时可查阅 Qt 的在线帮助。

QPalette 类使用最多，最重要的函数是 setColor() 函数，其原型如下：

```
QPalette::setColor(ColorRole r, const QColor & c)  
QPalette::setColor(ColorGroup gr, ColorRole r, const QColor & c)
```

第一个 setColor() 函数对某个主题的颜色进行设置，并不区分状态；第二个 setColor() 函数在对主题颜色进行设置的同时，还区分了状态，即对某个主题在某个状态下的颜色进行了设置。

QPalette 类同时还提供了 setBrush()函数，通过画刷的设置来对显示进行更改，这样就有可能使用图片而不仅是单一的颜色来对主题进行填充了。

Qt 之前版本中有关背景色设置的函数如 setBackgroundColor()或是前景色设置的函数如 setForegroundColor()在 Qt4 中都被废止，统一由 QPalette 类进行管理。

如 setBackgroundColor()函数可由以下语句代替：

```
xxx->setAutoFillBackground(true);
QPalette p = xxx->palette();
p.setColor(QPalette::Window, color); // p.setBrush(QPalette::Window, brush);
xxx->setPalette(p);
```

如果并不是用单一的颜色填充背景，也可将 setColor()函数换成 setBrush()函数对背景主题进行设置。

 小贴士：注意要先调用 setAutoFillBackground(true)设置窗体自动填充背景。

本实例实现的窗体分为两个部分，一部分用于对不同主题颜色的选择，另一部分用于显示选择的颜色对于窗体外观的改变。

具体实现代码如下：

```
class Palette : public QDialog
{
    Q_OBJECT
public:
    Palette(QWidget *parent = 0);

    void createCtrlFrame();
    void createContentFrame();
    void fillColorList(QComboBox *);

public slots:
    void slotWindow();
    void slotWindowText();
    void slotButton();
    void slotButtonText();
    void slotBase();

private:
    QFrame *ctrlFrame;           //颜色选择面板
    QFrame *contentFrame;        //具体显示面板
    QComboBox * windowComboBox;
    QComboBox * windowTextComboBox;
    QComboBox * buttonComboBox;
    QComboBox * buttonTextComboBox;
```



```
    QComboBox * baseComboBox;  
};
```

定义 Palette 类，继承自 QDialog 类，类声明中对所用到的函数和控件进行了声明。

其中，createCtrlFrame()函数完成窗体左半部分颜色选择区的创建，createContentFrame()函数完成窗体右半部分的创建，fillColorList()函数完成向颜色下拉列表框中插入颜色的工作。

createCtrlFrame()函数用于创建颜色选择区。具体代码如下：

```
void Palette::createCtrlFrame()  
{  
    .....  
    // QPalette::Window 颜色选择下拉列表框  
1    windowComboBox = new QComboBox;  
2    fillColorList(windowComboBox);  
3    connect(windowComboBox,SIGNAL(activated(int)),this,SLOT(slotWindow()));  
  
    //QPalette::WindowText 颜色选择下拉列表框  
4    windowTextComboBox = new QComboBox;  
5    fillColorList(windowTextComboBox);  
6    connect(windowTextComboBox,SIGNAL(activated(int)),this,  
             SLOT(slotWindowText()));  
    .....  
}
```

第 1~3 行创建对 QPalette::Window 背景色的选择下拉列表框。

第 1 行创建一个 QComboBox 对象。

第 2 行调用 fillColorList() 函数向下拉列表框中插入各种不同的颜色选项。

第 3 行连接下拉列表框的 activated() 信号与改变背景色的槽函数 slotWindow()。

其他颜色选择控件的创建与此类似，这里不再重复。

slotWindow() 函数用于响应对背景颜色的选择。具体代码如下：

```
void Palette::slotWindow()  
{  
1    QStringList colorList = QColor::colorNames();  
2    QColor color = QColor(colorList>windowComboBox->currentIndex());  
3    QPalette p = contentFrame->palette();  
4    p.setColor(QPalette::Window,color);  
5    contentFrame->setPalette(p);  
}
```

第1、2行获得当前选择的颜色值。

第3行调用 QWidget 类的 palette()函数，获得右部窗体 contentFrame 的调色板信息。

第4行调用 QPalette 类的 setColor()函数，设置 contentFrame 窗体的 Window 类颜色，即背景色，setColor()的第一个参数为设置的颜色主题，第二个参数为具体的颜色值。

第5行调用 QWidget 的 setPalette()把修改好的调色板信息应用回 contentFrame 窗体中，更新显示。

其他颜色选择响应槽函数与此函数类似。

slotWindowText()函数响应对文字颜色的选择，也即对前景色进行设置。具体代码如下：

```
void Palette::slotWindowText()
{
    QStringList colorList = QColor::colorNames();
    QColor color = QColor(colorList>windowTextComboBox->currentIndex());
    QPalette p = contentFrame->palette();
    p.setColor(QPalette::WindowText,color);
    contentFrame->setPalette(p);
}
```

slotButton()函数响应对按钮背景色的选择。具体代码如下：

```
void Palette::slotButton()
{
    QStringList colorList = QColor::colorNames();
    QColor color = QColor(colorList[buttonComboBox->currentIndex()]);
    QPalette p = contentFrame->palette();
    p.setColor(QPalette::Button,color);
    contentFrame->setPalette(p);
}
```

slotButtonText()函数响应对按钮上文字颜色的选择。具体代码如下：

```
void Palette::slotButtonText()
{
    QStringList colorList = QColor::colorNames();
    QColor color = QColor(colorList[buttonTextComboBox->currentIndex()]);
    QPalette p = contentFrame->palette();
    p.setColor(QPalette::ButtonText,color);
    contentFrame->setPalette(p);
}
```

slotBase()函数响应对可输入文本框背景色的选择。具体代码如下：



```
void Palette::slotBase()
{
    QStringList colorList = QColor::colorNames();
    QColor color = QColor(colorList[baseComboBox->currentIndex()]);
    QPalette p = contentFrame->palette();
    p.setColor(QPalette::Base,color);
    contentFrame->setPalette(p);
}
```

fillColorList()函数用于插入颜色。具体代码如下：

```
void Palette::fillColorList(QComboBox * combobox)
{
    1   QStringList colorList = QColor::colorNames();

    2   QString color;
    3   foreach(color,colorList)
    {
        4       QPixmap pix(QSize(70,20));
        5       pix.fill(QColor(color));
        6       combobox->addItem(QIcon(pix),NULL);
        7       combobox->setIconSize(QSize(70,20));
        8       combobox->setSizePolicy(QComboBox::AdjustToContents);
    }
}
```

第 1 行调用 QColor 类的 colorNames() 函数获得 Qt 所有知道名称的颜色名称列表，返回的是一个字符串列表 colorList。

第 2 行新建一个 QString 对象，为循环遍历作准备。

第 3~8 行采用 foreach 的方式对颜色名称列表进行遍历。

第 4 行新建一个 QPixmap 对象 pix 作为显示颜色的图标。

第 5 行为 pix 填充当前遍历的颜色。

第 6 行调用 QComboBox 的 addItem() 函数为下拉列表框插入一个条目，并以准备好的 pix 作为插入条目的图标，名称设为 NULL，即不显示颜色的名称。

第 7 行设置图标的尺寸，图标默认尺寸是一个方形，把它设置为与 pix 相同尺寸的长方形。

第 8 行调用 QComboBox 的 setSizePolicy() 设置下拉列表框的尺寸调整策略为 AdjustToContents，符合内容的大小。

实例 20 窗体的淡入淡出效果

知识点：

- 淡入淡出效果实现的原理
- 利用定时器控制淡入淡出窗体的变化过程

本实例实现一个窗体淡入淡出效果的例子，当窗体进行页面切换时，原页面的消失和新页面的显现并不是瞬间切换的，而是逐渐消隐和逐渐显现的过程。

本实例实现淡入淡出效果的基本原理可由图 3-3 描述。

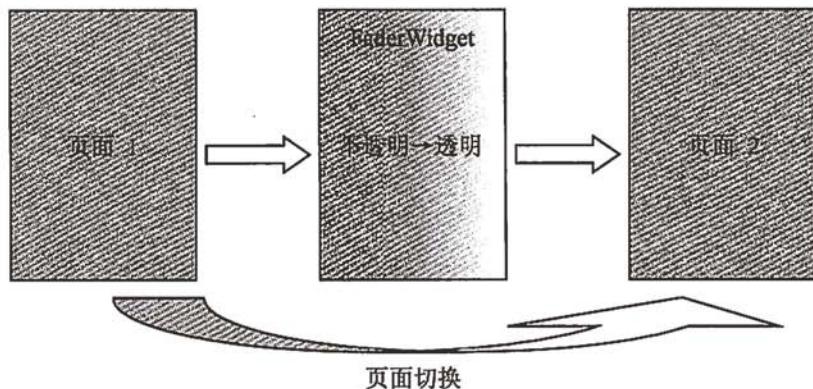


图 3-3 淡入淡出效果实现原理图

当对话框由页面 1 切换至页面 2 时，在响应页面切换命令的同时，新建一个 FaderWidget 窗体，此窗体是一个与对话框等尺寸的空白窗体，此窗体由不透明逐渐变为完全透明，即实现页面的淡入淡出效果。

本实例将实例 17 的实现改造成淡入淡出效果，主对话框由一个列表框 QListWidget 和一个堆栈窗体 QStackedWidget 组成，列表框中列出了可显示的 3 个页面名称：“基本资料”、“联系方式”和“详细资料”，如图 3-4 所示。堆栈窗体中包含了对应的 3 个页面。

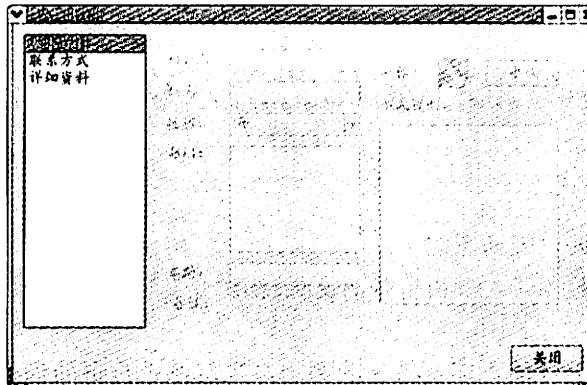


图 3-4 实例效果图

窗体的实现在实例 17 中已经介绍过了，此处重点分析淡入淡出效果窗体 FaderWidget。具体代码如下：

```
class FaderWidget : public QWidget
{
    Q_OBJECT
public:
    FaderWidget(QWidget *parent=0);
    void start();
protected:
    void paintEvent(QPaintEvent *event);
private:
    QTimer *timer;
    int currentAlpha;
    QColor startColor;
    int duration;
};
```

FaderWidget 继承自 QWidget，包含一个 start() 函数，调用 start() 函数即开始 FaderWidget 窗体的渐变过程；重新实现了 paintEvent() 函数，并声明了一个定时器变量定时调用 paintEvent() 函数。

FaderWidget 的构造函数：

```
FaderWidget::FaderWidget(QWidget *parent)
    : QWidget(parent)
{
    if (parent)
        startColor = parent->palette().window().color();
```

```

3     else
4         startColor = Qt::white;
5
6         currentAlpha = 0;
7         duration = 1000;
8
9         timer = new QTimer(this);
10        connect(timer, SIGNAL(timeout()), this, SLOT(update()));
11
12        setAttribute(Qt::WA_DeleteOnClose);
13        resize(parent->size());
14    }

```

在 FaderWidget 的构造函数中，第 1~4 行首先判断是否有父窗体存在，若有父窗体，则设置起始窗体颜色为父窗体的背景色；若没有父窗体，则设置起始窗体颜色为白色。

第 5 行设置透明度的初始值。

第 6 行设置渐变时长为 1000 毫秒。

第 7 行创建一个定时对象。

第 8 行连接定时器响应 timeout() 信号，每间隔时间结束后调用一次 update() 函数重画窗体。

第 9 行设置 FaderWidget 的窗体属性为 Qt::WA_DeleteOnClose，当整个对话框关闭时，此渐变窗体也同时关闭。

第 10 行设置 FaderWidget 窗体的尺寸为父窗体的大小。

start() 函数的实现代码如下：

```

void FaderWidget::start()
{
    currentAlpha = 255;
    timer->start(100);
    show();
}

```

FaderWidget 的 start() 函数显示此窗体并开始渐变。首先设置窗体显示的最初透明度为 255，即不透明；启动定时器，以 100 毫秒为周期进行重画；最后调用 show() 函数显示此窗体。

FaderWidget 的重画函数：

```

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

```

```

2     QColor semiTransparentColor = startColor;
3     semiTransparentColor.setAlpha(currentAlpha);
4     painter.fillRect(rect(), semiTransparentColor);

5     currentAlpha -= 255 * timer->interval() / duration;
6     if (currentAlpha <= 0)
7     {
8         timer->stop();
9         close();
10    }
11}

```

第 1 行首先以 FaderWidget 窗体创建一个 QPainter 对象。

第 2 行新建一个 QColor 对象保存重绘窗体的颜色，设置为 startColor，即父窗体的颜色或是白色。

第 3 行设置此颜色的透明度。

第 4 行利用此颜色填充整个窗体。

第 5 行修改 currentAlpha 值，每重绘一次透明度降低一定的值，直到透明度为 0，即完全透明。

第 6~8 行对透明度的值进行判断，若透明度已小于或等于零，则停止定时器，也即不再调用重画命令，并关闭此 FaderWidget 窗体。

完成 FaderWidget 演变窗体后，需在主对话框实现的合适位置和时机调用此渐变窗体以实现淡入淡出效果。

本实例中主对话框类为 ConfigDialog。

```

class ConfigDialog : public QDialog
{
    Q_OBJECT

public:
    ConfigDialog();
public slots:
    void changePage(QListWidgetItem *current, QListWidgetItem *previous);
private:
    QListWidget *contentsWidget;
    QStackedWidget *pagesWidget;
    QPointer<FaderWidget> faderWidget;
private slots:
    void fadeInWidget(int index);
};

```

此对话框继承自 QDialog 类，在类中声明了两个槽函数，一个 changePage()完成页面切换的工作，一个 fadeInWidget()完成页面切换时的淡入淡出效果；在类的私有变量中声明了一个 QPointer<FaderWidget>变量 faderWidget，此变量用于保存实现淡入淡出效果的渐变窗体，QPointer<>提供的是一种保护型指针，它会在使用结束后自动为指针赋零，避免了野指针的产生。

ConfigDialog()函数的实现：

```
ConfigDialog::ConfigDialog()
{
    1 contentsWidget = new QListWidget;           //页面名称列表框
    /* 插入各页面的名称 */
    2 contentsWidget->addItem(tr("base info"));
    3 contentsWidget->addItem(tr("contact"));
    4 contentsWidget->addItem(tr("detail"));
    5 connect(contentsWidget,
              SIGNAL(currentItemChanged(QListWidgetItem *, QListWidgetItem *)),
              this, SLOT(changePage(QListWidgetItem *, QListWidgetItem*)));

    6 pagesWidget = new QStackedWidget;           //页面堆栈窗
    /* 插入页面 */
    7 pagesWidget->addWidget(new BaseInfo);
    8 pagesWidget->addWidget(new Contact);
    9 pagesWidget->addWidget(new Detail);
    10 connect(pagesWidget, SIGNAL(currentChanged(int)),
               this, SLOT(fadeInWidget(int)));

    11 faderWidget = new FaderWidget(this);        //创建一个渐变窗体对象
    12 faderWidget->start();                     //以淡入淡出的效果显示主对话框
}
```

主对话框中新建了一个 QListWidget 对象和一个 QStackedWidget 对象，分别保存有可显示的页面名称和具体页面。

第 5 行的 connect 函数，为列表框 currentItemChanged()信号连接响应槽函数 changePage()，即当用户选择下拉列表框中的条目时，调用 changePage()更改右侧堆栈窗中的显示页面。

第 10 行的 connect 函数，为堆栈窗的 currentChanged()信号连接响应槽函数 fadeInWidget()，即当堆栈窗的页面发生改变时调用 fadeInWidget()来实现页面的淡入淡出效果。

第 11、12 行使对话框自身初始显示时也以淡入淡出的效果显现。

下面是两个槽函数的代码分析：



```
void ConfigDialog::changePage(QListWidgetItem *current,
                             QListWidgetItem *previous)
{
    if (!current)
        current = previous;
    pagesWidget->setCurrentIndex(contentsWidget->row(current));
}
```

changePage()槽函数主要完成页面的切换工作。

```
void ConfigDialog::fadeInWidget(int index)
{
    if (faderWidget)
        faderWidget->close();
    faderWidget = new FaderWidget(
        pagesWidget->widget(index));
    faderWidget->start();
}
```

fadeInWidget()槽函数在堆栈窗的页面发生变化时被调用，是实现页面淡入淡出的关键函数，函数首先判断是否已有渐变窗体对象存在，若已有则关闭已存在的渐变窗体，再以堆栈窗的当前窗体为父窗口创建渐变窗体对象，并调用 start()函数开始窗体渐变，实现淡入淡出效果。

实例 21 不规则窗体

知识点：

- 利用 setMask()为窗体设置遮罩，实现不规则窗体
- 设置遮罩后的窗体尺寸仍是原窗体大小，只是被遮罩的地方不可见

常见的窗体通常是各种方形的对话框，如前面实例中实现的所有对话框都是这样的。但有时也会需要用到非方形的窗体，如圆形、椭圆形甚至是不规则形状的对话框。

本实例即实现了一个以 PNG 图形外沿为形状的不规则形状对话框，如图 3-5 所示。

图 3-5 中所示的企鹅即为一个不规则窗体，实例在不规则窗体中绘制了作为窗体形状的 PNG 图片，也可在不规则窗体上放置按钮等控件，可以通过鼠标左键拖动窗体，鼠标右键关闭窗体。

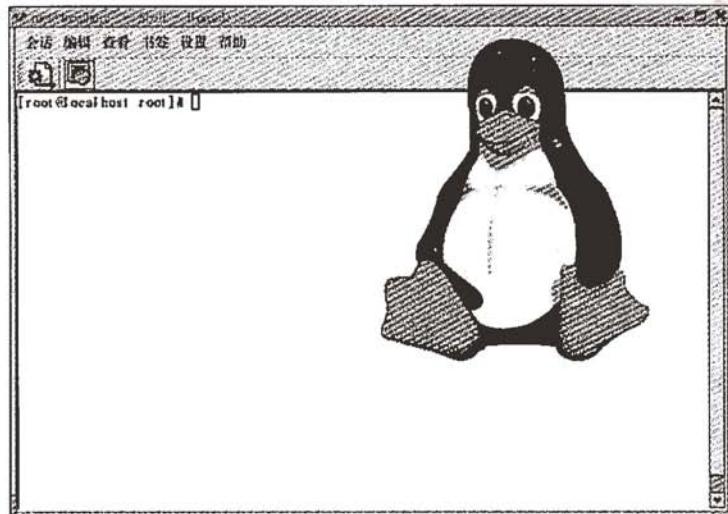


图 3-5 不规则窗体

具体实现代码如下：

```
class ShapeWidget : public QWidget
{
    Q_OBJECT
public:
    ShapeWidget(QWidget *parent=0);

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

private:
    QPoint dragPosition;
};
```

ShapeWidget 即为此不规则窗体类，继承自 QWidget 类。在类中重定义的鼠标事件 mousePressEvent()、mouseMoveEvent() 以及绘制函数 paintEvent()，使不规则窗体能用鼠标随意拖动。

ShapeWidget 的构造函数是实现不规则窗体的关键。具体代码如下：

```
ShapeWidget::ShapeWidget(QWidget *parent)
    : QWidget(parent,Qt::FramelessWindowHint)
```



```

{
1   QPixmap pix;
2   pix.load("./images/tux.png",0,
3       Qt::AvoidDither|Qt::ThresholdDither|Qt::ThresholdAlphaDither);
4   resize(pix.size());
5   setMask(pix.mask());
}

```

第 1 行新建一个 QPixmap 对象。

第 2 行调用 QPixmap 的 load() 函数为 QPixmap 对象填入图像值，load() 函数的原型如下：

```
bool load ( const QString & fileName, const char * format = 0, Qt::ImageConversionFlags flags = Qt::AutoColor )
```

第一个参数 fileName 为图片文件名；第二个参数 format 表示读取图片文件采用的格式，此处为 0 即采用默认的格式；第三个参数 flags 表示读取图片的方式，由 Qt::ImageConversionFlag 定义，此处设置的标识为避免图片的抖动方式。

第 3 行重设主窗体的尺寸为所读取的图片的大小。

第 4 行的 setMask() 命令是实现不规则窗体的关键，setMask() 的作用是为调用它的控件增加一个遮罩，遮住所选区域以外的部分使之看起来是透明的，它的参数可为一个 QBitmap 对象（如本实例）或一个 QRegion 对象，此处调用 QPixmap 的 mask() 函数获得图片自身的遮罩，为一个 QBitmap 对象。实例中使用的是 png 格式的图片，它的透明部分实际上即是一个遮罩。

重定义鼠标按下响应函数 mousePressEvent(QMouseEvent *) 和鼠标移动响应函数 mouseMoveEvent(QMouseEvent *)，使不规则窗体能响应鼠标事件，随意拖动。具体代码如下：

```

void ShapeWidget::mousePressEvent(QMouseEvent * event)
{
    if(event->button() == Qt::LeftButton)
    {
        dragPosition = event->globalPos() - frameGeometry().topLeft();
        event->accept();
    }
    if(event->button() == Qt::RightButton)
    {
        close();
    }
}

```

鼠标按下响应函数中，首先判断按下的是否为鼠标左键，若是则保存当前鼠标点所在的位置相对于窗体左上角的偏移值 dragPosition；如果按下鼠标右键，则关闭窗体。

 小贴士：此处的 frameGeometry().topLeft()仍然表示整个窗体的左上角，而并不是所见的不规则窗体的左上角。

```
void ShapeWidget::mouseMoveEvent(QMouseEvent * event)
{
    if (event->buttons() & Qt::LeftButton)
    {
        move(event->globalPos() - dragPosition);
        event->accept();
    }
}
```

在鼠标移动响应函数中，首先判断当前鼠标状态，调用 event->buttons()返回鼠标的状态，若为左侧按钮则调用 QWidget 的 move()函数把窗体移动至鼠标当前点，由于 move()函数的参数指的是窗体的左上角的位置，因此要用鼠标当前点的位置减去相对窗体左上角的偏移值 dragPosition。

```
void ShapeWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawPixmap(0,0,QPixmap(":/images/tux.png"));
}
```

ShapeWidget 的重画函数主要完成在窗体上绘制图片的工作，此处为方便显示在窗体上绘制的即是用来确定窗体外形的 PNG 图片。

实例 22 电子钟

知识点：

- 利用 QTimer 显示时钟
- QTime 时间显示格式的转换

本实例实现一个数字电子钟程序，效果图如图 3-6 所示显示于桌面上，并可随意拖动至桌面任意位置。

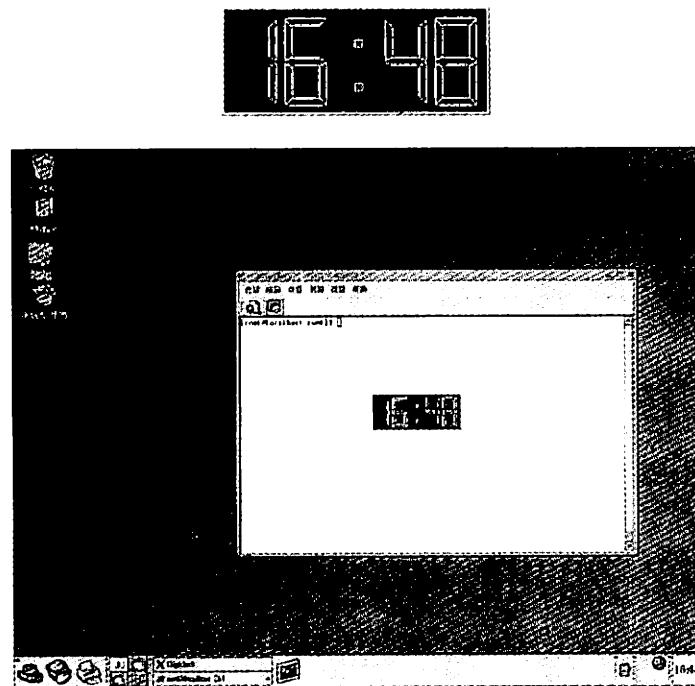


图 3-6 电子钟效果图

具体实现代码如下：

```
class DigiClock : public QLCDNumber
{
    Q_OBJECT
public:
    DigiClock(QWidget *parent=0);

    void mousePressEvent(QMouseEvent *);
    void mouseMoveEvent(QMouseEvent *);

public slots:
    void showTime();

private:
    QPoint dragPosition;
    bool showColon;
};
```

数字电子钟类 DigiClock 继承自 QLCDNumber 类，类中重定义了鼠标按下事件和鼠标移动事件以使电子钟可随意拖动；showTime()函数用于显示当前的时间；私有变量 dragPosition 用于保存鼠标点相对电子钟窗体左上角的偏移值；showColon 表示显示时间时是否显示“：“。

DigiClock 的构造函数：

```
DigiClock::DigiClock(QWidget *parent)
    : QLCDNumber(parent)
{
1   QPalette p = palette();
2   p.setColor(QPalette::Window, Qt::blue);
3   setPalette(p);

4   setWindowFlags(Qt::FramelessWindowHint);

5   setWindowOpacity(0.5);

6   QTimer *timer = new QTimer(this);
7   connect(timer, SIGNAL(timeout()), this, SLOT(showTime()));
8   timer->start(1000);

9   showTime();

10  resize(150, 60);
11  showColon=true;
}
```

在 DigiClock 的构造函数中，完成外观的设置及定时器的初始化工作。

第 1~3 行完成电子钟窗体背景色的设置，在前面的实例中已经分析过利用调色板类 QPalette 进行控件颜色控制的方法，此处设置背景色为蓝色。

第 4 行调用 QWidget 类的 setWindowFlags() 函数设置窗体的标识，此处设置为 Qt::FramelessWindowHint，表示此窗体为一个没有面板边框和标题栏的窗体。

第 5 行调用 setWindowOpacity() 函数设置窗体的透明度为 0.5，即半透明，但此函数在 X11 系统中并不起作用，当程序在 Windows 系统下编译运行时，此函数起作用，即电子钟半透明显示。

第 6~8 行完成电子钟中用于时间显示的定时器部分。

第 6 行新建一个定时器对象。

第 7 行连接定时器的 timeout() 信号与显示时间的槽函数 showTime()。



第 8 行调用 start() 函数以 1000 毫秒为周期启动定时器。

第 9 行调用一次 showTime() 函数，初始时间显示。

第 10 行设置电子钟显示的尺寸。

第 11 行初始化 showColon 为 true。

DigiClock 的 showTime() 函数完成电子钟的显示时间的功能。具体代码如下：

```
void DigiClock::showTime()
{
    1     QTime time = QTime::currentTime();
    2     QString text = time.toString("hh:mm");
    3     if(showColon)
    4     {
    5         text[2] = ':';
    6         showColon = false;
    7     }
    8     else
    9     {
    10        text[2] = '';
    11        showColon = true;
    12    }
    13    display(text);
}
```

第 1 行调用 QTime 的 currentTime() 函数获取当前的系统时间，保存在一个 QTime 对象中。

第 2 行调用 QTime 的 toString() 函数把获取的当前时间转换成字符串类型。为便于显示，toString() 函数的参数需指定转换后时间的显示格式。

- H/h：小时（若使用 H 表示小时，则无论何时都以 24 小时制显示小时；若使用 h 表示小时，则当同时指定 AM/PM 时，采用 12 小时制显示小时，其他情况下仍采用 24 小时制进行显示）。
- m：分钟。
- s：秒钟。
- AP/A：显示 AM 或 PM。
- Ap/a：显示 am 或 pm。

可根据实际显示需要进行格式设置，如：

hh:mm:ss A	22:30:08 PM
H:mm:s a	10:30:8 pm

 小贴士：QTime 的 `toString()` 函数也可直接利用 `Qt::DateFormat` 作为参数指定时间显示的格式，如 `Qt::TextDate`、`Qt::ISODate`、`Qt::LocaleDate` 等。

第 3~12 行完成电子钟“时”与“分”之间的表示秒钟的两个点的闪显功能。

第 13 行调用 `QLCDNumber` 类的 `display()` 函数显示转换好的字符串时间。

鼠标按下事件响应函数 `mousePressEvent(QMouseEvent *)` 和鼠标移动事件响应函数 `mouseMoveEvent(QMouseEvent *)` 的重定义使电子钟能用鼠标在桌面上随意拖动。具体代码如下：

```
void DigiClock::mousePressEvent(QMouseEvent * e)
{
    if (e->button() == Qt::LeftButton)
    {
        dragPosition = e->globalPos() - frameGeometry().topLeft();
        e->accept();
    }
    if (e->button() == Qt::RightButton)
    {
        close();
    }
}
```

鼠标按下响应函数中，首先判断按下的是否为鼠标左键，若是则保存当前鼠标点所在的位置相对于窗体左上角的偏移值 `dragPosition`；如果按下鼠标右键，则退出窗体。

```
void DigiClock::mouseMoveEvent(QMouseEvent * e)
{
    if (e->buttons() & Qt::LeftButton)
    {
        move(e->globalPos() - dragPosition);
        e->accept();
    }
}
```

在鼠标移动响应函数中，首先判断当前鼠标状态，调用 `e->buttons()` 返回鼠标的状态，若为左侧按钮则调用 `QWidget` 的 `move()` 函数把窗体移动至鼠标当前点。由于 `move()` 函数的参数指的是窗体的左上角的位置，因此要用鼠标当前点的位置减去相对窗体左上角的偏移值 `dragPosition`。



实例 23 程序启动画面

知识点：

QSplashScreen 的使用

多数大型应用程序启动时都会在程序完全启动前显示一个启动画面，在程序完全启动后消失。程序启动画面可以显示一些有关产品的信息，让用户在等待程序启动的同时了解有关产品的功能，也是一个宣传的方式。

QSplashScreen 类提供了在程序启动过程中显示的启动画面的功能。本实例实现一个出现程序启动画面的例子，效果图如图 3-7 所示。

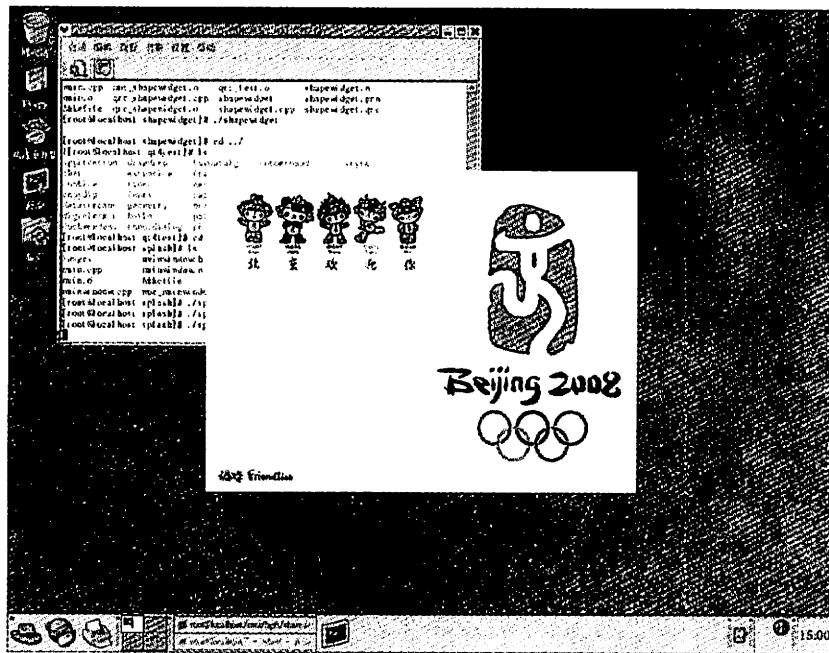


图 3-7 程序启动画面的效果图

当运行程序时，在显示屏的中央出现一个启动画面，经过一段时间，应用程序完成初始化工作后，启动画面隐去，出现程序的主窗口界面，如图 3-8 所示。

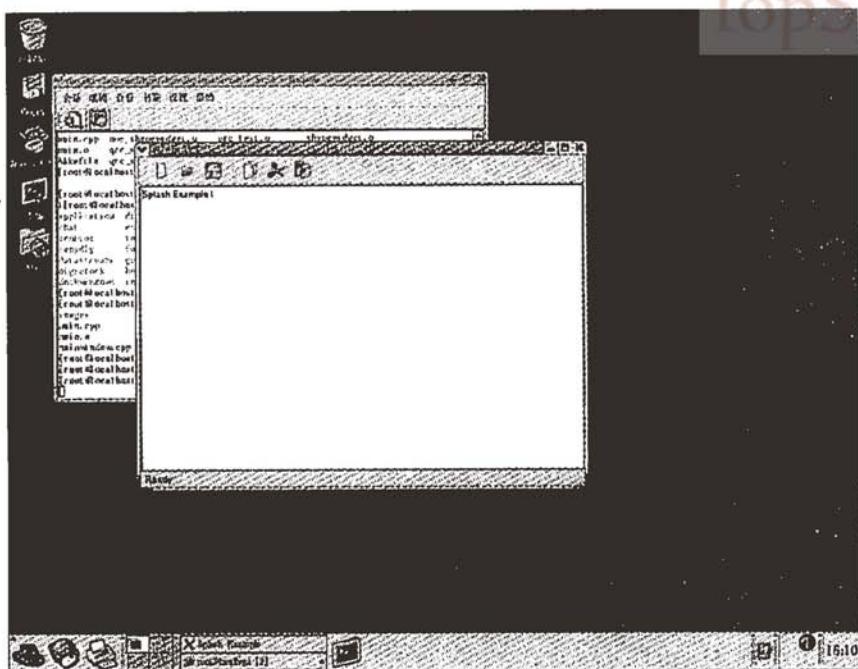


图 3-8 实例效果图

具体实现代码如下：

```
int main(int argc, char *argv[])
{
    1   QApplication app(argc, argv);
    2   QPixmap pixmap("./splash.jpg");
    3   QSplashScreen splash(pixmap);
    4   splash.show();
    5   app.processEvents();

    6   MainWindow window;
    7   window.show();
    8   splash.finish(&window);
    9   return app.exec();
}
```

启动画面主要在 main() 函数中实现。

第 1 行创建一个 QApplication 对象。

第 2 行创建一个 QPixmap 对象，splash.jpg 为启动图片。



第3行利用 QPixmap 对象创建一个 QSplashScreen 对象。
第4行调用 show()函数显示此启动图片。
第5行调用 processEvents()使程序在显示启动画面的同时仍能响应鼠标等其他事件。
第6、7行正常创建主窗体对象，并调用 show()函数显示。
第8行调用 QSplash 类的 finish()函数，表示在主窗体对象初始化完成后，结束启动画面。

第9行正常地调用 exec()函数。

由于启动画面一般在当程序初始化时间较长的情况下出现，本实例中为了使程序初始化时间加长以显示启动画面，在主窗体的构造函数中调用 sleep()函数，使主窗口程序在初始化时休眠几秒钟。具体代码如下：

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle(tr("Splash Example"));

    QTextEdit *edit = new QTextEdit;
    edit->setText("Splash Example !");
    setCentralWidget(edit);

    createToolBars();
    createStatusBar();
    resize(600,450);

    sleep(3);
}
```

第4章 QMainWindow

本章对 QMainWindow 类窗口进行分析。QMainWindow 类是一个经常用到的类，为用户提供了一个主窗口程序，可包含一个菜单条、一个工具栏、一个状态条以及一个中央窗体，是许多应用程序的基础，如文本编辑器、图片浏览器等都是以 QMainWindow 为基础实现的。

在本章中，首先分析了一个基本的主窗口程序的实现，只包含基本的菜单和工具栏，实现了基本的主窗口功能，如打开、新建等。随后的实例对在 QMainWindow 窗口中经常用到的功能实现进行了分析。

本章包括 7 个实例：

- 基本 QMainWindow 主窗口程序
- 打印文本
- 打印图像
- 图片的缩放与旋转
- 在工具栏中嵌入控件
- 设置字体、字号等格式属性
- 设置文本排序及对齐



实例 24 基本 QMainWindow 主窗口程序

知识点：

- 创建 QMainWindow 主窗口的基本步骤
- QAction 动作的创建
- 如何在菜单及工具栏中加入动作
- 工具栏的属性

本实例实现一个基本主窗口程序，包含一个菜单条、一个工具栏、中央可编辑窗体及状态栏。实现的效果图如图 4-1 所示。

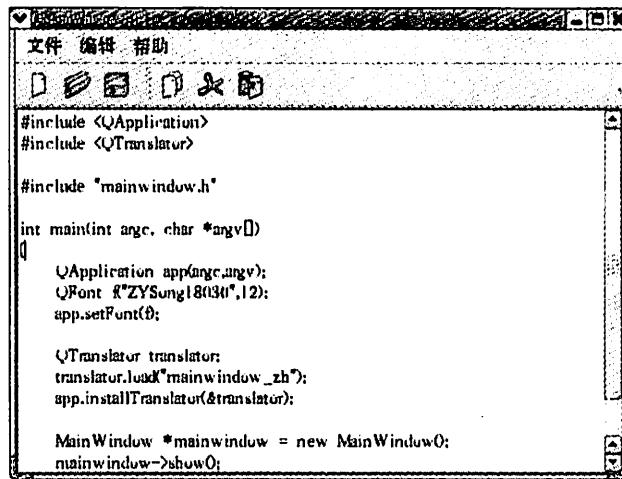


图 4-1 基本主窗口实例效果图

具体实现代码如下：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow();
    void createMenus();      //创建菜单
    void createActions();    //创建动作
```

```
void createToolBars(); //创建工具栏

public slots:
    void slotNewFile();
    void slotOpenFile();
    void slotSaveFile();

private:
    QMenu *fileMenu;
    QMenu *editMenu;
    QMenu *aboutMenu;
    QToolBar *fileTool;
    QToolBar *editTool;
    QAction *fileOpenAction;
    QAction *fileNewAction;
    QAction *fileSaveAction;
    QAction *exitAction;
    QAction *copyAction;
    QAction *cutAction;
    QAction *pasteAction;
    QAction *aboutAction;

    QTextEdit * text;
};
```

类声明中，createActions()函数用于创建所有的动作，createMenus()函数用于创建菜单，createToolBars()函数用于创建工具栏。接着声明了用到的槽函数，如“新建文件”、“打开文件”等。最后声明了实现主窗口所需的各个元素，包括菜单项、工具条以及各个动作等。

主窗口构造函数：

```
MainWindow::MainWindow()
{
1   setWindowTitle(tr("QMainWindow"));
2   text = new QTextEdit(this);
3   setCentralWidget(text);

4   createActions();
5   createMenus();
6   createToolBars();
}
```



第 1 行设置主窗口的标题栏。

第 2 行新建一个 QTextEdit 对象，以主窗口为父窗口。

第 3 行把 QTextEdit 作为主窗口的中央窗体。

第 4 行调用创建动作的函数。

第 5 行调用创建菜单的函数。

第 6 行调用创建工具栏的函数。

菜单与工具栏都与 QAction 类密切相关，工具栏上的功能按钮与菜单中的选项条目相对应，完成相同的功能，使用相同的快捷键与图标。QAction 类为用户提供了一个统一的命令接口，无论是从菜单触发还是从工具栏触发，或快捷键触发都调用同样的操作接口，达到同样的目的。以下是各个动作（Action）的实现代码：

```
void MainWindow::createActions()
{
    // “打开” 动作
1   fileOpenAction = new QAction(QIcon(":/images/open.png"), tr("Open"),this);
2   fileOpenAction->setShortcut(tr("Ctrl+O"));
3   fileOpenAction->setStatusTip(tr("open a file"));
4   connect(fileOpenAction,SIGNAL(triggered()),this,SLOT(slotOpenFile()));

    // “新建” 动作
5   fileNewAction = new QAction(QIcon(":/images/new.png"), tr("New"),this);
6   fileNewAction->setShortcut(tr("Ctrl+N"));
7   fileNewAction->setStatusTip(tr("new file"));
8   connect(fileNewAction,SIGNAL(triggered()),this,SLOT(slotNewFile()));

    // “保存” 动作
9   fileSaveAction = new QAction(QIcon(":/images/save.png"), tr("Save"),this);
10  fileSaveAction->setShortcut(tr("Ctrl+S"));
11  fileSaveAction->setStatusTip(tr("save file"));
12  connect(fileSaveAction,SIGNAL(activated()),this,SLOT(slotSaveFile()));

    // “退出” 动作
13  exitAction = new QAction(tr("Exit"), this);
14  exitAction->setShortcut(tr("Ctrl+Q"));
15  exitAction->setStatusTip(tr("exit"));
16  connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));

    // “剪切” 动作
17  cutAction = new QAction(QIcon(":/images/cut.png"), tr("Cut")this);
```

```

18 cutAction->setShortcut(tr("Ctrl+X"));
19 cutAction->setStatusTip(tr("cut to clipboard"));
20 connect(cutAction, SIGNAL(triggered()), text, SLOT(cut()));

// “复制”动作
21 copyAction = new QAction(QIcon(":/images/copy.png"), tr("Copy"), this);
22 copyAction->setShortcut(tr("Ctrl+C"));
23 copyAction->setStatusTip(tr("copy to clipboard"));
24 connect(copyAction, SIGNAL(triggered()), text, SLOT(copy()));

// “粘贴”动作
25 pasteAction = new QAction(QIcon(":/images/paste.png"), tr("Paste"), this);
26 pasteAction->setShortcut(tr("Ctrl+V"));
27 pasteAction->setStatusTip(tr("paste clipboard to selection"));
28 connect(pasteAction, SIGNAL(triggered()), text, SLOT(paste()));

// “关于”动作
29 aboutAction = new QAction(tr("About"), this);
30 connect(aboutAction, SIGNAL(triggered()), this, SLOT(slotAbout()));
}

```

第1~4行实现的是“打开文件”动作，第1行在创建这个动作时，指定了此动作使用的图标、名称以及父窗口。

第2行设置了此动作的快捷键为Ctrl+O。

第3行设定了状态条显示，当鼠标光标移至此动作对应的菜单条目或工具栏按钮上时，在状态条上显示“打开文件”的提示。

第4行连接此动作触发时所调用的槽函数slotOpenFile()。

 小贴士：第1行指定动作所使用的图标时，使用的是 QIcon ("::/images/open.png")，需在程序所在文件夹下新增一个 mainwindow.qrc 文件用于说明程序中所用到的图标，文件内容为：

```

<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/copy.png</file>
    <file>images/cut.png</file>
    <file>images/new.png</file>
    <file>images/open.png</file>
    <file>images/paste.png</file>
    <file>images/save.png</file>
</qresource>
</RCC>

```



并在.pro 文件中加入一行：
RESOURCES = mainwindow.qrc

“剪切”、“复制”和“粘贴”动作连接的触发响应槽函数，分别直接使用 QTextEdit 对象的 cut()、copy() 和 paste() 函数即可。

第 30 行“关于”动作的触发响应槽函数使用的是 QApplication 的 slotabout()。

在创建动作时，也可不指定图标，这类动作一般只在菜单中出现，而不是工具栏上使用。如第 13 行创建“退出”动作及第 29 行创建“关于”动作。

在实现了各个动作之后，需要把它通过菜单及工具栏快捷键的方式体现出来，以下是菜单的实现函数 createMenus()：

```
void MainWindow::createMenus()
{
    //文件菜单
    1 fileMenu = menuBar()->addMenu(codec->toUnicode("文件"));
    2 fileMenu->addAction(fileNewAction);
    3 fileMenu->addAction(fileOpenAction);
    4 fileMenu->addAction(fileSaveAction);
    5 fileMenu->addAction(exitAction);

    //编辑菜单
    6 editMenu = menuBar()->addMenu(codec->toUnicode("编辑"));
    7 editMenu->addAction(copyAction);
    8 editMenu->addAction(cutAction);
    9 editMenu->addAction(pasteAction);

    //帮助菜单
   10 aboutMenu = menuBar()->addMenu(codec->toUnicode("帮助"));
   11 aboutMenu->addAction(aboutAction);
}
```

其中，第 1~5 行实现文件菜单，Qt4 的菜单实现与 Qt3 有所不同，简化了菜单的实现过程。

第 1 行直接调用 QMainWindow 的 menuBar() 函数即可得到主窗口的菜单条指针，再调用菜单条 QMenuBar 的 addMenu() 函数，即完成在菜单条中插入一个新菜单 fileMenu，fileMenu 为一个 QMenu 类对象。

第 2~5 行调用 QMenu 的 addAction() 函数在菜单中加入菜单条目“打开”、“新建”、“保存”和“退出”。

第 6~9 行实现编辑菜单。

第 10、11 行实现帮助菜单。

接下来实现相对应的工具栏，即主窗口程序的 createToolBars() 函数。

```
void MainWindow::createToolBars()
{
    //文件工具栏
    1 fileToolBar = addToolBar("File");
    2 fileToolBar->addAction(fileNewAction);
    3 fileToolBar->addAction(fileOpenAction);
    4 fileToolBar->addAction(fileSaveAction);

    //编辑工具栏
    5 editTool = addToolBar("Edit");
    6 editTool->addAction(copyAction);
    7 editTool->addAction(cutAction);
    8 editTool->addAction(pasteAction);
}
```

第 1~4 行实现文件工具栏，第 5~8 行实现编辑工具栏。

主窗口的工具栏上可以有多个工具条，一般采用一个菜单对应一个工具条的方式，也可根据需要进行工具条的划分。

第 1 行直接调用 QMainWindow 的 addToolBar() 函数即可获得主窗口的工具条对象，每新增一个工具条调用一次 addToolBar() 函数，赋予不同的名称，即可在主窗口中新增一个工具条。

第 2~4 行调用 QToolBar 的 addAction() 函数在工具条中插入属于本工具条的动作。

第 5~8 行编辑工具条的实现与文件工具条类似。

两个工具条的显示可以由用户进行选择，在工具栏上单击鼠标右键将弹出工具条显示的选择菜单，如图 4-2 所示。

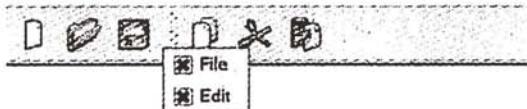


图 4-2 工具条选择

用户对需要显示的工具条进行选择。

工具条是一个可移动的窗口，它可停靠的区域由 QToolBar 的 allowAreas 决定，包括 Qt::LeftToolBarArea、Qt::RightToolBarArea、Qt::TopToolBarArea、Qt::BottomToolBarArea 和 Qt::AllToolBarAreas。



默认为 Qt::AllToolBarAreas，启动默认出现于主窗口的顶部。

可通过调用 setAllowAreas() 函数来指定工具条可停靠的区域，如：

```
fileToolBar->setAllowAreas(Qt::TopToolBarArea|Qt::LeftToolBarArea);
```

此函数限定文件工具条只可出现在主窗口的顶部或左侧。

工具条也可通过调用 setMovable() 函数设定工具条的可移动性，如：

```
fileToolBar->setMovable(false);
```

指定文件工具条不可移动，只出现于主窗口的顶部。

实现动作的响应的槽函数：

```
void MainWindow::slotNewFile()
{
    MainWindow *newWin = new MainWindow;
    newWin->show();
}
```

新建一个空白文件。

```
void MainWindow::slotOpenFile()
{
    fileName = QFileDialog::getOpenFileName(this);
    if (!fileName.isEmpty())
    {
        if (text->document()->isEmpty())
            loadFile(fileName);
        else
        {
            MainWindow *newWin = new MainWindow;
            newWin->show();
            newWin->loadFile(fileName);
        }
    }
}
```

打开一个文件，利用标准文件对话框 QFileDialog 打开一个已存在的文件，若当前中央窗体中已有打开的文件，则在一个新的窗口中打开选定的文件；若当前中央窗体是空的，则在当前窗体中打开。

具体读取文件内容的工作在 loadFile() 函数中完成：

```
void MainWindow::loadFile(QString fileName)
{
```

```

printf("file name: %s\n", fileName.data());
QFile file(fileName);
if (file.open(QIODevice::ReadOnly|QIODevice::Text))
{
    QTextStream textStream(&file);
    while(!textStream.atEnd())
    {
        text->append(textStream.readLine());
        printf("read line \n");
    }
    printf("end\n");
}
}

```

利用 QFile 和 QTextStream 读取文件内容。

本实例的重点是如何搭建一个基本的 QMainWindow 主窗口，因此对于菜单或工具栏的具体功能实现并没有做太多的分析，这些功能可在此基本主窗口程序的基础之上逐步完善。

实例 25 打印文本

知识点：

标准打印对话框 QPrintDialog 的使用

打印文本在文本编辑工作中经常使用，本实例实现使用打印机打印文本的功能，如图 4-3 所示。

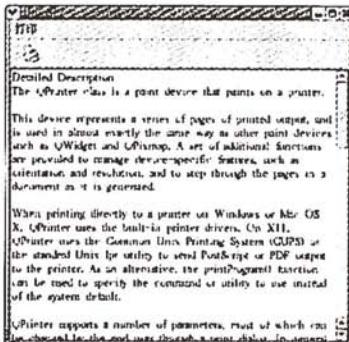


图 4-3 打印文本



具体实现代码如下：

```
PrintText::PrintText()
    : QMainWindow()
{
    1   QFont f("ZYSong18030";12);
    2   setFont(f);
    3   setWindowTitle(tr("Printer"));

    4   contentTextEdit = new QTextEdit(this);
    5   setCentralWidget(contentTextEdit);

    6   createActions();
    7   createMenus();
    8   createToolBars();

    9   QFile file("QPainter.txt");
10   if(file.open(QIODevice::ReadOnly|QIODevice::Text))
    {
11       QTextStream textStream(&file);
12       while(!textStream.atEnd())
13       {
14           contentTextEdit->append(textStream.readLine());
15       }
16       file.close();
17   }
}
```

第 1~3 行设置窗体使用的字体和窗体标题。

第 4、5 行创建一个 QTextEdit 控件 contentTextEdit，并设置为中央窗体。

第 6~8 行创建菜单、工具条等部件。

第 9~14 行从文本文件 QPainter.txt 中逐行读取数据并显示在 contentTextEdit 中。

slotPrint() 实现文本打印功能。具体代码如下：

```
void PrintText::slotPrint()
{
    1   QPrinter printer;
    2   QPrintDialog printDialog(&printer, this);
    3   if (printDialog.exec())
    {
        4       QTextDocument *doc = contentTextEdit->document();
        5       doc->print(&printer);
    }
```

```
}
```

第 1 行新建一个 QPrinter 对象。

第 2 行创建一个 QPrintDialog 对象，参数为 QPrinter 对象。

QPrintDialog 是 Qt 提供的标准对话框，为打印机的使用提供了一种方便、规范的方法。如图 4-4 所示，QPrintDialog 标准对话框提供了打印机的选择、配置功能，并允许使用者改变文档有关的设置，如页面大小、方向、打印类型（彩色/灰度）、页边距以及打印份数等。

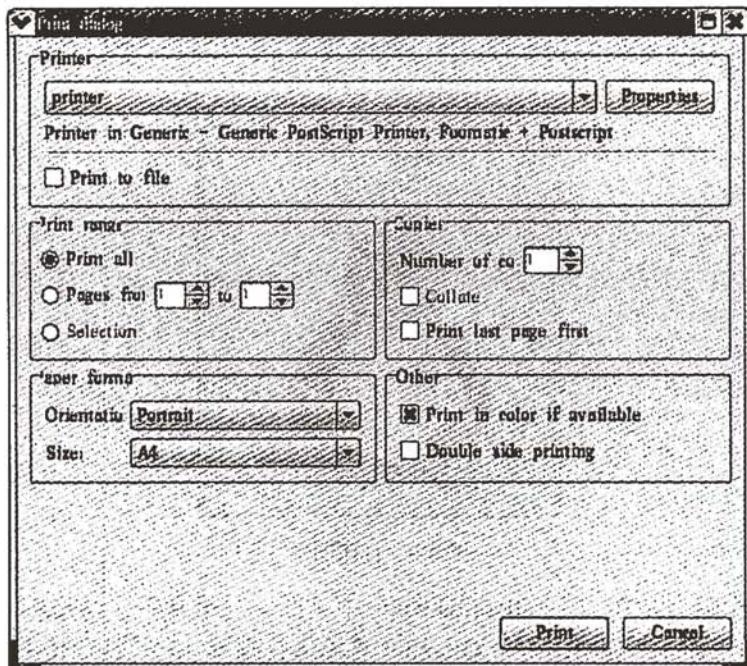


图 4-4 QPrintDialog 标准对话框

第 3 行判断打印对话框显示后用户是否单击“打印”按钮，若单击“打印”按钮，则相关打印属性将可以通过创建 QPrintDialog 对象时使用的 QPrinter 对象获得；若用户单击“取消”按钮，则不执行后续的打印操作。

第 4 行获得 QTextEdit 对象的文档。

第 5 行打印。

实例 26 打印图像

知识点：

- 标准打印对话框 QPrintDialog 的使用
- 以 QPrinter 作为 QPainter 画图实现图像打印

打印图像是图像处理软件中的一个常用功能，本实例实现使用打印机打印图像的功能，如图 4-5 所示。



图 4-5 打印图像

打印图像实际上是在一个 QPainterDevice 中画图，与平常在 QWidget、QPixmap 和 QImage 中画图一样，都是创建一个 QPainter 对象进行画图，只是打印使用的是 QPrinter，QPrinter 本质上也是一个绘图设备 QPainterDevice。

具体实现代码如下：

```
PrintImage::PrintImage()
    : QMainWindow()
{
    1   QFont f("ZYSong18030",12);
    2   setFont(f);
    3   setWindowTitle(tr("PrintImage"));
    4   imageLabel = new QLabel(this);
```

```
5  imageLabel->setSizePolicy(QSizePolicy::Ignored, QSizePolicy::Ignored);
6  imageLabel->setScaledContents(true);
7  setCentralWidget(imageLabel);

8  createActions();
9  createMenus();
10 createToolBars();

11 if(image.load("tux.png")){
12     {
13         imageLabel->setPixmap (QPixmap::fromImage(image));
14         resize(image.width(), image.height());
15     }
16 }
```

第 1~3 行设置窗体使用的字体和窗体标题。

第 4~7 行创建一个放置图像的 QLabel 对象 imageLabel，并将该 QLabel 对象设置为
中心窗体。

第 8~10 行创建菜单、工具条等部件。

第 12、13 行在 imageLabel 对象中放置图像。

slotPrint()实现打印功能的代码如下：

```
void PrintImage::slotPrint ()
{
1  QPrinter printer
2  QPrintDialog printDialog(&printer, this);
3  if (printDialog.exec())
{
4      QPainter painter(&printer);
5      QRect rect = painter.viewport();
6      QSize size = image.size();
7      size.scale(rect.size(), Qt::KeepAspectRatio);
8      painter.setViewport(rect.x(), rect.y(), size.width(), size.height());
9      painter.setWindow(image.rect());
10     painter.drawImage(0, 0, image);
11 }
}
```

第 1 行新建一个 QPrinter 对象。

第 2 行创建一个 QPrintDialog 对象，参数为 QPrinter 对象。

第 3 行判断打印对话框显示后用户是否单击“打印”按钮，若单击“打印”按钮，则



相关打印属性将可以通过创建 `QPrintDialog` 对象时使用的 `QPrinter` 对象获得；若用户单击“取消”按钮，则不执行后续的打印操作。

第 4 行创建一个 `QPainter` 对象，并指定绘图设备为一个 `QPrinter` 对象。

第 5 行获得 `QPainter` 对象的视口矩形。

第 6 行获得图像的大小。

第 7、8 行按照图形的比例大小重新设置视口矩形。

第 9 行设置 `QPainter` 窗口大小为图像的大小。

第 10 行打印。

实例 27 图片的缩放与旋转

知识点：

使用 `QMatrix` 实现坐标变换

图片的缩放与旋转是图像处理的常用功能，本实例通过 `QMatrix` 实现图片的缩放、旋转以及镜像等功能。

`QMatrix` 类提供了世界坐标系统的 2D 转换功能，可以使窗体转换变形，经常在绘图程序中使用，`QMatrix` 可以实现坐标系统的移动、缩放、变形以及旋转功能。

实现的效果图如图 4-6 所示。



图 4-6 图片的缩放与旋转

在本实例中，选择需要打开的图形文件，支持的文件格式包括.bmp、.jpg、.pbm、.png、.ppm、.xbm、.xpm 等，选中后，图形将以初始状态显示。实例中实现的功能包括图形的放大和缩小、顺时针旋转 90°、顺时针旋转 180°、顺时针旋转 270°、垂直镜像和水平镜像，并在工具条上提供相应的快捷按钮。

具体实现代码如下所示。

头文件 imgprocessor.h:

```
class ImgProcessor : public QMainWindow      // image processor widget
{
    Q_OBJECT
public:
    ImgProcessor();
    ~ImgProcessor();

    void createMenus();
    void createActions();
    void createToolBars();

private:
    QImage img;
    QLabel *imageLabel;

    QMenu *fileMenu;
    QMenu *zoomMenu;
    QMenu *rotateMenu;
    QMenu *mirrorMenu;

    QAction *openFileAction;
    QAction *zoomInAction;
    QAction *zoomOutAction;
    QAction *rotate90Action;
    QAction *rotate180Action;
    QAction *rotate270Action;
    QAction *mirrorVerticalAction;
    QAction *mirrorHorizontalAction;

    QToolBar *fileTool;
    QToolBar *zoomTool;
    QToolBar *rotateTool;
    QToolBar *mirrorTool;

protected slots:
```



```
void slotOpenFile();
void slotZoomIn();
void slotZoomOut();
void slotRotate90();
void slotRotate180();
void slotRotate270();
void slotMirrorVertical();
void slotMirrorHorizontal();
};
```

头文件定义了实例中需要用到的各种窗体控件，包括菜单、工具条、动作以及各种操作的槽函数。

实现文件 imgprocessor.cpp：

```
ImgProcessor::ImgProcessor()
    : QMainWindow()
{
    1   QFont f("ZYSong18030",12);
    2   setFont(f);

    3   setWindowTitle(tr("Image Processor"));

    4   imgLabel = new QLabel(this);
    5   imgLabel->setSizePolicy(QSizePolicy::Ignored, QSizePolicy::Ignored);
    6   imgLabel->setScaledContents(true);

    7   setCentralWidget(imgLabel);

    8   createActions();
    9   createMenus();
    10  createToolBars();
}
```

第 1、2 行设置字体。

第 3 行设置主窗体的标题。

第 4~6 行创建了一个 QLabel，用来放置和显示图形，setScaledContents 用来设置该控件的 scaledContents 属性，是否根据其大小自动调节内容大小，以使内容充满整个有效区域。若设置为 true，当显示图片时，控件会根据其大小对图片进行调节。该属性默认值为 false。另外可以通过 hasScaledContents () 来获取该属性值。

第 7 行将 imgLabel 设置为主窗体的中央窗体。

第 8~10 行分别是创建动作、菜单和工具条的实现函数。具体实现方式与前面的例子

类似，在本实例中不详细讲述。

slotZoomIn 函数主要实现图形的放大功能，代码如下：

```
void ImgProcessor::slotZoomIn()
{
    1 if(img.isNull())
    2     return;

    3 QMatrix martix;
    4 martix.scale(2,2);
    5 img=img.transformed(martix);
    6 imageLabel->setPixmap(QPixmap::fromImage(img));
    7 resize(img.width(),img.height());
}
```

第 1、2 行为有效性判断。

第 3~5 行声明一个 QMatrix 类的实例，按照两倍比例对水平和垂直方向进行放大，并将当前显示的图形按照该坐标矩阵进行转换。

QMatrix & QMatrix::scale (qreal sx, qreal sy) 函数返回缩放后的 martix 对象引用，若要实现两倍比例的缩小，则参数 sx 和 sy 改成 0.5 即可。

第 6、7 行重新设置显示图形并调整大小。



小贴士： scale (qreal sx, qreal sy) 函数的参数是 qreal 类型值。qreal 定义了一种 double 数据类型，该数据类型适用于所有的平台。要注意的是，对于 ARM 体系结构的平台，qreal 是一种 float 类型。在 Qt4 中还声明了一些指定位长度的数据类型，目的是保证程序能在 Qt 支持的所有平台上能够正常运行。如 qint8 表示一个有符号的 8 位字节。qlonglong 表示 long long int 类型，与 qint64 相同。详细情况可以参考 Qt 在线帮助。

slotRotate90 () 函数的实现代码如下：

```
void ImgProcessor::slotRotate90()
{
    if(img.isNull())
        return;

    QMatrix martix;
    martix.rotate(90);
    img=img.transformed(martix);
    imageLabel->setPixmap(QPixmap::fromImage(img));
```



```
    resize(img.width(),img.height());  
}
```

slotRotate90()函数实现的是图形的旋转，matrix.rotate(90)实现坐标的逆时针旋转90°。这里需要注意的是，在窗口设计中，由于坐标系的Y轴是向下的，因此，用户看到的图形是逆时针旋转90°，而实际上是顺时针旋转90°。

slotMirrorVertical()函数的实现代码如下：

```
void ImgProcessor::slotMirrorVertical()  
{  
    if(img.isNull())  
        return;  
  
    img=img.mirrored(false,true);  
    imageLabel->setPixmap(QPixmap::fromImage(img));  
    resize(img.width(),img.height());  
}
```

slotMirrorVertical()函数实现的是图形的垂直镜像，本实例通过 QImage::mirrored(bool horizontal = false, bool vertical = true)实现图形的镜像功能，参数 horizontal 和 vertical 分别指定了镜像的方向。

实例 28 在工具栏中嵌入控件

知识点：

如何在工具栏中插入控件

本实例实现一个继承自 QMainWindow 的窗体，包括一个工具栏和一个中央窗体，在工具栏上插入一个 QComboBox 控件和一个 QSpinBox 控件，并实现 QComboBox 控件和 QSpinBox 控件的动作响应。实例效果图如图 4-7 所示。

当改变下拉列表框或 SpinBox 的值时，值变化的情况会在下面的中央窗体中显示出来。具体实现代码如下所示。

头文件 toolbar.h:

```
classToolBar : public QMainWindow  
{
```

```

Q_OBJECT
public:
    ToolBar();
public slots:
    void slotComboBox(QString);
    void slotSpinBox(QString);
private:
    QTextEdit *text;
    QSpinBox *spin;
    QComboBox *box;
};

```



图 4-7 在工具栏中嵌入控件实例效果图

头文件中声明了两个需插入工具栏的控件（QComboBox 和 QSpinBox）和作为中央窗体的 QTextEdit 控件。

声明了两个用于响应 QComboBox 和 QSpinBox 控件动作的槽函数 slotComboBox() 和 slotSpinBox()。

实现文件 toolbar.cpp：

```

ToolBar::ToolBar()
{
1   setWindowTitle(tr("Insert a ComboBox to toolbar"));

2   text = new QTextEdit(this);           //中央窗体
3   text->setReadOnly(true);
4   setCentralWidget(text);

5   QToolBar *toolBar = addToolBar("control");
6   QLabel *label1 = new QLabel(tr("ctrl1:")); //控件 1: QLabel 和 QComboBox

```



```
7   box = new QComboBox;
8   box->insertItem(0,tr("ComboBox 1"));
9   box->insertItem(1,tr("ComboBox 2"));
10  box->insertItem(2,tr("ComboBox 3"));
11  toolBar->addWidget(label1);
12  toolBar->addWidget(box);
13  toolBar->addSeparator();
14  QLabel *label2 = new QLabel(tr("ctrl2: ")); //控件 2: QLabel 和 QSpinBox
15  spin = new QSpinBox;
16  spin->setRange(1,10);
17  toolBar->addWidget(label2);
18  toolBar->addWidget(spin);
19  connect(box,SIGNAL(currentIndexChanged(QString)),
20          this,SLOT(slotComboBox(QString)));
21  connect(spin,SIGNAL(valueChanged(QString)),
22          this,SLOT(slotSpinBox(QString)));
}
```

第 1 行设置窗体的标题。

第 2~4 行创建一个 QTextEdit 对象，作为中央窗体，并设置为只读。

第 5 行在主窗口中插入一个名为 control 的工具栏。

第 6~12 行完成往工具栏上插入一个标签和一个下拉列表框控件，首先分别创建一个 QLabel 控件和一个 QComboBox 控件，并在 QComboBox 控件中加入条目，调用工具栏对象的 addWidget() 函数插入控件，工具栏会根据调用的顺序，依次插入控件。

第 13 行在工具栏中加入一个分割符。

第 14~18 行完成往工具栏上插入一个标签和一个 QSpinBox 控件，同样调用 addWidget() 函数插入。

第 19 行连接 QComboBox 对象的 currentIndexChanged() 信号与响应槽函数 slotComboBox()，当单击下拉列表框并改变它的显示值时，调用槽函数 slotComboBox() 作出相应的反映。

第 20 行连接 QSpinBox 对象的 valueChanged() 信号与响应槽函数 slotSpinBox()，当选 择 QSpinBox 对象并改变它显示的值时，调用槽函数作出相应的反映。

```
void ToolBar::slotComboBox(QString combo)
{
    QString doc;
    QString spinStr;
    doc = "QComboBox: " + combo + "\n" + "QSpinBox: " + spinStr.setNum(spin->value());
```

```
    text->setText(doc);
}

voidToolBar::slotSpinBox(QString value)
{
    QString doc;
    doc = "QComboBox: " + box->currentText() + "\n" + "QSpinBox: " + value;
    text->setText(doc);
}
```

两个响应控件动作的槽函数，主要完成显示更新的功能，当控件显示值发生改变时，将值的变化显示在中央窗体中。

实例 29 设置字体、字号等格式属性

知识点：

- 与文本编辑相关的各个类之间的关系
- 通过 QTextCharFormat 修改文本的字体信息
- 利用 QTextCursor 设置字体信息

本实例实现一个设置文字字体、字号的对话框，程序继承自 QMainWindow 类，在工具栏上实现设置文字字体、字号大小、加粗、斜体、下划线以及字体颜色等快捷按钮，实例效果图如图 4-8 所示。



图 4-8 设置字体等文字属性实例效果图

在编辑框中输入一段文字，用鼠标选取文字，修改工具栏上的字体、字号大小、加粗等属性，选取的文字即发生相应的变化。

在分析本实例之前，先明确几个基本的概念，在编写包含格式设置的文本编辑程序时，经常用到的 Qt 类有：QTextEdit、QTextDocument、QTextCharFormat、QTextCursor、QTextBlock、QTextList、QTextFrame、QTextTable、QTextBlockFormat、QTextListFormat、QTextFrameFormat、QTextTableFormat 等。

刚看到如此多的相关类，可能会感到有些混乱，但只要弄清了它们之间的关系，运用起来就会非常方便，Qt 已经为用户完成了几乎所有与编辑有关的具体工作，我们所要做的就是运用合适的类，调用合适的函数接口。

首先，任何一个文本编辑的程序都要用到 QTextEdit 作为输入文本的容器，在它里面输入的可编辑文本由 QTextDocument 作为载体，而 QTextBlock、QTextList、QTextFrame 等则用来表示 QTextDocument 的元素，也可理解为 QTextDocument 的不同表现形式，可能为字符串、段落、列表、表格或是图片等。每种元素都有自己的格式，这些格式则用 QTextCharFormat、QTextBlockFormat、QTextListFormat、QTextFrameFormat 等类来描述与实现。例如，QTextBlockFormat 类对应于 QTextBlock 类，QTextBlock 类用于表示一块文本，一般可以理解为一个段落，但它并不只指段落，QTextBlockFormat 类则用于表示这一块文本的格式，如缩进的值、与四边的边距等。

各类之间的划分与关系可用图 4-9 来进行描述。

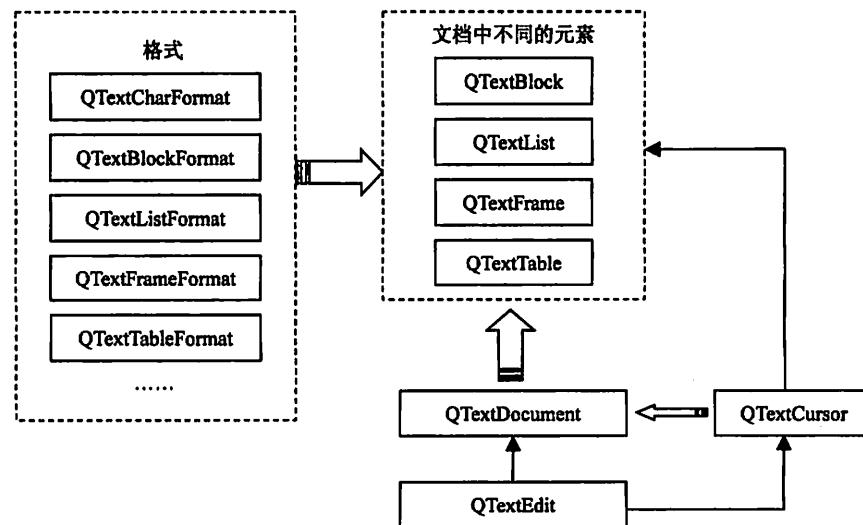


图 4-9 文本格式关系图

从图 4-9 中可知，`QTextCursor` 类是一个非常重要也经常会用到的类，它用于表示编辑文本中的光标。这个类提供了对 `QTextDocument` 文档的修改接口，所有对文档格式的修改，说到底都与光标有关，如改变字符的格式，指的是改变光标处字符的格式；改变段落的格式，指的是改变光标所在段落的格式，因此，`QTextCursor` 类在文档编辑类程序中有着重要的作用，所有对 `QTextDocument` 的修改能够通过 `QTextCursor` 类实现。这在本实例以及后续实例的分析中会逐步体现。

本实例使用了 `QTextCharFormat` 类来对文本字符的格式进行修改，在后续的章节中会讲述利用 `QTextBlockFormat`、`QTextListFormat` 以及 `QTextCursor` 实现设置其他风格的实例分析。具体实现文件如下所示。

头文件 fonts.h:

```
class FontSet : public QMainWindow
{
    Q_OBJECT
public:
    FontSet();

    QLabel *label1,
    QLabel *label2;

    QFontComboBox *fontBox;
    QComboBox *sizeBox;
    QToolButton *boldBtn;
    QToolButton *italicBtn;
    QToolButton *underBtn;
    QToolButton *colorBtn;

    void mergeFormat(QTextCharFormat);

public slots:
    void slotFont(QString);
    void slotSize(QString);
    void slotBold();
    void slotItalic();
    void slotUnder();
    void slotColor();
    void slotCurrentFormatChanged(const QTextCharFormat &fmt);

private:
    QTextEdit *text;
};
```



头文件中声明了一个用于进行文字输入的文字编辑框和用于设置各种文字属性的控件，包括一个字体选择下拉列表框 QFontComboBox 对象，一个用于设置字号的 QComboBox 对象，各快捷按钮以及相应的槽函数。

实现文件 fonts.cpp:

```
FontSet::FontSet()
{
    1   setWindowTitle(tr("font"));

    2   text = new QTextEdit(this);
    3   setCentralWidget(text);

    4   QToolBar *toolBar = addToolBar("Font");

        /* 字体 */
    5   QLabel *label1 = new QLabel(tr("ZiTi: "));
    6   fontBox = new QFontComboBox;
    7   fontBox->setFontFilters(QFontComboBox::ScalableFonts);
    8   toolBar->addWidget(label1);
    9   toolBar->addWidget(fontBox);

        /* 号字 */
   10  QLabel *label2 = new QLabel(tr("number: "));
   11  sizeBox = new QComboBox;
   12  toolBar->addWidget(label2);
   13  toolBar->addWidget(sizeBox);
   14  QFontDatabase db;
   15  foreach(int size, db.standardSizes())
   16      sizeBox->addItem(QString::number(size));

   17  toolBar->addSeparator();

        /* 加粗、斜体、下划线、颜色 */
   18  boldBtn = new QToolButton;
   19  boldBtn->setIcon(QIcon(":/images/bold.png"));
   20  boldBtn->setCheckable(true);
   21  toolBar->addWidget(boldBtn);
   22  italicBtn = new QToolButton;
   23  italicBtn->setIcon(QIcon(":/images/italic.png"));
   24  italicBtn->setCheckable(true);
   25  toolBar->addWidget(italicBtn);
   26  underBtn = new QToolButton;
   27  underBtn->setIcon(QIcon(":/images/underline.png"));
   28  underBtn->setCheckable(true);
```

```

29 toolbar->addWidget(underBtn);

30 toolbar->addSeparator();
31 colorBtn = new QToolButton;
32 colorBtn->setIcon(QIcon("./images/color.png"));
33 toolbar->addWidget(colorBtn);

/* 连接各信号与槽函数 */
34 connect(fontBox,SIGNAL(activated(QString)),this,SLOT(slotFont(QString)));
35 connect(sizeBox,SIGNAL(activated(QString)),this,SLOT(slotSize(QString)));
36 connect(boldBtn,SIGNAL(clicked()),this,SLOT(slotBold()));
37 connect(italicBtn,SIGNAL(clicked()),this,SLOT(slotItalic()));
38 connect(underBtn,SIGNAL(clicked()),this,SLOT(slotUnder()));
39 connect(colorBtn,SIGNAL(clicked()),this,SLOT(slotColor()));
40 connect(text,
           SIGNAL(currentCharFormatChanged(const QTextCharFormat&)),this,
           SLOT(slotCurrentFormatChanged(const QTextCharFormat&)));
}

```

第 1 行设置对话框的标题。

第 2、3 行创建一个 QTextEdit 对象，并把它设置为中央窗体。

第 4 行在对话框中插入一个名为 Font 的工具栏。

第 5~9 行完成在工具栏中插入设置文字字体的控件。包括一个标签和一个 QFontComboBox 对象，调用 QFontComboBox 的 setFontFilters 接口过滤只在下拉列表框中显示某一类字体，默认情况下为 QFontComboBox::AllFonts 列出所有字体。

第 10~13 行完成在工具栏中插入设置文字字号大小的控件。包括一个标签和一个 QComboBox 对象。

第 14~16 行完成在字号下拉列表框中填充各不同的字号条目，此处使用 QFontDatabase 来实现，QFontDatabase 类用于表示当前系统中所有可用的格式信息，主要是字体和字号大小，本实例只用它来列出字号，调用 standardSizes() 函数返回可用标准字号的列表，并将它们插入到字号下拉列表框中。



小贴士：在第 15 行使用了 foreach，foreach 是 Qt 提供的替代 C++ 中 for 循环的关键字，它的使用方法：

```

foreach(variable,container) container 表示程序中需要循环读取的一个列表,
variable 用于表示每个元素的变量；如:
foreach(int, QList<int>)
{
//process

```

```
}
```

循环至列表尾结束循环

第 17 行在工具栏中插入一个分割条。

第 18~29 行分别插入了 3 个工具按钮，按钮只显示图标，分别为加粗按钮、斜体按钮和下划线按钮。

第 30 行插入一个分割条。

第 31~33 行在工具栏中插入设置文字颜色的工具按钮。

第 34~40 行连接工具栏上各控件触发的信号与相应的槽函数，其中，第 40 行为 QTextEdit 对象的 currentCharFormatChanged() 信号连接了响应槽函数，当光标所在位置的字符格式发生变化时触发此信号，即光标位置发生改变后，新位置的字符格式与之前位置的不一样时触发此信号。

slotFont() 槽函数完成设置选定文字字体的工作，代码如下：

```
void FontSet::slotFont(QString f) //设置字体
{
    1   QTextCharFormat fmt;
    2   fmt.setFontFamily(f);
    3   mergeFormat(fmt);
}
```

第 1 行首先创建一个 QTextCharFormat 对象。

第 2 行调用 QTextCharFormat 的 setFontFamily() 函数选择的字体名称设置给 QTextCharFormat 对象。

第 3 行调用 mergeFormat() 函数把新的格式应用到光标选区内的字符。

mergeFormat() 函数的实现：

```
void FontSet::mergeFormat(QTextCharFormat fmt)
{
    QTextCursor cursor = text->textCursor();
    if (!cursor.hasSelection())
        cursor.select(QTextCursor::WordUnderCursor);
    cursor.mergeCharFormat(format);
    text->mergeCurrentCharFormat(format);
}
```

前面介绍过，所有对于 QTextDocument 进行的修改都通过 QTextCursor 类来完成， mergeFormat() 函数的作用即利用 QTextCursor 类把所有新的格式应用到光标选区内的字符上，每个格式设置槽函数最后都需调用此函数来应用更新。

首先获得编辑框中的光标，若光标没有高亮选区则把光标所在处的词作为选区，由前后空格或“，”、“。”等标点符号区分词，调用 QTextCursor 的 mergeCharFormat() 函数把参数 format 所表示的格式应用到光标所在处的字符上，最后调用 QTextEdit 的 mergeCurrentCharFormat() 函数把格式应用到选区内的所有字符上。后面其他的格式设置也可采用此种方法。

其他各 font 属性的设置与字体属性的设置类似。

设置选定文字的字号大小的代码如下：

```
void FontSet::slotSize(QString num) //设置字号
{
    QTextCharFormat fmt;
    fint.setFontPointSize(num.toFloat());
    text->mergeCurrentCharFormat(fmt);
}
```

设置选定文字为加粗显示的代码如下：

```
void FontSet::slotBold() //设置文字显示加粗
{
    QTextCharFormat fmt;
    fint.setFontWeight(boldBtn->isChecked() ? QFont::Bold : QFont::Normal);
    text->mergeCurrentCharFormat(f);
}
```

文字的粗细值由 QFont::Weight 表示，它是一个整型值，可为 0~99，它有 5 个预置的值，分别为 QFont::Light(25)、QFont::Normal(50)、QFont::DemiBold(63)、QFont::Bold(75) 和 QFont::Black(87)，一般在 QFont::Normal 和 QFont::Bold 之间转换。此处调用 QTextCharFormat 的 setFontWeight() 函数设置粗细值，若检测加粗按钮按下则设置字符的 Weight 值为 QFont::Bold，可直接设为 75；反之，则设为 QFont::Normal。

设置选定文字为斜体显示的代码如下：

```
void FontSet::slotItalic() //设置文字显示斜体
{
    QFont f = text->currentFont();
    f.setItalic(!f.italic());
    text->mergeCurrentCharFormat(f);
}
```

在选定文字下方加下划线的代码如下：

```
void FontSet::slotUnder() //设置文字加下划线
{
```



```

QTextCharFormat fmt;
fmt.setFontUnderline(underBtn->isChecked());
text->mergeCurrentCharFormat(fmt);
}
}

```

设置选定文字的颜色的代码如下：

```

void FontSet::slotColor() //设置文字颜色
{
    QColor color = QColorDialog::getColor(Qt::red, this);

    if(color.isValid())
    {
        QTextCharFormat fmt;
        fmt.setForeground(color);
        text->mergeCurrentCharFormat(fmt);
    }
}

```

设置选定文字的颜色使用了标准颜色对话框的方式，当单击触发颜色按钮时，弹出标准颜色对话框选择颜色。

```

void FontSet::slotCurrentFormatChanged(const QTextCharFormat &fmt)
{
    fontBox->setcurrentIndex(fontBox->findText(fmt.fontFamily()));
    sizeBox->setcurrentIndex(sizeBox->findText(QString::number(fmt.fontPointSize())));
    boldBtn->setChecked(fmt.font().bold());
    italicBtn->setChecked(fmt.font().italic());
    underBtn->setChecked(fmt.fontUnderline());
}

```

当光标所在处的字符格式发生变化时调用此槽函数，函数根据新的字符格式把工具栏上各个格式控件的显示更新。

本实例只是简单地实现了对文本编辑中几种格式的设置，分析了设置各种格式的基本方法，在实际的编程应用中还需要加入更多细节的考虑和控制。

实例 30 设置文本排序及对齐

知识点：

实现文本排序的基本流程

通过 QTextListFormat 改变文本排序的相关信息

在编写文本编辑器等应用中，常需要对文本进行排序，以列表的方式显示各段文本。本实例即实现文本的排序功能，并实现文本的对齐及撤销和恢复功能，如图 4-10 所示。

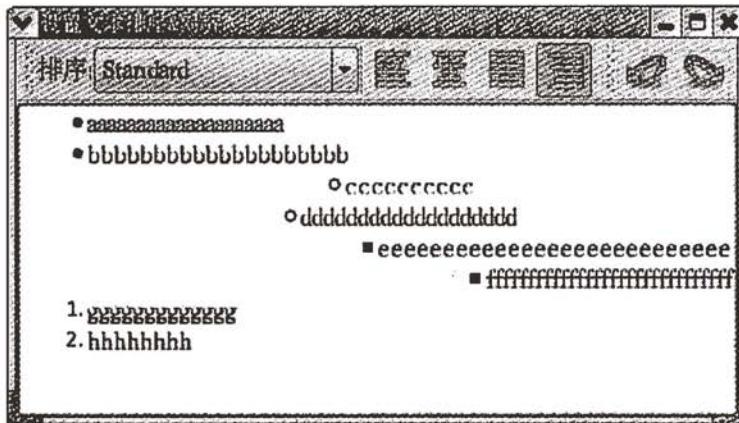


图 4-10 设置文本排序及对齐

在编辑框中任意输入几段文字，单击工具栏上的下拉列表框，选择某种排序方式，则光标所在的文本段以所选排序方式自动缩进排列显示，并且紧接着的文本段以同样的方式排列；工具栏中部的 4 个快捷按钮实现文本的对齐功能，分别为左对齐、右对齐、居中和两端对齐；工具栏右部的快捷按钮实现文本操作的前进/回退功能。

本实例中，主要用到的 Qt 类有：QTextCursor、QTextListFormat 以及 QTextBlockFormat，在前面的实例中介绍过 QTextListFormat 主要用于描述文本排序的格式，它主要包含两个基本的属性，一个为 QTextListFormat::style，表示文本采用哪种排序方式；另一个为 QTextListFormat::indent，表示排序后的缩进值。因此，要实现文本排序的功能只需设置好 QTextListFormat 的这两个属性，并把整个格式通过 QTextCursor 类应用到文本中即可。

文本排序功能实现的基本流程如图 4-11 所示，在一般的文本编辑器中，QTextListFormat 的缩进值 indent 都是预设好的，并不需要用户来设定。因此，在本实例中采用的也是在程序中通过获取当前文本段 QTextBlockFormat 的缩进值来进行相应的计算以获得排序文本的缩进值。

其余两个功能，文本对齐和前进/回退功能相对简单，Qt 已经完成了所有的工作，只要正确地调用函数接口即可。

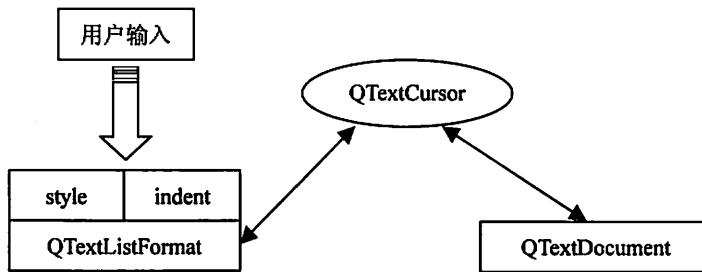


图 4-11 文本排序功能实现的基本流程

具体实现代码如下所示。

头文件 listalign.h:

```
class ListAlign : public QMainWindow
{
    Q_OBJECT
public:
    ListAlign(QWidget *parent=0);

public slots:
    void slotAlignment(QAction*);
    void slotList(int);
    void slotCursorPositionChanged();

private:
    QTextEdit *text;
    QLabel *label;
    QComboBox *listBox;
    QAction *leftAction;
    QAction *rightAction;
    QAction *centerAction;
    QAction *justifyAction;
    QAction *redoAction;
    QAction *undoAction;
};
```

头文件中声明了程序中用到的各窗体控件，包括一个文本输入框 QTextEdit，一个用于选择排序方式的下拉列表框和一系列在工具栏上使用的动作，以及相关的槽函数。

实现文件 listalign.cpp:

```
ListAlign::ListAlign(QWidget *parent)
```

```
1 : QMainWindow(parent)
2 {
3     setWindowTitle(tr("List&Alignment"));
4
5     QToolBar *toolBar = addToolBar("List");
6
7     QLabel *label = new QLabel(tr("List: "));
8     listBox = new QComboBox(toolBar);
9     /* 插入可选的排序方式 */
10    ..... //与 QTextListFormat 的 Style 条目相对应
11    toolBar->addWidget(label);
12    toolBar->addWidget(listBox);
13
14    toolBar->addSeparator();
15    QActionGroup *actGrp = new QActionGroup(this);
16    leftAction = new QAction(QIcon(":/images/left.png"), "left", actGrp);
17    leftAction->setCheckable(true);
18
19    centerAction = new QAction(QIcon(":/images/center.png"), "left", actGrp);
20    centerAction->setCheckable(true);
21
22    justifyAction = new QAction(QIcon(":/images/justify.png"), "left", actGrp);
23    justifyAction->setCheckable(true);
24
25    rightAction = new QAction(QIcon(":/images/right.png"), "left", actGrp);
26    rightAction->setCheckable(true);
27    toolBar->addAction(actGrp->actions());
28
29    QToolBar *editBar = addToolBar("Edit");
30    undoAction = new QAction(QIcon(":/images/undo.png"), "redo", this);
31    editBar->addAction(undoAction);
32    redoAction = new QAction(QIcon(":/images/redo.png"), "redo", this);
33    editBar->addAction(redoAction);
34    editBar->addAction(undoAction);
35    editBar->addAction(redoAction);
36
37    text = new QTextEdit(this);
38    text->setFocus();
39    setCentralWidget(text);
40
41    connect(listBox, SIGNAL(activated(int)), this, SLOT(slotList(int)));
42    connect(actGrp, SIGNAL(triggered(QAction*)), this,
```



```

        SLOT(slotAlignment(QAction*));
30 connect(redoAction,SIGNAL(triggered()),text,SLOT(redo()));
31 connect(undoAction,SIGNAL(triggered()),text,SLOT(undo()));
32 connect(text->document(),SIGNAL(redoAvailable(bool)),redoAction,
           SLOT(setEnabled(bool)));
33 connect(text->document(),SIGNAL(undoAvailable(bool)),undoAction,
           SLOT(setEnabled(bool)));
34 connect(text,SIGNAL(cursorPositionChanged()),this,
           SLOT(slotCursorPositionChanged())));
}

```

第 1 行设置主窗体的标题。

第 2 行插入一个工具栏。

第 3~6 行在工具栏中插入文本排序方式选择的控件，包括一个 QLabel 对象和一个 QComboBox 对象，并在 QComboBox 对象中插入与 QTextListFormat 类的 Style 属性相对应的排序方式条目。

第 7 行插入一个分割条。

第 8~17 行完成工具栏上文本对齐快捷按钮的插入，由于这 4 个快捷按钮不能同时被选中，为单选按钮，因此采用了 QActionGroup 的形式实现，QActionGroup 表示一个动作的集合，集合中同时只能有一个动作有效，它们的触发信号 triggered() 也是以组的形式触发，在第 29 行连接中可以看到它的信号与槽函数的连接方式，最后第 17 行调用 QToolBar 的 addActions() 函数插入这个动作集合。

第 18 行插入一个新的工具条 Edit，用来完成撤销/恢复的功能。

第 19~24 行在工具条中插入撤销/恢复的快捷按钮。

第 25~27 行创建一个 QTextEdit 对象，设置为主窗体的中央窗体，作为文本的输入框。

第 28~34 行完成各相关信号与槽函数的连接，其中，第 30~33 行的连接即完成了撤销/恢复的工作，第 30、31 行把撤销/恢复快捷按钮的 triggered() 信号与 QTextEdit 对象的 redo() 和 undo() 槽函数相连完成实际的撤销/恢复工作；第 32、33 行的连接完成撤销/恢复按钮状态的显示更新，根据当前文本是否可前进或是否可回退，从而决定撤销/恢复按钮是否可用；第 34 行连接函数响应光标位置改变信号 cursorPositionChanged()，主要用于更新工具栏快捷按钮的状态显示。

slotAlignment() 函数的实现代码如下：

```

void ListAlign::slotAlignment(QAction *act)
{
    if (act == leftAction)
        text->setAlignment(Qt::AlignLeft);
}

```

```

if (act == centerAction)
    text->setAlignment(Qt::AlignCenter);
if (act == justifyAction)
    text->setAlignment(Qt::AlignJustify);
if (act == rightAction)
    text->setAlignment(Qt::AlignRight);
}

```

slotAlignment()函数完成对按下某个对齐按钮的响应，根据比较判断触发的是哪个对齐按钮，调用 QTextEdit 的 setAlignment()函数可以实现当前段落的对齐调整。

slotCursorPositionChanged()函数的实现代码如下：

```

void ListAlign::slotCursorPositionChanged()
{
    if (text->alignment() == Qt::AlignLeft)
        leftAction->setChecked(true);
    if (text->alignment() == Qt::AlignCenter)
        centerAction->setChecked(true);
    if (text->alignment() == Qt::AlignJustify)
        justifyAction->setChecked(true);
    if (text->alignment() == Qt::AlignRight)
        rightAction->setChecked(true);
}

```

slotCursorPositionChanged()函数响应文本中光标位置发生改变的信号，完成 4 个对齐按钮的状态更新。通过调用 QTextEdit 类的 alignment()函数获得当前光标所在处段落的对齐方式，设置相应的对齐为按下状态。

slotList()函数实现根据用户选择的不同排序方式对文本进行排序，代码如下：

```

void ListAlign::slotList(int index)
{
1   QTextCursor cursor = text->textCursor();
2   if (index != 0) {
3       QTextListFormat::Style style = QTextListFormat::ListDisc;
4       switch (index) {
5           default:
6           case 1:
7               style = QTextListFormat::ListDisc;
8               break;
}

```



```
9         case 2:
10            style = QTextListFormat::ListCircle;
11            break;
12        case 3:
13            style = QTextListFormat::ListSquare;
14            break;
15        case 4:
16            style = QTextListFormat::ListDecimal;
17            break;
18        case 5:
19            style = QTextListFormat::ListLowerAlpha;
20            break;
21        case 6:
22            style = QTextListFormat::ListUpperAlpha;
23            break;
24    }
25
26    cursor.beginEditBlock();
27
28    QTextBlockFormat blockFmt = cursor.blockFormat();
29    QTextListFormat listFmt;
30    if (cursor.currentList())
31    {
32        listFmt = cursor.currentList()->format();
33    }
34    else
35    {
36        listFmt.setIndent(blockFmt.indent() + 1);
37        blockFmt.setIndent(0);
38        cursor.setBlockFormat(blockFmt);
39    }
40    listFmt.setStyle(style);
41    cursor.createList(listFmt);
42
43    cursor.endEditBlock();
44}
45else
46{
47    QTextBlockFormat bfmt;
48    bfmt.setObjectIndex(-1);
49    cursor.mergeBlockFormat(bfmt);
50}
51}
```

第1行获得编辑框的 QTextCursor 对象指针。

第2~23行从下拉列表框的选择确定 QTextListFormat 的 style 属性的值，Qt 共提供了6种文本排序的方式，分别是 QTextListFormat::ListDisc、QTextListFormat::ListCircle、QTextListFormat::ListSquare、QTextListFormat::ListDecimal、QTextListFormat::ListLowerAlpha 和 QTextListFormat::ListUpperAlpha，这6种方式的显示效果可以通过运行实例看到。

第24~34行完成 QTextListFormat 的另一个属性 indent，即缩进值的设定，并把设置的格式应用到光标所在的文本处，这段程序代码以 beginEditBlock() 开始，以 endEditBlock() 结束，这两个函数的作用是设定这两个函数之间的所有操作相当于一个动作，如果需要进行撤销或恢复，则这两个函数之间的所有操作将同时被撤销或恢复，这两个函数一般成对出现。设置 QTextListFormat 的缩进值首先通过 QTextCursor 获得 QTextBlockFormat 对象，由 QTextBlockFormat 获得段落的缩进值，在此基础上定义 QTextListFormat 的缩进值，本实例是在段落缩进的基础上加1，也可根据需要进行其他的设定。

第5章 图形与图画

Qt 的画图机制为屏幕显示和打印显示提供了统一的 API 接口，主要由 3 部分组成：QPainter 类、QPaintDevice 类和 QPaintEngine 类，如图 5-1 所示。QPainter 类提供了画图操作的各种接口；QPaintDevice 类提供可用于画图的空间，它是一个抽象类；而 QPaintEngine 类则为 QPainter 类和 QPaintDevice 类提供内部使用的抽象接口定义，对于开发者而言，一般是不会用到的。



图 5-1 API 接口的 3 个类

QPainter 类提供了丰富的操作接口，可以很方便地绘制各种各样的图形，从简单的直线、方形、圆形等，到文字和图片，与 QPainterPath 类配合，甚至可以绘制出任意复杂的图形。

QPaintDevice 类是所有可作为画图容器，即所有可用 QPainter 类进行绘制的类的基本类。目前，Qt 提供的具有此属性的类有：QWidget 类、 QImage 类、 QPixmap 类、 QPicture 类、 QGLWidget 类、 QPrinter 类和 QGLPixelBuffer 类，如图 5-2 所示。其中， QWidget 类和 QPixmap 类是最常用的画图容器。

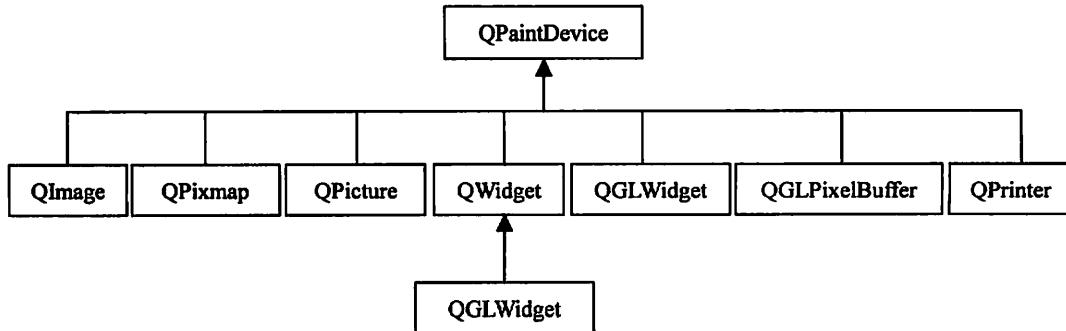


图 5-2 Qt 提供的类

本章包括 8 个实例：

- 利用 QPainter 绘制各种图形
- 利用 QPainterPath 进行画图
- 渐变效果
- QPainter 坐标系的变形
- SVG 格式图片的显示
- 一个简单的绘图工具
- 改变图片的透明度
- 橡皮筋线



实例 31 利用 QPainter 绘制各种图形

知识点：

- 使用 QPainter 提供的 draw 函数绘制基本图形
- QPen 的使用方法
- QBrush 的使用方法

本实例利用 QPainter 类提供的各种 draw 函数，绘制各种类型的图形，包括对图形的形状、颜色、填充风格等的选择，如图 5-3 所示。

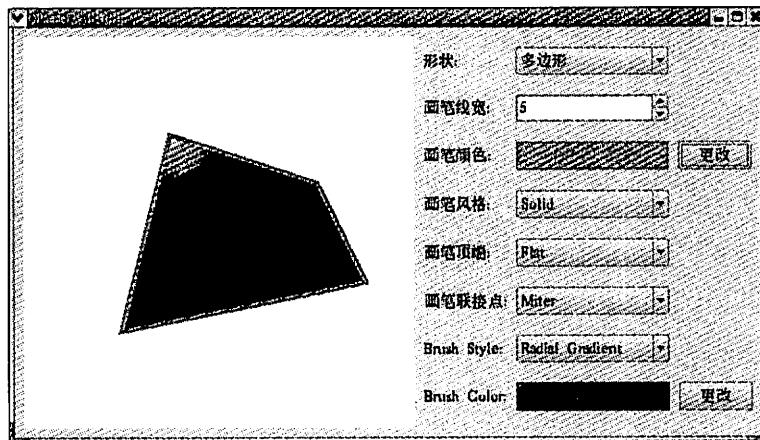
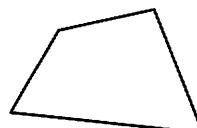


图 5-3 利用 QPainter 绘制各种图形

本实例主要是利用了 QPainter 提供的各种 draw 函数来实现。Qt 为开发者提供了丰富的绘制基本图形的 draw 函数，如图 5-4 所示。



QPainter::drawLine()



QPainter::drawPolygon()



QPainter::drawRect()

图 5-4 draw 函数

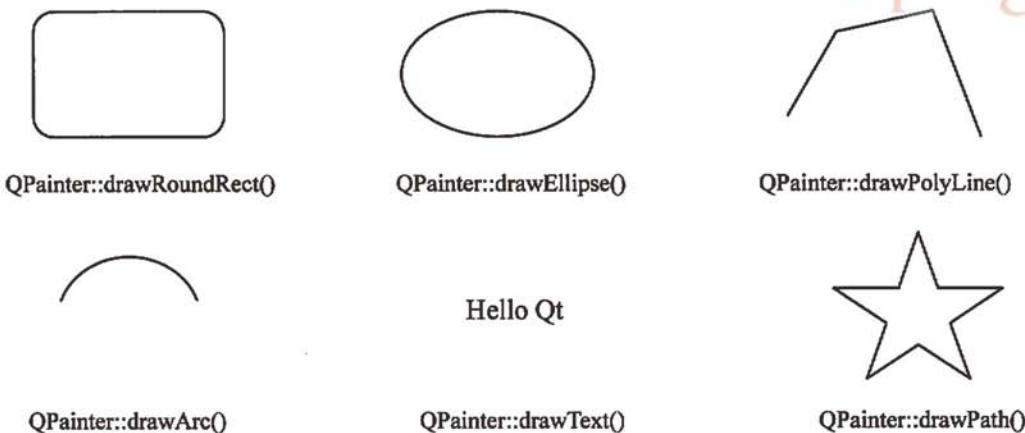


图 5-4 draw 函数 (续)

此外，QPainter 类还提供了一个 drawPixmap() 函数，可直接把图片画到可画控件上。本实例的具体实现包括两个部分的内容，一个是用于画图的区域 PaintArea 类，一个是主窗口 MainWidget 类，如图 5-5 所示。



图 5-5 PaintArea 类和 MainWidget 类

PaintArea 类的实现：

```

class PaintArea : public QWidget
{
    Q_OBJECT
public:
1   enum Shape {Line, Rectangle, RoundRect, Ellipse, Polygon,
              Polyline, Points, Arc, Path, Text,Pixmap};
2   PaintArea(QWidget *parent=0);
  
```

```
3 void setShape(Shape);
4 void setPen(QPen);
5 void setBrush(QBrush);
6 void paintEvent(QPaintEvent *);  
  
private:  
7 Shape shape;  
8 QBrush brush;  
9 QPen pen;  
};
```

PaintArea 类的类声明中，第 1 行声明了一个枚举型数据 Shape，列举了所有本实例中可能用到的图形形状。

第 2 行声明了 PaintArea 类的构造函数。

第 3~5 行声明的函数为主窗口类提供传递参数的接口，设置形状、画笔和画刷。

第 6 行重定义 paintEvent() 函数。

第 7~9 行声明了重画所需的 3 个参数。

PaintArea 类的构造函数：

```
PaintArea::PaintArea(QWidget *parent)
{
    setPalette(QPalette(Qt::white));
    setAutoFillBackground(true);

    setMinimumSize(400,400);
}
```

PaintArea 类的构造函数，设置了图形显示区域的背景色以及最小显示尺寸。

PaintArea 类提供的 3 个设置函数：

```
void PaintArea::setShape(Shape s)
{
    shape = s;
    update();
}

void PaintArea::setPen(QPen p)
{
    pen = p;
    update();
}
```

```
}

void PaintArea::setBrush(QBrush b)
{
    brush = b;
    update();
}
```

PaintArea 类提供了 3 个设置函数：setShape()、setPen()和 setBrush()用于设置重画时用到的 3 个基本参数，包括所画图形的形状以及画笔和画刷的属性。

PaintArea 类的重画函数：

```
void PaintArea::paintEvent(QPaintEvent *)
{
    1   QPainter p(this);
    2   p.setPen(pen);
    3   p.setBrush(brush);

    4   QRect rect(50,100,300,200);

    5   static const QPoint points[4] = {
        QPoint(150, 100),
        QPoint(300, 150),
        QPoint(350, 250),
        QPoint(100, 300)
    };

    6   int startAngle = 30 * 16;
    7   int spanAngle = 120 * 16;

    8   QPainterPath path;
    9   path.addRect(150,150,100,100);
    10  path.moveTo(100,100);
    11  path.cubicTo(300,100,200,200,300,300);
    12  path.cubicTo(100,300,200,200,100,100);

    13  switch(shape)
    14  {
    15      case Line:           //直线
    16          p.drawLine(rect.topLeft(),rect.bottomRight());
    17      break;
    18      case Rectangle:       //长方形
    19          p.drawRect(rect);
```

```
19     break;
20 case RoundRect:           //圆角方形
21     p.drawRoundRect(rect);
22     break;
23 case Ellipse:             //椭圆形
24     p.drawEllipse(rect);
25     break;
26 case Polygon:              //多边形
27     p.drawPolygon(points,4);
28     break;
29 case Polyline:             //多边线
30     p.drawPolyline(points,4);
31     break;
32 case Points:               //点
33     p.drawPoints(points,4);
34     break;
35 case Arc:                  //弧
36     p.drawArc(rect,startAngle,spanAngle);
37     break;
38 case Path:                  //路径
39     p.drawPath(path);
40     break;
41 case Text:                  //文字
42     p.drawText(rect.Qt::AlignCenter,tr("Hello Qt!"));
43     break;
44 casePixmap:                //图片
45     p.drawPixmap(150,150,QPixmap("./images/butterfly.png"));
46     break;
47 default:
48     break;
}
}
```

第 1 行新建一个 QPainter 对象。

第 2、3 行设置 QPainter 对象的画笔和画刷。

第 4 行设定一个方形区域，为画长方形、圆角方形、椭圆等作准备。

第 5 行创建一个 QPoint 的数组，其中包含 4 个点，为画多边形、多边线以及点作准备。

第 6、7 行新建的整型变量 startAngle 和 spanAngle，是为画弧作的准备，其中，startAngle 表示起始角，为弧形的起始点与圆心之间连线与水平方向的夹角；spanAngle 表示的是跨度角，为弧形起点、终点分别与圆心连线之间的夹角，如图 5-6 所示。

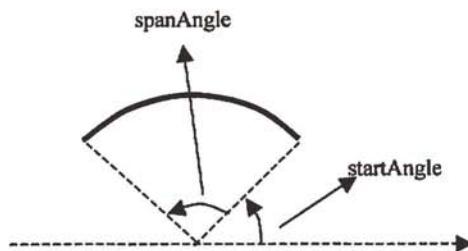


图 5-6 startAngle 和 spanAngle

 小贴士：用 QPainter 画弧形所使用的角度值，是以 $1/16$ 度为单位的，即 1° 在画弧时用 16 表示。

- 第 8~12 行新建一个 QPainterPath 对象，为画路径作准备。
- 第 13~47 行用一个 switch()语句，对所要画的形状作判断，调用 QPainter 的各个 draw() 函数完成图形的绘制。

主窗口类 MainWidget 的实现：

```
class MainWidget : public QWidget
{
    Q_OBJECT
public:
    1   MainWidget(QWidget *parent=0);

public slots:
    2   void slotShape(int);
    3   void slotPenWidth(int);
    4   void slotPenColor();
    5   void slotPenStyle(int);
    6   void slotPenCap(int);
    7   void slotPenJoin(int);
    8   void slotBrushColor();
    9   void slotBrush(int);

private:
    10  PaintArea *area;

    11  QComboBox *shapeComboBox;
    12  QSpinBox *widthSpinBox;
    13  QComboBox *penStyleComboBox;
    14  QComboBox *penCapComboBox;
    15  QComboBox *penJoinComboBox;
```



```
16 QComboBox *brushStyleComboBox;
17 QFrame *penColorFrame;
18 QFrame *brushColorFrame;
};
```

MainWidget 类的构造函数中,第 2~9 行声明了一系列设置与画图相关参数的槽函数。第 10 行声明了一个 PaintArea 对象。

MainWidget 类的构造函数中创建了各参数选择控件,大多数采用下拉列表框的形式。具体代码如下:

```
MainWidget::MainWidget(QWidget *parent)
    : QWidget(parent)
{
    area = new PaintArea;
    .....
    //形状选择下拉列表框
1    shapeComboBox = new QComboBox;
2    shapeComboBox->addItem(tr("Line"), PaintArea::Line);
    .....
3    connect(shapeComboBox,SIGNAL(activated(int)),this,SLOT(slotShape(int)));

    //画笔线宽选择控件
4    widthSpinBox = new QSpinBox;
5    widthSpinBox->setRange(0,20);
6    connect(widthSpinBox,SIGNAL(valueChanged(int)),this,
              SLOT(slotPenWidth(int)));

    //画笔颜色选择控件
7    penColorFrame = new QFrame;
8    penColorFrame->setAutoFillBackground(true);
9    penColorFrame->setPalette(QPalette(Qt::blue));
10   QPushButton *penColorPushButton = new QPushButton(tr("Change"));
11   connect(penColorPushButton,SIGNAL(clicked()),this,SLOT(slotPenColor()));

    //画笔风格选择下拉列表框
12  penStyleComboBox = new QComboBox;
13  penStyleComboBox->addItem(tr("Solid"), Qt::SolidLine);
    .....
14  connect(penStyleComboBox,SIGNAL(activated(int)),this,
            SLOT(slotPenStyle(int)));

    //画笔顶端风格选择下拉列表框
```

```

15 penCapComboBox = new QComboBox;
16 penCapComboBox->addItem(tr("Flat"), Qt::FlatCap);
.....
17 connect(penCapComboBox,SIGNAL(activated(int)),this,SLOT(slotPenCap(int)));

```

Qt 为画笔的顶端预置了 3 种风格，分别是 Qt::SquareCap、Qt::FlatCap 和 Qt::RoundCap，如图 5-7 所示。



图 5-7 Qt 为画笔的顶端预置的 3 种风格

其中，Qt::SquareCap 表示在线条的顶点处是方形的，且线条绘制的区域包括了端点，并且再往外延伸半个线宽的长度；Qt::FlatCap 表示在线条的顶点处是方形的，但线条绘制区域不包括端点在内；Qt::RoundCap 表示在线条的顶点处是圆形的，且线条绘制区域包含了端点。

```

//画笔连接点风格选择下拉列表框
18 penJoinComboBox = new QComboBox;
19 penJoinComboBox->addItem(tr("Miter"), Qt::MiterJoin);
.....
20 connect(penJoinComboBox,SIGNAL(activated(int)),this,SLOT(slotPenJoin(int)));

```

对于画笔连接点的风格，Qt 提供了 3 种预置的效果，分别是 Qt::BevelJoin、Qt::MiterJoin 和 Qt::RoundJoin，如图 5-8 所示。

其中，Qt::BevelJoin 风格连接是指两条线的中心线顶点相汇，相连处依然保留线条各自的方形顶端；Qt::MiterJoin 风格连接是指两条线的中心线顶点相汇，相连处线条延长到线的外侧汇集至点，形成一个尖顶的连接；Qt::RoundJoin 风格连接是指两条线的中心线顶点相汇，相连处以圆弧形连接。

```

//画刷风格选择下拉列表框
21 brushStyleComboBox = new QComboBox;
22 brushStyleComboBox->addItem(tr("Linear Gradient"),
    Qt::LinearGradientPattern);
.....
23 connect(brushStyleComboBox,SIGNAL(activated(int)),this,
    SLOT(slotBrush(int)));

```

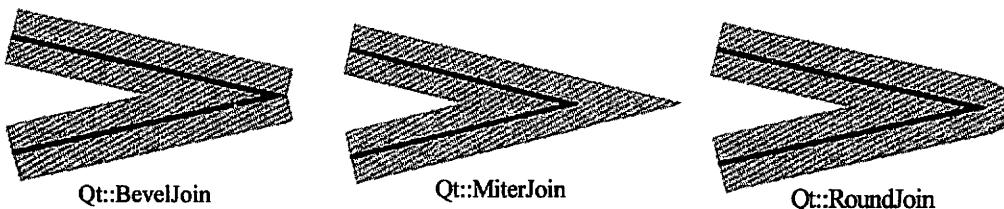


图 5-8 Qt 提供的 3 种预置效果

Qt 为画刷预置了十几种填充风格，此处不一一列举，如果需要可查阅 QBrush 类的帮助文档。

```
//画刷颜色选择控件  
24 brushColorFrame = new QFrame;  
25 brushColorFrame->setAutoFillBackground(true);  
26 brushColorFrame->setPalette(QPalette(Qt::green));  
27 QPushButton *brushColorPushButton = new QPushButton(tr("Change"));  
28 connect(brushColorPushButton,SIGNAL(clicked()),this,SLOT(slotBrushColor()));  
  
//布局  
.....  
29 setWindowTitle(tr("Basic Paint"));  
  
30 slotShape(shapeComboBox->currentIndex());  
31 slotPenWidth(widthSpinBox->value());  
32 slotBrush(brushStyleComboBox->currentIndex());  
}
```

其中，第 7~11 行画笔颜色的选择和第 24~28 行画刷颜色的选择，采用了 QFrame 和 QPushButton 对象组合完成，QFrame 对象负责显示当前所选择的颜色，QPushButton 对象用于触发颜色标准对话框进行颜色的选择。

第 30~32 行完成对形状、画笔、画刷属性的初始化工作。

slotShape() 函数根据当前下拉列表框中选择的选项，并调用 PaintArea 类的 setShape() 函数设置 PaintArea 对象中的形状参数。具体代码如下：

```
void MainWidget::slotShape(int value)  
{  
    PaintArea::Shape shape = PaintArea::Shape(shapeComboBox->itemData(  
        value, Qt::UserRole).toInt());  
    area->setShape(shape);  
}
```

```

void MainWidget::slotPenWidth(int value)
{
    QColor color = penColorFrame->palette().color(QPalette::Window);
    Qt::PenStyle style = Qt::PenStyle(penStyleComboBox->itemData(
        penStyleComboBox->currentIndex(), Qt::UserRole).toInt());
    Qt::PenCapStyle cap = Qt::PenCapStyle(penCapComboBox->itemData(
        penCapComboBox->currentIndex(), Qt::UserRole).toInt());
    Qt::PenJoinStyle join = Qt::PenJoinStyle(penJoinComboBox->itemData(
        penJoinComboBox->currentIndex(), Qt::UserRole).toInt());
    area->setPen(QPen(color, value, style, cap, join));
}

```

此函数响应画笔线宽选择下拉列表框变化的信号，在此函数中获得与画笔相关的所有属性值，包括画笔颜色、画笔线宽、画笔风格、画笔顶端风格以及画笔连接点风格，共同构成 QPen 对象，并调用 PaintArea 对象的 setPen() 函数设置 PaintArea 对象的画笔属性。

其他与画笔参数相关的响应函数完成的工作与此类似。如：

```

void MainWidget::slotPenStyle(int value)
void MainWidget::slotPenCap(int value)
void MainWidget::slotPenJoin(int value)

```

slotPenColor() 函数响应画笔颜色更改按钮，利用颜色标准对话框 QColorDialog 获取所选的颜色，设置显示，获得画笔其他属性，并调用 PaintArea 对象的 setPen() 函数设置 PaintArea 对象的画笔属性。具体代码如下：

```

void MainWidget::slotPenColor()
{
    QColor color = QColorDialog::getColor(Qt::blue);
    penColorFrame->setPalette(QPalette(color));
    /* 获得画笔的其他属性 */
    area->setPen(QPen(color, value, style, cap, join));
}

```

```

void MainWidget::slotBrushColor()
{
    QColor color = QColorDialog::getColor(Qt::blue);
    brushColorFrame->setPalette(QPalette(color));

    slotBrush(brushStyleComboBox->currentIndex());
}

```

slotBrushColor() 函数设置画刷颜色函数，与设置画笔颜色函数类似，但选定颜色后并



不直接调用 PaintArea 对象的 setBrush() 函数，而是调用 slotBrush() 槽函数，与画刷风格选择一起形成 QBrush 对象，再调用 PaintArea 对象的 setBrush() 函数设置显示区的画刷属性。

slotBrush() 函数完成对画刷风格的选择，代码如下：

```
void MainWidget::slotBrush(int value)
{
    1     QColor color = brushColorFrame->palette().color(QPalette::Window);
    2     Qt::BrushStyle style = Qt::BrushStyle(brushStyleComboBox->itemData(value, Qt::UserRole)
    .toInt());
    3
    4     if (style == Qt::LinearGradientPattern)
    5     {
    6         QLinearGradient linearGradient(0, 0, 400, 400);
    7         linearGradient.setColorAt(0.0, Qt::white);
    8         linearGradient.setColorAt(0.2, color);
    9         linearGradient.setColorAt(1.0, Qt::black);
   10        area->setBrush(linearGradient);
   11    }
   12    else if (style == Qt::RadialGradientPattern)
   13    {
   14        QRadialGradient radialGradient(200, 200, 150, 150, 100);
   15        radialGradient.setColorAt(0.0, Qt::white);
   16        radialGradient.setColorAt(0.2, color);
   17        radialGradient.setColorAt(1.0, Qt::black);
   18        area->setBrush(radialGradient);
   19    }
   20    else if (style == Qt::ConicalGradientPattern)
   21    {
   22        QConicalGradient conicalGradient(200, 200, 30);
   23        conicalGradient.setColorAt(0.0, Qt::white);
   24        conicalGradient.setColorAt(0.2, color);
   25        conicalGradient.setColorAt(1.0, Qt::black);
   26        area->setBrush(conicalGradient);
   27    }
   28    else if (style == Qt::TexturePattern)
   29    {
   30        area->setBrush(QBrush(QPixmap(":/images/cheese.jpg")));
   31    }
   32    else
   33    {
   34        area->setBrush(QBrush(color, style));
   35    }
}
```

```
}
```

第1行获得画刷的颜色。

第2行获得所选的画刷风格，若选择的是渐变或是纹理图案，则需要进行一定的处理。

第3~8行即针对选择的是线性渐变风格情况的处理，新建一个线性渐变对象，并对它变化范围内的颜色进行设定。

第9~14行针对选择的是圆形渐变风格情况的处理。

第15~20行针对选择的是锥形渐变风格情况的处理。

第21、22行针对选择的是纹理图案情况的处理。

第23行是针对其余风格的处理。

实例 32 利用 QPainterPath 进行画图

知识点：

- 使用 QPainterPath 绘制任意形状图形
- QPainterPath 的填充模式

本实例实现一个利用 QPainterPath 绘制的简单图形，并可选择 QPainterPath 的填充模式、边线线宽及颜色，如图 5-9 所示。通过本实例的分析，介绍与 QPainterPath 类相关应用。



图 5-9 利用 QPainterPath 画图



QPainterPath 类为 QPainter 类提供了一个存储容器，里面包含了所要画的内容的集合，以及画的顺序，如长方形、多边形、曲线等各种任意图形。在需要绘制此预先存储在 QPainterPath 对象中的内容时，只需调用 QPainter 类的 drawPath() 函数即可。

本实例的具体实现与前面介绍的例子类似，本实例的实现也是由两个部分组成，一个是由显示图形的画图区域 PaintArea 类，另一个是主窗口类 MainWidget。

```
class PaintArea : public QWidget
{
    Q_OBJECT
public:
    1   PaintArea(QWidget *parent=0);
    2   void setFillRule(int);
    3   void setPenWidth(int);
    4   void setPenColor(QColor);
    5   void setBrushColor(QColor);

    6   void paintEvent(QPaintEvent *); . . .

private:
    7   Qt::FillRule rule;
    8   int width;
    9   QColor penColor;
    10  QColor brushColor;
    11  QPainterPath path;
};
```

在 PaintArea 类的声明中，第 2~5 行声明了一系列用于参数设置的函数，为主窗口传递各种画图属性提供接口。

第 6 行重定义了 paintEvent() 函数。

第 7~10 行声明了与画图有关的变量，包括图形的填充规则、画笔线宽、画笔颜色及画刷颜色。

第 11 行声明了一个 QPainterPath 对象，用于保存需绘制的内容。

PaintArea 类的构造函数的代码如下：

```
PaintArea::PaintArea(QWidget *parent)
{
    1   setPalette(QPalette(Qt::white));
    2   setAutoFillBackground(true);
    3   setMinimumSize(400,400);
```

```
4     width = 1;
5     penColor = Qt::red;
6     brushColor = Qt::blue;
7     rule = Qt::OddEvenFill;
8
9
10
11 }
```

第 1、2 行设置画图区域的背景色为白色。

第 3 行设置画图区域的最小显示尺寸。

第 4~7 行初始化有关变量。

第 8~11 行在 QPainterPath 对象中保存了一个需绘制的图形。

第 8 行直接调用 QPainterPath 类的 addRect() 函数在 QPainterPath 对象中加入一个方形的图形，并在参数中指定此方形的位置和大小。QPainterPath 类提供了许多函数接口可以很方便地加入一些规则图形，如 addRect() 加入一个方形；addEllipse() 加入一个椭圆形；addText() 加入一个字符串；addPolygon() 加入一个多边形等。同时，QPainterPath 类还提供了 addPath() 函数用于加入另一个 QPainterPath 对象中保存的内容。

第 9 行把当前点移至(100,100)处，QPainterPath 对象的当前点自动处在上一部分图形内容的结束点上，若下一部分图形的起点不在此结束点上，则需调用 moveTo() 函数把当前点移到下一部分图形的起点上。

第 10、11 行调用 cubicTo() 函数绘制了两条曲线，cubicTo() 函数绘制的是贝塞尔曲线，如图 5-10 所示。它需要 3 个参数，分别表示 3 个点 cubicTo(c1, c2, endPoint)。

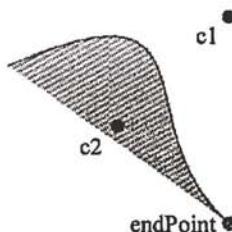


图 5-10 绘制贝塞尔曲线

利用 QPainterPath 类可以实现 QPainter 类的所有 draw() 函数所能实现的图形，如对于 QPainter::drawRect() 函数，除了可用上面介绍的 QPainterPath::addRect() 的方式实现，还可



以用如下方式实现：

```
QPainterPath path;
path.moveTo(0,0);
path.lineTo(200,0);
path.lineTo(200,100);
path.lineTo(0,100);
path.lineTo(0,0);
```

这是一个更通用的方法，其他诸如多边形、多边线等图形都能用这种方式实现。

```
void PaintArea::setFillRule(int index)
{
    rule = Qt::FillRule(index);
    update();
}

void PaintArea::setPenWidth(int w)
{
    width = w;
    update();
}

void PaintArea::setPenColor(QColor c)
{
    penColor = c;
    update();
}

void PaintArea::setBrushColor(QColor c)
{
    brushColor = c;
    update();
}
```

这 4 个参数设置函数分别设置填充规则、画笔线宽、画笔颜色和画刷颜色，并在设置完后立即调用 update() 函数进行重画，使新设置的参数立即生效。

PaintArea 类的重画函数完成具体的重画工作，代码如下：

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

```
2 path.setFillRule(rule);
3 QPen pen;
4 pen.setColor(penColor);
5 pen.setWidth(width);
6 p.setPen(pen);
7 p.setBrush(QBrush(brushColor));
8 p.drawPath(path);
}
```

第 1 行新建一个 QPainter 对象。

第 2 行设置 QPainterPath 的填充规则。

第 3~6 行设置画笔的属性。

第 7 行设置画刷的属性。

第 8 行调用 QPainter 类的 paintPath() 函数把 QPainterPath 对象中保存的图形绘制到图画区域中。

主窗口类 MainWidget 的实现：

```
class MainWidget : public QWidget
{
    Q_OBJECT
public:
1   MainWidget(QWidget *parent=0);
2   void createCtrlWidget();

public slots:
3   void slotPenColor();
4   void slotBrushColor();
5   void slotFillRule();
6   void slotPenWidth(int);

private:
7   PaintArea *area;
8   QWidget *ctrlWidget;
9   QComboBox *fillRuleComboBox;
10  QFrame *penColorFrame;
11  QFrame *brushColorFrame;
};
```

MainWidget 类的类声明中，第 1 行声明了构造函数。

第 2 行声明了一个用于创建参数选择区的函数。



第 3~6 行声明一系列响应参数选择的槽函数。

第 7 行声明了一个 PaintArea 对象私有变量。

MainWidget 类的构造函数：

```
MainWidget::MainWidget(QWidget *parent)
    : QWidget(parent)
{
    1 area = new PaintArea;
    2 ctrlWidget = new QWidget;
    3 createCtrlWidget();
    .....
}
```

第 1 行新建一个 PaintArea 对象。

第 2、3 行调用 createCtrlWidget() 函数创建参数选择区。

```
void MainWidget::createCtrlWidget() //创建参数选择控制区
{
    .....
    //填充规则选择下拉列表框
    fillRuleComboBox = new QComboBox;
    fillRuleComboBox->addItem(tr("Odd Even"), Qt::OddEvenFill);
    fillRuleComboBox->addItem(tr("Winding"), Qt::WindingFill);
    connect(fillRuleComboBox, SIGNAL(activated(int)), this, SLOT(slotFillRule()));
    .....
}
```

在创建参数选择区的函数中，关于 QPainterPath 类的填充规则，Qt 提供了两种填充规则，分别是 Qt::OddEvenFill 和 Qt::WindingFill。这两种填充规则在判定图形中某一点是处于内部还是外部时，判断依据不同。

其中，Qt::OddEvenFill 填充规则判断的依据是从图形中某一点画一条水平线到图形外，若这条水平线与图形边线的交点数目为奇数，则说明此点位于图形的内部；若为偶数，则此点位于图形的外部，如图 5-11 所示。

而 Qt::WindingFill 填充规则的判断依据则是从图形中某一点画一条水平线到图形外，每个交点处边线的方向可能向上，也可能向下，把这些交点数累加，方向相反的相互抵消，若最后结果不为 0 则说明此点在图形内，若为 0 则说明在图形外，如图 5-12 所示。

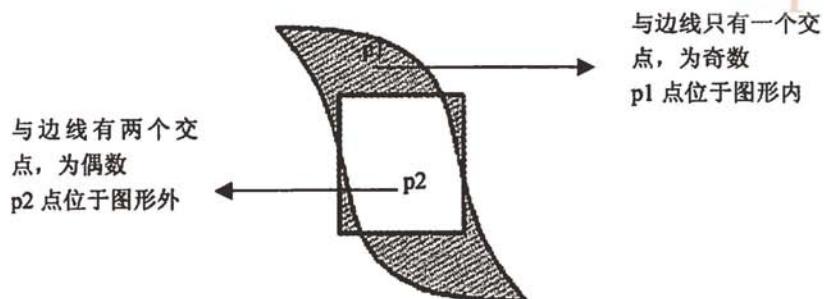


图 5-11 Qt::OddEvenFill 填充规则的判断依据

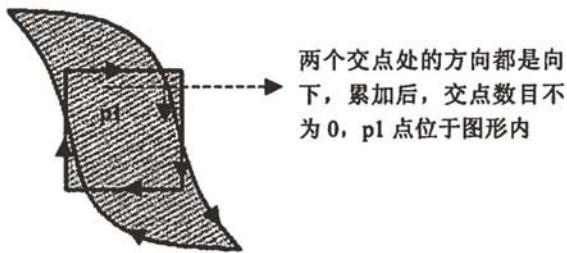


图 5-12 Qt::WindingFill 填充规则的判断依据

其中，边线的方向是由 QPainterPath 创建时，根据描述的顺序决定的，如果是采用 addRect() 或 addPolygon() 等函数加入的图形，默认是按顺时针方向。

实例 33 渐变效果

知识点：

- 3 种渐变模式
- 两种铺展效果
- 渐变颜色的设置

本实例实现一个画渐变色的例子，可选择渐变色的起止颜色及渐变的效果。通过本实例讲解渐变效果类 QGradient，及其 3 个子类 QLinearGradient、QRadialGradient 和

QConicalGradient 的使用方法。渐变效果如图 5-13 所示。

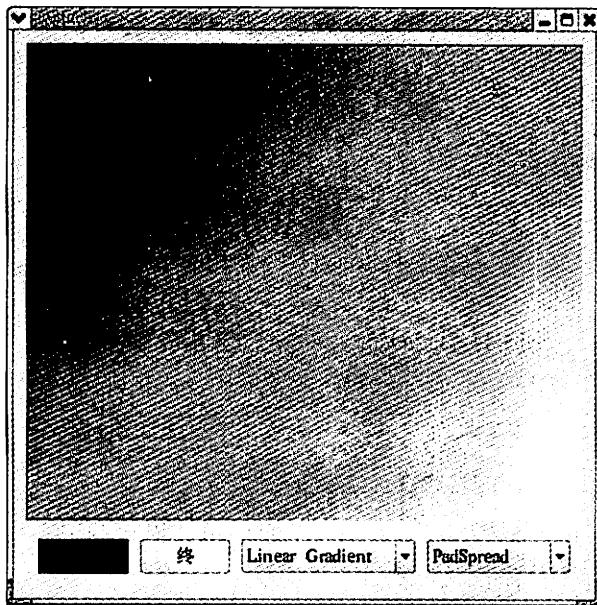


图 5-13 渐变效果

窗体下方左侧的两个按钮分别选择渐变色起点颜色和终点颜色，右侧的两个下拉列表框控件可选择渐变色的效果。在上方的显示区域，通过鼠标拖动确定渐变色的范围和方向，以鼠标的按下点和松开点作为渐变色显示范围的起点和终点，以鼠标拖动的方向作为渐变显示的方向。

Qt 提供了一个与渐变效果相关的 QGradient 类，根据目前 Qt 所支持的 3 种渐变效果：线性渐变、圆形渐变及锥形渐变，QGradient 类继承了 3 个子类，分别是 QLinearGradient 类、QRadialGradient 类和 QConicalGradient 类。QGradient 类是一个基类，在实际应用时，通常都使用它的 3 个子类，如图 5-14 所示。

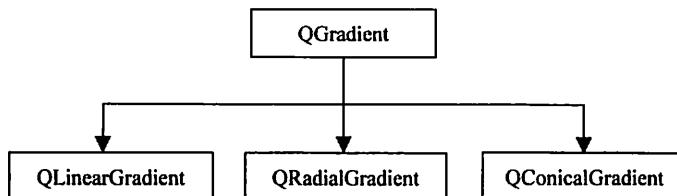


图 5-14 QGradient 的 3 个子类

本实例对 Qt 提供的各种渐变效果进行了显示和分析，具体实现代码如下。

本实例由两部分组成，分别是主窗口部分和用于渐变效果显示的自定义的一个 PaintArea 类，如图 5-15 所示。

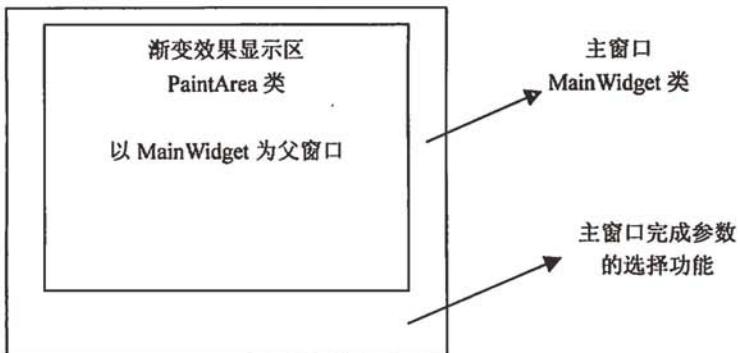


图 5-15 主窗口和 PaintArea 类

主窗口的类声明：

```
class MainWidget : public QWidget
{
    Q_OBJECT
public:
    1 MainWidget(QWidget *parent=0);
    2 void createCtrlWidget();

    3 QColor startColor;
    4 QColor endColor;
    5 Qt::BrushStyle style;
    6 QGradient::Spread spread;

public slots:
    7 void slotStartColor();
    8 void slotEndColor();
    9 void slotSetStyle(int);
   10 void slotSetSpread(int);

private:
    11 QWidget *ctrlWidget;

   12 QPushButton *startPushButton;
   13 QPushButton *endPushButton;
```

```
14 QCComboBox *gradientComboBox;
15 QCComboBox *spreadComboBox;
};
```

其中,第3~6行与显示相关的参数,声明为公有参数以便在另一个显示类 PaintArea 对象中能够使用,分别是渐变起始颜色 startColor、渐变终止颜色 endColor、画刷风格 style (主要用以指定是哪种类型的渐变)以及渐变的铺展效果 spread。

第11行声明的 ctrlWidget 用于放置参数选择控件。

渐变效果显示类 PaintArea 的类声明:

```
#include "mainwidget.h"
```

由于需要在 PaintArea 对象中使用主窗口中的变量,因此需包含主窗口的头文件 mainwidget.h。

```
class PaintArea : public QWidget
{
    Q_OBJECT
public:
1   PaintArea(MainWidget *parent);
2   void mousePressEvent(QMouseEvent * );
3   void mouseReleaseEvent(QMouseEvent * );
4   void paintEvent(QPaintEvent * );
private:
5   MainWidget *mainWidget;
6   QPoint startPoint;
7   QPoint endPoint;
8   QColor startColor;
9   QColor endColor;
10  QGradient::Type style;
};
```

第1行 PaintArea 类把主窗口类 MainWidget 作为父窗口,在构造函数中指定。

第2~4行声明 PaintArea 类需重定义的虚函数。

第5行声明一个主窗口的对象,便于调用主窗口中的变量。

第6、7行声明了两个 QPoint 变量,在响应鼠标事件时使用。

第8~10行声明了一系列与显示相关的变量。

PaintArea 类的构造函数:

```
PaintArea::PaintArea(MainWidget *parent)
{
    1   setPalette(QPalette(Qt::white));
    2   setAutoFillBackground(true);
    3   setMinimumSize(400,400);

    4   mainWidget = parent;
    5   startPoint = QPoint(0,0);
    6   endPoint = QPoint(400,400);
}
```

第1、2行设置此显示区的背景色为白色。

第3行设定显示区的最小显示尺寸。

第4行保存父窗口的对象指针。

第5、6行初始化变量。

响应鼠标按下事件和松开事件的函数：

```
void PaintArea::mousePressEvent(QMouseEvent * e)
{
    startPoint = e->pos();
}

void PaintArea::mouseReleaseEvent(QMouseEvent * e)
{
    endPoint = e->pos();
    update();
}
```

当鼠标按下时，记录下鼠标所在点的位置信息 startPoint；当鼠标松开时，记录下鼠标松开所在点的位置信息 endPoint，并调用 update() 函数进行重画。

PaintArea 类的重画函数：

```
void PaintArea::paintEvent(QPaintEvent *)
{
    1   QPainter p(this);
    2   QRect r = rect();
    3   if (mainWidget->style == QGradient::LinearGradient)      //线性渐变
    {
        4       QLinearGradient linearGradient(startPoint,endPoint);
        5       LinearGradient.setColorAt(0.0, mainWidget->startColor);
        6       linearGradient.setColorAt(1.0, mainWidget->endColor);
        7       linearGradient.setSpread(mainWidget->spread);
    }
}
```

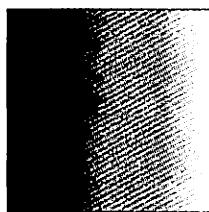
```
8     p.setBrush(linearGradient);
...
9     else if (mainWidget->style == QGradient::RadialGradient) //圆形渐变
{
10    int r = sqrt(pow(endPoint.x()-startPoint.x(),2)
11                + pow(endPoint.y()-startPoint.y(),2));
12    QRadialGradient radialGradient(startPoint, r, startPoint);
13    radialGradient.setColorAt(0.0, mainWidget->startColor);
14    radialGradient.setColorAt(1.0, mainWidget->endColor);
15    radialGradient.setSpread(mainWidget->spread);
16    p.setBrush(radialGradient);
}
16 else if (mainWidget->style == QGradient::ConicalGradient) //锥形渐变
{
17    double angle = atan2(endPoint.y()-startPoint.y(), endPoint.x()-startPoint.x());
18    QConicalGradient conicalGradient(startPoint, -(180*angle)/PI);
19    conicalGradient.setColorAt(0.0, mainWidget->startColor);
20    conicalGradient.setColorAt(1.0, mainWidget->endColor);
21    p.setBrush(conicalGradient);
}
22 p.drawRect(r);
}
```

第 1 行新建一个 QPainter 对象。

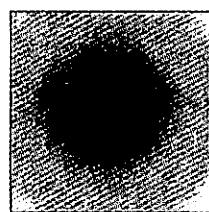
第 2 行获得画图区域的大小 rect。

第 3~21 行，根据主窗口中 style 变量值和 spread 变量值，决定所采用的渐变类型以及渐变铺展类型。

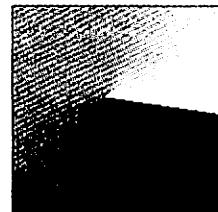
3 种渐变类型的效果如图 5-16 所示。



QGradient::LinearGradient
线性渐变



QGradient::RadialGradient
圆形渐变



QGradient::ConicalGradient
锥形渐变

图 5-16 3 种渐变类型的效果

铺展效果有 3 种，分别为：QGradient::PadSpread、QGradient::RepeatSpread 和

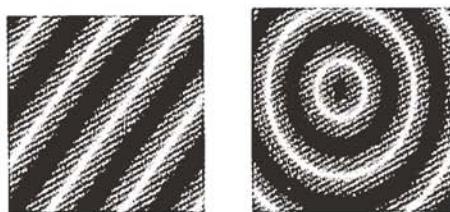
QGradient::ReflectSpread。其中，PadSpread 是默认的铺展效果，也是最常见的铺展效果，没有被渐变覆盖的区域填充单一的起始颜色或终止颜色；RepeatSpread 效果和 ReflectSpread 效果只对线性渐变和圆形渐变起作用。

铺展类型的效果如图 5-17 所示。



QGradient::RepeatSpread

(a) 线性渐变和圆形渐变的 RepeatSpread 铺展效果



QGradient::ReflectSpread

(b) 线性渐变和圆形渐变的 ReflectSpread 铺展效果

图 5-17 铺展类型的效果

第 3~8 行，若主窗口的 style 变量值为 QGradient::LinearGradient，表明选择的是线性渐变。

第 4 行利用鼠标的拖动的起止位置点 startPoint 和 endPoint 新建一个 QLinearGradient 对象。

第 5、6 行分别设置起止的颜色，调用 QGradient 的 setColorAt() 函数进行设置，其中第一个参数表示所设颜色点的位置，取值范围为 0.0~1.0 之间，0.0 表示起点，1.0 表示终点，第二个参数表示该点的颜色值。本实例中只设置了起点和终点的颜色，如需要也可设置中间任意位置的颜色，如 setColorAt(0.3,Qt::white)，设置起、终点之间 1/3 位置的颜色为白色。

第 7 行设置渐变的铺展效果。

第 8 行调用 QPainter 的 setBrush() 函数把准备好的渐变效果应用到画刷中。



第 9~15 行若主窗口的 style 变量值为 QGradient::RadialGradient，表明选择的是圆形渐变，其实现步骤与前面线性渐变效果大体一致。区别在于第 11 行圆形渐变类对象的创建需要 3 个参数，分别表示圆心位置、半径值和焦点位置。本实例中以 startPoint 作为圆心和焦点的位置，以 startPoint 和 endPoint 之间的距离为半径，当然圆心和焦点的位置也可以不重合。

第 16~21 行若主窗口的 style 变量值为 QGradient::ConicalGradient，表明选择的是锥形渐变，第 18 行锥形渐变对象创建需要两个参数，分别是锥形的顶点位置和渐变分界线与水平方向的夹角，如图 5-18 所示。锥形渐变不用设置铺展效果，它的铺展效果只能是 QGradient::PadSpread。

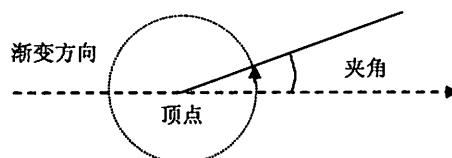


图 5-18 类角

小贴士：锥形渐变的方向默认是逆时针方向。

第 22 行，调用 drawRect() 函数重画整个显示区域。

主窗口类 MainWidget 的实现：

```
MainWidget::MainWidget(QWidget *parent)
    : QWidget(parent)
{
    1   startColor = Qt::green;
    2   endColor = Qt::blue;
    3   style = QGradient::LinearGradient;
    4   spread = QGradient::PadSpread;

    5   PaintArea *area = new PaintArea(this);

    6   ctrlWidget = new QWidget;
    7   createCtrlWidget();

    //布局
    .....
    8   setWindowTitle(tr("Gradient"));
}
```

第1~4行行为变量赋初值。

第5行新建一个PaintArea对象作为显示区。

第6、7行新建参数选择区，并调用createCtrlWidget()函数进行创建。

创建参数选择区函数：

```
void MainWidget::createCtrlWidget()
{
    //创建起、止颜色选择按钮
    1 startPushButton = new QPushButton(tr("start"));
    2 startPushButton->setAutoFillBackground(true);
    3 startPushButton->setPalette(QPalette(Qt::green));
    4 endPushButton = new QPushButton(tr("end"));
    5 endPushButton->setAutoFillBackground(true);
    6 endPushButton->setPalette(QPalette(Qt::blue));
    7 connect(startPushButton,SIGNAL(clicked()),this,SLOT(slotStartColor()));
    8 connect(endPushButton,SIGNAL(clicked()),this,SLOT(slotEndColor()));

    //创建渐变类型选择控件
    9 gradientComboBox = new QComboBox;
    10 gradientComboBox->addItem(tr("Linear Gradient"),
                                QGradient::LinearGradient);
    11 gradientComboBox->addItem(tr("Radial Gradient"),
                                QGradient::RadialGradient);
    12 gradientComboBox->addItem(tr("Conical Gradient"),
                                QGradient::ConicalGradient);
    13 connect(gradientComboBox,SIGNAL(activated(int)),this,
              SLOT(slotSetStyle(int)));

    //创建铺展效果选择控件
    14 spreadComboBox = new QComboBox;
    15 spreadComboBox->addItem(tr("Pad Spread"),QGradient::PadSpread);
    16 spreadComboBox->addItem(tr("Repeat Spread"),QGradient::RepeatSpread);
    17 spreadComboBox->addItem(tr("Reflect Spread"),QGradient::ReflectSpread);
    18 connect(spreadComboBox,SIGNAL(activated(int)),this,
              SLOT(slotSetSpread(int)));
    //布局
    .....
}
```

第1~8行创建两个按钮用于选择起、止颜色，按钮的背景色即所选择的颜色，利用setPalette()函数改变按钮的背景色，必须同时调用setAutoFillBackground()函数设置为



true，对背景颜色的改变才会生效。

第 9~13 行创建一个下拉列表框控件，列出可供选择的 3 种渐变效果，并连接相应的信号与槽函数。

小贴士：QComboBox 的 addItem() 函数可以只插入文本，也可同时插入与文本相对应的具体数据，一般为枚举型数据，便于后面操作时确定选择的是哪个数据。

第 14~18 行创建铺展效果选择下拉列表框，列出 3 种铺展效果。

slotStartColor() 为选择起始颜色的槽函数，通过标准颜色对话框进行选择，并更新按钮的背景色。具体代码如下：

```
void MainWidget::slotStartColor()
{
    startColor = QColorDialog::getColor(Qt::green);
    startPushButton->setPalette(QPalette(startColor));
}
```

slotEndColor() 为选择终止颜色的槽函数，通过标准颜色对话框进行选择，并更新按钮的背景色。具体代码如下：

```
void MainWidget::slotEndColor()
{
    endColor = QColorDialog::getColor(Qt::blue);
    endPushButton->setPalette(QPalette(endColor));
}
```

slotSetStyle 为选择渐变类型的槽函数，通过 QComboBox 的 itemData() 函数获得当前所选择的数据，是一个 QVariant 对象，通过.toInt() 函数转换成在枚举数据集合中的序号，并利用这个序号获得所选的渐变类型。具体代码如下：

```
void MainWidget::slotSetStyle(int value)
{
    style = QGradient::Type(gradientComboBox->itemData(
        value, Qt::UserRole).toInt());
}
```

slotSetSpread() 为选择铺展类型的槽函数，实现与前面渐变类型的选择类似。具体代码如下：

```
void MainWidget::slotSetSpread(int value)
{
    spread = QGradient::Spread(spreadComboBox->itemData(
```

```
    value, Qt::UserRole).toInt());  
}
```

实例 34 QPainter 坐标系的变形

知识点：

- QPainter 类为坐标系变形提供的函数的应用
- rotate(qreal angle): 坐标系旋转
- scale(qreal sx, qreal sy): 坐标系缩放
- translate(qreal dx, qreal dy): 坐标系平移
- shear(qreal sh, qreal sv): 坐标系切变

本实例利用 QPainter 类提供的坐标系变形函数，实现对所画对象的变形，如图 5-19 所示。

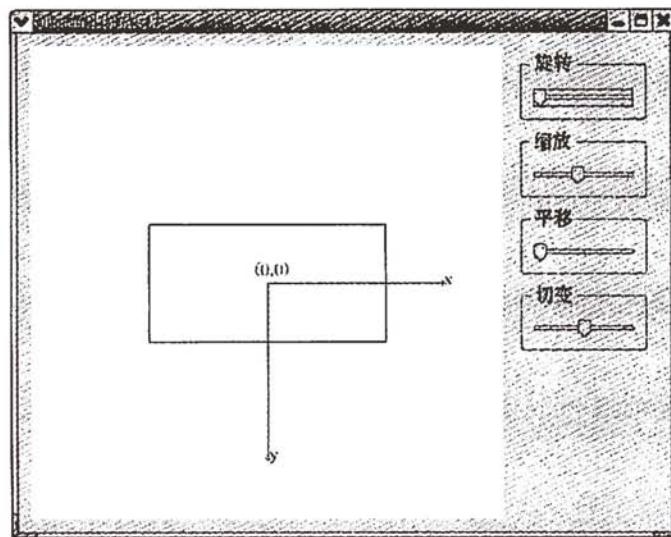


图 5-19 QPainter 坐标系的变形

Qt 画图的坐标系默认是以左上角为原点，X 轴向右，Y 轴向下。此坐标系可受 QPainter 类控制，对之进行变形，QPainter 类提供了相应的变形函数，包括旋转、缩放、平移和切



变。调用这些函数时，显示设备的坐标系会发生相应变形，但绘制内容相对坐标系的位置并不会发生改变，因此看起来像是绘制内容进行了变形，实质是坐标系的变形。若还需实现更加复杂的变形，则可采用 QMatrix 类实现。

本实例具体实现：创建一个完成图形显示功能的 Widget。

```
class PaintArea : public QWidget
{
    Q_OBJECT
public:
    PaintArea(QWidget *parent=0);

    void setScale(int);
    void setTranslate(int);
    void setRotate(int);
    void setShear(int);

    void paintEvent(QPaintEvent *);
```

private:

```
    qreal scale;
    int translate;
    int angle;
    qreal shear;
};
```

第 2~5 行声明了 4 个设置函数，用于主窗口传递变形参数。

第 6 行重定义 paintEvent() 函数。

第 7~10 行声明的 4 个私有变量用于保存与坐标系变形相关的参数。

显示窗体 PaintArea 的构造函数：

```
PaintArea::PaintArea(QWidget *parent)
{
    setPalette(QPalette(Qt::white));
    setAutoFillBackground(true);
    setMinimumSize(400,400);

    scale = 1;
    angle = 0;
    translate = 0;
    shear=0;
}

void PaintArea::setScale(int s)
```

```
{  
    scale = s/5.0;  
    update();  
}  
  
void PaintArea::setRotate(int r)  
{  
    angle = r;  
    update();  
}  
  
void PaintArea::setTranslate(int t)  
{  
    translate = t;  
    update();  
}  
  
void PaintArea::setShear(int h)  
{  
    shear = (h-10.0)/10.0;  
    update();  
}
```

4个设置函数用于设置坐标系的变形参数 scale、rotate、translate、shear。

其中，切变参数的值应控制在-1~1之间。

显示窗体的重画函数：

```
void PaintArea::paintEvent(QPaintEvent *)  
{  
1   QPainter p(this);  
2   p.translate(200,200); //把窗体的坐标原点移到中心点  
  
3   QPainterPath path;  
4   path.addRect(-100,-50,200,100);  
  
5   p.rotate(angle);  
6   p.scale(scale,scale);  
7   p.translate(translate,translate);  
8   p.shear(shear,shear);  
  
9   p.setPen(Qt::red);  
10  p.drawLine(0, 0, 150, 0);  
11  p.drawLine(148, -2, 150, 0);
```

```
12 p.drawLine(148, 2, 150, 0);
13 p.drawText(150, 2, m("x"));
14 p.drawLine(0, 0, 0, -50);
15 p.drawLine(-2, 148, 0, 150);
16 p.drawLine(2, 148, 0, 150);
17 p.drawText(2, 150, m("y"));
18 p.drawText(-10, -6, "(0,0)");
19 p.setPen(QPen(blue));
20 p.drawPath(path);
```

第 1 行创建一个 QPainter 对象。

第 2 行为了方便显示，利用 translate() 函数把坐标系的原点移至显示设备的中央。

第 3、4 行准备绘制方形的路径。

第 5~8 行根据设置的变形参数调用 QPainter 对象的各变形函数进行变形。

第 9~20 行绘制 QPainter 的坐标系及显示图形。

主窗口部分参数设置区的实现较简单，在其他实例中亦进行过介绍，此处不再赘述。

实例 35 SVG 格式图片的显示

知识点：

- 什么是 SVG 文件
- 利用 QSvgWidget 显示 SVQ 格式图片
- 在显示窗体中对 SVG 图片进行缩放
- 在显示窗体中实现对 SVG 图片的拖动

本实例实现一个 SVG 图片浏览器，显示以.svg 结尾的文件，如图 5-20 所示。

SVG 全称是 Scalable Vector Graphics，即可缩放的矢量图形。它是由 W3C（World Wide Web Consortium，万维网联盟）在 2000 年 8 月制定的一种新的二维矢量图形格式，也是规范中的网络矢量图形标准，是一个开放的图形标准。

SVG 格式的特点：

- (1) 基于 XML。
- (2) 采用文本来描述对象。

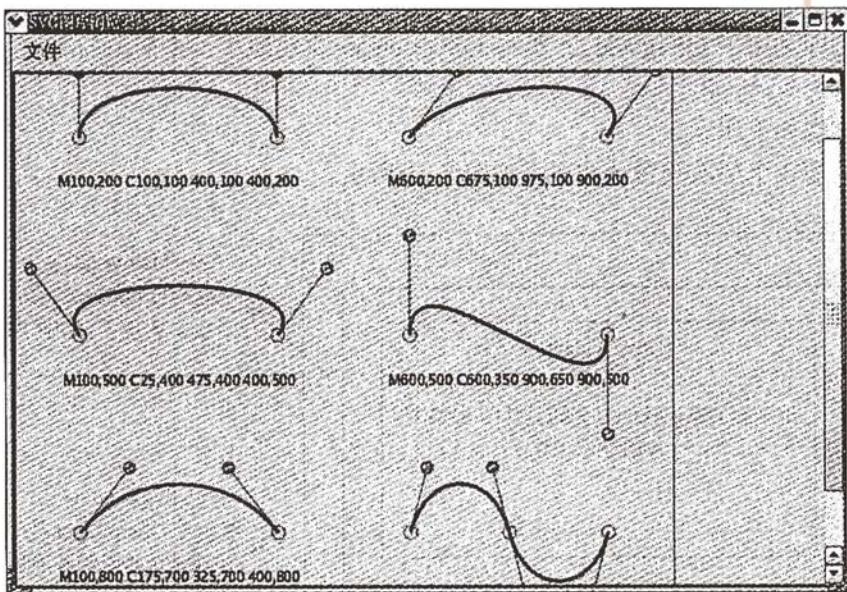


图 5-20 SVG 图片浏览器

(3) 具有交互性和动态性。

(4) 完全支持 DOM。

SVG 相对于 GIF、JPEG 格式的优势：SVG 是一种矢量图形格式，比 GIF、JPEG 等栅格格式具有众多优势，如文件小，对于网络而言下载速度快；可任意缩放而不会破坏图像的清晰度和细节；图像中的文字独立于图像，文字保留可编辑和可搜寻的状态，也没有字体的限制，用户系统即使没有安装某一体字，也会看到和他们制作时完全相同的画面等。正是基于 SVG 格式的各种优点和开放性，它得到了众多组织和知名厂商的支持与认可，因此能够迅速地开发和推广应用。

Qt 为 SVG 格式的图形显示与生成提供了专门的 QtSvg 模块，此模块中包含了与 SVG 图形相关的所有类，主要有 QSvgWidget、QSvgRender 和 QGraphicsSvgItem。

要在程序中使用与 SVG 相关的类，必须在程序中包含 SVG 相关的头文件：

```
#include <QtSvg>
```

由于 Qt 默认生成的 Makefile 中只会加入 QtGui、QtCore 模块的库，因此，必须在工程文件 xxx.pro 中加入一行：

```
QT += svg
```



这样会在编译时加入 QtSvg 的库。

本实例即利用了 QSvgWidget 类与 QSvgRender 类实现了一个简单的 SVG 图片浏览器程序。

具体实现如下所示。

本实例由 3 个层次的窗体构成，如图 5-21 所示。

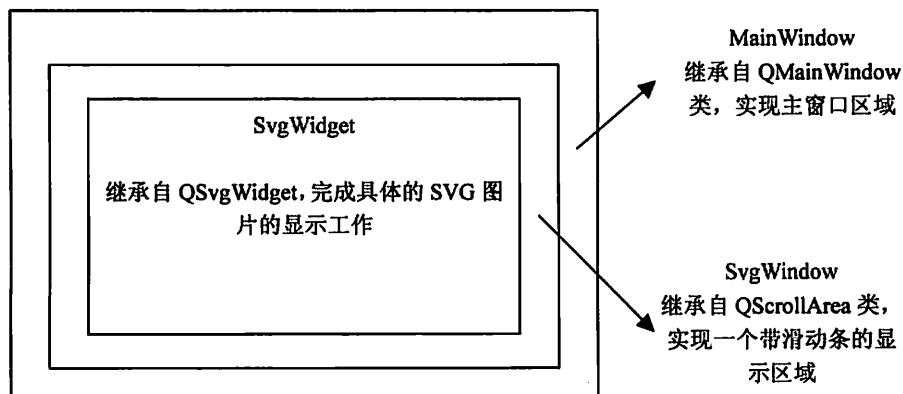


图 5-21 3 个层次的窗体

SvgWidget 类的实现：

```
#include <QtSvg>

class SvgWidget : public QSvgWidget
{
    Q_OBJECT
public:
    1   SvgWidget(QWidget *parent=0);
    2   void wheelEvent(QWheelEvent *);
private:
    3   QSvgRenderer *render;
};
```

SvgWidget 继承自 QSvgWidget 类，QSvgWidget 类的主要作用即显示 SVG 图片。

第 2 行重定义了 wheelEvent() 函数，响应鼠标的滚轮事件，使 SVG 图片能通过鼠标滚轮进行缩放。

第 3 行声明了一个 QSvgRenderer 对象 render，用于图片显示尺寸的确定。

```
SvgWidget::SvgWidget(QWidget *parent)
{
    render = renderer();
}
```

SvgWidget 的构造函数中只需获得本窗体的 QSvgRenderer 对象即可。

```
void SvgWidget::wheelEvent(QWheelEvent * e)
{
    1 const double diff = 0.1;
    2 QSize size = render->defaultSize();
    3 int width = size.width();
    4 int height = size.height();
    5 if (e->delta() > 0)           //向前 放大
    {
        6     width = int(this->width() + this->width()*diff);
        7     height = int(this->height() + this->height()*diff);
    }
    else                      //向后 缩小
    {
        8         width = int(this->width() - this->width()*diff);
        9         height = int(this->height() - this->height()*diff);
    }
    10    resize(width, height);
}
```

第 1 行确定的 diff 值表示每次滚轮滚动一定的值，图片大小改变的比例。

第 2~4 行获取图片显示区的尺寸大小，以便进行下一步的缩放操作。

第 5 行利用 QWheelEvent 的 delta() 函数获得滚轮滚动的距离值，通过此值来判断滚轮滚动的方向，若 delta() 值大于零，则表示滚轮向前（远离用户的方向）滚动；若小于零则表示向后（靠近用户的方向）滚动。

 小贴士：鼠标滚轮事件，滚轮每滚动 1° 相当于移动 8，而常见的滚轮鼠标拨动一下滚动的角度一般为 15°，因此滚轮拨动一下相当于移动了 120 ($=15*8$)。

第 6、7 行对图片的长、宽值进行处理，放大一定的比例。

第 8、9 行对图片的长、宽值进行处理，缩小一定的比例。

第 10 行利用新的长、宽值对图片进行 resize() 操作。

SvgWindow 类的实现：

```
#include "svgwidget.h"
```



在 SvgWindow 实现中包含 SvgWidget 类的头文件。

```
class SvgWindow : public QScrollArea
{
    Q_OBJECT
public:
    1   SvgWindow(QWidget *parent=0);
    2   void setFile(QString);
    3   void mousePressEvent(QMouseEvent *event);
    4   void mouseMoveEvent(QMouseEvent *event);

private:
    5   SvgWidget *svgWidget;
    6   QPoint mousePressPos;
    7   QPoint scrollBarValuesOnMousePress;
};
```

SvgWindow 类继承自 QScrollArea 类，是一个带滚动条的显示区域。使图片在放大到超过主窗口大小时，能通过拖动滚动条的方式进行查看。

第 2 行的 setFile() 函数用于接收主窗口对 SVG 文件的选择设置。

第 3、4 行重定义鼠标响应事件，使窗口可用鼠标对图片进行拖动查看。

第 5 行声明一个 SvgWidget 对象 svgWidget。

第 6 行声明的 QPoint 对象 mousePressPos 用于记录鼠标按下时的位置。

第 7 行声明的 QPainter 对象 scrollBarValuesOnMousePress 用于记录鼠标位置所对应的滑动条的位置。

在构造函数中新建 SvgWidget 对象，并调用 QScrollArea 类的 setWidget() 函数设置滚动区的窗体，使 svgWidget 成为 SvgWindow 的子窗口。具体代码如下：

```
SvgWindow::SvgWindow(QWidget *parent)
{
    svgWidget = new SvgWidget;
    setWidget(svgWidget);
}
```

当主窗口中对文件进行了选择或修改时，会调用 setFile() 函数设置新的文件。具体代码如下：

```
void SvgWindow::setFile(QString fileName)
{
    1   svgWidget->load(fileName);
```

```

2   QSvgRenderer *render = svgWidget->renderer();
3   svgWidget->resize(render->defaultSize());
}

```

第1行调用 `QSvgWidget` 类的 `load()` 函数，把新的 SVG 文件加载到 `svgWidget` 中进行显示。

第2、3行通过获得 `svgWidget` 的 `QSvgRenderer` 对象，使 `svgWidget` 窗体按 SVG 图片的默认尺寸进行显示。

鼠标按下时为 `mousePressPos` 和 `scrollBarValuesOnMousePress` 进行初始化，`QScrollArea` 类的 `horizontalScrollBar()` 和 `verticalScrollBar()` 函数可以分别获得 `svgWindow` 的水平滑动条和垂直滑动条。具体代码如下：

```

void SvgWindow::mousePressEvent(QMouseEvent *event)
{
    mousePressPos = event->pos();
    scrollBarValuesOnMousePress.rx() = horizontalScrollBar()->value();
    scrollBarValuesOnMousePress.ry() = verticalScrollBar()->value();
    event->accept();
}

```

当鼠标按下拖动时触发此响应函数，通过滑动条的位置设置来实现图片拖动的效果。具体代码如下：

```

void SvgWindow::mouseMoveEvent(QMouseEvent *event)
{
1   horizontalScrollBar()->setValue(scrollBarValuesOnMousePress.x()-
                                         event->pos().x() + mousePressPos.x());
2   verticalScrollBar()->setValue(scrollBarValuesOnMousePress.y()-
                                         event->pos().y() + mousePressPos.y());
3   horizontalScrollBar()->update();
4   verticalScrollBar()->update();
5   event->accept();
}

```

第1行对水平滑动条的新位置进行设置。

第2行对垂直滑动条的新位置进行设置。

`MainWindow` 类的实现：

```

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:

```



```

    MainWidget(QWidget *parent=0);
    void createMenu();
public slots:
    void slotOpenFile();
private:
    SvgWindow *svgWindow;
};

```

主窗口 MainWindow 继承自 QMainWindow 类；包含一个菜单栏，其中只有一个“文件”菜单条，包含一个“打开”菜单项。

MainWindow 类中声明了一个 SvgWindow 对象 svgWindow，以调用相关函数传递选择的文件名。

构造函数中，创建一个 SvgWindow 对象作为主窗口的中央窗体。具体代码如下：

```

MainWidget::MainWidget(QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle(tr("SVG Viewer"));
    createMenu();

    svgWindow = new SvgWindow;
    setCentralWidget(svgWindow);
}

```

创建菜单栏。具体代码如下：

```

void MainWidget::createMenu()
{
    QMenu *fileMenu = menuBar()->addMenu(tr("File"));

    QAction *openAct = new QAction(tr("Open"),this);
    connect(openAct,SIGNAL(triggered()),this,SLOT(slotOpenFile()));
    fileMenu->addAction(openAct);
}

```

通过标准文件对话框选择 SVG 文件，并调用 SvgWindow 的 setFile() 函数把选择的文件名传递给 svgWindow 进行显示。具体代码如下：

```

void MainWidget::slotOpenFile()
{
    QString name = QFileDialog::getOpenFileName(
        this,
        "open file dialog",

```

```
"/",
"svg files (*.svg)");
svgWindow->setFile(name);
}
```

 小贴士：在 Qt4 版本中，关于 SVG 又新增了 QSvgGenerator 类用于生成 SVG 文件，QSvgGenerator 类继承自 QPaintDevice，可直接作为画图设备，利用 QPainter 正常地进行各种绘制，最后可将画图指令存为 SVG 文件。

实例 36 一个简单的绘图工具

知识点：

- 双缓冲机制
- 如何利用双缓冲机制实现绘图
- 如何实现鼠标绘图

本实例实现一个简单的绘图工具，可以选择线型、线宽及颜色等基本要素，如图 5-22 所示。



图 5-22 一个简单的绘图工具

本实例的实现包括一个 QMainWindow 对象作为主窗口，一个工具栏 QToolBar 对象，一个 QWidget 对象作为主窗口的中央窗体 centralWidget，也就是绘图区，如图 5-23 所示。

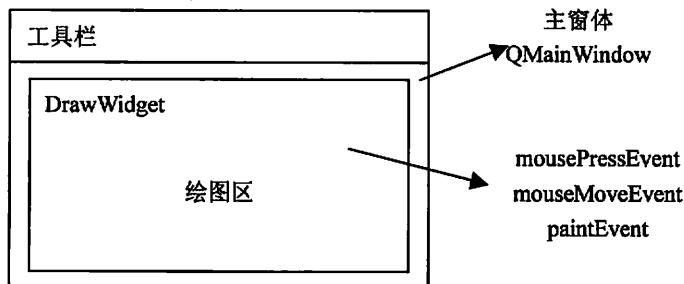


图 5-23 主窗体、工具栏和绘图区

本实例在绘图实现时，采用了双缓冲机制。所谓双缓冲机制，即在绘制控件时，先把所要绘制的内容绘制在一个图片中，再把图片一次性绘制到控件上。双缓冲机制是 GUI 编程中经常使用的方法，可以有效地消除显示时的闪烁现象，在早期的 Qt 版本中，若直接在控件上进行绘制工作，在控件重画时会产生闪烁的现象，在控件重画频繁时，闪烁尤其明显，采用双缓冲机制后，可以有效消除这种闪烁现象。在 Qt4 版本之后，QWidget 控件已经能够自动处理闪烁的问题，因此在控件上直接画图时，不用再操心显示的闪烁问题，但双缓冲机制在很多场合仍然有其用武之地，当所需绘制的内容较复杂并需要频繁刷新，或每次只要刷新整个控件的一小部分时，仍应尽量采用双缓冲机制。

具体实现如下：

绘图区窗体主要是通过响应鼠标事件来完成图形的绘制，因此需要对鼠标事件进行重定义。实例的第一步工作应实现一个响应鼠标事件进行绘图的窗体 DrawWidget。

```
class DrawWidget : public QWidget
{
    Q_OBJECT
public:
    DrawWidget();
    void mousePressEvent(QMouseEvent * );
    void mouseMoveEvent(QMouseEvent * );
    void paintEvent(QPaintEvent * );
    void resizeEvent(QResizeEvent * );
public slots:
    void setStyle(int);
    void setWidth(int);
    void setColor(QColor);
    void clear();
};
```

```
private:  
    QPixmap *pix;  
    QPoint startPos;  
    QPoint endPos;  
    int style;  
    int weight;  
    QColor color;  
};
```

DrawWidget 类继承自 QWidget 类，在类声明中对鼠标事件 mousePressEvent()和 mouseMoveEvent()、重画事件 paintEvent()以及尺寸变化事件 resizeEvent()进行了重定义。setStyle()、setWidth()以及 setColor()函数主要用于为主窗口传递各种与绘图有关的参数。

DrawWidget 类的构造函数：

```
DrawWidget::DrawWidget()  
{  
    1    setAutoFillBackground(true);  
    2    setPalette(QPalette(Qt::white));  
  
    3    pix = new QPixmap(size());  
    4    pix->fill(Qt::white);  
  
    5    setMinimumSize(600,400);  
}
```

第 1、2 行完成对窗体背景色的设置，此处调用 setPalette(QPalette(Qt::white))实现，与下面的代码效果一致：

```
QPalette p = palette();  
p.setColor(QPalette::Window,Qt::white);  
setPalette(p);
```

第 3 行为绘图准备一个 QPixmap 对象，并填充为背景色，此 QPixmap 对象用来准备随时接收绘制的内容。

第 5 行设置绘制区窗体的最小尺寸。

setStyle()函数接收主窗体传来的线型风格参数。具体代码如下：

```
void DrawWidget::setStyle(int s)  
{  
    style = s;  
}
```

setWidth()函数接收主窗体传来的线宽参数值。具体代码如下：

```
void DrawWidget::setWidth(int w)
{
    weight = w;
}
```

setColor()函数接收主窗体传来的画笔颜色值。具体代码如下：

```
void DrawWidget::setColor(QColor c)
{
    color = c;
}
```

重定义鼠标按下事件 mousePressEvent(), 在按下鼠标时, 记录当前的鼠标位置值 startPos。具体代码如下：

```
void DrawWidget::mousePressEvent(QMouseEvent * e)
{
    startPos = e->pos();
}
```

重定义鼠标移动事件 mouseMoveEvent(), 鼠标移动事件在默认情况下, 是在当鼠标按下的同时拖动鼠标被触发。QWidget 的 mouseTracking 属性指示窗体是否追踪鼠标, 默认为 false, 不追踪, 即在至少有一个鼠标按键按下的前提下移动鼠标才触发 mouseMoveEvent()事件, 可以通过 setMouseTracking (bool enable)方法对该属性值进行设置, 如果设置为追踪, 则无论是否有鼠标按下, 只要鼠标移动, 都会触发 mouseMoveEvent()事件。在此事件处理函数中, 完成向 QPixmap 对象中画图的工作。具体实现代码如下：

```
void DrawWidget::mouseMoveEvent(QMouseEvent * e)
{
    1   QPainter *painter = new QPainter(pix);
    2   QPen pen;
    3   pen.setStyle((Qt::PenStyle)style);
    4   pen.setWidth(weight);
    5   pen.setColor(color);
    6   painter->setPen(pen);

    7   painter->drawLine(startPos,e->pos());
    8   startPos = e->pos();
    9   update();
}
```

第1行以 QPixmap 对象为 QPainter 参数新建一个 QPainter 对象。

第2行新建一个 QPen 对象。

第3行设置画笔的线型, style 表示的是当前选择的线型是 Qt::PenStyle 枚举数据中的第几个元素。

第4行设置画笔的线宽值。

第5行设置画笔的颜色。

第6行把 QPen 对象应用到绘制对象中。

第7行调用 QPainter 的 drawLine() 函数绘制从 startPos 到鼠标当前位置的直线。

第8行更新 startPos 的值为鼠标的当前位置, 为下次绘制作准备。

第9行调用 update() 函数重画绘制区窗体。

重画函数 paintEvent() 完成绘制区窗体的更新工作, 只需调用 drawPixmap() 函数把用来接收图形绘制的 QPixmap 对象画在绘制区窗体控件上即可。具体代码如下:

```
void DrawWidget::paintEvent(QPaintEvent *)  
{  
    QPainter painter(this);  
    painter.drawPixmap(QPoint(0,0),*pix);  
}
```

当窗体的大小发生改变时, 效果看起来虽然像是绘制区大小改变了, 但实际能够进行绘制的区域仍然没变, 因为绘图的大小并没有改变, 还是原来绘制区窗口的大小, 因此在窗体尺寸变化时应及时调整用于绘制的 QPixmap 对象的尺寸大小。

```
void DrawWidget::resizeEvent(QResizeEvent * event)  
{  
    if(height() > pix->height() || width() > pix->width())  
    {  
        2    QPixmap *newPix = new QPixmap(size());  
        3    newPix->fill(Qt::white);  
        4    QPainter p(newPix);  
        5    p.drawPixmap(QPoint(0,0),*pix);  
        6    pix = newPix;  
    }  
    7    QWidget::resizeEvent(event);  
}
```

第1行判断改变后的窗体长或者宽是否大于原窗体的长和宽, 若大于则进行相应的调整, 否则直接调用 QWidget 的 resizeEvent() 函数返回。

第2行创建一个新的 QPixmap 对象 newPix。



第3行填充此新QPixmap对象newPix的颜色为背景色。
第4、5行在newPix中绘制原pix中的内容。
第6行把newPix赋值给pix作为新的绘制图形接收对象。
第7行调用QWidget类的resizeEvent()函数完成其余的工作。
clear()函数完成绘制区的清除工作，只需用一个新的、干净的QPixmap对象来代替pix，并调用update()重画即可。具体代码如下：

```
void DrawWidget::clear()
{
    QPixmap *clearPix = new QPixmap(size());
    clearPix->fill(Qt::white);
    pix = clearPix;
    update();
}
```

至此，一个能响应鼠标事件进行绘图功能的窗体类DrawWidget已实现，接下来的工作，即在主窗口中应用此窗体类。

主窗口类Painter继承自 QMainWindow类，只包含一个工具栏和一个中央窗体。具体代码如下：

```
class Painter : public QMainWindow
{
    Q_OBJECT
public:
    Painter(QWidget *parent=0);
    void createToolBar();

public slots:
    void slotColor();

private:
    DrawWidget *widget;
    QToolButton *colorBtn;
};
```

Painter类中声明了一个构造函数，一个用于创建工具栏的函数createToolBar()，一个用于进行颜色选择的槽函数slotColor()，并声明一个DrawWidget类对象作为主窗口的私有变量。

Painter类的构造函数：

```
Painter::Painter(QWidget *parent)
    : QMainWindow(parent)
{
    .....
1    widget = new DrawWidget;
2    setCentralWidget(widget);

3    createToolBar();
4    setMinimumSize(600,400);

5    slotStyle();
6    widget->setWidth(widthSpinBox->value());
7    widget->setColor(Qt::black);
}
```

第1、2行新建一个 DrawWidget 对象作为主窗口的中央窗体。

第3行调用 createToolBar() 函数实现一个工具栏。

第4行设置主窗口的最小尺寸。

第5、6、7行为绘制初始化线型、线宽和颜色参数，把各参数设置控件中的当前值作为初始参数。

工具栏的实现函数：

```
void Painter::createToolBar()
{
1    QToolBar *toolBar = addToolBar("Tool");
    //选择线型风格
2    styleComboBox = new QComboBox;
3    styleComboBox->addItem("SolidLine",Qt::SolidLine);
4    styleComboBox->addItem("DashLine",Qt::DashLine);
5    styleComboBox->addItem("DotLine",Qt::DotLine);
6    styleComboBox->addItem("DashDotLine",Qt::DashDotLine);
7    styleComboBox->addItem("DashDotDotLine",Qt::DashDotDotLine);
8    connect(styleComboBox,SIGNAL(activated(int)),this,SLOT(slotStyle()));

    .....
    //选择线宽
9    widthSpinBox = new QSpinBox;
10   connect(widthSpinBox,SIGNAL(valueChanged(int)),
            widget,SLOT(setWidth(int)));
    .....
    //选择颜色
11   colorBtn = new QToolButton;
12   QPixmap pixmap(20,20);
```



```
13 pixmap.fill(Qt::black);
14 colorBtn->setIcon(QIcon(pixmap));
15 connect(colorBtn,SIGNAL(clicked()),this,SLOT(slotColor()));
16 ...
17 //清除按钮
18 QToolButton *clearBtn = new QToolButton;
19 connect(clearBtn,SIGNAL(clicked()),widget,SLOT(clear()));
20 }
```

第 1 行为主窗口新建一个工具栏对象。

第 2~8 行创建线型选择控件，Qt 提供的线型选择有 Qt::SolidLine、Qt::DashLine、Qt::DotLine、Qt::DashDotLine 和 Qt::DashDotDotLine 5 种，如图 5-24 所示。此处采用一个 QComboBox 对象作为线型选择的控件，并关联相应的槽函数改变绘制区的线型参数。

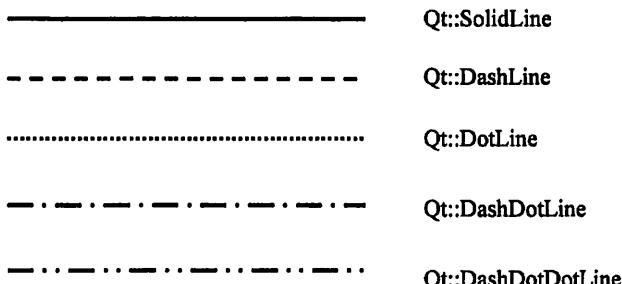


图 5-24 Qt 提供的 5 种线型

第 9、10 行创建线宽选择控件，用一个 QSpinBox 控件实现，并关联相应的槽函数改变绘制区的线宽参数。

第 11~15 行创建颜色选择按钮，采用一个 QToolButton 控件实现，并在此工具按钮的图标上显示所选择的颜色，关联相应的槽函数改变绘制区的画笔颜色。

第 16、17 行创建清除按钮，用一个 QToolButton 实现。

改变线型参数的槽函数，调用 DrawWidget 类的 setStyle() 函数把当前线型选择控件中的线型参数传给绘制区。具体代码如下：

```
void Painter::slotStyle()
{
    widget->setStyle(styleComboBox->itemData(
        styleComboBox->currentIndex(),Qt::UserRole).toInt());
}
```

QComboBox 类的 itemData 方法返回当前显示的下拉列表框数据，是一个 QVariant 对象，此对象与控件初始化时插入的枚举型数据相关，调用 QVariant 类的 toInt() 函数获得此数据在枚举型数据集合中的序号。

设置画笔颜色的槽函数：

```
void Painter::slotColor()
{
    1   QColor color = QColorDialog::getColor(Qt::black, this);
    2   if (color.isValid())
    3       {
    4           widget->setColor(color);
    5           QPixmap p(20,20);
    6           p.fill(color);
    7           colorBtn->setIcon(QIcon(p));
    8       }
}
```

第 1 行使用标准颜色对话框 QColorDialog 获得一个颜色值。

第 3 行把新选择的颜色传给绘制区，以改变画笔的颜色值。

第 4~6 行更新颜色选择按钮上的颜色显示。

实例 37 改变图片的透明度

知识点：

颜色混合算法

本实例实现了一个图片透明度可调的功能，如图 5-25 所示，此功能的实现主要是利用了颜色的混合概念。

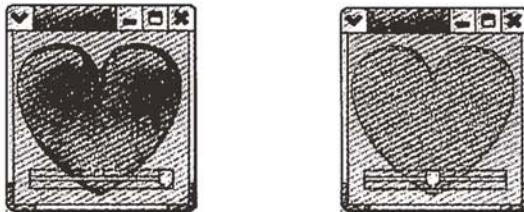


图 5-25 改变图片的透明度



用户可通过调节对话框下部的滑动条，来对上面的心形图片的透明度进行调节，当滑动条处于最左侧时，表示心形图片完全透明，随着滑动条的滑动透明度逐渐增大，到最右边则完全不透明。

在分析此实例之前，需先弄清 RGB 颜色的一些基本概念。顾名思义，RGB 颜色是由红色（R）、绿色（G）和蓝色（B）3 种基本颜色组合而成，所有的颜色都可通过这 3 种基本色的不同比例混合配置而成，但除了 R、G、B 3 种基本颜色之外还有一个重要的颜色属性即透明度（Alpha），利用 3 种基本色和透明度值的混合即可获得各式各样的颜色效果。

下面简单介绍颜色混合的算法。

众所周知，RGB 颜色的每种基本色的取值范围为 0~255 之间，透明度的取值可以用百分比的方式表示，颜色的混合是以像素为单位进行的，每个像素点都包含了 R、G、B 值和 Alpha 值。Alpha 值表示图形的透明度，如某个像素点的 RGB 值为(100,200,100)，Alpha 值为 70% 即 0.7，则这个像素点的颜色应为：

```
int red = 100*0.7 = 70;  
int green = 200*0.7 = 140;  
int blue = 100*0.7 = 70;
```

由此，某个像素点进行颜色混合的算法可表示为以下公式：

```
int red = int( qred(color1.rgb())*(1-alpha) + qred(color2.rgb())*alpha );  
int green = int( qgreen(color1.rgb())*(1-alpha) + qgreen(color2.rgb())*alpha );  
int blue = int( qblue(color1.rgb())*(1-alpha) + qblue(color2.rgb())*alpha );
```

通常，颜色的混合是在两个图片之间进行的，如图 5-26 所示。red、green、blue 分别表示混合后的图片在此像素点的 RGB 颜色值；color1 表示其中一幅图片此像素点的 QColor 颜色；color2 表示另一幅图片此像素点的 QColor 颜色。

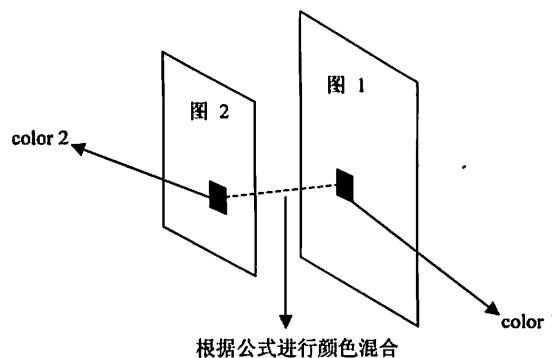


图 5-26 颜色的混合

了解了颜色混合的基本原理与算法之后，下面具体分析本实例的实现过程。

主窗口 PicTrans 继承自 QDialog 类，类中重写了绘制函数 paintEvent(QPaintEvent *); 声明了一个用于实现改变图片透明度的槽函数 slotChanged(int)；声明了两个私有变量 QImage，其中，img 用于保存混合后的图片，dst 用于保存可调透明度的目标图片。具体代码如下：

```
class PicTrans : public QDialog
{
    Q_OBJECT
public:
    PicTrans(QWidget *parent=0);
    void paintEvent(QPaintEvent *);

public slots:
    void slotChanged(int);

private:
    QImage *img;
    QImage dst;
};
```

主窗体的构造函数完成控件的初始化及布局工作。具体代码如下：

```
PicTrans::PicTrans(QWidget *parent)
    : QDialog(parent)
{
    1   QPalette p = palette();
    2   p.setColor(QPalette::Window,QColor(160,250,200));
    3   setPalette(p);

    4   QSlider *slider = new QSlider;
    5   slider->setRange(0,100);
    6   slider->setTickInterval(10);
    7   slider->setOrientation(Qt::Horizontal);
    8   slider->setValue(100);

    9   img = new QImage(":/images/heart.png");

    10  QVBoxLayout *layout = new QVBoxLayout;
    11  layout->addStretch(1);
    12  layout->addWidget(slider);
    13  setLayout(layout);
```

```
14 slotChanged(slider->value());  
15 connect(slider,SIGNAL(valueChanged(int)),this,SLOT(slotChanged(int)));  
}
```

第 1~3 行设置主窗体的背景色。

第4行新建一个滑动条 QSlider 对象，用于调节图形的透明度。

第 5 行设置滑动条对象的滑动范围为 0~100。

第 6 行设置滑动条的步进值为 10，即滑动条的每个移动单位为 10。

第 7 行设置滑动条的方向为水平方向。

第 8 行设置滑动条初始值为 100，对应为完全不透明。

第 9 行创建需在窗体中显示的图片对象 img。

第 10~13 行对窗体进行布局。

第 14 行调用一次 slotChanged() 函数利用颜色混合显示图片 img。

第 15 行连接滑动条对象的 valueChanged(int) 信号与 slotChanged(int) 槽函数。

`slotChanged()`函数是主要完成图片颜色混合的部分，也是本实例实现的关键代码。

```

void PicTrans::slotChanged(int value)
{
    dst = *img;
    QColor bkColor = palette().color(QPalette::Window);
    float v = 0.01 * value;

    int width = img->width();
    int height = img->height();
    for (int h=0;h<height;h++)
    {
        for (int w=0;w<width;w++)
        {
            int alpha = qAlpha(img->pixel(w,h));
            if (alpha == 0) //完全透明
            {
                int red = qRed(bkColor.rgb());
                int green = qGreen(bkColor.rgb());
                int blue = qBlue(bkColor.rgb());
                dst.setPixel(w,h,qRgb(red,green,blue));
            }
            else
            {

```

```
15         int red = (int)((qRed(bkColor.rgb())*(1-v)+  
16                     (qRed(img->pixel(w,h)))*v);  
17         int green = (int)((qGreen(bkColor.rgb())*(1-v)+  
18                     (qGreen(img->pixel(w,h)))*v);  
19         int blue = (int)((qBlue(bkColor.rgb())*(1-v)+  
20                     (qBlue(img->pixel(w,h)))*v);  
21         dst.setPixel(w,h,qRgb(red,green,blue));  
22     }  
23 }  
24  
25  
26 19 resize(dst.width(),dst.height());  
27 20 repaint();  
28 }
```

第 1 行为用于保存混合后图片的 QImage 对象 dst 赋初值，让它等于用于调节透明度的图片。

第 2 行获得窗体的背景色，本实例将背景色作为进行颜色混合的其中一个图片。

第 3 行把传入的滑动条的数值赋给变量 v，转换成调节的当前透明度值，滑动条的值是一个 0~100 之间的整数，则需乘以 0.01 进行转换。

第 4、5 行获得需要进行颜色混合的图片的长、宽值，由于颜色混合是以像素点为单位逐点进行，因此以长、宽值进行循环逐点进行操作。

第 8~18 行即完成某个像素点的颜色混合工作。

第 8、9 行获得心形图片当前像素点的透明度值，若此透明度值为 0，则说明此点是完全透明的，则在此像素点只要直接显示背景色即可，不用进行颜色的混合工作。

第 10~13 行直接用背景色对用于保存混合后图片的此像素点进行赋值即可。若心形图片在此像素点的透明度值不为 0 则需进行颜色的混合，第 15~18 行完成此部分工作。

第 15 行利用前面介绍过的颜色混合公式，获得混合后的 R 值 red。

第 16 行利用前面介绍过的颜色混合公式，获得混合后的 G 值 green。

第 17 行利用前面介绍过的颜色混合公式，获得混合后的 B 值 blue。

第 18 行调用 QImage 的 setPixel() 函数设置某一点的颜色，用混合后的 RGB 值对保存混合后图片的此像素点进行赋值。

第 19 行重设 dst 图片的尺寸为心形图片的大小。.

第 20 行调用 repaint() 重画窗口，把混合后的图片 dst 重画到窗口中。

窗口的绘制函数，主要完成把保存颜色混合后的图片重画至窗口的工作。具体代码如下：

```
void PicTrans::paintEvent(QPaintEvent * e)
{
    QPainter painter(this);
    painter.drawImage(0,0,dst);
}
```

实例 38 橡皮筋线

知识点：

QRubberBand 的使用方法

在图形编辑应用中常会需要用到橡皮筋线，如选择图形的某个区域等，最常见的就是在系统桌面上用鼠标拖动，可以绘制一个类似蚂蚁线的选区，并且选区线能够跟随鼠标的移动而伸缩，因此叫作橡皮筋线。

本实例即实现一个橡皮筋线的例子，如图 5-27 所示。

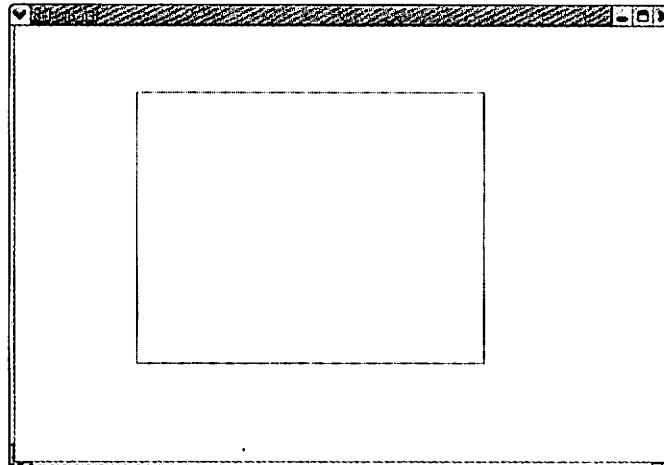


图 5-27 橡皮筋线

Qt 为橡皮筋线专门提供了一个 QRubberBand 类。橡皮筋线实现的关键是用 QRubberBand 类提供的函数与鼠标事件的响应函数相结合，在按下鼠标时新建 QRubberBand 对象，在移动鼠标时利用 QRubberBand 的 setGeometry() 函数绘制蚂蚁线，在松开鼠标时隐藏 QRubberBand 对象。

主窗口 RubberBand 类继承自 QMainWindow，在类中重写了 3 个鼠标响应函数，声明了一个 QRubberBand 对象作为私有变量。具体代码如下：

```
class RubberBand : public QMainWindow
{
    Q_OBJECT
public:
    RubberBand(QWidget *parent=0);

    void mousePressEvent(QMouseEvent * );
    void mouseMoveEvent(QMouseEvent * );
    void mouseReleaseEvent(QMouseEvent * );

private:
    QRubberBand *rubberBand;
    QPoint origin;
};
```

构造函数完成对 RubberBand 中央窗体背景色的设置以及大小尺寸的设置工作。具体代码如下：

```
RubberBand::RubberBand(QWidget *parent)
    : QMainWindow(parent)
{
    QWidget *mainWidget = new QWidget;
    mainWidget->setBackgroundRole(QPalette::Light);
    mainWidget->setAutoFillBackground(true);
    setCentralWidget(mainWidget);

    resize(600,400);
    setWindowTitle(tr("Rubberband"));
    rubberBand = NULL;
}
```

实现橡皮筋线的关键在于各个鼠标事件响应函数。

鼠标按下事件响应函数主要完成 QRubberBand 对象的创建工作。具体代码如下：

```
void RubberBand::mousePressEvent(QMouseEvent * e)
{
1    origin = e->pos();
2    if (!rubberBand)
3        rubberBand = new QRubberBand(QRubberBand::Rectangle, this);
4    rubberBand->setGeometry(QRect(origin, QSize()));
```

```

5     rubberBand->show();
}

```

第 1 行记录鼠标按下时的位置信息 origin。

第 2、3 行对变量 rubberBand 进行判断，若未创建 QRubberBand 对象，则新建一个 QRubberBand 对象，构建 QRubberBand 对象的第一个参数描述了橡皮筋线的类型，可有两种选择，若为 QRubberBand::Rectangle，则为一个描绘了四周边线的方形区域，若设为 QRubberBand::Line，则为一个由直线填满的方形区域；第二个参数为橡皮筋线的父窗口指针。

第 4 行调用 setGeometry() 函数设置橡皮筋线的位置和尺寸大小。

鼠标移动事件响应函数主要完成随鼠标移动伸缩橡皮筋线的工作。具体代码如下：

```

void RubberBand::mouseMoveEvent(QMouseEvent * e)
{
    if (rubberBand)
        rubberBand->setGeometry(QRect(origin, e->pos()).normalized());
}

```

第 2 行调用 QRubberBand 的 setGeometry() 函数调整橡皮筋线的大小，以按下鼠标时保存的点和当前鼠标点组成一个 QRect 对象作为参数描述橡皮筋线选框的大小。

 小贴士：QRect 的 normalized() 函数返回值也是一个 QRect 对象，确保返回的 QRect 对象的 width() 值和 height() 值都是大于零的。

因为在鼠标拖动时可能向两个相反的方向拖动，此函数会自动交换顶点以确保长、宽值大于零。

鼠标松开事件响应函数主要完成橡皮筋线隐藏的工作。在拖动鼠标结束，划定一个区域后松开鼠标，此时橡皮筋线也应随之消失。具体代码如下：

```

void RubberBand::mouseReleaseEvent(QMouseEvent * e)
{
    if (rubberBand)
        rubberBand->hide();
}

```

第 6 章 Graphics View

在 Qt4 版本中，废弃了之前版本中的 QCanvas 系列类，而是用 Graphics View 框架结构的类替代。Graphics View 框架结构主要包含了 3 个主要的类，分别是 QGraphicsScene、QGraphicsView、QGraphicsItem。QGraphicsScene 场景类提供了一个用于管理位于其中的众多项目 QGraphicsItem 容器，QGraphicsView 视口类用于显示场景中的项目，它们三者之间的关系可用图 6-1 表示。

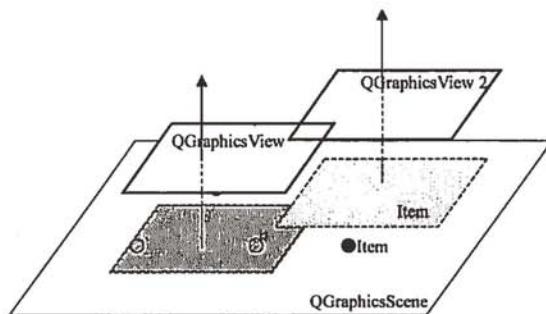


图 6-1 Graphics View 三元素之间的关系

QGraphicsScene 类是一个用于放置项目的容器，它本身是不可见的，必须通过与之相连的 **QGraphicsView** 视口类来显示及与外界进行互操作。**QGraphicsScene** 主要完成的工作包括提供对它所包含的项目的操作接口，为它所包含的项目传递事件，管理各个项目的状态等。

QGraphicsView 类提供一个可视的窗口，用于显示场景中的项目，在同一个场景中可以有多个视口。

QGraphicsItem 类是场景中各个项目的基础类，在它的基础上可以继承出各种项目类，Qt 已经预置的有如 **QGraphicsLineItem**、**QGraphicsEllipseItem**、**QGraphicsRectItem** 等，当然也可以在 **QGraphicsItem** 类的基础上实现自定义的项目类。

在弄清楚以上基本概念后，要进行 Graphics View 的开发，还有一个重要的概念需要弄清楚，即坐标系的问题，3 个 Graphics View 基本类有各自不同的坐标系。

QGraphicsScene 类的坐标系是以中心为原点(0,0)，如图 6-2 所示。

QGraphicsView 类继承自 QWidget 类，因此它和其他的 QWidget 类一样以窗口的左上角作为自己坐标系的原点，如图 6-3 所示。

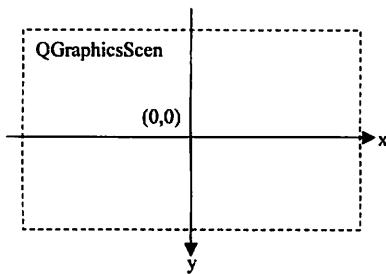


图 6-2 QGraphicsScene 的坐标系

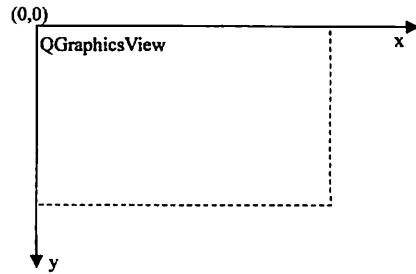
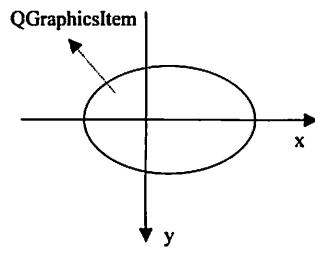
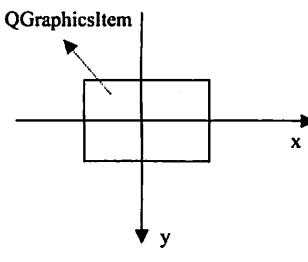


图 6-3 QGraphicsView 的坐标系

而 QGraphicsItem 类则有自己的坐标系，在调用 QGraphicsItem 类的 paint() 函数重画项目时则是以此坐标系为基准，如图 6-4 所示。



(a)



(b)

图 6-4 QGraphicsItem 的坐标系

根据需要，Qt 提供了这 3 个坐标系之间的相互转换函数，以及项目与项目之间的转换函数，如需要从 QGraphicsItem 的坐标系中的某一点坐标转换到在场景中的坐标，则可调用 QGraphicsItem 的 mapToScene() 函数进行映射。而 QGraphicsItem 的 mapToParent() 函数则可把 QGraphicsItem 坐标系中的某点坐标映射至它的上一级坐标系中，有可能是场景坐标，也有可能是另一个 QGraphicsItem 坐标。

本章包括 4 个实例：

- ❑ 地图浏览器
- ❑ 各种 Graphics Item
- ❑ Graphics Item 的各种变形
- ❑ 飞舞的蝴蝶

实例 39 地图浏览器

知识点：

- 利用 Graphics View 显示地图
- QGraphicsView 和 QGraphicsScene 之间的坐标转换
- QGraphicsScene 坐标与地图坐标的转换
- 利用 QGraphicsView 的 scale() 实现地图的缩放

本实例实现一个地图浏览器的基本功能，包括地图的浏览、放大、缩小，以及显示各点的坐标等，如图 6-5 所示。

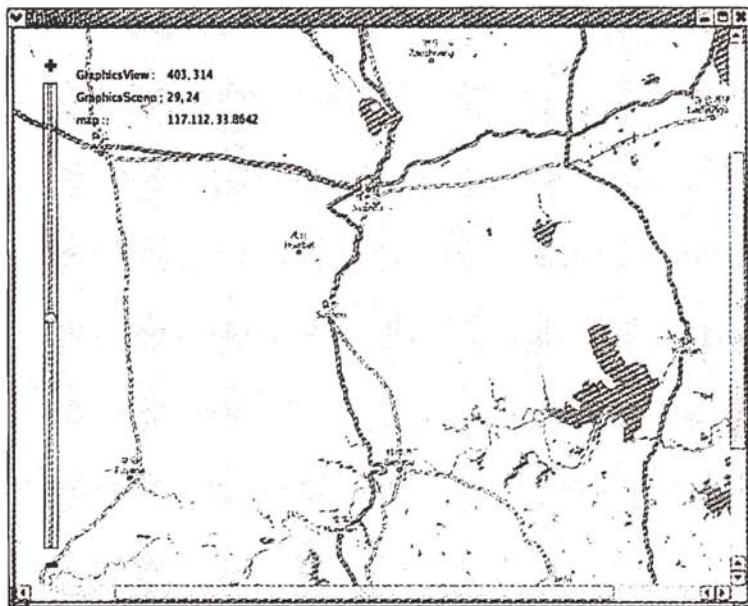


图 6-5 地图浏览器

本实例主要是利用了 QGraphicsView 的 drawBackground() 函数重画场景的背景显示地图，利用 scale() 函数实现地图的缩放，利用各坐标系之间的转换函数实现某点在各层中的坐标及地图经、纬度的显示。

具体实现代码如下：

```
class MapWidget : public QGraphicsView
{
    Q_OBJECT
public:
    1   MapWidget();
    2   void readMap();
    3   QPointF mapToMap(QPointF);
public slots:
    4   void slotZoom(int);

protected:
    5   void drawBackground(QPainter *painter, const QRectF &rect);
    6   void mouseMoveEvent(QMouseEvent *);

private:
    7   QPixmap map;
    8   qreal zoom;
    9   QLabel *viewCoord;
    10  QLabel *sceneCoord;
    11  QLabel *mapCoord;

    12  double x1,y1; // map lefttop lon&lat
    13  double x2,y2; // map rightbottom lon&lat
};
```

MapWidget 作为地图浏览器的主窗体，由于本实例只是实现地图浏览器的一些基本功能，因此不实现类似 QMainWindow 的菜单或工具栏等，只继承自 QGraphicsView。在实际应用时，通常应继承 QMainWindow 来实现。本实例重点是为了分析 Graphics View 的应用，把其他功能的实现隐去。

在头文件的类声明中，第 2 行的 readMap() 函数用于读取描述地图信息的文件，此文件中描述了所要显示的地图名，以及经、纬度等信息。

第 3 行的 mapToMap() 函数用于实现场景坐标系与地图坐标之间的映射，以获得某点的经、纬度值。

第 5 行的 drawBackground() 函数完成地图显示的功能。

主窗口的构造函数：

```
MapWidget::MapWidget()
{
    1   readMap(); //读取地图信息
```

```
2     zoom = 50;

3     int width = map.width();
4     int height = map.height();
5     QGraphicsScene *scene = new QGraphicsScene(this);
6     scene->setSceneRect(-width/2,-height/2,width,height);
7     setScene(scene);
8     setCacheMode(CacheBackground);

//用于地图缩放的滑动条
9     QSlider *slider = new QSlider;
10    slider->setOrientation(Qt::Vertical);
11    slider->setRange(1,100);
12    slider->setTickInterval(10);
13    slider->setValue(50);
14    connect(slider,SIGNAL(valueChanged(int)),this,SLOT(slotZoom(int)));

15    QLabel *zoominLabel = new QLabel;
16    zoominLabel->setScaledContents(true);
17    zoominLabel->setPixmap(QPixmap("./images/zoomin.png"));

18    QLabel *zoomoutLabel = new QLabel;
19    zoomoutLabel->setScaledContents(true);
20    zoomoutLabel->setPixmap(QPixmap("./images/zoomout.png"));

//坐标值显示区
21    QFrame *coordFrame = new QFrame;
22    QLabel *label1 = new QLabel(tr("GraphicsView :"));
23    viewCoord = new QLabel;
24    QLabel *label2 = new QLabel(tr("GraphicsScene :"));
25    sceneCoord = new QLabel;
26    QLabel *label3 = new QLabel(tr("map :"));
27    mapCoord = new QLabel;
28    QGridLayout *grid = new QGridLayout;           //坐标显示区布局
29    grid->addWidget(label1,0,0);
30    grid->addWidget(viewCoord,0,1);
31    grid->addWidget(label2,1,0);
32    grid->addWidget(sceneCoord,1,1);
33    grid->addWidget(label3,2,0);
34    grid->addWidget(mapCoord,2,1);
35    grid->setSizeConstraint(QLayout::SetFixedSize);
36    coordFrame->setLayout(grid);
```



```
//缩放控制子布局  
37 QVBoxLayout *zoomLayout = new QVBoxLayout;  
38 zoomLayout->addWidget(zoominLabel);  
39 zoomLayout->addWidget(slider);  
40 zoomLayout->addWidget(zoomoutLabel);  
  
//缩放控制子布局  
41 QVBoxLayout *coordLayout = new QVBoxLayout;  
42 coordLayout->addWidget(coordFrame);  
43 coordLayout->addStretch();  
  
//主布局  
44 QHBoxLayout *layout = new QHBoxLayout;  
45 layout->addLayout(zoomLayout);  
46 layout->addLayout(coordLayout);  
47 layout->addStretch();  
48 layout->setMargin(30);  
49 layout->setSpacing(10);  
50 setLayout(layout);  
  
51 setWindowTitle("Map Widget");  
52 setMinimumSize(600,400);  
}
```

第 1 行调用 `readMap()` 函数读取地图的相关信息。

第 2 行初始化缩放的比例。

第 3~8 行新建一个 `QGraphicsScene` 对象为主窗口连接一个场景，并限定场景的显示区域为地图的大小。

第 9~14 行新建一个 `QSlider` 对象作为地图的缩放控制，设置地图缩放比例值范围为 0~100，前面初始值 50 表示正常比例显示，并把缩放控制条的 `valueChanged()` 信号与地图缩放槽函数相连。

第 15~20 行显示缩放滑动条两端的放大、缩小图标。

第 21~36 行完成用于各层坐标显示的 `QFrame`。

第 37~50 行完成布局工作。

`readMap()` 函数的代码如下：

```
void MapWidget::readMap()           //读取地图信息函数  
{  
1   QFile mapFile("maps.txt");  
2   QString mapName;
```

```
3 int ok = mapFile.open(QIODevice::ReadOnly);
4 if(ok)
{
5     QTextStream t(&mapFile);
6     if(!t.atEnd())
    {
7         t >> mapName;
8         t >> x1 >> y1 >> x2 >> y2;
    }
}
9 map.load(mapName);
}
```

本实例中利用一个文本文件来描述与地图相关的信息，文件内容为：

```
map.png 114.4665527 35.96022297 119.9597168 31.3911575
```

依次包含地图的名称，地图左上角的经、纬度值，地图右下角的经、纬度值。

第1行以maps.txt（描述地图信息的文本文件）为参数新建一个QFile对象。

第3行以“只读”方式打开此文件。

第4~8行分别读取地图的名称和4个经、纬度信息。

第9行把地图读取至私有变量map中。

slotZoom()函数完成地图的缩放功能。根据缩放滑动条的当前值，确定缩放的比例，调用QGraphicsView的scale()函数实现地图缩放。具体代码如下：

```
void MapWidget::slotZoom(int value)           //地图缩放函数
{
    qreal s;
    if(value>zoom) // zoom in
    {
        s = pow(1.01,(value-zoom));
    }
    else      // zoom out
    {
        s = pow((1/1.01),(zoom-value));
    }

    scale(s,s);
    zoom = value;
}
```

QGraphicsView 和 QGraphicsScene 都有 drawBackground()虚函数，它们做的工作都是



重画场景的背景，因此需要以整幅图片设置场景的背景时，可以实现 `QGraphicsScene` 的 `drawBackground()` 函数，也可实现 `QGraphicsView` 的 `drawBackground()` 函数。本实例中即实现了 `QGraphicsView` 类的 `drawBackground()` 函数，以地图图片重画场景的背景实现地图的显示。具体代码如下：

```
void MapWidget::drawBackground(QPainter *painter, const QRectF &rect)
{
    painter->drawPixmap(int(sceneRect().left()),int(sceneRect().top()),map);
}
```

`mouseMoveEvent()` 函数响应鼠标移动事件，在此函数中完成某点在各层坐标中的映射及显示。具体代码如下：

```
void MapWidget::mouseMoveEvent(QMouseEvent * event)
{
    //QGraphicsView 坐标
    1   QPoint viewPoint = event->pos();
    2   viewCoord->setText(QString::number(viewPoint.x()) + "," +
                           QString::number(viewPoint.y()));

    //QGraphicsScene 坐标
    3   QPointF scenePoint = mapToScene(viewPoint);
    4   sceneCoord->setText(QString::number(scenePoint.x()) + "," +
                           QString::number(scenePoint.y()));

    //地图坐标（经、纬度）
    5   QPointF latLon = mapToMap(scenePoint);
    6   mapCoord->setText(QString::number(latLon.x()) + "," +
                           QString::number(latLon.y()));
}
```

第 1、2 行显示鼠标所在点在视口中的坐标。

第 3、4 行显示鼠标所在点在场景中的坐标。

第 5、6 行显示鼠标所在点在地图中的经、纬度值，调用 `mapToMap()` 函数实现从场景坐标到地图坐标的转换。

`mapToMap()` 函数完成从场景坐标至地图坐标的转换。具体代码如下：

```
QPointF MapWidget::mapToMap(QPointF p)           //由场景坐标映射至地图坐标
{
    QPointF latLon;
    qreal w = sceneRect().width();
    qreal h = sceneRect().height();
```

```
qreal lon = y1 - ((h/2 + p.y())*abs(y1-y2)/h);
qreal lat = x1 + ((w/2 + p.x())*abs(x1-x2)/w);
latLon.setX(lat);
latLon.setY(lon);
return latLon;
}
```

实例 40 各种 Graphics Item

知识点：

- 标准 QGraphicsItem 的使用
- 自定义 QGraphicsItem 的实现
- 利用 QTimer 实现动画 QGraphicsItem
- 利用 QGraphicsItemAnimation 和 QTimeLine 类实现动画 QGraphicsItem

本实例实现一个窗体，在其中显示各种类型的 QGraphicsItem，包括 Qt 预定义的，如 QGraphicsEllipseItem、QGraphicsRectItem 等；也有自定义的项目，如不停闪烁的圆、透明的蝴蝶图片以及来回移动的星星，如图 6-6 所示。

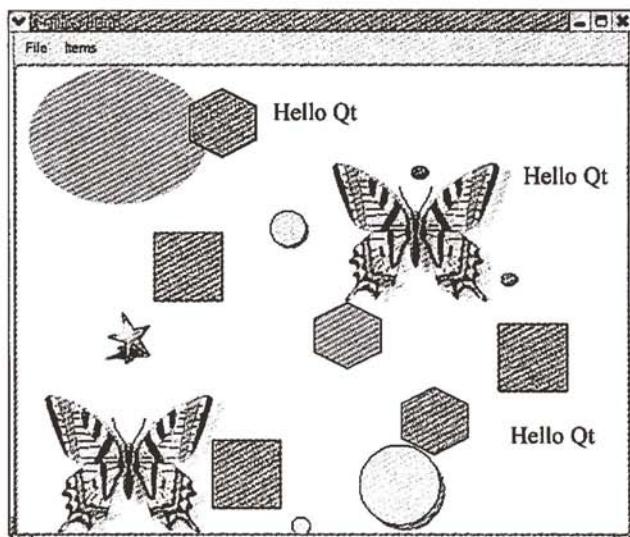


图 6-6 各种 Graphics Item



整个实例以一个 QMainWindow 作为主窗体，包含一个菜单栏，进行加入各种显示项目的操作，以一个 QGraphicsView 作为主窗体的 centralWidget，显示各种类型的项目。

具体实现如下：

在进行主窗体显示前，先实现自定义的 QGraphicsItem 类，分别是闪烁的圆和星星。

```
class FlashItem : public QGraphicsItem, QObject           //闪烁的项目
{
public:
    FlashItem();                                         //第1行

    QRectF boundingRect() const;                         //第2行
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget);
    void timerEvent(QTimerEvent *);                      //第4行

private:
    bool flash;                                         //第5行
    QTimer *timer;                                       //第6行
};
```

闪烁项目类继承自 QGraphicsItem 类和 QObject 类。闪烁效果是利用定时器，定时地重画圆的颜色实现的。

第 1 行声明闪烁项目类的构造函数。

第 2 行声明了 boundingRect() 函数，所有继承自 QGraphicsItem 的自定义项目都必须实现此函数。

第 3 行声明了重画函数，填充圆中的颜色。

第 4 行声明了定时器函数。

第 5 行声明了布尔型变量 flash 作为颜色切换的标识。

小贴士：Butterfly 类需要继承 QObject 类，因为在 Butterfly 类中需要用到定时器，而定时器是继承自 QObject 类的。

闪烁项目的构造函数：

```
FlashItem::FlashItem()
{
    flash = true;                                     //第1行
    setFlag(ItemIsMoveable);                          //第2行
    startTimer(50);                                   //第3行
}
```

第 1 行初始化颜色切换标识。

第2行调用 `setFlag()` 函数设置项目的属性，`ItemIsMoveable` 表示此项目是可移动的，即可用鼠标进行拖动操作。

第3行启动一个定时器，以50毫秒为时间间隔。

`boundingRect()` 函数返回自定义项目的边界，以项目坐标系为基础，增加两个像素点的冗余。具体代码如下：

```
QRectF FlashItem::boundingRect() const
{
    qreal adjust = 2;
    return QRectF(-10 - adjust, -10 - adjust,
                  43 + adjust, 43 + adjust);
}
```

`paint()` 函数为自定义项目的绘制函数。具体代码如下：

```
void FlashItem::paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget)
{
    1   painter->setPen(Qt::NoPen);
    2   painter->setBrush(Qt::darkGray);
    3   painter->drawEllipse(-7,-7,40,40);

    4   painter->setPen(QPen(Qt::black,0));
    5   painter->setBrush(flash ? (Qt::red):(Qt::yellow));
    6   painter->drawEllipse(-10,-10,40,40);
}
```

第1~3行绘制闪烁项目的阴影。

第1行设置 `Qt::NoPen` 表示阴影区不绘制边线。

第2行设置画刷的颜色，即阴影的颜色为 `Qt::darkGray`，深灰色；调用 `drawEllipse()` 函数绘制阴影区。

第4~6行绘制闪烁区的椭圆。

第4行设置边线的颜色为黑色，线宽为0。

第5行设置画刷的颜色，此处画刷的颜色根据颜色切换标识 `flash` 决定在椭圆中填充何种颜色，在红色与黄色间选择。

第6行绘制与阴影区同样形状和大小的椭圆，并错开一定的距离实现立体的感觉。

`timerEvent()` 为定时器响应函数，完成颜色切换标识的反置，并在每次反置后调用 `update()` 函数，重画项目以实现闪烁的效果。具体代码如下：

```
void FlashItem::timerEvent(QTimerEvent *)
{
```



```
    flash = !flash;
    update();
}
```

星形项目类 StarItem，实际上是一个图片项目，可以用 `QGraphicsPixmapItem` 类来实现，如蝴蝶项目，也可继承 `QGraphicsItem` 新建一个项目类来实现，这样的实现方式为项目的实现提供更大的灵活性。具体代码如下：

```
class StarItem : public QGraphicsItem //星形项目
{
public:
    StarItem(); //构造函数

    QRectF boundingRect() const;
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget);

private:
    QPixmap pix;
};
```

StarItem 的构造函数，只完成读取图片信息的工作。具体代码如下：

```
StarItem::StarItem()
{
    pix.load(":/images/image.png");
}
```

boundingRect()所有自定义项目都必须实现的函数。具体代码如下：

```
QRectF StarItem::boundingRect() const
{
    return QRectF(-pix.width()/2,-pix.height()/2,pix.width(),pix.height());
}
```

星形项目来回移动的效果，是在显示时利用 `QGraphicsItemAnimation` 和 `QTimeLine` 来实现，这将在后面的函数分析中介绍。代码如下：

```
void StarItem::paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget)
{
    painter->drawPixmap(boundingRect().topLeft(),pix);
}
```

主窗口类继承自 `QMainWindow`。具体代码如下：

```
class MainWindow : public QMainWindow //显示主窗口
```

```

{
    Q_OBJECT
public:
    MainWindow(QWidget *parent=0);

    void initScene();
    void createActions();
    void createMenus();
public slots:
    .....
/* 各动作对应的槽函数 */
private:
    .....
};

```

主窗体的构造函数：

```

MainWindow::MainWindow(QWidget *parent)           //主窗体构造函数
    : QMainWindow(parent)
{
    1   createActions();
    2   createMenus();

    3   scene = new QGraphicsScene;
    4   scene->setSceneRect(-200,-200,400,400);

    5   initScene();

    6   QGraphicsView *view = new QGraphicsView;
    7   view->setScene(scene);
    8   view->setMinimumSize(400,400);
    9   view->show();

    10  setCentralWidget(view);
    11  resize(550,450);
    12  setWindowTitle(tr("Graphics Items"));
}

```

第1行调用 createActions()函数创建主窗体的所有动作。

第2行调用 createMenus()函数创建主窗体的菜单栏。

```

void MainWindow::createActions()           //创建主窗体的动作
{

```



```
// “新建” 动作  
newAct = new QAction(tr("New"),this);  
// “清除” 动作  
clearAct = new QAction(tr("Clear"),this);  
// “退出” 动作  
exitAct = new QAction(tr("Exit"),this);  
// “增加一个椭圆项目” 动作  
addEllipseItemAct = new QAction(tr("Add Ellipse"),this);  
// “增加一个多边形” 动作  
addPolygonItemAct = new QAction(tr("Add Polygon"),this);  
// “增加一个文字项目” 动作  
addTextItemAct = new QAction(tr("Add Text"),this);  
// “增加一个闪烁项目” 动作  
addFlashItemAct = new QAction(tr("Add Flash"),this);  
// “增加一个方形项目” 动作  
addRectItemAct = new QAction(tr("Add Rectangle"),this);  
// “增加一个来回移动的星星” 动作  
addAnimItemAct = new QAction(tr("Add Animation"),this);  
// “增加一个蝴蝶图片” 动作  
addAlphaItemAct = new QAction(tr("Add Alpha-png"),this);  
.....  
}  
  
void MainWindow::createMenus() //创建主窗体的菜单栏  
{  
    QMenu * fileMenu = menuBar()->addMenu(tr("File"));  
    fileMenu->addAction(newAct);  
    fileMenu->addAction(clearAct);  
    fileMenu->addSeparator();  
    fileMenu->addAction(exitAct);  
  
    QMenu * itemsMenu = menuBar()->addMenu(tr("Items"));  
    itemsMenu->addAction(addEllipseItemAct);  
    itemsMenu->addAction(addRectItemAct);  
    itemsMenu->addAction(addPolygonItemAct);  
    itemsMenu->addAction(addTextItemAct);  
    itemsMenu->addAction(addFlashItemAct);  
    itemsMenu->addAction(addAlphaItemAct);  
    itemsMenu->addAction(addAnimItemAct);  
}
```

```
void MainWindow::initScene()           //初始化场景
{
    int i;
    for (i=0; i<3; i++)
        slotAddFlashItem();
    for(i=0;i<3;i++)
        slotAddEllipseItem();
    for(i=0;i<3;i++)
        slotAddRectItem();
    for(i=0;i<2;i++)
        slotAddAlphaItem();
    for(i=0;i<3;i++)
        slotAddPolygonItem();
    for(i=0;i<3;i++)
        slotAddTextItem();
}

void MainWindow::slotNew()           //新建一个显示窗体
{
    slotClear();
    initScene();
    MainWindow *newWin = new MainWindow;
    newWin->show();
}

void MainWindow::slotClear()          //清除场景中所有的项目
{
    QList<QGraphicsItem*> listItem = scene->items();

    while (!listItem.empty())
    {
        scene->removeItem(listItem.at(0));
        listItem.removeAt(0);
    }
}

void MainWindow::slotAddEllipseItem() //在场景中加入一个椭圆形项目
{
    QGraphicsEllipseItem *item = new QGraphicsEllipseItem(QRectF(0,0,80,60));
    item->setPen(Qt::NoPen);
    item->setBrush(QColor(qrand()%256,qrand()%256,qrand()%256));
    qreal scale = ((qrand()%10)+1)/5.0;
    item->scale(scale,scale);
```



```
item->setFlag(QGraphicsItem::ItemIsMovable);
scene->addItem(item);
item->setPos((qrand()%int(scene->sceneRect().width())-200,
               (qrand()%int(scene->sceneRect().height())-200));
}

void MainWindow::slotAddPolygonItem()           //在场景中加入一个多边形项目
{
    QVector<QPoint> v;
    v << QPoint(30,-15) << QPoint(0,-30) << QPoint(-30,-15) << QPoint(-30,15) << QPoint(0,30) <<
    QPoint(30,15);
    QGraphicsPolygonItem *item = new QGraphicsPolygonItem(QPolygonF(v));
    item->setBrush(QColor(qrand()%256,qrand()%256,qrand()%256));
    item->setFlag(QGraphicsItem::ItemIsMovable);
    scene->addItem(item);
    item->setPos((qrand()%int(scene->sceneRect().width())-200,
                  (qrand()%int(scene->sceneRect().height())-200));
}

void MainWindow::slotAddRectItem()           //在场景中加入一个长方形项目
{
    QGraphicsRectItem *item = new QGraphicsRectItem(QRectF(0,0,60,60));
    QPen pen;
    pen.setWidth(3);
    pen.setColor(QColor(qrand()%256,qrand()%256,qrand()%256));
    item->setPen(pen);
    item->setBrush(QColor(qrand()%256,qrand()%256,qrand()%256));
    item->setFlag(QGraphicsItem::ItemIsMovable);
    scene->addItem(item);
    item->setPos((qrand()%int(scene->sceneRect().width())-200,
                  (qrand()%int(scene->sceneRect().height())-200));
}

void MainWindow::slotAddTextItem()           //在场景中加入一个文字项目
{
    QFont font("Times",16);
    QGraphicsTextItem *item = new QGraphicsTextItem("Hello Qt");
    item->setFont(font);
    item->setFlag(QGraphicsItem::ItemIsMovable);
    item->setDefaultTextColor(QColor(qrand()%256,qrand()%256,qrand()%256));
    scene->addItem(item);
    item->setPos((qrand()%int(scene->sceneRect().width())-200,
                  (qrand()%int(scene->sceneRect().height())-200));
```

```
}

void MainWindow::slotAddFlashItem() //在场景中加入一个闪烁项目
{
    FlashItem *item = new FlashItem;
    qreal scale = ((qrand()%10)+1)/5.0;
    item->scale(scale,scale);
    scene->addItem(item);
    item->setPos((qrand()%int(scene->sceneRect().width())-200,
                   (qrand()%int(scene->sceneRect().height())-200));
}

void MainWindow::slotAddAlphaItem() //在场景中加入一个透明蝴蝶图片
{
    QGraphicsPixmapItem *item =
        scene->addPixmap(QPixmap("./images/butterfly"));
    item->setFlag(QGraphicsItem::ItemIsMovable);
    item->setPos((qrand()%int(scene->sceneRect().width())-200,
                   (qrand()%int(scene->sceneRect().height())-200));
}

void MainWindow::slotAddAnimationItem() //在场景中加入一个动画星星
{
    StarItem *item = new StarItem;
    QGraphicsItemAnimation *anim = new QGraphicsItemAnimation;
    anim->setItem(item);
    QTimeLine *timeLine = new QTimeLine(4000);
    timeLine->setCurveShape(QTimeLine::SineCurve);
    timeLine->setLoopCount(0);
    anim->setTimeLine(timeLine);

    int y = (qrand()%400) - 200;
    for (int i=0; i<400; i++)
    {
        anim->setPosAt(i/400.0, QPointF(i-200,y));
    }
    timeLine->start();
    scene->addItem(item);
}
```

项目的动画显示有两种方式可以实现：一种是利用 `QGraphicsItemAnimation` 类和 `QTimeLine` 类来实现；另一种是在项目类中利用定时器 `QTimer` 和项目的重画函数 `paint()`

来实现。本实例中来回移动的星星采用的是第一种方式，而闪烁的圆和后面的“飞舞的蝴蝶”实例中采用的是第二种方式。

本实例中，包括了 Graphics View 中基本所有类型的应用，有标准的 QGraphicsItem 类型，也有自定义的 QGraphicsItem 类型，通过本实例的分析，对于如何自定义一个 QGraphicsItem 应有一个基本的了解。

实例 41 Graphics Item 的各种变形

知识点：

- 自定义 QGraphicsItem
- QGraphicsItem 各种变形函数的用法

本实例实现一个对 QGraphicsItem 类实现各种变形操作的例子，包括旋转、缩放、切变以及位移，如图 6-7 所示。

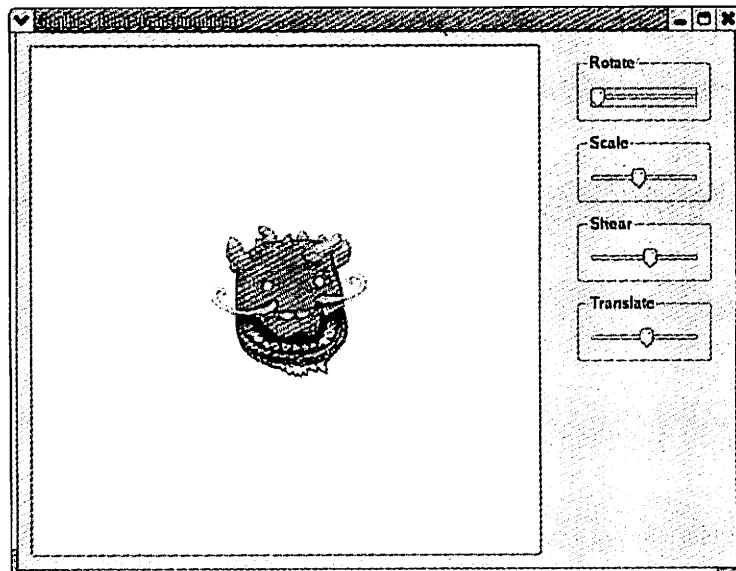


图 6-7 Graphics Item 的各种变形

QGraphicsItem 类为开发者提供了许多变形的函数，可以很方便地通过这些函数对项目进行变形。

具体实现如下：

首先实现一个自定义的 QGraphicsItem 类作为进行变形的对象。

```
class PixItem : public QGraphicsItem
{
public:
    PixItem(QPixmap * pixmap);

    QRectF boundingRect() const;
    void paint(QPainter * painter, const QStyleOptionGraphicsItem * option, QWidget * widget);

private:
    QPixmap pix;
};
```

实现一个自定义的 PixItem 类，继承自 QGraphicsItem 类，并声明 boundingRect() 函数及 paint() 函数，这也是实现一个自定义类所要实现的最基本的内容。声明一个 QPixmap 变量作为项目显示的图片。

类的构造函数，以一个 QPixmap 对象作为传入参数，并把它赋值给自己的私有变量 pix。具体代码如下：

```
PixItem::PixItem(QPixmap * pixmap)
{
    pix = *pixmap;
}
```

boundingRect() 函数限定自定义项目的边界，此时以图片的大小加上两个像素点的冗余作为项目的边界。具体代码如下：

```
QRectF PixItem::boundingRect() const
{
    return QRectF(-2-pix.width()/2,-2-pix.height()/2,pix.width()+4,pix.height()+4);
}
```

重画函数只需调用 QPainter 的 drawPixmap() 函数把项目的图片画出即可。具体代码如下：

```
void PixItem::paint(QPainter * painter, const QStyleOptionGraphicsItem * option, QWidget * widget)
{
```



```
    painter->drawPixmap(-pix.width()/2,-pix.height()/2,pix);
}
```

完成用于变形显示的项目后，实现一个主窗口对项目进行显示，并完成各种变形的工作。

MainWidget 类为主窗体类，继承自 QWidget。它包括一个控制面板区及一个显示区。具体代码如下：

```
class MainWidget : public QWidget
{
    Q_OBJECT
public:
    MainWidget(QWidget *parent=0);
    void createControlIFrame();

public slots:
    void slotRotate(int);
    void slotScale(int);
    void slotShear(int);
    void slotTranslate(int);

private:
    QGraphicsView *view;
    QFrame *ctrlFrame;
    PixItem *item;
    int angle;
    qreal scale;
    qreal shear;
    qreal translate;
};
```

主窗体类的构造函数：

```
MainWidget::MainWidget(QWidget *parent)
    : QWidget(parent)
{
    1    angle = 0;
    2    scale = 5;
    3    shear = 5;
    4    translate = 50;

    5    QGraphicsScene *scene = new QGraphicsScene;
    6    scene->setSceneRect(-200,-200,400,400);
```

```
7   QPixmap * pixmap = new QPixmap(":/images/rotate.png");
8   item = new PixItem(pixmap);
9   scene->addItem(item);
10  item->setPos(0,0);

11  view = new QGraphicsView;
12  view->setScene(scene);
13  view->setMinimumSize(400,400);

14  ctrlFrame = new QFrame;
15  createControllFrame();

    //布局
16  QHBoxLayout * mainLayout = new QHBoxLayout;
17  mainLayout->setMargin(10);
18  mainLayout->setSpacing(20);
19  mainLayout->addWidget(view);
20  mainLayout->addWidget(ctrlFrame);
21  setLayout(mainLayout);

22  setWindowTitle(tr("Graphics Item Transformation"));
}
```

第 1~4 行对各个私有变量赋初值。

第 5、6 行新建一个 `QGraphicsScene` 对象，并调用 `setSceneRect()` 函数限定它的显示区域。

第 7、8 行新建一个 `PixItem` 对象，也即上面自定义的项目，并为它传入一个图片用于显示。

第 9 行把这个自定义项目对象加入到场景中。

第 10 行设置项目在场景中的位置为中心(0,0)。

第 11~13 行新建一个视口对象，并与场景相连，设置视口的最小尺寸为(400,400)。

第 14、15 行调用 `createControllFrame()` 函数新建主窗体右侧的控制面板区。

第 16~21 行对主窗体进行布局。

第 22 行设置主窗体的标题。

右侧的控制面板区分为 4 块，分别是旋转控制区、缩放控制区、切变控制区以及位移控制区，每个区都用一个 `QGroupBox` 对象包含一个 `QSpinBox` 对象来实现，并为每个 `QSpinBox` 对象的 `valueChanged()` 信号连接一个槽函数，具体完成变形的工作。具体代码



如下：

```
void MainWidget::createControlFrame()
{
    //旋转控制
    QGroupBox *rotateGroup = new QGroupBox(tr("Rotate"));
    QSlider *rotateSlider = new QSlider;
    rotateSlider->setOrientation(Qt::Horizontal);
    rotateSlider->setRange(0,360);
    connect(rotateSlider,SIGNAL(valueChanged(int)),this,SLOT(slotRotate(int)));
    QHBoxLayout *l1 = new QHBoxLayout;
    l1->addWidget(rotateSlider);
    rotateGroup->setLayout(l1);

    //缩放控制
    QGroupBox *scaleGroup = new QGroupBox(tr("Scale"));
    QSlider *scaleSlider = new QSlider;
    .....
    connect(scaleSlider,SIGNAL(valueChanged(int)),this,SLOT(slotScale(int)));
    .....

    //切变控制
    QGroupBox *shearGroup = new QGroupBox(tr("Shear"));
    QSlider *shearSlider = new QSlider;
    .....
    connect(shearSlider,SIGNAL(valueChanged(int)),this,SLOT(slotShear(int)));
    .....

    //位移控制
    QGroupBox *translateGroup = new QGroupBox(tr("Translate"));
    QSlider *translateSlider = new QSlider;
    .....
    connect(translateSlider,SIGNAL(valueChanged(int)),this,
           SLOT(slotTranslate(int)));
    .....

    //主布局
}

```

slotRotate()函数实现项目的旋转。调用 QGraphicsItem 类的 rotate()函数实现，它的参数为旋转角度的度数值。具体代码如下：

```
void MainWidget::slotRotate(int value)
{
    item->rotate(value-angle);
    angle = value;
}
```

slotScale()函数实现项目的缩放。调用 QGraphicsItem 类的 scale()函数实现，它的参数为缩放的比例。具体代码如下：

```
void MainWidget::slotScale(int value)
{
    qreal s;
    if (value>scale)
        s = pow(1.1,(value-scale));
    else
        s = pow(1/1.1,(scale-value));

    item->scale(s,s);

    scale = value;
}
```

slotShear()函数实现项目的切变。调用 QGraphicsItem 类的 shear()函数实现，它的参数为切变的比例。具体代码如下：

```
void MainWidget::slotShear(int value)
{
    item->shear((value-shear)/10.0,0);
    shear = value;
}
```

slotTranslate()函数实现项目的切变。调用 QGraphicsItem 类的 translate()函数实现，它的参数为位移的大小。具体代码如下：

```
void MainWidget::slotTranslate(int value)
{
    item->translate(value-translate,value-translate);
    translate = value;
}
```

本实例对 QGraphicsItem 的各种变形进行了总结，分析了各种变形函数的应用方式，在调用各种变形函数时，应注意参数的选择，尤其是在需要复原的场合中。

实例 42 飞舞的蝴蝶

知识点：

- 自定义 QGraphicsItem
- 利用定时器实现 QGraphicsItem 的动画效果

本实例实现了一个在显示框中不停上下飞舞的蝴蝶，如图 6-8 所示。

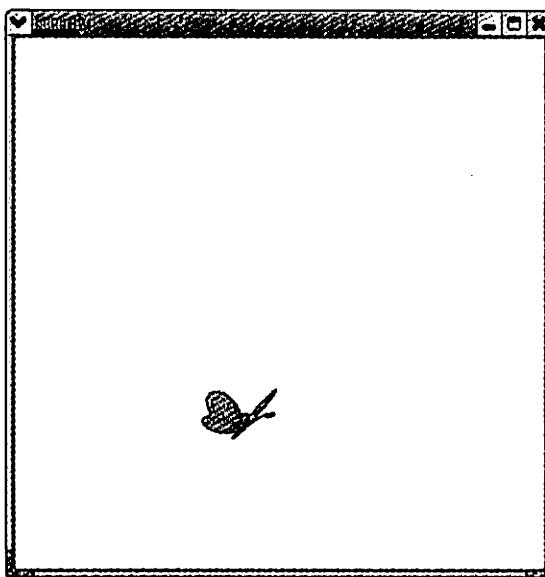


图 6-8 飞舞的蝴蝶

本实例主要是利用了定时器实现动画的原理，在定时器的 timerEvent() 中对 QGraphicsItem 进行重画来实现。具体实现如下：

首先自定义一个 Butterfly 类显示蝴蝶。

```
class Butterfly : public QObject, public QGraphicsItem
{
    Q_OBJECT
public:
    Butterfly();
```

```
2 void timerEvent(QTimerEvent *);  
3 QRectF boundingRect() const;  
  
protected:  
4 void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,  
    QWidget *widget);  
private:  
5 bool up;  
6 QPixmap pix_up;  
7 QPixmap pix_down;  
  
8 qreal angle;  
};
```

Butterfly 类继承自 QGraphicsItem 类和 QObject 类。

第 1 行声明 Butterfly 类的构造函数。

第 2 行声明了定时器的 timerEvent() 函数。

第 3 行声明了 boundingRect() 函数，所有继承自 QGraphicsItem 的自定义项目都必须实现此函数。

第 4 行声明了重画函数。

第 5 行声明了一个布尔变量用于标识蝴蝶翅膀的位置，因为本实例中飞舞的蝴蝶是由两幅图片组成的。

第 6、7 行声明的两个 QPixmap 变量即用于表示两幅蝴蝶的图片。

Butterfly 类的构造函数：

```
Butterfly::Butterfly()  
{  
1 pix_up.load(":/images/butterfly1.png");  
2 pix_down.load(":/images/butterfly2.png");  
3 up = true;  
4 startTimer(100);  
}
```

第 1、2 行首先调用 QPixmap 的 load() 函数把两幅蝴蝶的图片分别保存到 pix_up 和 pix_down 变量中。

第 3 行为图片切换的标识变量赋初值。

第 4 行启动一个定时器，并设置时间间隔为 100 毫秒。

boundingRect() 函数为蝴蝶项目限定区域范围。此范围是以项目自身的坐标系为基础设定的。具体代码如下：



```
QRectF Butterfly::boundingRect() const
{
    qreal adjust = 2;
    return QRectF(-pix_up.width()/2-adjust,-pix_up.height()/2-adjust,
                  pix_up.width()+adjust*2,pix_up.height()+2*adjust);
}
```

蝴蝶的重画函数：

```
void Butterfly::paint(QPainter *painter, const QStyleOptionGraphicsItem *option,QWidget *widget)
{
    if(up)
    {
        painter->drawPixmap(boundingRect().topLeft(),pix_up);
        up = !up;
    }
    else
    {
        painter->drawPixmap(boundingRect().topLeft(),pix_down);
        up = !up;
    }
}
```

对蝴蝶项目重画时，首先对 up 变量进行判断，确定当前已显示的是 pix_up 图片还是 pix_down 图片，以此决定此次重画哪幅蝴蝶图片，若当前显示的是 pix_up 也即蝴蝶翅膀在上的图片，则此次重画 pix_down 翅膀在下的图片，反之亦然，以实现蝴蝶翅膀上下翻飞的效果。注意在 drawPixmap() 调用后要设置 up 变量。

定时器函数用于完成蝴蝶的移动，蝴蝶飞舞的边界作一个限定。具体代码如下：

```
void Butterfly::timerEvent(QTimerEvent *)
{
    //边界控制
1    qreal edgex = scene()->sceneRect().right()+boundingRect().width()/2;
2    qreal edgetop = scene()->sceneRect().top()+boundingRect().height()/2;
3    qreal edgebottom = scene()->sceneRect().bottom()+boundingRect().height()/2;

4    if (pos().x() >= edgex)
5        setPos(scene()->sceneRect().left(),pos().y());
6    if (pos().y() <= edgetop)
7        setPos(pos().x(),scene()->sceneRect().bottom());
8    if (pos().y() >= edgebottom)
9        setPos(pos().x(),scene()->sceneRect().top());
```

```
10    angle += (qrand()%10)/20.0;
11    qreal dx = fabs(sin(angle*PI)*10.0);
12    qreal dy = (qrand()%20)-10.0;
13    setPos(mapToParent(dx,dy));
}
```

第1~3行分别限定了蝴蝶飞舞的右边界、上边界和下边界。

第4、5行若蝴蝶飞行到了超过右边界时，则调用 `setPos()` 函数，把蝴蝶水平移回左边界处。

第6、7行若蝴蝶飞行到了超过上边界时，则把蝴蝶垂直移回至下边界处。

第8、9行若蝴蝶飞行到了超过下边界时，则把蝴蝶垂直移回至上边界处。

第10~13行为蝴蝶的飞行加入一些随机的起伏。

注意，此时的 `dx`、`dy` 是相对蝴蝶的坐标系而言的，因此在调用 `setPos()` 函数时，应使用 `mapToParent()` 函数映射成场景的坐标。

完成了蝴蝶类的实现后，把它应用到场景中，并关联一个视口。具体代码如下：

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

    QGraphicsScene *scene = new QGraphicsScene;
    scene->setSceneRect(QRectF(-200,-200,400,400));

    Butterfly *butterfly = new Butterfly;
    butterfly->setPos(-100,0);
    scene->addItem(butterfly);

    QGraphicsView *view = new QGraphicsView;
    view->setScene(scene);
    view->resize(400,400);
    view->show();
    return app.exec();
}
```

第 7 章 Model/View

Qt4 版本中，引进了一个全新的 Model/View 结构，使数据管理与相应的数据显示相互独立，并提供了一系列标准的函数接口和用于 Model 模块与 View 模块之间的通信。数据与显示的分离为开发者定制数据项的显示带来了更大的灵活性。

本章的实例即围绕 Model/View 结构进行编写，为更好地理解实例，在进行实例分析之前，有必要对 Model/View 结构作一些概念性的介绍，使读者对 Model/View 结构有一个宏观的认识。

1. Model/View与MVC

Model/View 结构是从 MVC 演化而来，MVC (Model-View-Controller) 是从 Smalltalk 发展而来的一种设计模式，常被用于构建用户界面。它由 3 种对象组成。Model 是应用程序对象，View 是它的屏幕表示，Controller 定义了用户界面如何对用户输入进行响应。在 MVC 出现之前，用户界面设计倾向于三者揉合在一起，MVC 对它们进行了解耦，提高了灵活性与重用性。把 MVC 中的 View 和 Controller 合在一起，就形成了 Model/View 结构。这种结构的指导思想仍然是把数据管理与显示分离，但比 MVC 更简单一些。

2. Model/View的基本框架

为了更灵活地对用户的输入进行处理，在 Model/View 结构中引入了 Delegate 的概念，所有与数据编辑或定制显示相关的操作都由 Delegate 这个代理来处理，它的最大好处在于可以通过它对数据的编辑和显示进行定制。Model、View 及 Delegate 3 个模块之间的结构关系如图 7-1 所示。

Model 负责与实际的数据源通信，数据源对于 View 模块和 Delegate 模块来说是透明的，它们都通过 Model 模块提供的 Model Indexs 接口来对数据进行访问。这种分离方式使得不同的 View 能够对同一个 Model 进行显示，也可独立地对 View 进行修改而不对实际数据结构造成影响。

三者之间的通信通过信号与槽的机制实现。当自身的状态发生改变时，会发出信号通知其他模块。

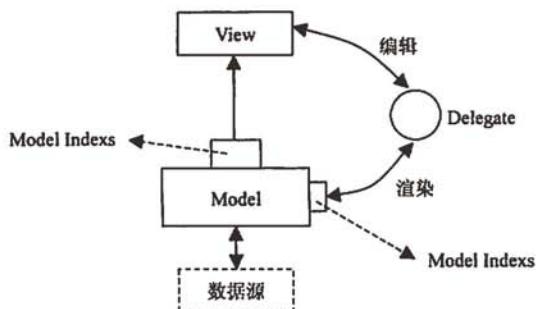


图 7-1 Model、View、Delegate 的关系

(1) Model

Model 模块的基本接口都在 `QAbstractItemModel` 中定义。`QAbstractItemModel` 类是所有 Model 的基类。在开发中一般不直接使用 `QAbstractItemModel`，而是使用它的子类，如图 7-2 所示。

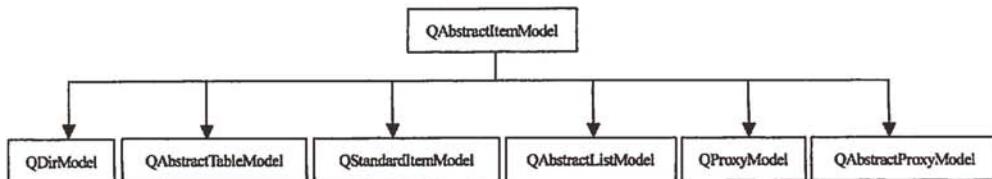


图 7-2 QAbstractItemModel 及其子类

若 Qt 提供的 Model 类不能满足对数据管理的要求，可自定义一个 Model 类，可从 `QAbstractItemModel` 类继承，也可直接从 `QAbstractListModel` 或 `QAbstractTableModel` 类继承，本章实例 45 即实现一个继承自 `QAbstractListModel` 的自定义 Model 类。

Model 模块本身并不存储数据，而是为 View 和 Delegate 访问数据提供标准的接口，无论底层的数据源采用何种结构存储数据，Model 模块都会以数据表项的方式进行管理，上层的 View 或 Delegate 通过这种表项的方式进行访问，以 `QModelIndex` 作为访问的索引。通过确定行、列号获得 `QModelIndex`，如图 7-3 所示。

(2) View

View 模块从 Model 中获得数据项显示给用户。数据显示的方式不必与 Model 提供的表示方式相同，可与底层数据源的数据结构完全不同。View 模块的基本接口都在 `QAbstractItemView` 中定义。`QAbstractItemView` 类是所有 View 的基类。在开发中一般不直接使用 `QAbstractItemView`，而是使用它的子类，如图 7-4 所示。

Qt 提供了一些常用的 View 模型，如 `QTreeView`、`QTableView` 和 `QListView`，若这



些 Qt 提供的 View 模型不能满足开发的显示要求, 可继承 `QAbstractItemView` 类自定义一个 View 类, 本章实例 46 即实现了一个自定义的柱状统计图的 View 类。

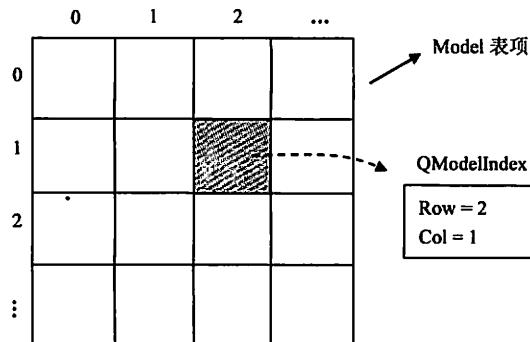


图 7-3 通过确定行、列号获得 QModelIndex

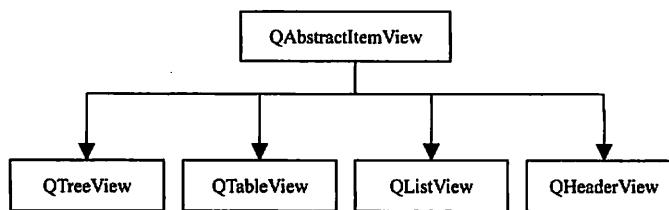


图 7-4 QAbstractItemView 及其子类

(3) Delegate

Delegate 用于渲染数据的显示及编辑, 当然也可不使用 Delegate 而让 View 模块按默认方式处理显示及编辑。但当对数据项的显示与编辑方式有特定要求时, 需采用 Delegate 实现。

Delegate 的基本接口在 `QAbstractItemDelegate` 类中定义。Delegate 通过实现 `paint()` 和 `sizeHint()` 以达到渲染数据项的目的。然而, 简单的基于 Widget 的 Delegate, 可以从 `QItemDelegate` 类继承, 而不是 `QAbstractItemDelegate`, 这样可以使用它提供的上述函数的默认实现。Delegate 可以使用 Widget 来处理编辑过程, 也可以直接对事件进行处理。本章的实例 44 实现一个利用 Delegate 的例子。

本章包括 4 个实例:

- 文件目录浏览器
- 利用特定控件进行表项编辑
- 自定义 Model
- 柱状统计图

实例 43 文件目录浏览器

知识点：

- 利用现有的 Model 和 View 进行连接
- 如何让不同的 View 模块使用相同的 Model
- View 的选择模式
- 不同的 View 显示相同的选择

本实例利用 Qt 现有的 QDirModel、QTreeView、QtableView 和 QListview 实现一个简单的文件目录浏览器，如图 7-5 所示。

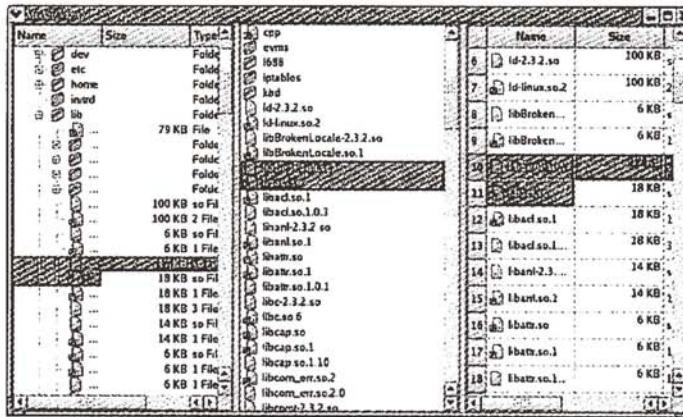


图 7-5 文件目录浏览器

QDirModel 类继承自 QAbstractItemModel 类，是为访问本地文件系统提供的数据模型，它提供了一系列与文件操作相关的函数，如新建、删除、创建目录等。本实例仅利用它来显示本地文件系统。

具体实现代码如下：

```

int main(int argc, char *argv[])
{
    1   QApplication app(argc, argv);
    2   QDirModel model;

```



第 1、2 行新建一个 QDirModel 对象，为数据访问做好准备，QDirModel 的创建还可设置过滤器，设置只有符合条件的文件或目录才可被访问。

```
3   QListWidget list;
4   QTreeView tree;
5   QTableView table;
```

第 3~5 行新建 3 种不同的 View 对象，分别是 QListWidget、QTreeView 和 QTableView，以便文件目录可以以 3 种不同的方式显示。

```
6   tree.setModel(&model);
7   list.setModel(&model);
8   table.setModel(&model);
```

第 6~8 行调用 setModel() 函数设置 3 个 View 对象的 Model 都为 QDirModel 对象 model。

```
9   tree.setSelectionModel(QAbstractItemView::MultiSelection);
10  list.setSelectionModel(tree.selectionModel());
11  table.setSelectionModel(tree.selectionModel());
```

第 9 行首先调用 QTreeView 的 setSelectionModel() 函数，设置 QTreeView 对象的选择方式为多选。QAbstractItemView 提供了 5 种选择模式：QAbstractItemView::SingleSelection、QAbstractItemView::ContiguousSelection、QAbstractItemView::ExtendedSelection、QAbstractItemView::MultiSelection 和 QAbstractItemView::NoSelection。

第 10、11 行设置 QListWidget 和 QTableView 对象与 QTreeView 对象使用相同的选择模型。

```
12  QObject::connect(&tree,SIGNAL(doubleClicked(QModelIndex)),&list,
                     SLOT(setRootIndex(QModelIndex)));
13  QObject::connect(&tree,SIGNAL(doubleClicked(QModelIndex)),&table,
                     SLOT(setRootIndex(QModelIndex)));
```

分别连接 QTreeView 对象的 doubleClicked() 信号与 QListWidget 对象和 QTableView 对象的 setRootIndex() 槽函数，实现当双击 QTreeView 对象中的某个目录时，QListWidget 对象和 QTableView 对象中显示此选定目录下的所有文件和目录。

```
14  QSplitter *splitter = new QSplitter;
15  splitter->addWidget(&tree);
16  splitter->addWidget(&list);
17  splitter->addWidget(&table);
```

```

18 splitter->setWindowTitle(QObject::tr("Model/View"));
19 splitter->show();
20 return app.exec();
}

```

实例 44 利用特定控件进行表项编辑

知识点：

- 如何实现一个自定义的 Delegate
- 如何在程序中应用 Delegate

在实例 7 中，实现了在表格中嵌入各种不同控件的例子，可通过表格中的控件对编辑的内容进行限定。这种在表格中插入控件的方式控件始终会显示，当表格中控件数目较多时，会影响表格的美观。在本实例中，利用 Delegate 的方式实现同样的效果，但控件只在需要编辑数据项时才出现，如图 7-6 所示。



图 7-6 利用特定控件进行表项编辑



表格中有 3 种不同的控件，分别是用于生日输入的日历编辑框 QDateLineEdit、下拉列表框 QComboBox 和一个 QSpinBox，因此需实现 3 个 Delegate 对象：DateDelegate、ComboDelegate 和 SpinDelegate，这 3 个 Delegate 都是 Widget 类型的控件，因此都采用继承 QItemDelegate 的方式实现。下面分别对 3 个 Delegate 的实现进行介绍。

DateDelegate 的实现：

```
class DateDelegate : public QItemDelegate
{
    Q_OBJECT
public:
    DateDelegate(QObject *parent = 0);
    QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem &option,
                          const QModelIndex &index) const;

    void setEditorData(QWidget *editor, const QModelIndex &index) const;
    void setModelData(QWidget *editor, QAbstractItemModel *model,
                      const QModelIndex &index) const;

    void updateEditorGeometry(QWidget *editor,
                             const QStyleOptionViewItem &option, const QModelIndex &index) const;
};
```

DateDelegate 继承自 QItemDelegate 类，实现一个自定义的 Delegate，一般需要重定义声明中的几个虚函数。createEditor() 函数完成创建控件的工作，创建由参数中 QModelIndex 对象指定的表项数据的编辑控件，并对控件的内容进行限定；setEditorData() 函数设置控件显示的数据，把 Model 中的数据更新至 Delegate 中，相当于一个初始化的工作；setModelData() 函数把 Delegate 中对数据的改变更新至 Model 中。updateEditor() 函数更新控件区的显示。

```
QWidget *DateDelegate::createEditor(QWidget *parent,
                                    const QStyleOptionViewItem &/* option */,
                                    const QModelIndex &/* index */) const
{
    1   QDateTimeEdit *editor = new QDateTimeEdit(parent);
    2   editor->setDisplayFormat("yyyy-MM-dd");
    3   editor->setCalendarPopup(true);
    4   editor->installEventFilter(const_cast<DateDelegate*>(this));

    5   return editor;
}
```

第1行新建一个 QDateTimeEdit 对象作为编辑时的输入控件。

第2行设置此 QDateTimeEdit 对象的显示格式为 yyyy-MM-dd，此显示方式为 ISO 方式。

 小贴士：日期显示的格式可以有很多种，以 QString 的方式设置，可根据需要进行设定，如：

```
yy.MM.dd    08.01.01
d.MM.yyyy   1.01.2008
```

其中，y 表示年，M 表示月份，d 表示日，需要注意的是表示月的 M 一定要大写。

第3行设置日历选择的显示以 Popup 的方式，即下拉显示的方式。

第4行调用 QObject 类的 installEventFilter() 函数安装事件过滤器，使 DateDelegate 能捕获 QDateTimeEdit 对象的事件。

```
void DateDelegate::setEditorData(QWidget *editor,
                                 const QModelIndex &index) const
{
    1   QString dateStr = index.model()->data(index).toString();
    2   QDate date = QDate::fromString(dateStr, Qt::ISODate);

    3   QDateTimeEdit *edit = static_cast<QDateTimeEdit*>(editor);
    4   edit->setDate(date);
}
```

第1行获取指定 index 数据项的数据，调用 QModelIndex 的 model() 可获得提供此 index 的 Model 对象，data() 函数返回的是一个 QVariant 对象，把它转换成一个 QString 类型数据。

第2行通过 QDate 的 fromString() 函数把以 QString 类型表示的日期数据转换成 QDate 类型，Qt::ISODate 表示 QString 类型的日期是以 ISO 格式保存的，这样最终转换获得的 QDate 数据也是 ISO 格式，使控件显示与表格显示保持一致。

第3行把参数中的 editor 转换成 QDateTimeEdit 对象，以获得编辑控件的对象指针。

第4行设置控件的显示数据。

```
void DateDelegate::setModelData(QWidget *editor, QAbstractItemModel *model,
                               const QModelIndex &index) const
{
    1   QDateTimeEdit *edit = static_cast<QDateTimeEdit*>(editor);
    2   QDate date = edit->date();
```



```
3     model->setData(index, QVariant(date.toString(Qt::ISODate)));
}
```

第 1 行通过紧缩转换获得编辑控件的对象指针。

第 2 行获得编辑控件中的数据更新。

第 3 行调用 setData() 把数据修改更新到 Model 中。

ComboDelegate 的实现：

ComboDelegate 的类声明与 DateDelegate 的类似，需要重定义的函数也一样，此处只列出了重定义的函数。

在 createEditor() 函数中创建一个 QComboBox 控件，并插入可显示的条目，并安装事件过滤器。

```
QWidget *ComboDelegate::createEditor(QWidget *parent,
                                     const QStyleOptionViewItem, const QModelIndex) const
{
    QComboBox *editor = new QComboBox(parent);
    editor->addItem(QString::fromLocal8Bit("工人"));
    editor->addItem(QString::fromLocal8Bit("农民"));
    editor->addItem(QString::fromLocal8Bit("医生"));
    editor->addItem(QString::fromLocal8Bit("律师"));
    editor->addItem(QString::fromLocal8Bit("军人"));

    editor->installEventFilter(const_cast<ComboDelegate*>(this));
    return editor;
}
```

更新 Delegate 控件的数据显示：

```
void ComboDelegate::setEditorData(QWidget *editor,
                                   const QModelIndex &index) const
{
    QString str = index.model()->data(index).toString();

    QComboBox *box = static_cast<QComboBox*>(editor);
    int i = box->findText(str);
    box->setCurrentIndex(i);
}
```

更新 Model 中的数据：

```
void ComboDelegate::setModelData(QWidget *editor, QAbstractItemModel *model,
                                 const QModelIndex &index) const
```

```

    {
        QComboBox *box = static_cast<QComboBox*>(editor);
        QString str = box->currentText();

        model->setData(index, str);
    }

    void ComboDelegate::updateEditorGeometry(QWidget *editor,
        const QStyleOptionViewItem &option, const QModelIndex &/* index */) const
{
    editor->setGeometry(option.rect);
}

```

SpinDelegate 的实现与 ComboDelegate 的实现类似，此处不再赘述。

在 Model/View 结构的显示中应用准备好的 Delegate：

```

int main( int argc, char **argv )
{
1   QApplication app(argc, argv);
.....  

2   QStandardItemModel model(4, 4);
3   QTableView tableView;
4   tableView.setModel(&model);

```

第 2~4 行新建一个 QStandardItemModel 对象，并采用 QTableView 以表格的方式进行显示。

```

5   DateDelegate dateDelegate;
6   ComboDelegate comboDelegate;
7   SpinDelegate spinDelegate;
8   tableView.setItemDelegateForColumn(1,&dateDelegate);
9   tableView.setItemDelegateForColumn(2,&comboDelegate);
10  tableView.setItemDelegateForColumn(3,&spinDelegate);

```

第 5~10 行分别新建 3 个 Delegate，并调用 View 的 setItemDelegateForColumn() 函数为指定的列应用指定的 Delegate，此处对第 1 列应用 DateDelegate。第 2 列应用 ComboDelegate，第 3 列应用 SpinDelegate。

```

11  model.setHeaderData(0,Qt::Horizontal,QObject::tr("Name"));
12  model.setHeaderData(1,Qt::Horizontal,QObject::tr("Birthday"));
13  model.setHeaderData(2,Qt::Horizontal,QObject::tr("Job"));
14  model.setHeaderData(3,Qt::Horizontal,QObject::tr("Income"));

```

第 11~14 行对表格的表头显示进行设置。



```

15  QFile file("./data.tab");
16  if (file.open(QFile::ReadOnly | QFile::Text))
17  {
18      QTextStream stream(&file);
19      QString line;
20      model.removeRows(0, model.rowCount(QModelIndex()), QModelIndex());
21      int row = 0;
22      do {
23          line = stream.readLine();
24          if (!line.isEmpty())
25          {
26              model.insertRows(row, 1, QModelIndex());
27              QStringList pieces = line.split(".", QString::SkipEmptyParts);
28              model.setData(model.index(row, 0, QModelIndex()),
29                            pieces.value(0));
29              model.setData(model.index(row, 1, QModelIndex()),
30                            pieces.value(1));
30              model.setData(model.index(row, 2, QModelIndex()),
31                            pieces.value(2));
31              model.setData(model.index(row, 3, QModelIndex()),
32                            pieces.value(3));
32              row++;
33          }
34      } while (!line.isEmpty());
35      file.close();
36  }

```

第 15~32 行从文件中读取数据作为数据源，并把文件中的数据按数据表项的方式进行管理。

```

33  tableView.setWindowTitle(QObject::tr("Delegate"));
34  tableView.show();
35  return app.exec();
}

```

实例 45 自定义 Model

知识点：

- 如何实现一个自定义的 Model

- 如何在程序中应用自定义的 Model
- Model 中行、列概念的应用

本实例继承 QAbstractListModel 类实现了一个 QStringListModel，为字符串列表数据实现一个 Model 类。

可打开一个数据源文件，以列表的方式显示，并能保存对数据项的修改，如图 7-7 所示。



图 7-7 自定义 Model

在实现自定义的 Model 类时，可从 QAbstractItemModel 类继承，也可直接从 QAbstractListModel 和 QAbstractTableModel 类继承，QAbstractListModel 和 QAbstractTableModel 为实现提供了更多相应的默认实现，可减少实现的工作量。

QStringListModel 的实现：

```
class QStringListModel : public QAbstractListModel
{
    Q_OBJECT
public:
    QStringListModel(const QStringList &strings, QObject *parent = 0);

    //重定义基本虚函数
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role = Qt::DisplayRole) const;

    //重定义可编辑的虚函数
    Qt::ItemFlags flags(const QModelIndex &index) const;
    bool setData(const QModelIndex &index, const QVariant &value,
                 int role = Qt::EditRole);
```



```
//重定义插入与删除行虚函数  
bool insertRows(int position, int rows,  
                 const QModelIndex &index = QModelIndex());  
bool removeRows(int position, int rows,  
                 const QModelIndex &index = QModelIndex());  
  
private:  
    QStringList stringList;  
};
```

实现一个自定义 Model，只需重定义 rowCount() 和 data() 虚函数，重定义 headerData() 虚函数是为 QTreeView 和 QTableView 显示提供表头数据。

由于以字符串列表作为数据源，因此不用考虑父子继承的关系，只要有行号就能够指定数据项。

```
int StringListModel::rowCount(const QModelIndex &parent) const  
{  
    return stringList.count();  
}
```

由于本实例的数据源是一个 QStringList 类型，因此，QStringList 的长度即 Model 的行数。

```
QVariant StringListModel::data(const QModelIndex &index, int role) const  
{  
    if (!index.isValid())  
        return QVariant();  
  
    if (index.row() >= stringList.size())  
        return QVariant();  
  
    if (role == Qt::DisplayRole)  
        return stringList.at(index.row());  
    else  
        return QVariant();  
}
```

若要求的数据 Role 类型是 Qt::DisplayRole，则返回指定行的数据。Model 中的每个数据项都有一系列与之相关的数据元素，用 Role 值来表示，View 模块在显示数据项时，通过这些 Role 值来决定如何进行渲染，采用何种方式显示。常用的如 Qt::DisplayRole 一般表示用文本的方式显示，Qt::ToolTipRole 表示以提示的方式显示等，数量较多，具体可查阅 Qt 的相关帮助。

headerData()返回指定 Role 值、指定方向并且指定序号的表头数据值。具体代码如下：

```
QVariant StringListModel::headerData(int section, Qt::Orientation orientation, int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();

    if (orientation == Qt::Horizontal)
        return QString("Column %1").arg(section);
    else
        return QString("Row %1").arg(section);
}
```

flags()返回数据项的属性值。如可选 Qt::ItemIsSelectable、可编辑 Qt::ItemIsEditable 等。此处返回 QAbstractItemModel 的基本属性加上 Qt::ItemIsEditable 属性，QAbstractItemModel 的基本属性是 Qt::ItemIsEnabled|Qt::ItemIsSelectable。

```
Qt::ItemFlags StringListModel::flags(const QModelIndex &index) const
{
    if (!index.isValid())
        return Qt::ItemIsEnabled;

    return QAbstractItemModel::flags(index) | Qt::ItemIsEditable;
}
```

setData()设置指定序号、指定 Role 值的数据项的值。若设置成功则返回 true，并且发射 dataChanged()信号，否则返回 false。

```
bool StringListModel::setData(const QModelIndex &index, const QVariant &value, int role)
{
    if (index.isValid() && role == Qt::EditRole)
    {
        stringList.replace(index.row(), value.toString());
        emit dataChanged(index, index);
        return true;
    }
    return false;
}
```

实现一个可编辑的 Model，这个函数是必须被实现的。

insertRows()函数的实现代码如下：

```
bool StringListModel::insertRows(int position, int rows, const QModelIndex &index)
{
```



```

1 beginInsertRows(QModelIndex(), position, position+rows-1);
2 for (int row = 0; row < rows; ++row)
{
3     stringList.insert(position, "");
4 }
5 endInsertRows();
6 return true;
}

```

插入行，函数的 `position` 参数表明在哪一行之前插入行，`rows` 参数表示需要插入的行数，`index` 参数表示插入的行以哪个数据项作为父项。

若 `position==0` 则表示行插入在指定父项下的最前端；若 `position==rowCount` 即总行数，则表示行插入在指定父项下的最后。

第 1 行调用 `beginInsertRows()` 函数表示开始插入，此函数的第一个参数指明插入行的父项；第二和第三个参数指明所要插入行在插入后的第一和最后一个行号值。如在第 2 行之前插入 3 行：`insertRows(2,3,index);`，则开始插入行应调用 `beginInsertRows(index,2,4)`；插入的第 1 行行号为 2，插入的最后一行行号为 4，插入后，其后的行号依次顺延，如图 7-8 所示。

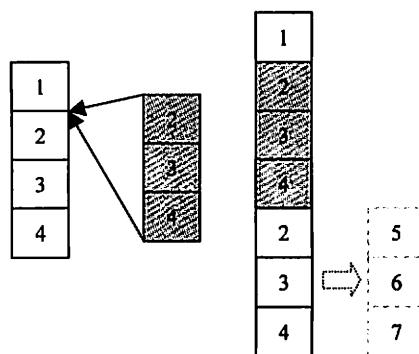


图 7-8 插入后，其后的行号依次顺延

第 2、3 行完成对实际数据源的数据修改。

第 4 行调用 `endInsertRows()` 结束插入行操作。

`removeRows()` 函数的实现代码如下：

```

bool QStringListModel::removeRows(int position, int rows, const QModelIndex &index)
{
    beginRemoveRows(QModelIndex(), position, position+rows-1);
    for (int row = 0; row < rows; ++row)

```

```

    {
        stringList.removeAt(position);
    }
    endRemoveRows();
    return true;
}

```

删除指定行是插入行的逆过程，position 参数表示从哪一行开始删除，rows 参数表示删除的行数，index 参数指明删除行的父项。

`beginRemoveRows()` 函数中需指明删除行的起始行号和末尾行号，如图 7-9 所示。

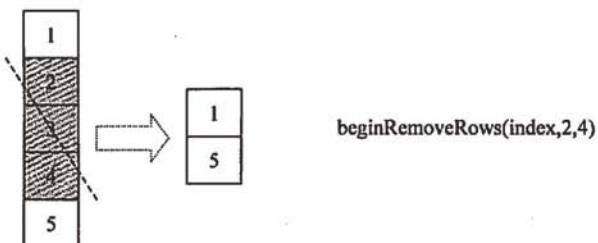


图 7-9 在 `beginRemoveRows()` 函数中指明删除行的起始行号和末尾行号

在主程序中应用 `QStringListModel`。

在主窗口类声明中，`setupModel()` 函数用于创建 `QStringListModel` 对象，`setupView()` 函数用于创建显示的 View 对象。具体代码如下：

```

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent=0);

    void createMenu();
    void setupModel();
    void setupView();
    void openFile(QString);
public slots:
    void slotOpenFile();
    void slotSaveFile();
    void slotInsertRows();
    void slotRemoveRows();
private:
    QListView *list;
}

```



```
    QStringListModel *model;
    QString name;
};
```

创建一个 QListView 对象，以列表的方式对数据进行显示，并将 QListView 对象设置为主窗口的中央窗体。具体代码如下：

```
void MainWindow::setupModel()
{
    QStringList strList;
    model = new QStringListModel(strList,this);
}

void MainWindow::setupView()
{
    list = new QListView;
    list->setModel(model);
    setCentralWidget(list);
}
```

主窗口的菜单栏有两个菜单（打开和保存），分别完成文件操作，以及编辑操作（插入和删除行）。

打开文件，利用标准对话框实现。具体代码如下：

```
void MainWindow::slotOpenFile()
{
    name = QFileDialog::getOpenFileName(
        this,
        "open file dialog",
        "",
        "strip files (*.txt)");
}

if(!name.isEmpty())
    openFile(name);
}
```

slotSaveFile()函数响应“保存”菜单，完成对文件内容修改的保存工作。具体代码如下：

```
void MainWindow::slotSaveFile()
{
    if(name.isEmpty())
        return;
```

```
QFile file(name);
if (!file.open(QFile::WriteOnly))
    return;

QTextStream ts(&file);

for(int i=0; i<model->rowCount(); i++)
{
    QModelIndex index = model->index(i);
    QString str = model->data(index, Qt::DisplayRole).toString();
    ts << str << ",";
}
}
```

openFile()函数完成具体打开文件的工作，并把内容按 QStringListModel 的列表方式进行管理。具体代码如下：

```
void MainWindow::openFile(QString path)
{
    if (!path.isEmpty())
    {
        QFile file(path);
        if (file.open(QFile::ReadOnly | QFile::Text))
        {
            QTextStream stream(&file);
            QString line;
            model->removeRows(0, model->rowCount(QModelIndex()),
                               QModelIndex());
            int pos = 0;
            line = stream.readLine();
            if (!line.isEmpty())
            {
                QStringList pieces = line.split(",",QString::SkipEmptyParts);
                QString str;
                foreach(str,pieces)
                {
                    model->insertRows(pos, 1, QModelIndex());
                    model->setData(model->index(pos),str);
                    pos++;
                }
            }
            file.close();
        }
    }
}
```



```
        }
    }
}
```

slotInsertRows()函数响应“插入”菜单，在当前行之前插入行，利用标准输入对话框QInputDialog 输入行数，调用 QStringListModel 的 insertRows()函数完成插入行的工作。具体代码如下：

```
void MainWindow::slotInsertRows()
{
    bool ok;
    QModelIndex index = list->currentIndex();
    int rows = QInputDialog::getInteger(this,tr("Insert Row Number"),
                                         tr("Please input number:"),1,1,10,1,&ok);
    if(ok)
    {
        model->insertRows(index.row(),rows,QModelIndex());
    }
}
```

slotRemoveRows()函数响应“删除”菜单，完成删除当前行的工作。具体代码如下：

```
void MainWindow::slotRemoveRows()
{
    QModelIndex index = list->currentIndex();
    model->removeRows(index.row(),1,QModelIndex());
}
```

实例 46 柱状统计图

知识点：

- 如何实现一个自定义的 View
- 如何在程序中应用自定义的 View

本实例利用自定义的 View 实现一个柱状统计图对 TableModel 的表格数据进行显示，如图 7-10 所示。

实现自定义的 View，可从 QAbstractItemView 继承子类，对所需的虚函数进行重定义与实现，对于 QAbstractItemView 类中的纯虚函数，在子类中必须进行重定义，但不一

定要实现，可根据需要选择。

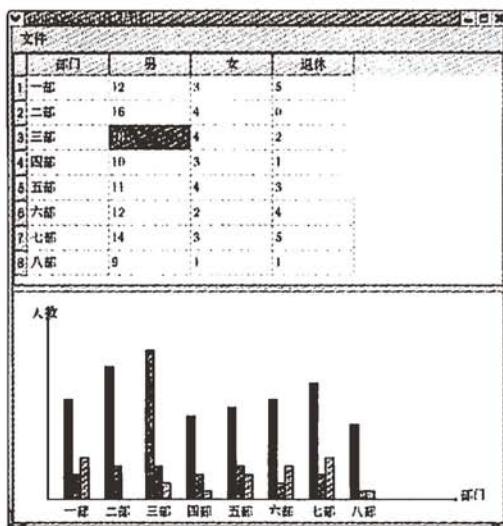


图 7-10 柱状统计图

本实例继承 QAbstractItemView 类，自定义了一个 HistogramView 类，用于对表格数据进行柱状图显示。

```
class HistogramView : public QAbstractItemView
{
    Q_OBJECT
public:
    HistogramView(QWidget *parent=0);

    QRect visualRect(const QModelIndex &index) const;
    void scrollTo(const QModelIndex &index, ScrollHint hint = EnsureVisible);
    QModelIndex indexAt(const QPoint &point) const;

    void paintEvent(QPaintEvent * );
    void mousePressEvent(QMouseEvent * );

    void setSelectionModel(QItemSelectionModel * selectionModel);
    QRegion itemRegion(QModelIndex index);

protected slots:
    void dataChanged(const QModelIndex &topLeft,
                     const QModelIndex &bottomRight);
```



```
void selectionChanged(const QItemSelection & selected,
                      const QItemSelection & deselected );

protected:
    QModelIndex moveCursor(QAbstractItemView::CursorAction cursorAction,
                           Qt::KeyboardModifiers modifiers);
    int horizontalOffset() const;
    int verticalOffset() const;
    bool isIndexHidden(const QModelIndex & index) const;
    void setSelection( const QRect& rect,
                       QItemSelectionModel::SelectionFlags flags );
    QRegion visualRegionForSelection(const QItemSelection & selection) const;

private:
    QItemSelectionModel *selections;

    QList<QRegion> listRegionM;
    QList<QRegion> listRegionF;
    QList<QRegion> listRegionS;
};
```

在类声明中，对父类 `QAbstractItemView` 中的所有纯虚函数都必须进行声明，这些必须声明的纯虚函数包括 `visualRect()`、`scrollTo()`、`indexAt()`、`moveCursor()`、`horizontalOffset()`、`verticalOffset()`、`isIndexHidden()`、`setSelection()` 和 `visualRegionForSelection()`，这些函数并不一定都要实现，根据功能要求选择实现。

本实例并没有对所有声明的函数都进行实现，选择了与数据选择及数据变更相关的函数进行了实现。类中声明了一个 `QItemSelectionModel` 对象 `selections` 用于保存与 View 的选择项相关的内容；另外 3 个私有变量 `listRegionM`、`listRegionF` 和 `listRegionS` 分别用于保存 3 个不同类型柱状图的区域范围，每个柱的区域是 `QList` 中的一个值。

`paintEvent()` 函数是具体完成绘制柱状图工作的关键函数：

```
void HistogramView::paintEvent(QPaintEvent *)
{
1   printf("paintEvent\n");
2   QPainter painter(viewport());
```

第 2 行以 `viewport()` 作为绘图设备新建一个 `QPainter` 对象。

```
3   painter.setPen(Qt::black);
4   int x0 = 40;
5   int y0 = 250;
```

```

//画坐标轴 y 轴
6 painter.drawLine(x0, y0, 40, 30);
7 painter.drawLine(38, 32, 40, 30);
8 painter.drawLine(40, 30, 42, 32);
9 painter.drawText(20, 30, tr("num"));
10 for (int i=1; i<5; i++)
{
    painter.drawLine(-1,-i*50,1,-i*50);
11    painter.drawText(-20,-i*50,tr("%1").arg(i*5));
}
//x 轴
13 painter.drawLine(x0, y0, 540, 250);
14 painter.drawLine(538, 248, 540, 250);
15 painter.drawLine(540, 250, 538, 252);
16 painter.drawText(545, 250, tr("department"));
17 int row;
//x 轴变量名
18 int posD = x0+20;
19 for (row = 0; row < model()->rowCount(rootIndex()); row++)
{
20     QModelIndex index = model()->index(row, 0, rootIndex());
21     QString dep = model()->data(index).toString();
22     painter.drawText(posD,y0+20,dep);
23     posD += 50;
}

```

第 6~23 行画 x、y 坐标轴，并标上坐标轴变量。

```

//男员工—柱状图
24 int posM = x0+20;
25 for (row = 0; row < model()->rowCount(rootIndex()); row++)
{
26     QModelIndex index = model()->index(row, 1, rootIndex());
27     int male = model()->data(index).toDouble();
28     int width = 10;
29     if (selections->isSelected(index))
30         painter.setBrush(QBrush(Qt::blue,Qt::Dense3Pattern));
31     else
32         painter.setBrush(Qt::blue);

```

```

33     painter.drawRect(QRect(posM,y0-male*10,width,male*10));
34     QRegion regionM(posM,y0-male*10,width,male*10);
35     listRegionM << regionM;
36
37     posM += 50;
}

```

第 24~36 行绘制表格第 1 列数据的柱状图。

第 29~32 行区分当前所要绘制的表格数据项是否已被选中，用不同的画刷区别选中与未被选中的数据项。

第 33 行调用 QPainter 的 drawRect() 根据当前数据项的值按比例绘制一个方形表示此数据项。

第 34、35 行把此数据所占据的区域保存到 listRegionM 列表中，为后面的数据项选择作准备。

```

//女员工—柱状图
37 int posF = x0+30;
38 for (row = 0; row < model()->rowCount(rootIndex()); row++)
{
39     QModelIndex index = model()->index(row, 2, rootIndex());
40     int female = model()->data(index).toDouble();
41
42     if (selections->isSelected(index))
43         painter.setBrush(QBrush(Qt::red, Qt::Dense3Pattern));
44     else
45         painter.setBrush(Qt::red);
46
47     painter.drawRect(QRect(posF,y0-female*10,width,female*10));
48     QRegion regionF(posF,y0-female*10,width,female*10);
49     listRegionF << regionF;
50
51     posF += 50;
}

```

第 37~49 行绘制表格第 2 列数据的柱状图。与绘制第 1 列数据的工作类似，用不同的画刷区分选中与未被选中的数据项，同时保存每个数据项所占的区域至 listRegionF 中。

```

//退休人员—柱状图
50 int posS = x0+40;
51 for (row = 0; row < model()->rowCount(rootIndex()); row++)

```

```

52     QModelIndex index = model()->index(row, 3, rootIndex());
53     int sum = model()->data(index).toDouble();

54     int width = 10;
55     if (selections->isSelected(index))
56         painter.setBrush(QBrush(Qt::green, Qt::Dense3Pattern));
57     else
58         painter.setBrush(QBrush(Qt::green));
59     painter.drawRect(QRect(posS,y0-sum*10,width,sum*10));
60     QRegion regionS(posS,y0-sum*10,width,sum*10);
61     listRegionS << regionS;

62     posS += 50;
}
}

```

第 50~62 行绘制表格第 3 列数据的柱状图。与绘制第 1 列数据的工作类似，用不同的画刷区分选中与未被选中的数据项，同时保存每个数据项所占的区域至 listRegionS 中。

重定义 QAbstractItemView 的 dataChanged() 函数是一个槽函数，当 Model 中的数据发生变更时，此槽函数会响应。此处当 Model 中数据更改时，调用绘图设备的 update() 函数进行更新，反映数据的变化。具体代码如下：

```

void HistogramView::dataChanged(const QModelIndex &topLeft, const QModelIndex &bottomRight)
{
    QAbstractItemView::dataChanged(topLeft, bottomRight);
    viewport()->update();
}

```

setSelectionModel() 函数的实现代码如下：

```

void HistogramView::setSelectionModel(QItemSelectionModel * selectionModel)
{
    selections = selectionModel; //为 selections 赋值
}

```

实现了以上几个函数，柱状图 View 已经能正确显示表格的统计数据，并且对表格中某项进行修改时，会及时将变化反映在柱状图中。

接下来实现的工作是对选择项的更新。

selectionChanged() 槽函数，当数据项选择发生变化时此函数会响应，重定义此函数，只需在数据项发生变化时调用 update() 函数重画绘图设备即可。



```
void HistogramView::selectionChanged(const QItemSelection & selected,
                                     const QItemSelection & deselected)
{
    viewport()->update();
}
```

`selectionChanged()`函数是将其他 View 中的操作引起的数据项选择变化反映到自身 View 的显示中。下面实现的几个函数的作用是将自身 View 中的操作引起的数据项选择变化反映出来。

本实例中的柱状图可被鼠标单击选择，选中后以不同的方式显示。因此重定义鼠标按下事件函数，在函数调用 `setSelection()` 函数确定鼠标单击点是否在某个数据项的区域内，并设置选择项。具体代码如下：

```
void HistogramView::mousePressEvent(QMouseEvent *e)
{
    QAbstractItemView::mousePressEvent(e);
    setSelection(QRect(e->pos().x(), e->pos().y(), 1, 1),
                 QItemSelectionModel::SelectCurrent);
}
```

`setSelection()` 函数调用有两个参数，一个是 `QRect`，另一个是 `QItemSelectionModel::SelectionFlags`。它们的作用是把位于 `QRect` 内的数据项按 `SelectionFlags` 指定的方式进行更新。`SelectionFlags` 描述了被选择的数据项以何种方式进行更新，`QItemSelectionModel` 类提供了多种可用的 `SelectionFlags`，常用的如 `QItemSelectionModel::Select`、`QItemSelectionModel::Current` 等。

```
void HistogramView::setSelection ( const QRect &rect,
                                   QItemSelectionModel::SelectionFlags flags )
{
1   int rows = model()->rowCount(rootIndex());
2   int columns = model()->columnCount(rootIndex());
3   QModelIndex selectedIndex;
4
4   for (int row = 0; row < rows; ++row)
5   {
5       for (int column = 1; column < columns; ++column)
6       {
6           QModelIndex index = model()->index(row, column, rootIndex());
7           QRegion region = itemRegion(index);
8           if (!region.intersected(contentsRect).isEmpty())
9               selectedIndex = index;
}
```

```

    }

10   if(selectedIndex.isValid())
11       selections->select(selectedIndex,flags);
12 else
13 {
14     QModelIndex noIndex;
15     selections->select(noIndex, flags);
16 }
}

```

第1、2行获取总行数和总列数。

第3行新建一个 QModelIndex 对象 selectedIndex，用于保存被选中的数据项 Index 值。本实例只实现用鼠标单击选择，而没有实现鼠标拖动框选，因此鼠标动作同时只可能选中一个数据项，若需实现框选，则可用一个 QModelIndexList 来保存所有被选中的数据项 Index。

第4~9行确定在 rect 中是否含有数据项，采用遍历的方式把每个数据项的区域与 rect 区域进行 intersected 操作，获得两者之间的交集，若此交集不为空则说明此数据项被选中，把它的 Index 值赋给 selectedIndex。其中，第7行的 itemRegion() 函数返回指定 Index 的数据项所占用的区域。

第10~14行完成对 QItemSelectionModel 的 select() 函数调用，此函数是在实现 setSelection() 函数时必须调用的，它完成最后对选择项的设置工作。

```

QRegion HistogramView::itemRegion(QModelIndex index)
{
    QRegion region;
    if (index.column() == 1)           //male
        region = listRegionM[index.row()];

    if (index.column() == 2)           //female
        region = listRegionF[index.row()];

    if (index.column() == 3)           //retire
        region = listRegionS[index.row()];

    return region;
}

```

```
QModelIndex HistogramView::indexAt(const QPoint &point) const
```



```
{
    QPoint newPoint(point.x(), point.y());

    QRegion region;
    foreach(region, listRegionM) // Male Column
    {
        if(region.contains(newPoint))
        {
            int row = listRegionM.indexOf(region);
            QModelIndex index = model()->index(row, 1, rootIndex());
            return index;
        }
    }
}
```

检查当前点是否处于第 1 列数据的区域中。

```
foreach(region, listRegionF) // Female Column
{
    if(region.contains(newPoint))
    {
        int row = listRegionF.indexOf(region);
        QModelIndex index = model()->index(row, 2, rootIndex());
        return index;
    }
}
```

检查当前点是否处于第 2 列数据的区域中。

```
foreach(region, listRegionS) // Retire Column
{
    if(region.contains(newPoint))
    {
        int row = listRegionS.indexOf(region);
        QModelIndex index = model()->index(row, 3, rootIndex());
        return index;
    }
}
```

检查当前点是否处于第 3 列数据的区域中。

```
return QModelIndex();
}
```

此函数当鼠标在 View 中单击或位置发生改变时被触发，它返回鼠标所在点的



QModelIndex 值，若鼠标处在某个数据项的区域中，则返回此数据项的 Index 值，否则返回一个空 Index。

在主程序中使用自定义的 View。

```
void MainWindow::setupModel()
{
    model = new QStandardItemModel(4,4,this);
    model->setHeaderData(0,Qt::Horizontal,tr("department"));
    model->setHeaderData(1,Qt::Horizontal,tr("male"));
    model->setHeaderData(2,Qt::Horizontal,tr("female"));
    model->setHeaderData(3,Qt::Horizontal,tr("retire"));

}
```

首先新建一个 Model，并设置表头数据。

```
void MainWindow::setupView()
{
    .....
1   QTableView *table = new QTableView;
2   HistogramView *histogram = new HistogramView(splitter);

3   table->setModel(model);
4   histogram->setModel(model);

5   QItemSelectionModel *selectionModel = new QItemSelectionModel(model);
6   table->setSelectionModel(selectionModel);
7   histogram->setSelectionModel(selectionModel);

8   connect(selectionModel,
             SIGNAL(selectionChanged(const QItemSelection, const QItemSelection)),
             histogram,
             SLOT(selectionChanged(const QItemSelection, const QItemSelection)));
9   connect(selectionModel,
             SIGNAL(selectionChanged(const QItemSelection, const QItemSelection)),
             table,
             SLOT(selectionChanged(const QItemSelection, const QItemSelection)));
    .....
}
```

第 1、2 行新建一个 QTableView 和一个 HistogramView 对象。

第 3、4 行为两个 View 对象设置相同的 Model。



第 5~7 行新建一个 QItemSelectionModel 对象作为两个 View 对象共同使用的选 择模型。

第 8 行连接选择模型的 selectionChanged() 信号与柱状图 HistogramView 对象的 selectionChanged() 槽函数，以使 QTableView 对象中的选择变化能反映到 HistogramView 对象的显示中。

第 9 行连接选择模型的 selectionChanged() 信号与 QTableView 对象的 selectionChanged() 槽函数，以使 HistogramView 对象中的选择变化能反映到 TableView 对象的显示中。

第8章 网络与通信

网络编程在应用程序开发中占据着重要的地位。Linux 操作系统提供了统一的套接字抽象，通过套接字可以编写不同层次的网络应用，但套接字的使用比较复杂，对于初学者来说不容易上手，为此，Qt4 提供了一个全新的网络模块 QtNetwork，大大降低了网络程序开发的难度。通常的网络编程涉及的协议包括 UDP、TCP、FTP、HTTP 等，这些在 Qt4 中都有相应的类与之对应，通过这些类可以方便快捷地编写网络程序。

本章通过几个简单的实例对网络编程中经常涉及的知识点作了详细的介绍，通过这些实例，读者可以对 Qt 网络编程有个大致的了解。

本章包括 5 个实例：

- 获取本机网络信息
- 基于 UDP 的网络广播程序
- 基于 TCP 的网络聊天室程序
- 实现 HTTP 文件下载
- 实现 FTP 上传和下载



实例 47 获取本机网络信息

知识点：

- QHostInfo 的使用
- QNetworkInterface 的使用
- QNetworkAddressEntry 的使用

在网络应用中，开发者经常需要获得本机的网络信息，如主机名、IP 地址、硬件地址等信息，本实例通过 QHostInfo、QNetworkInterface、QNetworkAddressEntry 实现上述功能，如图 8-1 所示。

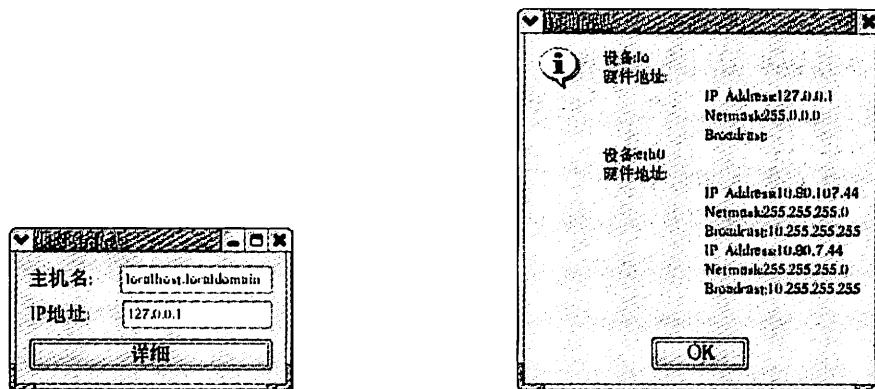


图 8-1 获取本机网络信息

具体实现代码如下：

```
void NetworkInformation::getHostInformation()
{
    1. QString localHostName=QHostInfo::localHostName();
    2. LineEditLocalHostName->setText(localHostName);

    3. QHostInfo hostInfo = QHostInfo::fromName(localHostName);
    4. QList<QHostAddress> listAddress = hostInfo.addresses();
    5. if(!listAddress.isEmpty())
    6. {
        7.     LineEditAddress->setText(listAddress.first().toString());
    }
}
```

```
8  }
}
```

getHostInformation()函数获得主机信息。QHostInfo 提供了一系列有关网络信息的静态函数，可以根据主机名获得分配的 IP 地址，也可以根据 IP 地址获得相应的主机名。

第 1、2 行通过 localHostName()获得本机主机名。

第 3 行根据主机名获得有关主机信息，包括 IP 地址等，QHostInfo::fromName()函数通过主机名查找 IP 地址信息。

第 4~8 行获得主机的 IP 地址列表，该列表可能为空，在不为空的情况下使用第一个 IP 地址。

```
void NetworkInformation::slotDetail()
{
    QString detail="";
    QList<QNetworkInterface> list=QNetworkInterface::allInterfaces();
    for(int i=0;i<list.count();i++)
    {
        QNetworkInterface interface=list.at(i);
        detail = detail+tr("Device:") + interface.name() + "\n";
        QString hardwareAddress=interface.hardwareAddress();
        detail = detail+tr("HardwareAddress:") + interface.hardwareAddress() + "\n";
        QList<QNetworkAddressEntry> entryList=interface.addressEntries();
        for(int j=0;j<entryList.count();j++)
        {
            QNetworkAddressEntry entry=entryList.at(j);
            detail = detail + "\t" + tr("IP Address:") + entry.ip().toString() + "\n";
            detail = detail + "\t" + tr("Netmask:") + entry.netmask().toString() + "\n";
            detail = detail + "\t" + tr("Broadcast:") + entry.broadcast().toString() + "\n";
        }
    }
    QMessageBox::information(this,tr("Detail"),detail);
}
```

slotDetail()函数获得与网络接口有关的信息。QNetworkInterface 类提供了一个主机 IP 地址和网络接口的列表，name()方法可以获得网络接口的名称，hardwareAddress()方法可以获得网络接口的硬件地址，每个网络接口包括 0 个或多个 IP 地址，每个 IP 地址有选择性地与一个子网掩码和（或）一个广播地址相关联。这样的一个列表可以通过 QNetworkInterface 的 addressEntries()方法获得。QNetworkAddressEntry 类存储了被网络接口所支持的一个 IP 地址，同时还有与之相关的子网掩码和广播地址。



实例 48 基于 UDP 的网络广播程序

知识点：

- ☛ UDP 通信协议
- ☛ QUdpSocket 的使用

网络信息广播是网络应用中的一个常用功能，本实例通过 QUdpSocket 实现基于 UDP 协议的广播应用，如图 8-2 所示。

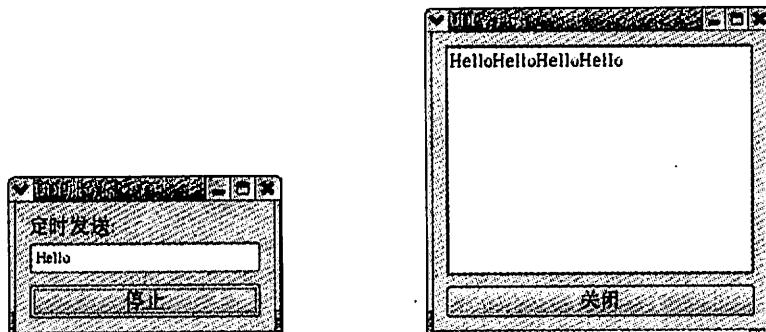


图 8-2 基于 UDP 的网络广播程序

UDP 是一种轻量级、不可靠、面向数据报、面向无连接的传输层协议，可以应用在可靠性不是十分重要的场合，如短消息、广播信息等。

在本实例中，服务器端实现一个定时器，定时向网络的某个端口广播信息，只要连接到该端口的客户程序都可以收到信息，如图 8-3 所示。



图 8-3 UDP 通信协议

具体代码如下所示。

服务器端：

```
UdpServer::UdpServer(QWidget *parent, Qt::WindowFlags f)
```

```
1   : QDialog( parent, f )
2   {
3     setWindowTitle(tr("UDP Server"));
4
5     QVBoxLayout *vbMain = new QVBoxLayout( this );
6
7     LabelTimer = new QLabel( this );
8     LabelTimer->setText(tr("Timer:"));
9     vbMain->addWidget( LabelTimer );
10
11    LineEditText = new QLineEdit(this);
12    vbMain->addWidget( LineEditText );
13
14    PushButtonStart = new QPushButton( this );
15    PushButtonStart->setText( tr( "Start" ) );
16    vbMain->addWidget( PushButtonStart );
17
18    connect(PushButtonStart,SIGNAL(clicked()),this,SLOT(PushButtonStart_clicked()));
19
20    port = 5555;
21
22    isStarted=false;
23
24    udpSocket = new QUdpSocket(this);
25
26    timer = new QTimer(this);
27    connect(timer,SIGNAL(timeout()),this,SLOT(timeout()));
28 }
```

第 1 行设置窗体的标题。

第 2~11 行初始化各个控件并设置布局。

第 12 行设置 UDP 的端口号参数，server 定时向此端口发送广播信息。

第 14 行创建一个 QUdpSocket。

第 15、16 行创建一个定时器，定时发送广播信息。

```
void UdpServer::timeout()
{
    QString msg = LineEditText->text();
    int length = 0;
    if(msg == "")
    {
        return;
    }
}
```

```

        }
        if((length=udpSocket->writeDatagram(
            msg.toLatin1(),msg.length(),QHostAddress::Broadcast, port))!=msg.length())
        {
            return ;
        }
    }
}

```

在服务器端的定时器响应槽中，向端口发送广播信息，`QHostAddress::Broadcast` 指定向广播地址发送。

客户端：

```

UdpClient::UdpClient( QWidget *parent, Qt::WindowFlags f )
    : QDialog( parent, f )
{
    1   setWindowTitle(tr("UDP Client"));

    2   QVBoxLayout *vbMain = new QVBoxLayout( this );
    3   QTextEditReceive = new QTextEdit( this );
    4   vbMain->addWidget( QTextEditReceive );
    5   PushButtonClose = new QPushButton( this );
    6   PushButtonClose->setText( tr( "Close" ) );
    7   vbMain->addWidget( PushButtonClose );
    8   connect(PushButtonClose,SIGNAL(clicked()),this,SLOT(PushButtonClose_clicked()));
    9   port = 5555;

    10  udpSocket = new QUdpSocket(this);
    11  connect(udpSocket, SIGNAL(readyRead()),this, SLOT(dataReceived()));

    12  bool result=udpSocket->bind(port);
    13  if(!result)
    14  {
    15      QMessageBox::information(this,tr("error"),tr("udp socket create error!"));
    16      return;
    17  }
}

```

第 1 行设置窗体的标题。

第 2~8 行初始化各个控件并设置布局。

第 9 行设置 UDP 的端口号参数，指定在此端口上监听数据。

第 10、11 行创建一个 `QUdpSocket`，并连接 `readyRead()` 信号，`readyRead()` 是 `QIODevice` 的信号，`QUdpSocket` 也是一个 I/O 设备，从 `QIODevice` 继承而来，当有数据到达 I/O 设

备时，发出 readyRead()信号。

第 12 行绑定到指定的端口上。

dataReceived()函数响应 QUpdSocket 的 readyRead()信号，一旦 UdpSocket 对象中有数据报可读时，即通过 readDatagram()方法将数据读出并显示。具体代码如下：

```
void UdpClient::dataReceived()
{
    1   while (udpSocket->hasPendingDatagrams())
    2   {
    3       QByteArray datagram;
    4       datagram.resize(udpSocket->pendingDatagramSize());

    5       udpSocket->readDatagram(datagram.data(), datagram.size());

    6       QString msg=datagram.data();
    7       TextEditReceive->insertPlainText(msg);
    }
}
```

第 1 行判断 UdpSocket 中是否有数据报可读，hasPendingDatagrams()方法当至少有一个数据报可读时返回 true，否则返回 false。

第 3~5 行读取第一个数据报，pendingDatagramSize()可以获得第一个数据报的长度。

第 6、7 行显示数据内容。

实例 49 基于 TCP 的网络聊天室程序

知识点：

- TCP 通信协议
- QTcpSocket 的使用

网络聊天是网络应用中一种常见的功能，本实例通过 Qt 提供的 QTcpServer 和 QTcpSocket 实现一个简单的网络聊天室。

实现的效果如图 8-4 所示。

TCP 是一种可靠的、面向连接、面向数据流的传输协议，多数高层网络协议都使用 TCP 协议，包括 HTTP 和 FTP，TCP 协议非常适合数据的连续传输，如图 8-5 所示。

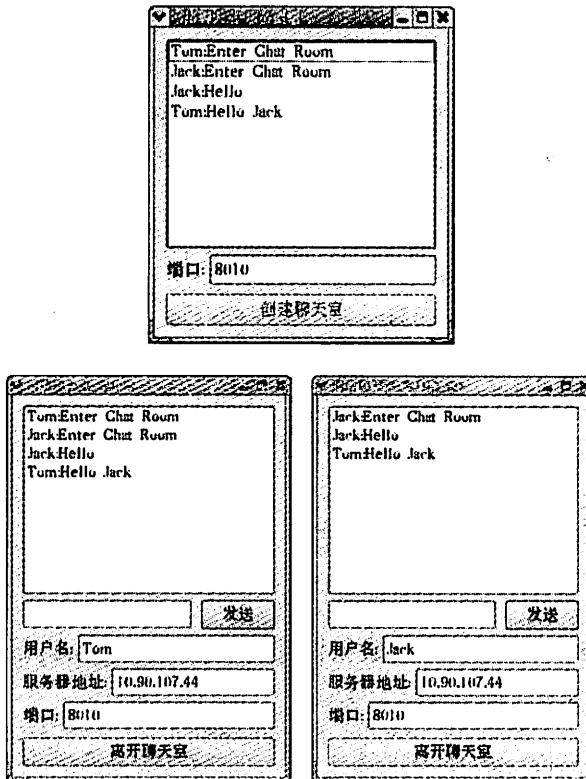


图 8-4 基于 TCP 的网络聊天室程序

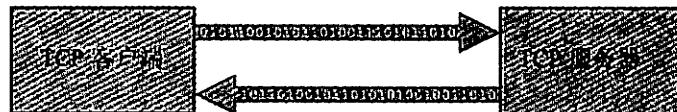


图 8-5 TCP 通信协议

本实例分别实现了网络聊天室的服务器程序和客户端程序，服务器程序可以创建一个聊天室，客户端程序可以输入登录的用户名、服务器地址以及使用的端口号，然后进入聊天室，聊天室中的每一位用户均可以看到发布的消息。

服务器端：

在服务器端实现了 3 个类，其中，Server 继承自 QTcpServer，实现一个 TCP 协议的服务器；TcpClientSocket 继承自 QTcpSocket，实现一个 TCP 套接字；TcpServer 是一个 QDialog，负责服务器端的对话框显示与控制。各个类之间的信号/槽连接情况如图 8-6 所示。

所示。

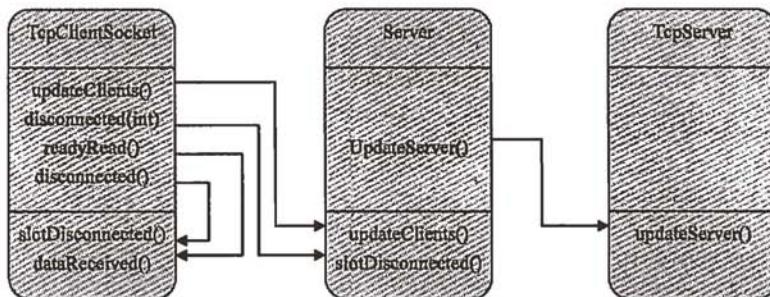


图 8-6 各个类之间的信号/槽连接情况

具体实现代码如下：

```
tcpclientsocket.h:
class TcpClientSocket : public QTcpSocket
{
    Q_OBJECT
public:
    TcpClientSocket( QObject *parent=0 );
    ~TcpClientSocket();
signals:
    void updateClients(QString,int);
    void disconnected(int);
protected slots:
    void dataReceived();
    void slotDisconnected();
};
```

TcpClientSocket 类实现的是一个 TCP 套接字，在本实例中的作用是在服务器端实现与客户端程序的通信。

实现文件 tcpclientsocket.cpp 如下：

```
TcpClientSocket::TcpClientSocket( QObject *parent )
{
    connect(this, SIGNAL(readyRead()),this, SLOT(dataReceived()));
    connect(this, SIGNAL(disconnected()),this, SLOT(slotDisconnected()));
}
```

在 TcpClientSocket 的构造函数中主要指定了信号与槽的连接关系。readyRead()是 QIODevice 的 signal，由 QTcpSocket 继承而来。QIODevice 所有输入/输出设备的一个抽象类，其中定义了基本的接口，在 Qt 中，QTcpSocket 也被看作是一个 QIODevice，readyRead()信号在有数据到来时发出。disconnected()信号在断开连接时发出。

```
void TcpClientSocket::dataReceived()
{
    while (bytesAvailable()>0)
    {
        char buf[1024];
        int length=bytesAvailable();
        read(buf, length);
        QString msg=buf;
        emit updateClients(msg,length);
    }
}
```

当有数据到来时，触发 dataReceived() 函数，从套接字中将有效数据取出，然后发出 updateClients() 信号，updateClients 信号是通知服务器向聊天室内的所有成员广播信息。

```
server.h
#include <QtNetwork>
#include "tcpclientsocket.h"
class Server : public QTcpServer
{
    Q_OBJECT
public:
    Server(QObject *parent = 0,int port=0);
    QList<TcpClientSocket*> tcpClientSocketList;
signals:
    void updateServer(QString,int);
public slots:
    void updateClients(QString,int);
    void slotDisconnected(int);
protected:
    void incomingConnection(int socketDescriptor);
};
```

Server 实现的是一个 TCP 服务器类，继承自 QTcpServer，QTcpServer 类提供了一种基于 TCP 协议的服务器。利用 QTcpServer，开发者可以监听到指定端口的 TCP 连接。在 Server 中定义了一个 QList 变量 tcpClientSocketList，用来保存与每一个客户端连接的



TcpClientSocket。

实现文件：server.cpp

```
Server::Server(QObject *parent,int port)
    : QTcpServer(parent)
{
    listen(QHostAddress::Any,port);
}
```

Server 类的构造函数，在指定的端口对任意地址进行监听。QHostAddress 定义了几种特殊的 IP 地址，如 QHostAddress::Null 表示一个空地址，QHostAddress::LocalHost 表示 IPv4 的本机地址 127.0.0.1，QHostAddress::LocalHostIPv6 表示 IPv6 的本机地址，QHostAddress::Broadcast 表示广播地址 255.255.255.255，QHostAddress::Any 表示 IPv4 的任意地址 0.0.0.0，QHostAddress::AnyIPv6 表示 IPv6 的任意地址。

当出现一个新的连接时，QTcpServer 触发 incomingConnection() 函数，参数 socketDescriptor 指定了连接的 socket 描述符。

```
void Server::incomingConnection(int socketDescriptor)
{
    1   TcpClientSocket *tcpClientSocket = new TcpClientSocket(this);
    2   connect(tcpClientSocket,SIGNAL(updateClients(QString,int)),
              this,SLOT(updateClients(QString,int)));
    3   connect(tcpClientSocket,SIGNAL(disconnected(int)),
              this,SLOT(slotDisconnected(int)));
    4   tcpClientSocket->setSocketDescriptor(socketDescriptor);
    5   tcpClientSocketList.append(tcpClientSocket);
}
```

第 1 行创建一个新的 TcpClientSocket 与客户端通信。

第 2、3 行连接 TcpClientSocket 的 updateClients 信号和 disconnected 信号。

第 4 行将新创建的 TcpClientSocket 的套接字描述符指定为参数 socketDescriptor。

第 5 行将 tcpClientSocket 加入保存列表。

updateClients() 函数的作用是将任意客户端发来的信息进行广播，保证聊天室的所有客户均可以看到其他人的发言。

```
void Server::updateClients(QString msg,int length)
{
    1   emit updateServer(msg,length);
    2   for(int i=0;i<tcpClientSocketList.count();i++)
    3   {
```

```

4     QTcpSocket *item=tcpClientSocketList.at(i);
5     if(item->write(msg.toLatin1(), length)!=length)
6     {
7         continue ;
8     }
9 }
}

```

第 1 行发出 updateServer 信号，用来通知服务器对话框更新相应的显示状态。

第 2~9 行实现信息的广播，tcpClientSocketList 中保存了所有与服务器相连的 TcpClientSocket 对象。

slotDisconnected() 函数的作用是从 tcpClientSocketList 列表中将断开连接的 TcpClientSocket 对象删除掉。

```

void Server::slotDisconnected(int descriptor)
{
    for(int i=0;i<tcpClientSocketList.count();i++)
    {
        QTcpSocket *item=tcpClientSocketList.at(i);
        if(item->socketDescriptor ()==descriptor)
        {
            tcpClientSocketList.removeAt(i);
            return;
        }
    }
    return;
}

```

TcpServer 类实现了服务器端的对话框显示与控制。

```

tcpserver.h
class TcpServer : public QDialog
{
    Q_OBJECT
public:
    TcpServer( QWidget *parent=0, Qt::WindowFlags f=0 );
    ~TcpServer();
public:
    QListWidget *ListWidgetContent;
    QLabel* LabelPort;
    QLineEdit* LineEditPort;
    QPushButton* PushButtonCreate;
    int port;
}

```

```

Server *server;
public slots:
void slotCreateServer();
void updateServer(QString,int);
};

tcpserver.cpp
TcpServer::TcpServer( QWidget *parent, Qt::WindowFlags f )
: QDialog( parent, f )
{
    setWindowTitle(tr("TCP Server"));

    QVBoxLayout *vbMain = new QVBoxLayout( this );

    ListWidgetContent = new QListWidget( this );
    vbMain->addWidget( ListWidgetContent );

    QHBoxLayout *hb = new QHBoxLayout( );

    LabelPort = new QLabel( this );
    LabelPort->setText(tr("Port:"));
    hb->addWidget( LabelPort );

    LineEditPort = new QLineEdit(this);
    hb->addWidget( LineEditPort );

    vbMain->addLayout(hb);

    PushButtonCreate = new QPushButton( this );
    PushButtonCreate->setText( tr( "Create" ) );
    vbMain->addWidget( PushButtonCreate );

    connect(PushButtonCreate,SIGNAL(clicked()),this,SLOT(slotCreateServer()));
    port = 8010;
    LineEditPort->setText(QString::number(port));
}

```

TcpServer 类的构造函数，主要实现窗体各控件的创建、布局以及信息槽的连接等。

```

void TcpServer::slotCreateServer()
{
1   server = new Server(this,port);
2   connect(server,SIGNAL(updateServer(QString,int)),

```



```
    this,SLOT(updateServer(QString,int)));
3   PushButtonCreate->setEnabled(false);
}
```

slotCreateServer()函数的作用是创建一个 TCP 服务器。

第 1 行创建一个 Server 对象。

第 2 行将 Server 对象的 updateServer 信号与相应的槽进行连接。

```
void TcpServer::updateServer(QString msg,int length)
{
    ListWidgetContent->addItem (msg.left(length));
}
```

updateServer()函数的作用是更新服务器端的信息显示。

客户端：

```
void TcpClient::slotEnter()
{
1   if(!status)
2   {
3       QString ip=LineEditServerIP->text();
4       if(!serverIP->setAddress(ip))
5       {
6           QMessageBox::information(this,tr("error"),tr("server ip address error!"));
7           return;
8       }
9       if(LineEditUser->text()=="")
10      {
11          QMessageBox::information(this,tr("error"),tr("User name error!"));
12          return ;
13      }
14      userName=LineEditUser->text();

15      tcpSocket = new QTcpSocket(this);
16      connect(tcpSocket,SIGNAL.connected(),this,SLOT(slotConnected()));
17      connect(tcpSocket,SIGNAL(disconnected()),this,SLOT(slotDisconnected()));
18      connect(tcpSocket, SIGNAL(readyRead()),this, SLOT(dataReceived()));

19      tcpSocket->connectToHost (*serverIP, port);
20      status=true;
21  }
22  else
```

```

23  {
24      int length = 0;
25
26      QString msg=userName+tr(":Leave Chat Room");
27      if((length=tcpSocket->write(msg.toLatin1(),msg.length())!=msg.length())
28      {
29          return ;
30      }
31      tcpSocket->disconnectFromHost();
32
33      status=false;
34  }
}

```

slotEnter()函数实现进入和离开聊天室的功能。

第1行 status 表示当前的状态, true 表示已经进入聊天室, false 表示已经离开聊天室。这里根据 status 的状态决定是执行“进入”还是“离开”的操作。

第3~14行完成输入合法性检验, !serverIP->setAddress()函数可以用来判断给定的 IP 地址是否能够被正确解析。

第15~18行创建一个 QTcpSocket 类对象, 并将信号/槽连接起来。

第19行与 TCP 服务器端连接, 连接成功后发出 connected()信号。

第25~29行构造一条离开聊天室的消息, 并通知服务器端。

第30行与服务器断开连接, 断开连接后发出 disconnected()信号。

第31行将 status 状态复位。

slotConnected()函数为 connected()信号的响应槽, 当与服务器连接成功后, 客户端构造一条进入聊天室的消息, 并通知服务器。具体代码如下:

```

void TcpClient::slotConnected()
{
    int length = 0;
    PushButtonSend->setEnabled( true );
    PushButtonEnter->setText(tr("Leave"));

    QString msg=userName+tr(":Enter Chat Room");
    if((length=tcpSocket->write(msg.toLatin1(),msg.length())!=msg.length())
    {
        return ;
    }
}

```



当有数据到来时，触发 `dataReceived()` 函数，从套接字中将有效数据取出，并显示。具体代码如下：

```
void TcpClient::dataReceived()
{
    while (tcpSocket->bytesAvailable()>0)
    {
        QByteArray datagram;
        datagram.resize(tcpSocket->bytesAvailable());
        QHostAddress sender;
        tcpSocket->read(datagram.data(), datagram.size());
        QString msg=datagram.data();
        ListWidgetContent->addItem (msg.left(datagram.size()));
    }
}
```

实例 50 实现 HTTP 文件下载

知识点：

- HTTP 通信协议
- QHttp 实现文件下载

文件下载是网络应用的常用功能，本实例实现 HTTP 协议的文件下载功能，如图 8-7 所示。

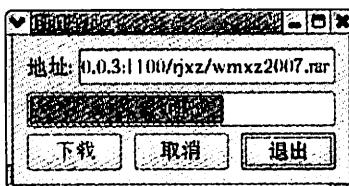


图 8-7 HTTP 文件下载

超文本传输协议 HTTP 是一个应用级的网络协议，主要用于下载 HTML 和 XML 文件，也可用于其他类型数据的高层下载协议，和 FTP 一样，都运行在 TCP 之上。HTTP 只使用一个 TCP 连接来处理请求、响应和文件的传输，如图 8-8 所示。

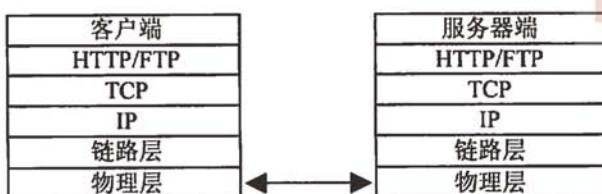


图 8-8 HTTP 通信协议

Qt 提供的 QHttp 类用于构建 HTTP 客户端，提供了许多常用的操作，QHttp 采用异步工作方式，如果某个操作不能立即执行完，函数仍然会马上返回，继续执行下一条指令，对执行结果的报告通过信号 signals 的方式进行。

```
HttpClient::HttpClient( QWidget *parent, Qt::WindowFlags f )
: QDialog( parent, f )
{
    /*控件初始化与布局*/
    .....
1   httpClient = new QHttp(this);

2   connect(httpClient, SIGNAL(requestFinished(int, bool)),
          this, SLOT(httpRequestFinished(int, bool)));
3   connect(httpClient, SIGNAL(dataReadProgress(int, int)),
          this, SLOT(httpDataReadProgress(int, int)));
4   connect(httpClient, SIGNAL(responseHeaderReceived (const QHttpResponseHeader &)),this,
          SLOT(httpResponse HeaderReceived(const QHttpResponseHeader &)));

5   cancelButton->setEnabled(false);
}

}

```

第 1 行创建一个 QHttp 对象 httpClient。

第 2~4 行连接 httpClient 的信号/槽，其中，requestFinished (int id, bool error) 信号是当某个请求被处理完毕后发出，参数 id 表示当前请求的标识；dataReadProgress (int done, int total) 信号是当对象从 HTTP 服务器端读取数据时发出，用来指示当前传输的进度，参数 done 为已传输的内容，total 为需要传输的总数，在某些时候 total 值可能不能计算出来，此时 total 为 0，在使用进度条显示时需要进行相应处理；responseHeaderReceived(const QHttpResponseHeader & resp) 信号在接收到服务器端的 HTTP 响应头时发出。

第 5 行设置取消按钮状态为不使能。

slotDownload() 实现文件的下载：



```
void HttpClient::slotDownload()
{
    1   QUrl url(urlLineEdit->text());
    2   QFileInfo fileInfo(url.path());
    3   QString fileName = fileInfo.fileName();
    4   if ( QFile::exists(fileName))
    5   {
    6       QMessageBox::information(this, tr("Error"),tr("File already exists!"));
    7       return;
    8   }
    9   file = new QFile(fileName);
10   if (!file->open(QIODevice::WriteOnly))
11   {
12       QMessageBox::information(this, tr("Error"),tr("Save file error!"));
13       delete file;
14       file = 0;
15       return;
16   }

17   httpClient->setHost(url.host(), url.port() != -1 ? url.port() : 80);
18   if (!url.userName().isEmpty())
19   {
20       httpClient->setUser(url.userName(), url.password());
21   }

22   httpRequestAborted = false;
23   requestId = httpClient->get(url.path(), file);

24   progressBar->reset();

25   downloadPushButton->setEnabled(false);
26   cancelPushButton->setEnabled(true);
}
```

第 1 行根据用户输入的需要下载的文件得到 QUrl 类型的值 url。

第 2、3 行根据 url 得到文件的有关信息。

第 4~8 行判断该文件名是否在当前路径下已经存在，若存在，则报错返回。

第 9~16 行在本地以只写方式创建一个文件。

第 17 行通过 QHttp 的 setHost()方法设置服务器的主机名和端口号，如果 url 中未指定端口号，则使用默认的 80 端口。

第 18~21 行如果用户输入的 url 包含用户名信息，则通过 QHttp 的 setUser()方法向服

服务器端发送用户名和密码信息，以获得认证。

第22行设置请求中断标志。

第23行通过QHttp的get()方法向服务器发送下载文件请求。

第24行复位进度条。

第25、26行设置按钮的属性，以区分当前的操作状态。

httpRequestFinished()槽处理请求结束的响应：

```
void HttpClient::httpRequestFinished(int id, bool error)
{
    1   if (httpRequestAborted)
    2   {
    3       if (file)
    4       {
    5           file->close();
    6           file->remove();
    7           delete file;
    8           file = 0;
    9       }
    10      progressBar->reset();
    11      return;
    12  }

    13 if (id == requestId)
    14 {
    15     progressBar->reset();
    16     file->close();
    17     if (error)
    18     {
    19         file->remove();
    20         QMessageBox::information(this, tr("Error"), tr("Download failed!"));
    21     }

    22     downloadPushButton->setEnabled(true);
    23     cancelPushButton->setEnabled(false);
    24     delete file;
    25     file = 0;
    26  }
}
```

第1~12行判断如果请求终端标志被置位，则表明下载中断，删除未完成的文件。

第13~26行判断当前的请求是否是下载请求，如果是则表明下载完成，复位进度条，



参数 error 指示下载过程中是否有错误发生。

httpDataReadProgress()响应传输进度状态指示，主要完成进度条的更新。具体代码如下：

```
void HttpClient::httpDataReadProgress(int done, int total)
{
    progressBar->setMaximum(total);
    progressBar->setValue(done);
}
```

httpResponseHeaderReceived()用于接收到服务器端的响应头时的处理。参数 responseHeader 包含了 HTTP 服务器的响应头信息，statusCode()可以获得服务器端的响应码，如 200 表示“请求成功”，404 表示“请求资源未找到”等，具体的值可以参考 RFC 1945 定义。具体代码如下：

```
void HttpClient::httpResponseHeaderReceived(const QHttpResponseHeader &responseHeader)
{
    if (responseHeader.statusCode() != 200)
    {
        QMessageBox::information(this, tr("Error"), tr("Download error!"));
        httpRequestAborted = true;
        progressBar->reset();
        httpClient->abort();
        return;
    }
}
```

slotCancel()中断下载，置位中断请求标志，并对按钮的状态进行设置。具体代码如下：

```
void HttpClient::slotCancel()
{
    httpRequestAborted = true;
    httpClient->abort();
    downloadPushButton->setEnabled(true);
    cancelPushButton->setEnabled(false);
}
```

实例 51 实现 FTP 上传和下载

知识点：

- ❑ FTP 通信协议
- ❑ QFtp 实现文件下载

本实例实现基于 FTP 协议的文件上传和下载，如图 8-9 所示。



图 8-9 实现 FTP 上传和下载

在实例中，输入服务器地址以及登录的用户名和密码，单击“登录”按钮后开始与服务器进行连接，如果连接上并登录成功，“登录”按钮变成不可单击，同时使能“上传”和“下载”功能，当选择“上传”或“下载”，输入需要上传或下载的文件名，即可实现 FTP 文件的上传或下载。

QFtp 提供了基于 FTP 协议的客户端功能，QFtp 以异步方式进行工作，如果某个操作不能立即执行完，函数仍然会马上返回，继续执行下一条指令，对执行结果的报告通过信号 signals 的方式进行，可以进行此类预定方式的函数包括 connectToHost()、login()、close()、list()、cd()、get()、put()、remove()、mkdir()、rmdir()、rename() 和 rawCommand()。

具体实现代码如下：

```
class FtpClient : public QDialog
{
    Q_OBJECT
public:
    FtpClient( QWidget *parent=0, Qt::WindowFlags f=0 );
    ~FtpClient();
public:
```



```
QLabel* LabelServer;
QLineEdit* LineEditServer;
QLabel* LabelUser;
QLineEdit* LineEditUser;
QLabel* LabelPassword;
QLineEdit* LineEditPassword;
QPushButton* PushButtonLogin;
QPushButton* PushButtonPut;
QPushButton* PushButtonGet;

enum STATUS{None,LOGIN,PUT,GET};
STATUS curStatus;

QFtp *ftpClient;
QFile *localFile;

public slots:
void slotLogin();
void slotPut();
void slotGet();
void slotStateChanged ( int state );
void slotDone ( bool error );
};
```

类的声明中定义了程序所使用的各种控件以及方法，enum STATUS 定义了当前 FTP 的状态，如登录、上传、下载状态，用来指示成功后的状态显示。

slotLogin()函数实现 FTP 服务器的登录。具体代码如下：

```
void FtpClient::slotLogin()
{
1   QString serverAddress = LineEditServer->text();
2   if(serverAddress.isEmpty())
3   {
4       QMessageBox::warning(this,tr("error"),tr("Please input server address!"));
5       return;
6   }
7   QString userName = LineEditUser->text();
8   if(userName.isEmpty())
9   {
10      QMessageBox::warning(this,tr("error"),tr("Please input user name!"));
11      return;
12  }
13  QString password = LineEditPassword->text();
```

```

14  ftpClient=new QFtp();
15  ftpClient->connectToHost(serverAddress);

16  connect(ftpClient, SIGNAL(stateChanged ( int )), this, SLOT(slotStateChanged ( int )));
17  connect(ftpClient, SIGNAL(done ( bool )), this, SLOT(slotDone ( bool )));

18  ftpClient->login(userName,password);
19  curStatus=LOGIN;
}

```

第 1~13 行进行输入数据的获取和校验。

第 14 行创建一个 QFtp 对象。

第 15 行通过 connectToHost()方法连接 FTP 服务器端。

第 16、17 行连接信号和槽，done (bool error)信号在最后一条命令处理完毕后发出，参数 error 指示是否有错误发生，如果为 true，表示有错误发生，错误信息可以通过 errorString()方法得到。stateChanged(int state)信号在连接状态发生变化时发出，参数 state 表示新的连接状态，可以是以下值。

- QFtp::Unconnected: 主机未连接。
- QFtp::HostLookup: 正在查询主机。
- QFtp::Connecting: 正在尝试连接主机。
- QFtp::Connected: 已成功连接主机。
- QFtp::LoggedIn: 已成功登录。
- QFtp::Closing: 连接正在断开（断开后的状态是 Unconnected）。

第 18 行以输入的用户名和密码登录服务器。

第 19 行设置当前的状态是“登录”状态。

这里只判断 QFtp::LoggedIn 状态，如果已成功登录，则使能“上传”和“下载”按钮。具体代码如下：

```

void FtpClient::slotStateChanged ( int state )
{
    if(state == QFtp::LoggedIn)
    {
        PushButtonPut->setEnabled(true);
        PushButtonGet->setEnabled(true);
    }
}

```

slotPut()是上传函数，根据用户输入的文件名上传文件，QFtp 的 put()方法实现文件的上传。put()方法的原型如下：

```
int put ( QIODevice * dev, const QString & file, TransferType type = Binary )
```

参数 dev 是一个 QIODevice 指针，在这里创建了一个 QFile 对象，file 参数指定文件名，type 参数指定传输类型，可以是 Binary 和 Ascii，默认采用二进制 Binary 方式。

```
void FtpClient::slotPut()
{
    bool ok;
    QString fileName = QInputDialog::getText(this, tr("Put File:"), tr("Please input file name:"), QLineEdit::Normal, QString(), &ok);
    if(ok && !fileName.isEmpty())
    {
        QFile *remoteFileName=new QFile(fileName);
        ftpClient->put(remoteFileName, fileName);
    }
    curStatus=PUT;
}
```

slotGet()是下载方法，QFtp 的 get()方法实现文件的下载。get()方法的原型如下：

```
int get ( const QString & file, QIODevice * dev = 0, TransferType type = Binary )
```

参数 file 指定保存在本地的文件名，dev 为一个 QIODevice 指针，这里是一个 QFile 对象指针，type 指定传输类型。

```
void FtpClient::slotGet()
{
    bool ok;
    QString fileName = QInputDialog::getText(this, tr("Get File:"), tr("Please input file name:"), QLineEdit::Normal, QString(), &ok);
    if(ok && !fileName.isEmpty())
    {
        localFile=new QFile(fileName);
        localFile->open(QIODevice::WriteOnly);
        ftpClient->get(fileName,localFile);
    }
    curStatus=GET;
}
```

slotDone()方法为响应 done()信号的槽函数。具体代码如下：

```
void FtpClient::slotDone(bool error)
```

```
1 if(error)
2 {
3     QMessageBox::warning(this,tr("error"), ftpClient-> errorString());
4     return;
5 }
6 if(curStatus == LOGIN)
7 {
8     PushButtonLogin->setEnabled(false);
9     curStatus=None;
10}
11 if(curStatus == PUT)
12 {
13     QMessageBox::warning(this,tr("succeed"),tr("Put file succeed!"));
14     curStatus=None;
15}
16 if(curStatus == GET)
17 {
18     localFile->close();
19     QMessageBox::warning(this,tr("succeed"),tr("Get file succeed!"));
20     curStatus=None;
21}
}
```

第1~5行行为处理过程中的错误处理。

第6~10行如果当前状态为“登录”状态，则表示登录成功，将“登录”按钮变成不可用。

第11~15行如果当前状态为“上传”状态，则表示上传成功。

第16~21行如果当前状态为“下载”状态，则表示下载成功。

第9章 磁盘文件

本章主要实现对磁盘文件的处理，通过两个实例分析了 Qt 对文件、目录及其属性的操作。

- 获取文件属性
- 文件浏览器

实例 52 获得文件属性

知识点：

- 文件信息的获取
- 文件路径的有效分离

在对文件进行操作时，经常需要获得文件的相关属性信息，包括文件名、文件大小、创建时间、最后修改时间、最后访问时间以及一些读写属性等，方便用户的使用。QFileInfo 类提供了系统独立的文件信息。

本实例通过 QFileInfo 类实现以上信息的获取，如图 9-1 所示。



图 9-1 获得文件信息

QFileInfo 提供了文件在文件系统中的文件名称与位置等信息，以及文件的权限、目录、文件或符号连接等，也提供文件的大小、创建时间、最后修改时间、最后访问时间等信息。

QFileInfo 可以使用绝对路径和相对路径来指向同一个文件，绝对路径以“/”开头（在 Windows 中以磁盘符号开头），相对路径则以目录名或文件名开头，isRelative()方法可以用来判断 QFileInfo 使用的是绝对路径还是相对路径。makeAbsolute()方法可以用来将相对路径转化为绝对路径。

为了加快执行的效率，QFileInfo 可以将文件信息进行一次读取缓存，这样后续的访问就不需要持续访问文件了，但是由于文件在读取信息之后可能被其他程序或本程序改变属性，因此 QFileInfo 通过 refresh()方法提供了一种刷新机制可以更新文件的信息，用户也可以通过 setCaching()方法关闭这种缓冲功能。



文件的属性可以通过 `isFile()`、`isDir()` 和 `isSymLink()` 获得，对于符号连接，`symLinkTarget()` 方法可以获得符号连接指向的文件名称。

`QFileInfo` 还提供了对文件路径进行有效分离的方法，如下：

```
QFileInfo fi("/tmp/archive.tar.gz");
QString absolutefilepath = fi.absoluteFilePath(); // absolutefilepath ="/tmp/archive.tar.gz"
QString absolutepath = fi.absolutePath(); //absolutepath = "/tmp"
QString basename = fi.baseName(); //basename = "archive"
QString completebasename = fi.completeBaseName(); //completebasename = "archive.tar"
QString completesuffix = fi.completeSuffix(); //completesuffix = "tar.gz"
QString filename = fi.fileName(); //filename = "archive.tar.gz"
QString filepath = fi.filePath(); //filepath = "/tmp/archive.tar.gz"
QString path = fi.path(); //path = "/tmp"
QString suffix = fi.suffix(); //suffix = "gz"
```

文件的日期可以由 `created()`、`lastModified()`、`lastRead()` 等方法获得，文件的存取权限可以由 `isReadable()`、`isWritable()` 和 `isExecutable()` 等方法获得，文件的所有权限可以由 `owner()`、`ownerId()`、`group()`、`groupId()` 等方法获得。`Permission()` 方法可以用来测试一个文件的权限。

相关实现代码如下：

```
void FileInformation::getFileInformation(QString file)
{
    1   QFileInfo info(file);

    2   qint64 size = info.size();
    3   QDateTime created = info.created();
    4   QDateTime lastModified = info.lastModified();
    5   QDateTime lastRead = info.lastRead();
    6   bool isDir = info.isDir();
    7   bool isFile = info.isFile();
    8   bool isSymLink = info.isSymLink();
    9   bool isHidden = info.isHidden();
   10  bool isReadable = info.isReadable();
   11  bool isWritable = info.isWritable();
   12  bool isExecutable = info.isExecutable();

   13  LineEditSize->setText(QString::number(size));
   14  LineEditCreated->setText(created.toString());
   15  LineEditLastModified->setText(lastModified.toString());
   16  LineEditLastRead->setText(lastRead.toString());
   17  CheckBoxIsDir->setCheckState(isDir?Qt::Checked:Qt::Unchecked);
```

```

18 CheckBoxIsFile->setCheckState (isFile?Qt::Checked:Qt::Unchecked);
19 CheckBoxIsSymLink->setCheckState (isSymLink?Qt::Checked:Qt::Unchecked);
20 CheckBoxIsHidden->setCheckState (isHidden?Qt::Checked:Qt::Unchecked);
21 CheckBoxIsReadable->setCheckState (isReadable?Qt::Checked:Qt::Unchecked);
22 CheckBoxIsWritable->setCheckState (isWritable?Qt::Checked:Qt::Unchecked);
23 CheckBoxIsExecutable->setCheckState (isExecutable?Qt::Checked:Qt::Unchecked);
}

```

第1行根据输入参数创建一个QFileInfo对象。

第2~5行获得QFileInfo对象的大小、创建时间、最后修改时间和最后访问时间。

第6~8行判断QFileInfo对象的文件类型属性，包括目录、文件和符号连接。

第9~12行判断QFileInfo对象的读写属性和可执行属性。

第13~23行根据上面的结果更新界面显示。

实例 53 文件浏览器

知识点：

- QDir 显示文件系统目录
- 过滤方式显示文件列表

文件系统的浏览是文件操作的一个常用功能，本实例实现一个文件系统的浏览功能，可以浏览所有的文件，如图9-2所示。



图9-2 文件浏览器

在本实例中，用户可以双击浏览器中显示的目录进入某一级目录，或单击“..”返回上一级目录，顶部的编辑框显示当前所在的目录路径，列表中显示该目录下的所有文件。



具体实现代码如下：

```
void FileViewer::showFileInfoList(QFileInfoList list)
{
    1   ListWidgetFile->clear();
    2   for(unsigned int i=0;i<list.count();i++)
    3   {
    4       QFileInfo tmpFileInfo=list.at(i);
    5       if((tmpFileInfo.isDir()))
    6       {
    7           QIcon icon(":/images/dir.png");
    8           QString fileName=tmpFileInfo.fileName();
    9           QListWidgetItem *tmp=new QListWidgetItem (icon,fileName);
   10           ListWidgetFile->addItem(tmp);
   11       }
   12       else if(tmpFileInfo.isFile())
   13       {
   14           QIcon icon(":/images/file.png");
   15           QString fileName=tmpFileInfo.fileName();
   16           QListWidgetItem *tmp=new QListWidgetItem (icon,fileName);
   17           ListWidgetFile->addItem(tmp);
   18       }
   19   }
}
```

slotShow()函数是负责将 QFileInfoList 对象中的所有项显示在 ListWidgetFile 控件中，ListWidgetFile 是一个 QListWidget 对象，QListWidget 类提供一种基于项目的列表控件。第 1 行先清空列表控件。

第 2~19 行依次从 QFileInfoList 对象中取出所有项，按目录和文件两种方式加入到列表控件中。

```
void FileViewer::slotShow(QDir dir)
{
    QStringList string;
    string << "*";
    QFileInfoList list=dir.entryInfoList (string,QDir::AllEntries,QDir::DirsFirst);
    showFileInfoList(list);
}
```

slotShow()函数负责显示目录 dir 下的所有文件，QDir 的 entryInfoList()方法是按照某种过滤方式获得目录下的文件列表，函数原型如下：

```
QFileInfoList QDir::entryInfoList ( const QStringList & nameFilters, Filters filters = NoFilter, SortFlags sort = NoSort ) const
```

`nameFilters` 参数指定了文件名的过滤方式，如“*”，“*.tar.gz”；`filters` 参数指定文件属性的过滤方式，如目录、文件、读写属性等，`QDir::Filter` 定义了一系列的过滤方式，如表 9-1 所示。

表 9-1 QDir::Filter 定义的过滤方式

过滤方式	说 明
<code>QDir::Dirs</code>	按照过滤方式列出所有目录
<code>QDir::AllDirs</code>	列出所有目录，不考虑过滤方式
<code>QDir::Files</code>	只列出文件
<code>QDir::Drives</code>	列出磁盘驱动器（UNIX 系统无效）
<code>QDir::NoSymLinks</code>	不列出符号连接（对不支持符号连接的操作系统无效）
<code>QDir::NoDotAndDotDot</code>	不列出“.” 和“..”
<code>QDir::AllEntries</code>	列出目录、文件和磁盘驱动器，相当于 <code>Dirs Files Drives</code>
<code>QDir::Readable</code>	列出所有具有“读”属性的文件和目录
<code>QDir::Writable</code>	列出所有具有“写”属性的文件和目录
<code>QDir::Executable</code>	列出所有具有“执行”属性的文件和目录
<code>QDir::Modified</code>	只列出被修改过的文件（UNIX 系统无效）
<code>QDir::Hidden</code>	列出隐藏文件（在 UNIX 系统下，隐藏文件的文件名以“.” 开始）
<code>QDir::System</code>	列出系统文件（在 UNIX 系统下指 FIFO、套接字和设备文件）
<code>QDir::CaseSensitive</code>	文件系统如果区分文件名大小写，则按大小写方式进行过滤

`Sort` 参数指定列表的排序情况，`QDir::SortFlag` 定义了一系列的排序方式，如表 9-2 所示。

表 9-2 QDir::SortFlag 定义的排序方式

排序方式	说 明
<code>QDir::Name</code>	按名称排序
<code>QDir::Time</code>	按时间排序（修改时间）
<code>QDir::Size</code>	按文件大小排序



续表

排序方式	说 明
QDir::Type	按文件类型排序
QDir::Unsorted	不排序
QDir::DirsFirst	目录优先排序
QDir::DirsLast	目录最后排序
QDir::Reversed	反序
QDir::IgnoreCase	忽略大小写方式排序
QDir::LocaleAware	使用当前本地排序方式进行排序

slotShowDir()函数根据用户的选择显示下一级目录的所有文件。具体实现代码如下：

```
void FileViewer::slotShowDir(QListWidgetItem * item)
{
    1   QString str=item->text();
    2   QDir dir;
    3   dir.setPath(LineEditDir->text());
    4   dir.cd(str);
    5   LineEditDir->setText(dir.absolutePath());
    6   slotShow(dir);
}
```

第 1 行先将下一级的目录名保存在 str 中。

第 2 行定义一个 QDir 对象。

第 3 行设置 QDir 对象的路径为当前目录路径。

第 4 行根据下一级目录名重新设置 QDir 对象的路径。

第 5 行刷新显示当前的目录路径，QDir 的 absolutePath()方法获得目录的绝对路径，即以“/”开头的路径名，同时忽略多余的“.”或“..”以及多余的分隔符。

第 6 行显示当前目录下的所有文件。

第 10 章 事件

本章通过 3 个实例分析了 Qt 中的几种常用事件，包括鼠标事件、键盘事件以及事件过滤器的使用方法。

- 获得鼠标事件
- 使用键盘控制移动
- 事件过滤器实现动态图片按钮



实例 54 获得鼠标事件

知识点：

各种鼠标事件的响应方式

鼠标事件包括鼠标的移动，鼠标按下、松开、鼠标双击等。本实例演示如何获得鼠标事件，如图 10-1 所示。

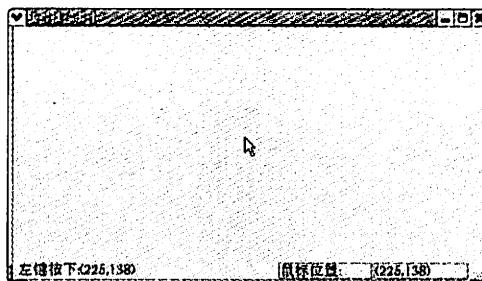


图 10-1 文件浏览器

鼠标事件发生在用户按下、松开鼠标按键或操作鼠标移动时，在本实例中，当用户操作鼠标在特定区域内移动时，状态栏右侧会实时显示当前鼠标的位置信息，当用户按下鼠标按键时，状态栏左侧会显示用户按下的鼠标按键属性，如左键、右键、中键，并显示按下的鼠标位置，当用户松开鼠标时，状态栏左侧会显示鼠标松开的位置。

具体实现代码如下：

```
class MouseEvent : public QMainWindow
{
    Q_OBJECT
public:
    MouseEvent();
    ~MouseEvent();
private:
    QLabel *labelStatus;
    QLabel *labelMousePos;
protected :
    void mouseMoveEvent ( QMouseEvent * e );
    void mousePressEvent ( QMouseEvent * e );
```

```

void mouseReleaseEvent ( QMouseEvent * e );
void mouseDoubleClick( QMouseEvent * e );
};

```

在类的声明中，重定义了 QWidget 类的 3 个鼠标事件方法：mouseMoveEvent、mousePressEvent 和 mouseReleaseEvent，这样当有鼠标事件发生时，就会响应相应的函数。

```

MouseEvent::MouseEvent()
: QMainWindow()
{
1   setWindowTitle(tr("Get Mouse Event"));

2   labelStatus = new QLabel();
3   labelStatus->setText(tr("Mouse Position:"));
4   labelStatus -> setFixedWidth (100);

5   labelMousePos = new QLabel();
6   labelMousePos->setText(tr(""));

7   labelMousePos -> setFixedWidth (100);

8   statusBar()->addPermanentWidget(labelStatus);
9   statusBar()->addPermanentWidget(labelMousePos);

10  this->setMouseTracking (true);
}

```

第 1 行设置窗体的标题。

第 2~4 行创建 QLabel 控件 labelStatus，用来显示鼠标移动时鼠标的位置。

第 5~7 行创建 QLabel 控件 labelMousePos，用来显示鼠标按下或释放时鼠标的位置。

第 8、9 行在 QMainWindow 的状态栏中增加控件。

第 10 行设置窗体追踪鼠标，setMouseTracking(bool enable)设置窗体是否追踪鼠标，默認為 false，不追踪，在此情况下当至少有一个鼠标按键按下时才响应鼠标移动事件，在前面的例子中有很多类似的情况，如画图程序。在本实例中，需要实时显示鼠标的位置，因此设置为 true，追踪鼠标。

mouseMoveEvent() 函数为鼠标移动事件响应函数，QMouseEvent 类的 x() 和 y() 方法可以获得鼠标的相对位置，即相对于应用程序的位置。具体代码如下：

```

void MouseEvent::mouseMoveEvent ( QMouseEvent * e )
{
    labelMousePos ->setText("(" + QString::number(e->x()) + "," + QString::number(e-> y()) + ")");
}

```



 小贴士：QMouseEvent 类的 x() 和 y() 方法可以获得鼠标相对于接收事件的窗体的位置，globalX() 和 globalY() 方法可以获得鼠标相对窗口系统（如 X11）的位置。

mousePressEvent() 函数为鼠标按下事件响应函数，QMouseEvent 类的 button() 方法可以获得发生鼠标事件的按键属性，如左键、右键、中键等。具体代码如下：

```
void MouseEvent::mousePressEvent ( QMouseEvent * e )
{
    QString str= "("+QString::number(e->x())+","+QString::number(e->y())+")";
    if(e->button()==Qt::LeftButton)
    {
        statusBar()->showMessage (tr("Mouse Left Button Pressed:") +str);
    }
    else if(e->button()==Qt::RightButton)
    {
        statusBar()->showMessage (tr("Mouse Right Button Pressed:") +str);
    }
    else if(e->button()==Qt::MidButton)
    {
        statusBar()->showMessage (tr("Mouse Middle Button Pressed:") +str);
    }
}
```

mouseReleaseEvent() 函数为鼠标松开事件响应函数。具体代码如下：

```
void MouseEvent::mouseReleaseEvent ( QMouseEvent * e )
{
    QString str= "("+QString::number(e->x())+","+QString::number(e->y())+")";
    statusBar()->showMessage (tr("Mouser Released:") +str,3000);
}
```

实例 55 使用键盘控制移动

知识点：

键盘事件的应用方法

在图像处理和游戏应用程序中有时需要通过键盘控制某个对象的移动，此功能可以通过对键盘事件的处理实现。本实例通过键盘控制界面上的一个图标，演示通过键盘控

制图标的移动，如图 10-2 所示。

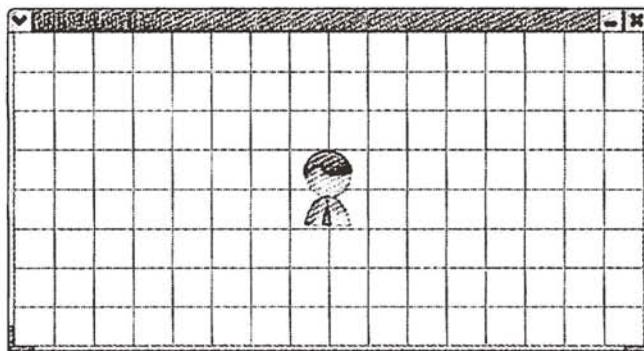


图 10-2 使用键盘控制移动

在实例中通过键盘的上下左右方向键可以控制图标的移动，移动的步进值为网格的大小，如果同时按下 Ctrl 键，则实现细微移动；按下 Home 键回到界面的左上点；按下 End 键达到界面的右下点。

键盘事件的获取是通过重定义 QWidget 的 keyPressEvent() 和 keyReleaseEvent() 来实现的。

```
class KeyEvent : public QWidget
{
    Q_OBJECT
public:
    KeyEvent(QWidget *parent=0);
    ~KeyEvent();
    void drawPix();
    void keyPressEvent(QKeyEvent *);
    void paintEvent(QPaintEvent *);
private:
    QPixmap *pix;
    QImage image;
    int startX;
    int startY;
    int width;
    int height;
    int step;
};
```



类的声明中定义了有关变量和函数，其中*pix 作为一个绘图设备，使用双缓冲机制实现图形的绘制；image 是界面中间的小图标；startX、startY 为图标的左上顶点位置；width、height 为界面的宽度和高度；step 为网格的大小，即移动的步进值。

drawPix()函数实现 在 QPixmap 对象上绘制图像：

```
void KeyEvent::drawPix()
{
    1   pix->fill(Qt::white);
    2   QPainter *painter = new QPainter(pix);
    3   QPen pen(Qt::DotLine);
    4   painter->setPen(pen);
    5   for(int i=step;i<width;)
    6   {
    7       painter->drawLine(QPoint(i,0),QPoint(i,height));
    8       i=i+step;
    9   }
    10  for(int j=step;j<height;)
    11  {
    12      painter->drawLine(QPoint(0,j),QPoint(width,j));
    13      j=j+step;
    14  }
    15  painter->drawImage(QPoint(startX,startY),image);
}
```

第 1 行重新刷新 pix 对象为白色底色。

第 2 行创建一个 QPainter 对象，并指定 pix 为绘图设备。

第 3、4 行创建一个 QPen 对象，设置画笔的线型为 Qt::DotLine，用于绘制网格。

第 5~9 行按照步进值的间隔绘制纵向的网格线。

第 10~14 行按照步进值的间隔绘制横向的网格线。

第 15 行在 pix 对象中绘制可移动的小图标。

keyPressEvent()函数处理键盘的按下事件：

```
void KeyEvent::keyPressEvent(QKeyEvent *event)
{
    1   if(event->modifiers() == Qt::ControlModifier)
    2   {
    3       if(event->key() == Qt::Key_Left)
    4       {
    5           startX=(startX-1<0)?startX:startX-1;
    6       }
    7       if(event->key() == Qt::Key_Right)
```

```
8     {
9         startX=(startX+1+image.width()>width)?startX:startX+1;
10    }
11   if(event->key()==Qt::Key_Up)
12   {
13       startY=(startY-1<0)?startY:startY-1;
14   }
15   if(event->key()==Qt::Key_Down)
16   {
17       startY=(startY+1+image.height()>height)?startY:startY+1;
18   }
19 }
```

第1行判断修饰键 Ctrl 是否被按下，Qt::KeyboardModifier 定义了一系列修饰键，如下所示。

- Qt::NoModifier: 没有修饰键按下。
- Qt::ShiftModifier: Shift 键按下。
- Qt::ControlModifier: Ctrl 键按下。
- Qt::AltModifier: Alt 键按下。
- Qt::MetaModifier: Meta 键按下。
- Qt::KeypadModifier: 小键盘按键按下。
- Qt::GroupSwitchModifier: Mode_switch 键按下 (X11)。

第3~18 行分别根据按下的上下左右方向键调节图标的左上顶点的位置，步进值为 1，即细微移动。

```
20 else
21 {
22     startX=startX-startX%step;
23     startY=startY-startY%step;
24
25     if(event->key()==Qt::Key_Left)
26     {
27         startX=(startX-step<0)?startX:startX-step;
28     }
29     if(event->key()==Qt::Key_Right)
30     {
31         startX=(startX+step+image.width()>width)?startX:startX+step;
32     }
33     if(event->key()==Qt::Key_Up)
34     {
```



```
35         startY=(startY-step<0)?startY:startY-step;
36     }
37     if(event->key() == Qt::Key_Down)
38     {
39         startY=(startY+step+image.height()>height)?startY:startY+step;
40     }
41     if(event->key() == Qt::Key_Home)
42     {
43         startX=0;
44         startY=0;
45     }
46     if(event->key() == Qt::Key_End)
47     {
48         startX=width-image.width();
49         startY=height-image.height();
50     }
51 }
52 drawPix();
53 update();
}
```

第 20 行对 Ctrl 修饰键没有按下的处理。

第 22、23 行首先调节图标左上顶点的位置至网格的顶点上。

第 25~40 行分别根据按下的上下左右方向键调节图标的左上顶点的位置，步进值为网格的大小。

第 41~45 行如果按下 Home 键则重新复位图标位置为界面的左上顶点。

第 46~50 行如果按下 End 键则将图标位置置为界面的右下顶点，这里注意需要考虑图标自身的大小。

第 52 行根据调整后的图标位置重新在 pix 中绘制图像。

第 53 行调用 update()，触发界面重画。

paintEvent() 为界面的重画函数，将 pix 绘制在界面上。具体代码如下：

```
void KeyEvent::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawPixmap(QPoint(0,0),*pix);
}
```

实例 56 事件过滤器实现动态图片按钮

知识点：

- 事件过滤器的使用方法
- 利用 `installEventFilter` 方法安装事件过滤器

Qt 提供的 `QPushButton` 提供一个普通的按钮类，如果在应用程序中需要实现动态图片按钮，即鼠标按下时按钮图片发生变化，同时响应鼠标的按下等事件。本实例通过事件过滤器实现动态图片按钮效果，如图 10-3 所示。

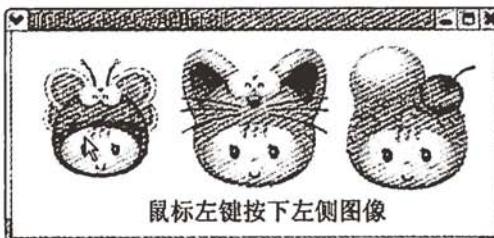


图 10-3 实现动态图片按钮

Qt 的事件模型中提供的事件过滤器功能使得一个 `QObject` 对象可以监视另一个 `QObject` 对象中的事件，通过在一个 `QObject` 对象中安装事件过滤器可以在事件到达该对象前捕获事件，从而起到监视该对象事件的效果。

在本实例中，3 个图片分别对应 3 个 `QLabel` 对象，当鼠标按下某个图片时，图片大小会发生变化，鼠标释放时，图片大小恢复初始大小，并且程序会提示当前事件的状态信息，如鼠标按键类型、按下的图片序号等。



小贴士：实现类似功能的另一种方式是通过分别继承不同的控件类，并重构各控件的事件响应函数，但若窗体中包含大量不同的控件时，每一个控件都必须要重新继承，然后分别重构不同的事件函数，实现比较复杂。事件过滤器可以实现在窗体中监视全部控件的不同事件，方便实现功能扩展。

```
class EventFilter : public QDialog
{
    Q_OBJECT
```

```

public:
    EventFilter( QWidget *parent=0, Qt::WindowFlags f=0 );
    ~EventFilter();
public:
    QLabel* Label1;
    QLabel* Label2;
    QLabel* Label3;
    QLabel* LabelState;
    QImage Image1;
    QImage Image2;
    QImage Image3;

public slots:
    bool eventFilter(QObject*,QEvent*);
};


```

类声明中定义了窗体中的各种控件以及槽，EventFilter()是 QObject 的事件监视函数。

```

EventFilter::EventFilter( QWidget *parent, Qt::WindowFlags f )
    : QDialog( parent, f )
{
    /*控件的创建和布局*/
    .....
    Label1->installEventFilter(this);
    Label2->installEventFilter(this);
    Label3->installEventFilter(this);
}

```

构造函数通过 installEventFilter()为每一个图片安装事件过滤器，installEventFilter()的函数原型如下：

```
void QObject::installEventFilter ( QObject * filterObj )
```

参数 filterObj 是监视事件的对象，此对象可以通过 EventFilter()函数接收事件，这里指定为整个窗体，如果某个事件需要被过滤，即停止正常的事件响应，则在 EventFilter()函数中返回 true，否则返回 false。QObject 的 removeEventFilter()可以解除已安装的事件过滤器。

```

bool EventFilter::eventFilter(QObject* watched,QEvent* event)
{
1   if(watched == Label1)
2   {
3       if(event->type() == QEvent::MouseButtonPress)

```

```
4  {
5      QMouseEvent *mouseEvent =(QMouseEvent *)event;
6      if(mouseEvent->buttons() &Qt::LeftButton)
7      {
8          LabelState->setText(tr("Left mouse button pressed on left image"));
9      }
10     else if(mouseEvent->buttons() &Qt::MidButton)
11     {
12         LabelState->setText(tr("Middle mouse button pressed on left image"));
13     }
14     else if(mouseEvent->buttons() &Qt::RightButton)
15     {
16         LabelState->setText(tr("Right mouse button pressed on left image"));
17     }
18     QMatrix martix;
19     martix.scale(0.8,0.8);
20     QImage tmp=Image1.transformed(martix);
21     Label1->setPixmap (QPixmap::fromImage(tmp));
22 }
23 if(event->type() == QEvent::MouseButtonRelease)
24 {
25     LabelState->setText(tr("Mouse button released from left image"));
26     Label1->setPixmap (QPixmap::fromImage(Image1));
27 }
28 }
/*处理其余控件的事件过滤*/
.....
29 return QDialog::eventFilter(watched,event);
}
```

第1行首先判断当前发生事件的对象。

第3行判断发生的事件类型。

第5行将事件 event 转化为鼠标事件。

第6~17行根据鼠标的按键类型分别显示。

第18~21行显示缩小的图片。

第23~27行为鼠标释放事件的处理，恢复图片的大小。

第29行调用 QDialog::eventFilter()将事件交给上层对话框。

第 11 章 其 他

本章包括 7 个实例，分别是：

- 利用 QSettings 保存程序窗口状态
- 利用 QDataStream 对文件进行存取
- 改变鼠标指针形状
- 改变窗体显示风格
- 拖拽图标
- 拖拽文字
- 字符串编码格式转换

实例 57 利用 QSettings 保存程序窗口状态

知识点：

QSettings 的使用方法

在实际应用中，常需要应用程序能保存程序的状态以及用户的设置，如应用程序显示的大小、位置、背景颜色或用户设置的参数等信息，以便在下次运行程序时，能保持上次关闭的状态。

本实例实现一个能保存关闭时状态的例子。实例的对话框如图 11-1 所示。

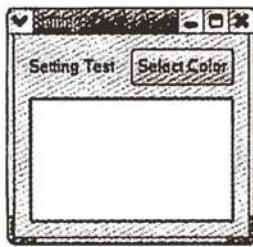


图 11-1 利用 QSettings 保存程序设置实例

例子实现的对话框包括一个标签（QLabel）、一个选择颜色（Select Color）按钮、一个文本编辑框（QTextEdit）。触发“选择颜色”按钮会弹出标准颜色对话框，可改变左侧标签文字（Setting Test）的颜色。

随意改变对话框的大小，移动对话框的位置，并单击“选择颜色”按钮，改变标签文字的颜色，在下面的文字编辑框中随意输入文字，如图 11-2 所示。

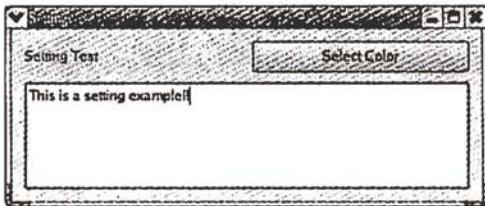


图 11-2 改变对话框的显示状态

此时，关闭对话框，当再重新运行此程序时，对话框仍旧保持关闭时的状态。



Qt 提供的 `QSettings` 类能很方便地实现保存程序设置的需求，并能很好地解决跨平台的问题，使移植变得简单方便。

具体实现代码如下所示。

头文件 `settings.h`:

```
#ifndef SETTINGS_H
#define SETTINGS_H

#include <QDialog>
class QPushButton;
class QLabel;
class QTextEdit;

class Settings : public QDialog
{
    Q_OBJECT
public:
    Settings();
    QLabel *label;
    QPushButton *colorBtn;
    QTextEdit *edit;

    void readSettings();
    void writeSettings();

protected:
    void closeEvent(QCloseEvent *); 

public slots:
    void slotColor();
};

#endif // SETTINGS_H
```

在头文件中声明了必要的对话框控件，一个 `label`、一个 `ColorBtn` 和一个文本编辑框 (`edit`)。

还声明了两个函数 `readSettings()`、`writeSettings()`，`readSettings()` 用于在对话框初始化时读取对话框上次关闭时保存的设置；`writeSettings()` 用于在程序关闭时保存对话框当时的状态。

另外还声明了一个 `protected` 函数 `closeEvent()`，此函数是 `QWidget` 的一个保护函数，当窗口关闭时会触发此函数；声明的 `setColor()` 函数用于选择颜色，以改变标签文字的颜色。

`readSettings()` 和 `writeSettings()` 是实现本实例功能的主要函数。

实现文件 `settings.cpp`：

```
Settings::Settings()
{
    1   setWindowTitle("Settings");

    2   label = new QLabel;
    3   label->setText("Setting Test");
    4   colorBtn = new QPushButton;
    5   colorBtn->setText("Select Color");
    6   edit = new QTextEdit;

    7   QGridLayout *layout = new QGridLayout(this);
    8   layout->addWidget(label,0,0);
    9   layout->addWidget(colorBtn,0,1);
   10  layout->addWidget(edit,1,0,1,2);

   11  readSettings();

   12  connect(colorBtn, SIGNAL(clicked()), this, SLOT(slotColor()));
}
```

`Settings()` 函数完成对话框创建初始化的工作。

第 1 行设置对话框的标题。

第 2~6 行完成对话框中控件的创建。

第 7~10 行完成对话框的布局管理。

第 11 行调用 `readSettings()` 函数，读取上次程序关闭时的用户设置，并用这些设置初始化本次程序的运行。

第 12 行连接“选择颜色”按钮的信号与槽函数。

`closeEvent()` 函数为程序的关闭事件函数，该函数只完成一件工作，即调用 `writeSettings()` 函数，保存当前的对话框状态。具体代码如下：

```
void Settings::closeEvent(QCloseEvent *e)
{
    writeSettings();
}
```



readSettings()函数完成读取上次程序关闭时状态的功能。具体代码如下：

```
void Settings::readSettings()
{
    1   QSettings setting("MyPro", "settings");
    2   QPoint pos = setting.value("position").toPoint();
    3   QSize size = setting.value("size").toSize();

    4   QColor color = setting.value("color").value<QColor>();
    5   QString text = setting.value("text").toString();

    6   move(pos);
    7   resize(size);
    8   QPalette p = label->palette();
    9   p.setColor(QPalette::Normal, QPalette::WindowText, color);
   10  label->setPalette(p);
   11  edit->setPlainText(text);
}
```

第 1 行新建一个 `QSettings` 对象，第一个参数分别为公司或组织名称，第二个参数表示应用程序的名称。若在程序中有多个地方需用到 `QSettings`，也可利用 `QCoreApplication` 一次性设置好应用程序的公司名和程序名，如：

```
QCoreApplication::setOrganizationName("MyPro");
QCoreApplication::setApplicationName("Settings");
QSettings setting;
```

这样，在每次用到 `QSetting` 时，即可直接创建对象而不用输入参数。

第 2、3 行读取对话框的位置与大小信息，调用 `QSetting` 的 `value()` 函数可获得一个 `QVariant` 类对象，`value()` 的输入参数为所要读取的信息关键字，`QVariant` 类提供了多种类型转换的函数如 `toInt()`、`toPoint()`、`toTime()`、`toDouble()` 等，如此可获得位置与大小信息。

第 4、5 行读取标签文字颜色和编辑框的内容，此处由于 `QVariant` 类并没有提供 `toColor()` 函数，因此颜色信息无法直接转换成 `QColor` 类型，而是调用了 `QVariant` 的 `value()` 函数以获得所需的类型。实际上，`toString()` 或 `toPoint()` 等 `QVariant` 提供的直接转换函数也即相当于调用 `QVariant::value()` 函数，如：

```
setting.value().toPoint();
```

也可写成：

```
setting.value().value<QPoint>();
```

第 6~11 行用读取到的信息初始化对话框的显示。

当保存的信息较多时也可使用组（Group）的方式，如位置（position）和大小（size）信息为一组（Dialog 组），标签文字颜色信息与编辑框文字信息为一组（Content 组），用 beginGroup()与 endGroup()进行限定，如：

```
setting.beginGroup("Dialog");
QPoint pos = setting.value("position").toPoint();
QSize size = setting.value(size").toSize();
setting.endGroup();

setting.beginGroup("Content");
QColor color = setting.value("color").value<QColor>();
QString text = setting.value("text").toString();
setting.endGroup();
```

这种方式一般在需保存的信息较多时使用，便于分类与书写。采用分组后，若只要保存某组中的一个信息时，应写成：

```
QPoint pos = setting.value("Dialog/position").toPoint();
```

这样就与

```
setting.beginGroup("Dialog");
QPoint pos = setting.value("position").toPoint();
setting.endGroup();
```

效果相同。

writeSettings()函数完成保存程序状态的功能。具体代码如下：

```
void Settings::writeSettings()
{
    1   QSettings setting("MyPro","settings");
    2   setting.setValue("position",pos());
    3   setting.setValue("size",size());

    4   setting.setValue("color",label->palette().color(QPalette::WindowText));
    5   setting.setValue("text",edit->toPlainText());
}
```

第 1 行新建一个 QSettings 对象。

第 2、3 行调用 QSettings 的 setValue()函数保存程序的位置与大小信息。

第 4、5 行保存颜色信息与文字编辑框内容信息。

保存时也可采用分组的方式进行编写，但注意保存与读取的信息关键字应保持一一对应。

 小贴士：利用 `QSettings` 保存信息，信息的关键字可以任意指定，但为了使程序的兼容性更好，应注意关键字的大小问题，因为有的平台对大小写不敏感，如 Windows 平台，而有的平台对大小写敏感如 Mac OS。因此，在指定关键字时应尽量使同一个关键字在不同的地方出现时都采用同样的大小写，并且不要用只有大小写区别的词表示不同的关键字。

`QSettings` 类为用户提供了方便存储程序各类信息的方式，在实际应用中，经常需要保存用户设置的参数等信息，以便在下次启动时对系统进行初始化，一般的方法是程序员自己定义数据格式，写入文件，而使用 `QSettings` 的方法能大大简化程序员的工作量。

实例 58 利用 `QDataStream` 对文件进行存取

知识点：

`QDataStream` 的使用方法

本实例利用 `QDataStream` 类通过对文件的读写操作，实现用户参数的保存与读取。

`QDataStream` 提供了一个二进制的数据流，并且与程序运行的操作系统平台无关。利用 `QDataStream` 类可以方便地保存和读取各类数据。例如，在实际应用中常需要保存用户设置的参数，以便下次运行时能恢复关闭时的参数设置，可需要与其他程序交互参数等。

本实例实现一个简单的对话框，包括信道、功率和频率 3 个参数的设置，当触发“保存”按钮后，程序会利用 `QDataStream` 类把当前的各个参数值保存在 `parameters.dat` 文件中，关闭程序再重新运行或随意修改参数值后，触发“读取”按钮，各个参数控件则会恢复显示上次保存的值，并显示上次保存的时间。实例的实现效果图如图 11-3 所示。

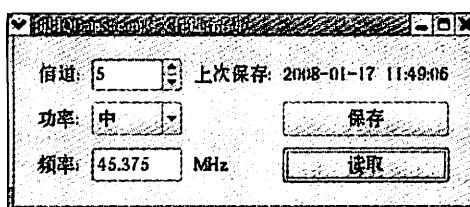


图 11-3 利用 `QDataStream` 对文件进行存取

具体实现代码如下所示。

头文件 pararw.h:

```
class ParaRW : public QDialog
{
    Q_OBJECT
public:
    ParaRW();

    QLabel *label1;
    QLabel *label2;
    QLabel *label3;
    QLabel *label4;
    QLabel *label5;
    QLabel *timeLabel;
    QPushButton *saveButton;
    QPushButton *getButton;
    QComboBox *powerComboBox;
    QSpinBox *channelSpinBox;
    QLineEdit *frequencyEdit;
public slots:
    void slotSave();
    void slotGet();
};
```

头文件中声明了程序用到的控件，以及两个槽函数。

实现文件 pararw.cpp:

```
ParaRW::ParaRW()
{
    //初始化
    .....
}
```

slotSave()槽函数响应用户单击“保存”按钮，保存当前各个参数的值。具体代码如下：

```
void ParaRW::slotSave()
{
    1 int channel = channelSpinBox->value();
    2 int power = powerComboBox->currentIndex();
    3 float frequency = frequencyEdit->text().toFloat();
    4 QDateTime *time = new QDateTime;
```



```

5   QFile file("parameters.dat");
6   file.open(QIODevice::WriteOnly);
7   QDataStream out(&file);

8   out.setVersion(QDataStream::Qt_4_0);
9   out << (quint32)0xa1a2a3a4;

10  out << (qint32)channel << (qint32)power << frequency << time->currentDateTime();
}

```

第 1~4 行获取当前需保存的各个参数值，包括信道、功率、频率和当前的时间，参数中有 int 类型、float 类型和 QDateTime 类型。

QDataStream 类支持多种数据类型，除基本的数据类型外，还包括如 QFont、QString、QImage、QColor 等，QDataStream 会根据所保存的数据类型以预定的编码方式写入二进制流中。

第 5 行创建一个 QFile 对象，作为 I/O 设备。

第 6 行以“只写”的方式打开这个设备。

第 7 行用打开的 I/O 设备创建一个 QDataStream 对象。

第 8 行设置 QDataStream 对象的版本号。由于自 Qt1.0 至今，数据类型的二进制格式在不断地发生改变，未来也许还会有更多的变化，在使用 QDataStream 进行存取时应使用相同版本的 Qt 数据类型，为了使程序有更好的兼容性，可在程序中设定版本号。

第 9 行在数据流中写入一个标识头，相当于定义自己的数据格式，用于区分。

第 10 行写入需要保存的数据。

写入 int 类型数据时，最好把它转换成 Qt 的数据类型，如 qint16 或 qint32。因为基本的 C++ 数据类型在不同的运行平台上，字节长度是不一样的，转换成 Qt 的数据类型能确保字节长度不随平台的变化而改变。

slotGet()槽函数响应用户单击“读取”按钮，读取已保存的各个参数的值。具体代码如下：

```

void ParaRW::slotGet()
{
1   QFile file("parameters.dat");
2   file.open(QIODevice::ReadOnly);
3   QDataStream in(&file);

4   in.setVersion(QDataStream::Qt_4_0);
5   qint32 magic;

```

```
6     in >> magic;
7     if(magic != 0xa1a2a3a4)
8     {
9         QMessageBox::information(this,"exception",tr("invalid file format"));
10    return;
11 }
12
13 qint32 channel;
14 qint32 power;
15 float frequency;
16 QDateTime time;
17
18 in >> channel >> power >> frequency >> time;
19
20 channelSpinBox->setValue(channel);
21 powerComboBox->setcurrentIndex(power);
22 QString freq;
23 frequencyEdit->setText(freq.setNum(frequency));
24 QString lastTime = time.date().toString(Qt::ISODate) + " " + time.time().toString();
25 timeLabel->setText(lastTime);
26 }
```

第 1 行创建一个 QFile 对象，作为 I/O 设备。

第 2 行以“只读”的方式打开这个设备。

第 3 行用打开的 I/O 设备创建一个 QDataStream 对象。

第 4 行设定数据类型版本号为 Qt_4_0，与写入的版本一致。

第 5~9 行从数据流中读取标识头，若读出的标识头与预设的不一致，则报告“文件格式错误”。

第 10~13 行创建所需参数的变量。

第 14 行从数据流中依次读取各个参数值。

第 15~20 行用读取的参数值更新显示。

实例 59 改变鼠标指针形状

知识点：

- 如何改变鼠标的外观
- 如何实现自定义鼠标外观



不同的鼠标指针形状可以提示用户当前进行的操作，或提示用户当前应用程序所处的状态，增强应用程序的可用性，方便用户的使用。本实例实现改变鼠标指针形状，如图 11-4 所示。



图 11-4 改变鼠标指针形状

具体代码如下：

```
void Cursor::slotArrow()
{
    setCursor(Qt::ArrowCursor);
}
```

setCursor()是 QWidget 的方法，其函数原型如下：

```
void setCursor ( const QCursor & )
```

QCursor 类提供了多种指针形状，如表 11-1 所示。

表 11-1 QCursor 类提供的多种指针形状

指针形状	说 明	指针形状	说 明
↖	Qt::ArrowCursor	↑	Qt::SizeVerCursor
↑	Qt::UpArrowCursor	↔	Qt::SizeHorCursor
+	Qt::CrossCursor	↗	Qt::SizeBDiagCursor
	Qt::IBeamCursor	↖	Qt::SizeFDiagCursor
☒	Qt::WaitCursor	⊕	Qt::SizeAllCursor
⚡	Qt::BusyCursor	㊂	Qt::SplitVCursor
🚫	Qt::ForbiddenCursor	㊃	Qt::SplitHCursor
👉	Qt::PointingHandCursor	👉	Qt::OpenHandCursor
⟲	Qt::WhatsThisCursor	⟲	Qt::ClosedHandCursor

除了 QCursors 提供的各种指针形状外，用户还可以自定义指针形状。

```
void Cursor::slotCustom()
{
    QCursor *myCursor= new QCursor(QPixmap(":/images/custom.png"),-1,-1);
    setCursor(*myCursor);
}
```

第1行首先创建一个 QCursors，指针形状指定为某个自定义的图片文件，QCursors 的构造函数原型如下：

```
QCursors::QCursors ( const QPixmap & pixmap, int hotX = -1, int hotY = -1 )
```

其中，参数(hotX,hotY)为指针的热点坐标，默认均为-1，表示热点位于图片的中心位置。

实例 60 改变窗体显示风格

知识点：

- 利用 QStyleFactory 获得当前系统支持的窗体风格
- 调用 QApplication::setStyle()改变窗体的风格

本实例实现一个显示风格可变的窗体，通过下拉列表框中的选择，改变窗体的显示风格，实例效果如图 11-5 所示。

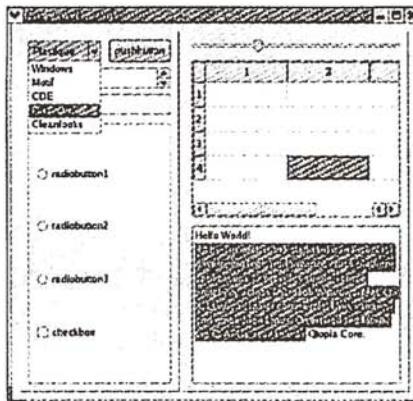


图 11-5 改变窗体显示风格



在图 11-5 左上角的下拉列表框中列出了所有系统可支持的预置窗体风格，根据对它的选择，从而改变整个窗体的显示风格。具体实现代码如下所示。

头文件 style.h:

```
class Style : public QDialog
{
    Q_OBJECT
public:
    Style(QWidget *parent=0);

    void createLeftLayout();
    void createRightLayout();
public slots:
    void slotChangeStyle(QString);
private:
    QFrame *leftFrame;
    QFrame *rightFrame;
};
```

头文件中声明了一个槽函数 `slotChangeStyle(QString)`，用于完成改变窗体风格的工作；定义了两个私有变量 `leftFrame` 和 `rightFrame`，分别用于左右两部分控件的显示；声明了两个公有函数用于具体完成左右两部分控件的创建与布局。

实现文件 style.cpp:

```
Style::Style(QWidget *parent)
    : QDialog(parent)
{
    1   setWindowTitle(tr("Change Window Style"));

    2   createLeftLayout();
    3   createRightLayout();

    4   QHBoxLayout *mainLayout = new QHBoxLayout;
    5   mainLayout->setMargin(10);
    6   mainLayout->setSpacing(5);
    7   mainLayout->addWidget(leftFrame);
    8   mainLayout->addWidget(rightFrame);
    9   setLayout(mainLayout);
}
```

`Style()` 函数为对话框类的构建函数，第 1 行设置窗体的标题。

第 2 行调用 `createLeftLayout()` 函数完成窗体左部的控件实现及布局。

第3行调用createRightLayout()函数完成窗体右部的控件实现及布局。

第4~9行利用QHBoxLayout类完成窗体的主布局。

```
void Style::createLeftLayout()
{
    1   leftFrame = new QFrame;
    2   leftFrame->setFrameStyle(QFrame::Panel|QFrame::Raised);
    3   QComboBox *styleComboBox = new QComboBox;
    4   styleComboBox->addItems(QStyleFactory::keys());
    5   QPushButton *button = new QPushButton(tr("pushbutton"));
    6   QHBoxLayout *hbox = new QHBoxLayout;
    7   hbox->addWidget(styleComboBox);
    8   hbox->addWidget(button);

    9   QSpinBox *spinBox = new QSpinBox;
    10  spinBox->setRange(0,9);
    11  QLineEdit *lineEdit = new QLineEdit;
    12  lineEdit->setText("Hello Hello");

    13  QGroupBox *group = new QGroupBox;
    14  QRadioButton *radio1 = new QRadioButton(tr("radiobutton1"),group);
    15  QRadioButton *radio2 = new QRadioButton(tr("radiobutton2"),group);
    16  QRadioButton *radio3 = new QRadioButton(tr("radiobutton3"),group);
    17  QCheckBox *checkBox = new QCheckBox(tr("checkbox"),group);
    18  QVBoxLayout *groupLayout = new QVBoxLayout;
    19  groupLayout->addWidget(radio1);
    20  groupLayout->addWidget(radio2);
    21  groupLayout->addWidget(radio3);
    22  groupLayout->addWidget(checkBox);
    23  group->setLayout(groupLayout);

    24  QVBoxLayout *vbox = new QVBoxLayout;
    25  vbox->addLayout(hbox);
    26  vbox->addWidget(spinBox);
    27  vbox->addWidget(lineEdit);
    28  vbox->addWidget(group);
    29  leftFrame->setLayout(vbox);

    30  connect(styleComboBox,SIGNAL(activated(QString)),this,
               SLOT(slotChangeStyle(QString)));
}
```



```
31 slotChangeStyle(QStyleFactory::keys()[0]);
}
```

createLeftLayout()函数完成窗体的左部控件实现及布局。

第 1 行创建一个 QFrame 类对象，作为窗体左部控件的容器。

第 2 行设置的 QFrame 类对象的显示风格，设置为 Panel&Raised。

第 3、4 行创建用于显示系统可用风格列表的下拉列表框控件，QStyleFactory::keys() 函数将会返回一个字符串序列，包含所有预置的可用的风格名称，调用 QComboBox 的 addItems() 函数把这些风格名称加入到下拉列表框控件中。

第 5~23 行创建了一系列常用控件，由于本实例的目的是为了说明如何进行风格的设置，并展示在不同风格下各常用控件的显示状态，对这些控件只进行了简单的创建和设置，包括按钮、单选按钮、复选框等。

第 24~29 行对这些控件进行了布局。

第 30 行为下拉列表框的选择变化连接了响应的槽函数 slotChangeStyle(QString)。

第 31 行调用改变风格的函数 slotChangeStyle() 初始化窗体的显示，此处用的是可用预置风格序列的第一个风格对窗体进行初始化。

```
void Style::createRightLayout()
{
    rightFrame = new QFrame;
    rightFrame->setFrameStyle(QFrame::Panel|QFrame::Raised);

    QSlider * slider = new QSlider(Qt::Horizontal);
    QTableWidget *table = new QTableWidget;
    table->setColumnCount(3);
    table->setRowCount(4);
    QTextEdit *edit = new QTextEdit;
    edit->setText("Hello World\n.....");
    QVBoxLayout *layout = new QVBoxLayout;
    layout->setSpacing(10);
    layout->addWidget(slider);
    layout->addWidget(table);
    layout->addWidget(edit);
    rightFrame->setLayout(layout);
}
```

createRightLayout()函数完成窗体右部的控件实现及布局，包括一个滑动条、一个表格和一个文字编辑框。

```
void Style::slotChangeStyle(QString style)
```

```
{  
    QApplication::setStyle(QStyleFactory::create(style));  
    QApplication::setPalette(QApplication::style()->standardPalette());  
}
```

slotChangeStyle()函数完成改变窗体风格的工作，首先利用 QStyleFactory 的 create() 函数创建一个 QStyle 对象，以风格名称为参数，调用 QApplication 的 setStyle() 函数把此程序的风格设置为刚创建的风格，调用 setPalette() 完成窗体显示的改变。

实例 61 拖拽图标

知识点：

- 实现拖拽效果的基本流程
- QDrag、QDragEnterEvent、QDropEvent、QDragMoveEvent 类的使用方法

本实例实现窗体内图标的拖拽效果，排列在窗体中的图标可以拖拽至窗体的任意位置。图标也可在两个独立运行的实例程序间相互拖拽图标，此时拖拽的图标将复制一份到拖拽目的程序窗体中。实例效果图如图 11-6 和图 11-7 所示。

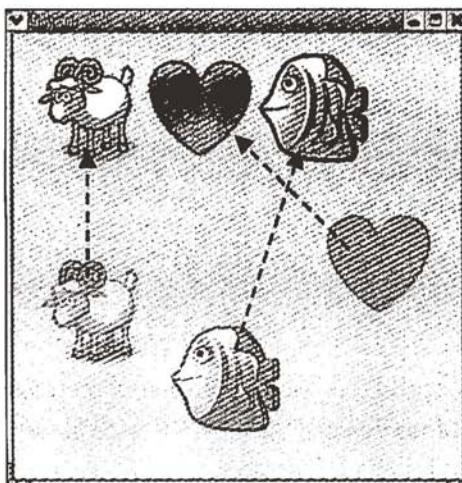


图 11-6 图标在窗体内任意拖拽

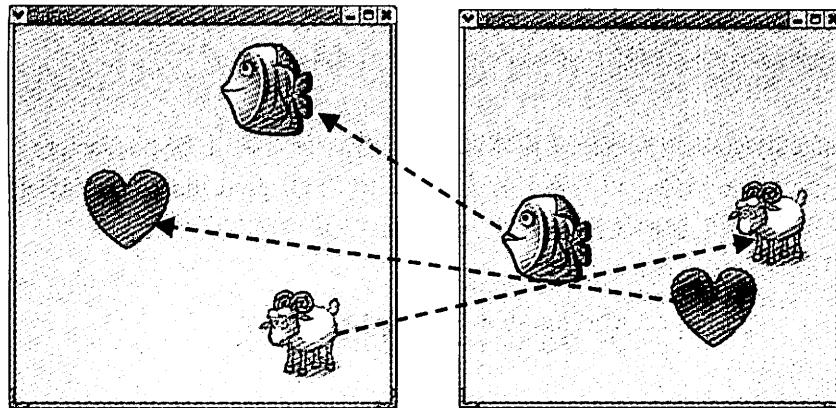


图 11-7 在程序独立运行的程序之间拖拽图标

拖拽机制的实现类似于剪贴板的机制，Qt 提供的与拖拽相关的类有 QDrag、QDragEnterEvent、QDropEvent、QDragMoveEvent 等。

下面以实例为基础逐步分析拖拽的实现过程。

在本实例中，为了实现图标的拖拽效果，首先需要构建一个可拖拽的图标类 DragIcon。

```
class DragIcon : public QLabel
{
    Q_OBJECT
public:
    DragIcon(QPixmap pix, QWidget *parent=0);

    void mousePressEvent(QMouseEvent *);
    void mouseMoveEvent(QMouseEvent *);

private:
    QPoint startPos;
};
```

DragIcon 类继承自 QLabel 类，此类重定义 mousePressEvent((QMouseEvent *))事件函数和 mouseMoveEvent((QMouseEvent *))事件函数，响应鼠标的按下和移动事件。私有变量 startPos 用于保存鼠标按下时的位置信息。

DragIcon 类的构建函数中初始化图标的图片，并设置图标标签符合图片的尺寸。具体代码如下：

```
DragIcon::DragIcon(QPixmap pix, QWidget *parent)
    : QLabel(parent)
{
    setScaledContents(true);
    setPixmap(pix);
}
```

响应鼠标按下事件函数中，首先判断按下的是不是鼠标的左键，若是则记录当时点的位置信息 startPos，为后面的判断作准备。具体代码如下：

```
void DragIcon::mousePressEvent(QMouseEvent * e)
{
    if(e->button() == Qt::LeftButton)
        startPos = e->pos();
}
```

创建拖拽类的工作主要在响应鼠标移动事件函数 mouseMoveEvent()中完成，代码如下：

```
void DragIcon::mouseMoveEvent(QMouseEvent * e)
{
1    if (!e->buttons() & Qt::LeftButton)
2        return;
3    if ((e->pos() - startPos).manhattanLength()
        < QApplication::startDragDistance())
4        return;

5    QPixmap pix = *pixmap();

6    QByteArray data;
7    QDataStream stream(&data,QIODevice::WriteOnly);
8    stream << pix << QPoint(e->pos()-rect().topLeft());
9    QMimeData *mimeData = new QMimeData;
10   mimeData->setData("Drag-Icon",data);

11  QDrag *drag = new QDrag(this);
12  drag->setMimeData(mimeData);
13  drag->setHotSpot(QPoint(e->pos() - rect().topLeft()));
14  drag->setPixmap(pic);

15  hide();

16  Qt::DropAction dropAction = drag->start(Qt::CopyAction | Qt::MoveAction);
```

```

17 if (dropAction == Qt::MoveAction)
18     close();
19 else
20     show();
}

```

第 1、2 行判断当前移动时是否是左侧按钮按下，若不是则不响应。

第 3、4 行判断响应鼠标移动事件时移动的距离是否超过判断为拖拽事件的最小距离， $(e->pos() - startPos).manhattanLength()$ 为按住鼠标移动的距离， $manhattanLength()$ 为 QPoint 类的函数，返回某两点间曼哈顿长度，大致相当于如图 11-8 所示的距离值。

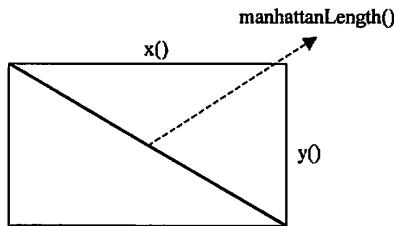


图 11-8 距离值

`QApplication::startDragDistance()` 表示产生拖拽事件的最小距离值（默认为 4 个像素值），用户按住鼠标移动大于 `QApplication::startDragDistance()` 的距离，则认为是一个拖拽事件，可避免由于用户手抖动而引起的误操作。与 `QApplication::startDragDistance()` 有同样作用的还有一个 `QApplication::startDragTime()` 值，表示产生拖拽事件的最短时间值，可根据实际情况选择合适的值作为产生拖拽事件的判断依据。

第 5 行获得可拖拽图标的图片。

第 6~10 行生成一个 `QMimeType` 对象，为下面创建可拖拽 `QDrag` 对象作准备，首先创建一个 `QByteArray` 对象用于存储需拖拽图标的信息，包含图片信息以及鼠标按下点位置相对图片左上角的偏移量。

第 7、8 行利用 `QStreamData` 类向 `QByteArray` 对象中写入数据。

第 10 行调用 `QMimeType` 类的 `setData()` 函数把包含拖拽图标信息的 `QByteArray` 对象 `data` 放入 `QMimeType` 对象中，`setData()` 函数的第一个参数用于说明数据的格式名，可为自定义的任意格式。

第 11~14 行生成一个可拖拽 `QDrag` 对象。

第 12 行调用 `setMimeData()` 函数设置 `QDrag` 对象的数据值。

第 13 行调用 `setHotSpot()` 函数设置拖拽图标的热点，也即拖动时鼠标相对于拖拽图

标左上角的位置。

第 14 行设置鼠标拖动时，QDrag 对象显示的图片，一般设置为与需拖动图标一致的图片。

第 15 行隐藏原位置上的图标。

第 16 行调用 QDrag 类的 start() 函数，函数参数设置此拖拽对象可使用的 DropAction（放动作），可为复制 CopyAction、移动 MoveAction、LinkAction 等，并等待拖拽结束时返回的 DropAction 值，默认为 CopyAction。

第 17~20 行对返回的 DropAction 值进行判断，若返回的为 CopyAction，则调用原图标的 show() 函数，重新在原位置显示图标；若返回的为 MoveAction，则表示是一个移动操作，则调用原图标的 close() 函数关闭原图标。

实现可拖拽的图标后，则可构建一个 DragWidget 类作为放置可拖拽图标的容器。

DragWidget 继承自 QWidget 类，在类中重定义 dragEnterEvent(QDragEnterEvent *) 函数响应拖拽进入事件，dragMoveEvent(QDragMoveEvent *) 函数响应拖拽移动事件，dropEvent(QDropEvent *) 函数响应拖拽的放事件。具体代码如下：

```
class DragWidget : public QWidget
{
    Q_OBJECT
public:
    DragWidget(QWidget *parent=0);

    void dragEnterEvent(QDragEnterEvent *);
    void dragMoveEvent(QDragMoveEvent *);
    void dropEvent(QDropEvent *);
};
```

DragWidget 的构成函数：

```
DragWidget::DragWidget(QWidget *parent)
    : QFrame(parent)
{
    1   setMinimumSize(400,400);
    2   setAcceptDrops(true);

    3   DragIcon *icon1 = new DragIcon(QPixmap(":/images/sheep.png"),this);
    4   DragIcon *icon2 = new DragIcon(QPixmap(":/images/heart.png"),this);
    5   DragIcon *icon3 = new DragIcon(QPixmap(":/images/fish.png"),this);

    6   icon1->move(20,20);
```



```
7     icon2->move(120,20);
8     icon3->move(220,20);
}
```

第 1 行设置窗体的最小大小。

第 2 行调用 setAcceptDrops(true) 设置此窗体可接受拖拽事件。

第 3~5 行新建 3 个 DragIcon 对象。

第 6~8 行排列 3 个可拖拽图标的位置。

dragEnterEvent() 函数响应拖拽进入事件的代码如下：

```
void DragWidget::dragEnterEvent(QDragEnterEvent * e)
{
    if (e->mimeData()->hasFormat("Drag-Icon"))
    {
        if(children().contains(e->source()))
        {
            e->setDropAction(Qt::MoveAction);
            e->accept();
        }
        else
            e->acceptProposedAction();
    }
}
```

该函数首先判断 QDragEnterEvent 的 mimeData 中是否包含数据格式名为 Drag-Icon 的数据，若包含则继续判断此拖拽对象是本窗体内部的对象还是外部拖入的拖拽对象，若为本窗体内部的对象则说明这是个移动操作 MoveAction，则调用 setDropAction() 函数设置 DropAction 为 MoveAction；否则说明这是个复制操作 CopyAction，则调用 acceptProposedAction() 表示接受默认的操作即 CopyAction。这样产生拖拽对象的 DragIcon 类即可根据此时返回的 DropAction（放动作）进行相应的操作。

dragMoveEvent() 函数响应拖拽移动事件的代码如下：

```
void DragWidget::dragMoveEvent(QDragMoveEvent * e)
{
    if (e->mimeData()->hasFormat("Drag-Icon"))
    {
        if(children().contains(e->source()))
        {
            e->setDropAction(Qt::MoveAction);
            e->accept();
        }
    }
}
```

```
        }  
    else  
        e->acceptProposedAction();  
    }  
}
```

该函数的实现与上面 dragEnterEvent() 函数实现基本一样，是为了对拖拽进行更细致的控制，在本实例中不实现此函数亦可。

`dropEvent()`函数响应拖拽的放事件，即拖拽动作的结束操作。具体代码如下：

```

void DragWidget::dropEvent(QDropEvent * e)
{
    if (e->mimeData()->hasFormat("Drag-Icon"))
    {
        QByteArray data = e->mimeData()->data("Drag-Icon");
        QDataStream stream(&data,QIODevice::ReadOnly);
        QPixmap pix;
        QPoint offset;
        stream >> pix >> offset;

        DragIcon *icon = new DragIcon(pix,this);
        icon->move(e->pos() - offset);
        icon->show();

        if (children().contains(e->source()))
        {
            e->setDropAction(Qt::MoveAction);
            e->accept();
        }
        else
            e->acceptProposedAction();
    }
    else
        e->ignore();
}

```

第1行判断事件的 `mimeData` 中是否包含格式名为 `Drag-Icon` 的数据，若包含则进行后续的工作，否则忽略此事件。

第 2 行创建一个 QByteArray 对象获得格式名为 Drag-Icon 的数据。

第3行利用QStreamData类从QByteArray对象中读取出所需的数据值，包括图片信息及位置偏移量值。



第 7~9 行新建一个 DragIcon 对象，并在减去偏移量的位置处显示。

第 10~13 行与前面 dragEnterEvent() 函数中一样对拖拽对象进行判断，是内部或外部对象，从而决定设置 DropAction 的类型。

小贴士：可能有读者会认为没有必要在 dragEnterEvent() 函数与 dropEvent() 函数中都进行拖拽对象的判断工作，只用在 dropEvent() 函数中一次性完成即可。事实上在两个函数中都进行判断是有用处的，若不在 dragEnterEvent() 函数中进行判断设置 DropAction，则当鼠标按住图标在本窗体内部拖动还未松开按钮时，箭头下方会显示一个“+”号，这一般是用于表示复制。

以上即一个简单的拖拽实例的实现过程，总结实现的过程，拖拽效果的关键步骤可由图 11-9 概括。

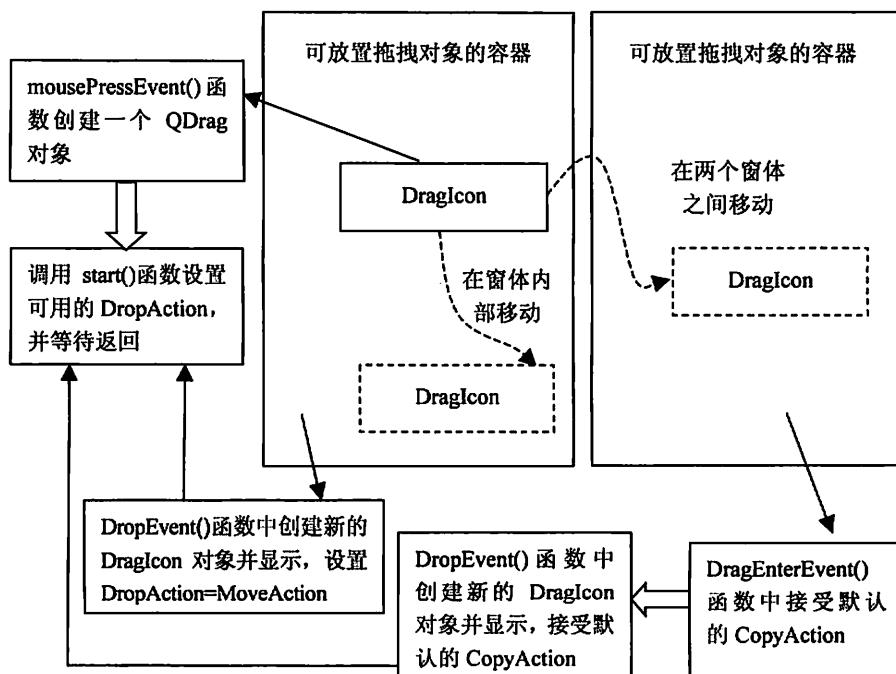


图 11-9 拖拽效果的关键步骤

由图 11-9 可知，实现拖拽功能的最基本的工作有 3 个部分：

(1) 在可拖拽对象的 mousePressEvent() 函数中构建 QDrag 对象，并调用 start() 函数等待 DropAction 的返回。

(2) 在放置可拖拽对象的容器类中，实现 dragEnterEvent() 函数，判断并设置应采用何种 DropAction，返回给 start() 函数。

(3) 在放置可拖拽对象的容器类中，实现 dragEnterEvent() 函数，创建一个新的可拖拽对象并在当前鼠标位置进行显示，同时判断并设置应采用何种 DropAction，返回给 start() 函数。

实例 62 拖拽文字

知识点：

- 拖拽数据类型
- QDrag、QDragEnterEvent、QDropEvent、QDragMoveEvent 类的使用方法

在实例 61 中实现了一个图标拖拽的例子，不仅图片可作为拖拽的对象，文字也同样可作为拖拽的对象。在本实例中即实现一个对文字进行拖拽的例子，如图 11-10 所示。

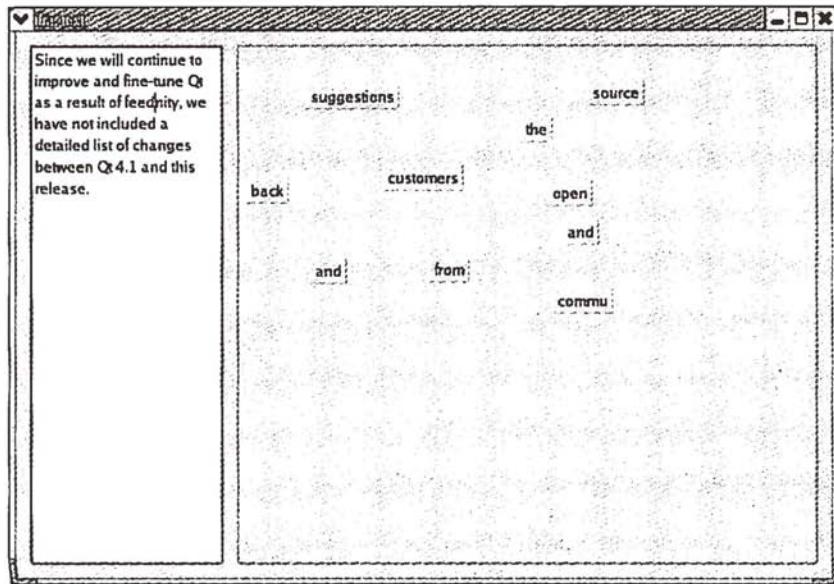


图 11-10 拖拽文字

在窗体的左侧为一个 QTextEdit 控件，在框内输入文字，用鼠标选定一段文字并拖拽



至右侧的窗体内，即会成为一个一个的文字标签，并且每个标签都可独立拖拽。

与拖拽图标一样，首先应构建一个可拖拽标签类 DragLabel。DragLabel 类继承自 QLabel 类，在类中重新实现 mousePressEvent(QMouseEvent*) 函数响应鼠标按下事件。具体代码如下：

```
class DragLabel : public QLabel
{
    Q_OBJECT
public:
    DragLabel(const QString &text, QWidget *parent=0);

    void mousePressEvent(QMouseEvent *);
};
```

在 DragLabel 的构造函数中，对标签的基本显示样式进行设置，包括它的背景、形状等。具体代码如下：

```
DragLabel::DragLabel(const QString &text, QWidget *parent)
    : QLabel(text,parent)
{
    setAutoFillBackground(true);
    setFrameShape(QFrame::Panel);
    setFrameShadow(QFrame::Raised);
}
```

mouseEvent() 函数响应鼠标按下事件，主要完成构建 QDrag 对象的工作。具体代码如下：

```
void DragLabel::mousePressEvent(QMouseEvent * e)
{
    1    QString str = text();
    2    QPixmap pix;
    3    pix = pix.grabWidget(this);

    4    QByteArray data;
    5    QDataStream stream(&data,QIODevice::WriteOnly);
    6    stream << str << QPoint(e->pos()-rect().topLeft());
    7    QMimeData *mimeData = new QMimeData;
    8    mimeData->setData("Drag-Text",data);
    9    mimeData->setText(str);

   10   QDrag *drag = new QDrag(this);
   11   drag->setMimeData(mimeData);
```



```
12     drag->setHotSpot(QPoint(e->pos() - rect().topLeft()));  
13     drag->setPixmap(pic);  
  
14     hide();  
  
15     Qt::DropAction dropAction = drag->start(Qt::CopyAction | Qt::MoveAction);  
  
16     if (dropAction == Qt::MoveAction)  
17         close();  
18     else  
19         show();  
}
```

第 1 行获得标签的文字。

第 2、3 行调用 QPixmap 类的 grabWidget() 函数把标签作为图片获取。

第 4~9 行代码创建一个 QMimeData 对象保存可拖拽对象的数据信息，首先新建一个 QByteArray 对象，利用 QStreamData 类把需保存的信息写入 QByteArray 对象中，包括文字信息及鼠标点相对于标签左上角的偏移值；构建一个 QMimeData 对象，调用它的 setData() 函数以 Drag-Text 为格式名把保存有信息的 QByteArray 对象写入，最后调用 setText() 函数直接以 plainText 格式写入文字信息。

第 10~13 行创建一个 QDrag 对象，首先调用 setMimeData() 函数写入前面已构建好的 QMimeData 对象，setHotSpot() 函数用于设置拖动时鼠标所在点相对标签图片左上角的偏移值，最后调用 setPixmap() 设置标签在拖动时显示的图片。

第 14 行调用 hide() 函数隐藏原位置的标签，实现标签随鼠标一起移动的效果。

第 15 行调用 QDrag 类的 start() 函数，设置可选的 DropAction，包括 CopyAction 和 MoveAction 两种，并等待返回。

第 16~19 行根据返回的 DropAction 类型进行相应的操作，若返回的是复制 CopyAction，则显示原位置的标签；若返回的是移动 MoveAction，则说明是在窗体内部进行移动，则关闭原位置的标签。

构建好可拖拽的标签类后，为右侧放置可拖拽对象的容器构建一个类 DragWidget。具体代码如下：

```
class DragWidget : public QFrame  
{  
    Q_OBJECT  
public:  
    DragWidget(QWidget *parent=0);
```



```
void dragEnterEvent(QDragEnterEvent *);  
void dragMoveEvent(QDragMoveEvent *);  
void dropEvent(QDropEvent *);  
};
```

在此容器类中重写 dragEnterEvent(QDragEnterEvent *)函数响应拖拽进入事件，dragMoveEvent(QDragMoveEvent *)函数响应拖拽移动事件，dropEvent(QDropEvent *)函数响应拖拽放事件。

```
DragWidget::DragWidget(QWidget *parent)  
    : QFrame(parent)  
{  
    setMinimumSize(300,300);  
    setAcceptDrops(true);  
    setFrameStyle(QFrame::StyledPanel|QFrame::Raised);  
}
```

在 DragWidget 类的构造函数中，调用拖拽容器的最小尺寸及面板风格，并调用 setAcceptDrops(true)设置此窗体可接受拖拽事件。

dragEnterEvent()函数代码如下：

```
void DragWidget::dragEnterEvent(QDragEnterEvent * e)  
{  
    if (e->mimeData()->hasText())  
    {  
        if(children().contains(e->source()))  
        {  
            e->setDropAction(Qt::MoveAction);  
            e->accept();  
        }  
        else  
            e->acceptProposedAction();  
    }  
}
```

该函数用于响应拖拽动作的进入事件，首先需对进行的拖拽对象进行数据格式的判断，决定此拖拽对象是否是本窗体可接受的对象类型。e->mimeData()->hasText()表示只要拖拽对象包含的是文字类型 (text/plain)，则窗体都接受。接下来对进入的拖拽对象产生的源进行判断，决定返回的是 CopyAction 还是 MoveAction，若拖拽对象是本窗体内部产生的，则设置放动作 (DropAction) 为移动 MoveAction 类型；若拖拽对象是由外部产

生的，则接受默认的放动作（DropAction）即复制 CopyAction 类型。

dropEvent()函数的代码如下：

```
void DragWidget::dropEvent(QDropEvent * e)
{
    if (e->mimeData()->hasFormat("Drag-Text"))
    {
        QByteArray data = e->mimeData()->data("Drag-Text");
        QDataStream stream(&data,QIODevice::ReadOnly);
        QString text;
        QPoint offset;
        stream >> text >> offset;

        DragLabel *label = new DragLabel(text,this);
        label->move(e->pos() - offset);
        label->show();

        if (children().contains(e->source()))
        {
            e->setDropAction(Qt::MoveAction);
            e->accept();
        }
        else
            e->acceptProposedAction();
    }
    else if (e->mimeData()->hasText())
    {
        QStringList strList = e->mimeData()->text().split(QRegExp("\s+"),
                                                       QString::SkipEmptyParts);
        QPoint pos = e->pos();

        foreach(QString str, strList)
        {
            DragLabel *dragLabel = new DragLabel(str,this);
            dragLabel->move(pos);
            dragLabel->show();
            pos += QPoint(dragLabel->width(),0);
        }

        if (children().contains(e->source()))
        {
            e->setDropAction(Qt::MoveAction);
            e->accept();
        }
    }
}
```



```
25     else
26         e->acceptProposedAction();
}
else
26     e->ignore();
}
```

该函数响应拖拽的放事件，主要可分为两个部分，对两种格式的拖拽对象分别进行处理，一种为自定义的 Drag-Text 格式，一种为基本的 text/plain。

对于自定义的 Drag-Text 格式，由第 1~13 行的代码进行处理。

第 2~6 行创建一个 QByteArray 对象，利用 QDataStream 把拖拽对象中保存的数据信息获取出来，包括文字信息和位置偏移值。

第 7~9 行新建一个 DragLabel 对象，设置获取的文字信息，并在鼠标所在点减去偏移量处显示。

第 10~13 行对拖拽对象的产生源进行判断，从而决定设置何种放动作（DropAction）。

对于包含基本数据格式 text/plain 的拖拽对象，由第 14~25 行的代码进行处理。

第 15 行获取了一个字符串列，由于拖拽进来的文字对象可能是一个字符，也可能是一个单词或一段文字，通过 split() 函数，以空格为参照，把拖拽进来的文字对象分割成一个个独立的单词。

第 16 行保存鼠标当前的位置，用于显示标签。

第 17~21 行针对刚获得的单词列，为每一个单词新建一个 DragLabel 对象，并一个接一个并排显示。

最后对拖拽对象的产生源进行判断，从而决定设置何种放动作（DropAction）。

完成了可拖拽标签和放置可拖拽对象的容器类的构建后，在 main() 函数中完成最后的结合工作。具体代码如下：

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

    QWidget *mainWidget = new QWidget;
    QTextEdit *edit = new QTextEdit;
    edit->setText("Since we will .....");
    DragWidget *dragWidget = new DragWidget;

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(edit);
    layout->addWidget(dragWidget);
}
```

```
layout->setStretchFactor(edit,1);
layout->setStretchFactor(dragWidget,3);
mainWidget->setLayout(layout);
mainWidget->show();

return app.exec();
}
```

在主窗口中新建一个 QTextEdit 控件用于输入文字，一个 DragWidget 控件用于放置可拖拽标签，以 QHBoxLayout 布局。

实例 63 字符串编码格式转换

知识点：

- Qt 中常用的编码格式
- QString 的各种常用编码转换函数

在不同操作系统之间传输字符串数据时，经常碰到因字符串的格式编码不同造成显示乱码的现象，这是由于不同操作系统使用的编码格式不一致带来的问题。本实例主要就 Qt 应用中经常碰到的字符串转换成 Unicode、Utf8 和字符内码 3 种编码格式的问题展开分析，如图 11-11 所示。

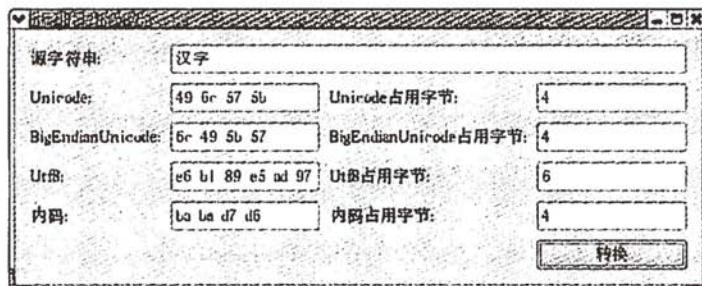


图 11-11 字符串编码格式转换

在实例中，用户在源字符串的编辑框中输入需要转换的字符串，可以是字符或汉字，单击“转换”按钮后，程序自动将该字符串转换成 Unicode、Utf8 和字符内码 3 种编码格式的二进制显示，即在内存中的实际存储方式，并显示不同编码格式占用的字节长度。



在分析实例之前，首先对几种编码格式作一些介绍，方便读者理解。

Unicode 是统一表示各种语言字符的编码标准，支持跨平台、跨程序、跨语言。它将世界上所有的字符都映射成一个数字，如“汉”为 0x6C49，Unicode 是一种 2 字节编码，由两个字节的大小存储一个字符。

Unicode 规范中推荐使用的标记字节顺序的方法是 BOM (Byte Order Mark)，根据内存中存放的字节顺序分为 Little-Endian 和 Big-Endian，Little-Endian 采用高值存放在高地址字节，低值存放在低地址字节的方式；Big-Endian 采用高值存放在低地址字节，低值存放在高地址字节的方式。不同的体系结构采用的字节存放顺序是不一样的，如 x86 中通常以 Little-Endian 方式存放，而 PowerPC 中通常以 Big-Endian 存放。

通常来讲，Unicode 指的是 Little-Endian，即高值存放在高地址字节，低值存放在低地址字节，如“汉”的 Unicode 值为 0x6C49，实际在内存中，6C 放在高地址字节，49 放在低地址字节，即 49 6C。另一种为 Big-Endian Unicode，即高值存放在低地址字节，低值存放在高地址字节，如“汉”的 Big-Endian Unicode 值为 0x6C49，实际在内存中，49 放在高地址字节，6C 放在低地址字节，即 6C 49。

UTF-8 是 UTF (UCS Transformation Format) 规范规定的一种 Unicode 编码表示方式，另外还有 UTF-16 等。UTF-8 是以 8 位为单元对 Unicode 进行编码，兼容 ASCII 字符；UTF-16 是以 16 位对 Unicode 进行编码，与 ASCII 字符不兼容，对于小于 0x10000 的 Unicode，UTF-16 编码就等于 Unicode 编码所对应 16 位无符号整数。UTF-8 是以字节流的方式表示，没有字节顺序的问题。UTF-8 与 unicode 编码之间的转换规则如下：

Unicode 编码(十六进制)	UTF-8 字节流(二进制)
0x0000~0x007F	0xxxxxxx
0x0080~0x07FF	110xxxxx 10xxxxxx
0x0800~0xFFFF	1110xxxx 10xxxxxx 10xxxxxx

例如，“汉”字的 Unicode 为 0x6C49，位于 0x0800~0xFFFF 之间，对应的 UTF-8 字节流为 1110xxxx 10xxxxxx 10xxxxxx，UTF-8 编码采用三字节表示，0x6C49 的二进制为 0x0110110001001001，将这 16 位分别替换上面的 x，可以得到 11100110 10110001 10001001，即 E6 B1 89，这就是中文“汉”字的 UTF-8 编码。

字符内码指的是用来代表字符的内码，包括单字节内码和双字节编码，单字节内码就是 ASCII 码，可以表示 256 个字符编码；双字节内码就是 ANSI，可以表示 65000 个字符编码。对于简体中文编码 GB2312，实际上对应的是 ANSI 的一个代码页 936。ANSI 有很多代码页，使用不同代码页的内码无法在其他代码页正常显示，如繁体中文、日文都对应 ANSI 的一个代码页，这些内码无法在简体中文 GB2312 的平台上正常显示。对于 GB2312 来说，用一字

节表示一个字符，每一个汉字由两个字符构成，因此每一个汉字占两个字节。

slotStart()函数的代码如下：

```
void StringCodec::slotStart()
{
    1   lineEditUnicode->clear();
    2   lineEditBigEndianUnicode->clear();
    3   lineEditUtf8->clear();
    4   lineEditLocal->clear();

    5   QString source=lineEditSource->text();

    6   const QChar *u=source.unicode();

    //Get Unicode
    7   lineEditUnicodeLen->setText(QString::number(source.length()*2));
    8   for(int i=0;i<source.length();i++)
    9   {
    10      const ushort unicode=u[i].unicode();
    11      lineEditUnicode->insert(QString::number(unicode%256,16)+" ");
    12      lineEditUnicode->insert(QString::number(unicode/256,16)+" ");
    13  }

    //Get BigEndian Unicode
    14  lineEditBigEndianUnicodeLen->setText(QString::number(source.length()*2));
    15  for(int i=0;i<source.length();i++)
    16  {
    17      const ushort unicode=u[i].unicode();
    18      lineEditBigEndianUnicode->insert(QString::number(unicode/256,16)+" ");
    19      lineEditBigEndianUnicode->insert(QString::number(unicode%256,16)+" ");
    20  }

    //Get Utf8
    21  QByteArray b=source.toUtf8();
    22  lineEditUtf8Len->setText(QString::number(b.length()));
    23  for(int i=0;i<b.length();i++)
    24  {
    25      const unsigned char a=b[i];
    26      lineEditUtf8->insert(QString::number(a,16)+" ");
    27  }

    //Get Local
```



```
28 QByteArray ba=source.toLocal8Bit();
29 lineEditLocalLen->setText(QString::number(ba.length()));
30 for(int i=0;i<ba.length();i++)
31 {
32     const unsigned char a=ba[i];
33     lineEditLocal->insert(QString::number(a,16)+" ");
34 }
35 }
```

第 1~4 行首先清空各控件的内容。

第 5 行获得用户输入的源字符串，保存在一个 QString 对象中。

第 6 行获得 QString 对象的 Unicode 编码，这是一个 QChar* 对象，QChar 提供了一种 16 位表示 Unicode 的方式，使用两个字节表示一个字符。

第 7 行获得源字符串的 Unicode 编码的长度，由于 Unicode 采用两个字节来表示一个字符，因此，这里通过源字符串长度的两倍得到 Unicode 的长度。

第 8~13 行获得 Unicode 编码。

第 10 行通过 QChar 的 unicode() 方法获得每一个字符的 Unicode 编码值，这是一个 ushort 类型的值，占两个字节。

第 11、12 行分别获得 ushort 值的低值和高值字节，由于 Unicode 采用的是 Little-Endian 方式，因此，低值字节存放在低地址字节上，高值字节存放在高地址字节上。

第 14 行获得源字符串的 Big-Endian 编码的长度，同 Unicode 长度。

第 15~20 行获得 Big-Endian Unicode 编码。

第 17 行通过 QChar 的 unicode() 方法获得每一个字符的 Unicode 编码值，这是一个 ushort 类型的值，占两个字节。

第 18、19 行分别获得 ushort 值的高值和低值字节，由于这里采用的是 Big-Endian 方式，因此，高值字节存放在低地址字节上，低值字节存放在高地址字节上。

第 21 行通过 QString 的 toUtf8() 方法得到源字符串的 UTF-8 编码序列，这是一个 QByteArray 对象，QByteArray 提供了一种字节流的表示方式。

第 22 行获得该字节流的长度信息，这就是源字符串的 UTF-8 编码所占用的字节长度。

第 23~27 行分别对字节流的每一个字节按十六进制的形式显示，UTF-8 没有字节顺序的问题，因此 QByteArray 中存放的字节流顺序就是内存中存放的字节顺序。

第 28 行获得源字符串的字符内码编码序列，这也是一个 QByteArray 对象，通过 QString 的 toLocal8Bit() 可以获得本地 8 位字符内码编码的字节流。

第 29 行获得该字节流的长度信息，这就是源字符串的内码编码所占用的字节长度。

第 30~34 行分别对字节流的每一个字节按十六进制的形式显示。