第 10 章　主窗口和动作

绝大多数的 QApplication 都会管理一个单独的 QMainWindow。如图 10.1 所示，QMainWindow 拥有一些与大多数桌面应用程序相同的特性：

- 中央窗件
- 菜单
- 工具栏
- 状态栏
- 停靠区域

在绝大多数的应用程序中，QMainWindow 都会是(在此主窗口内的)所有 QAction，QWidget 以及 QObject 型堆对象的(祖)父对象。如示例 10.1 所示，为应用程序扩展该类，是常见的做法。

示例 10.1　src/widgets/mainwindow/mymainwindow.h

```
[ . . . . ]
class MyMainWindow : public QMainWindow {
    Q_OBJECT
 public:
    explicit MyMainWindow(QWidget* parent=0);
    void closeEvent(QCloseEvent* event);                    1

 protected slots:
    virtual void newFile();
    virtual void open();
    virtual bool save();
[ . . . . ]
```

1 对基类的函数进行重写，用以捕捉用户打算关闭该窗口的时机。

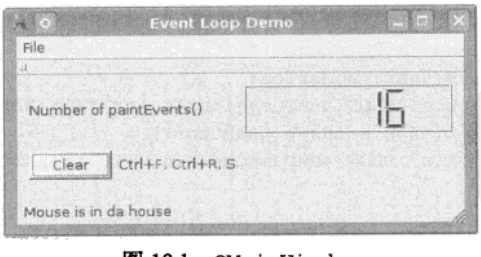

图 10.1　QMainWindow

10.1　QAction，QMenu 和 QMenuBar

QAction 从 QObject 派生而来，是用于用户选定动作的一个基类。它提供了丰富的接口，正如所看到的那样，它还可以用于许多种动作中。QWidget 接口使得每个窗件都可以维护一个 QList<QAction*>。

所有 QWidget 都可以拥有 QAction。一些窗件借助上下文菜单提供了 QAction 的清单，其他则借助菜单栏。有关如何为窗件提供上下文菜单的细节，可以参考 setContextMenuPolicy() 的 API 文档。

QMenu 是一个能够给 QAction 集合提供特殊视图的 QWidget。QMenuBar 是菜单的一个集合，常见于 QMainWindow 中。

若 QMenu 的父对象是 QMenuBar，QMenu 就会表现为一个拥有常规接口的下拉菜单；若其父对象不是 QMenuBar，它可以像对话框一样弹出，这种情况下，它就是一种上下文菜单[①]。当然，一个 QMenu 也可以把另外一个 QMenu 作为父对象，此时，前者就变成了子菜单。

为了帮助用户做出正确的选择，每个 QAction 都可拥有如下元素。

- 可显示于菜单或按钮上的文本或图标。
- 加速键或者快捷键。
- 一个“What's this?”和一个工具提示。
- 用来切换可见/不可见、启用/禁用、选中/未选中等动作状态的方法。
- changed()，hovered()，toggled() 和 triggered() 信号。

图 10.2 中的 QMainWindow 只有一个菜单栏，其中含有单一的菜单，提供了两个选项。

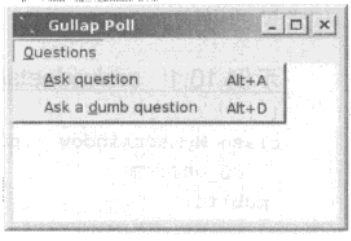

图 10.2 QMenu

示例 10.2 给出了创建该菜单栏的代码。QMainWindow::menuBar() 函数会返回一个指向 QMenuBar 的指针，它是 QMainWindow 的子对象。如果菜单栏尚未存在的话，menuBar() 函数会创建并返回一个指向空 QMenuBar 子对象的指针。

示例 10.2 src/widgets/dialogs/messagebox/dialogs.cpp

```
[ . . . . ]

    /* Insert a menu into the menubar. */
    QMenu* menu = new QMenu(tr("&Questions"), this);

    QMainWindow::menuBar()->addMenu(menu);

    /* Add some choices to the menu. */
    menu->addAction(tr("&Ask question"),
                    this, SLOT(askQuestion()), tr("Alt+A"));
    menu->addAction(tr("Ask a &dumb question"),
                    this, SLOT(askDumbQuestion()), tr("Alt+D"));
}
```

每次调用 QMenu::addAction(text, target, slot, shortcut) 函数，都会创建一个未命名的 QAction，并且会将其添加到这个 QMenu 上。然后会调用它的基类函数，QWidget::addAction(QAction *)，这样把新创建的 QAction 添加到用于上下文菜单的 QMenu 的 QAction 清单中。后者会在菜单的 QList<QAction*> 中添加该新动作。

① 上下文菜单一般通过单击鼠标右键进行调用，或者通过按下“菜单”按钮激活。之所以称为上下文菜单，是因为这种菜单总是依赖于上下文(哪一个 QWidget 或项被选中或者拥有焦点)。

10.1.1　Qaction，QToolbar 和 QActionGroup

因为应用程序可能会给用户提供各种不同的方式(例如，菜单、工具栏按钮、键盘快捷键)来发出同一个命令，将每个命令都封装成一个动作有助于确保整个应用程序拥有一致、同步的行为。QAction 发射信号并根据需要连接到槽上。

在 Qt GUI 应用程序中，动作通常使用以下几种方式进行触发。

- 用户单击菜单项。
- 用户按下快捷键。
- 用户单击工具栏按钮。

QMenu::addAction()有若干个重载形式。下面将使用继承自 QWidget 的版本，即 addAction(QAction *)，如示例 10.3 所示。

在这里，你会看到如何向菜单、动作群组和工具栏中添加动作。其中，首先是从 QMainWindow 派生一个类，并用数个 QAction 成员外加一个 QActionGroup 和一个 QToolBar 来对其进行封装。

示例 10.3　src/widgets/menus/study.h

```
[ . . . . ]
class Study : public QMainWindow {
    Q_OBJECT
 public:
    explicit Study(QWidget* parent=0);
 public slots:
    void actionEvent(QAction* act);
 private:
    QActionGroup* actionGroup;                                    1
    QToolBar* toolbar;                                            2

    QAction* useTheForce;
    QAction* useTheDarkSide;
    QAction* studyWithObiWan;
    QAction* studyWithYoda;
    QAction* studyWithEmperor;
    QAction* fightYoda;
    QAction* fightDarthVader;
    QAction* fightObiWan;
    QAction* fightEmperor;
protected:
    QAction* addChoice(QString name, QString text);

};
[ . . . . ]
```

1 为了捕捉信号。

2 为了将动作显示为按钮。

这个类的构造函数创建了菜单，然后将其安装到已经是基类一部分的 QMenuBar 中，具体可以参见示例 10.4。

示例 10.4 src/widgets/menus/study.cpp

```
[ . . . . ]

Study::Study(QWidget* parent) : QMainWindow(parent) {
    actionGroup = new QActionGroup(this);
    actionGroup->setExclusive(false);
    statusBar();

    QWidget::setWindowTitle( "to become a jedi, you wish?  " );
                                                                    1

    QMenu* useMenu = new QMenu("&Use", this);
    QMenu* studyMenu = new QMenu("&Study", this);
    QMenu* fightMenu = new QMenu("&Fight", this);

    useTheForce = addChoice("useTheForce", "Use The &Force");
    useTheForce->setStatusTip("This is the start of a journey...");
    useTheForce->setEnabled(true);
    useMenu->addAction(useTheForce);                                2
[ . . . . ]

    studyWithObiWan = addChoice("studyWithObiWan", "&Study With Obi Wan");
    studyMenu->addAction(studyWithObiWan);
    studyWithObiWan->setStatusTip("He will certainly open doors for you...");
    fightObiWan = addChoice("fightObiWan", "Fight &Obi Wan");
    fightMenu->addAction(fightObiWan);
    fightObiWan->setStatusTip("You'll learn some tricks from him"
                              " that way for sure!");
[ . . . . ]

    QMainWindow::menuBar()->addMenu(useMenu);
    QMainWindow::menuBar()->addMenu(studyMenu);
    QMainWindow::menuBar()->addMenu(fightMenu);

    toolbar = new QToolBar("Choice ToolBar", this);                 3
    toolbar->addActions(actionGroup->actions());

    QMainWindow::addToolBar(Qt::LeftToolBarArea, toolbar);


    QObject::connect(actionGroup, SIGNAL(triggered(QAction*)),
            this, SLOT(actionEvent(QAction*)));                     4

    QWidget::move(300, 300);
    QWidget::resize(300, 300);
}
```

1 这里所用到的一些“ClassName::”前缀并不是必须的，这是因为，这个函数可以根据“this”进行调用。类名称可以用来显式地调用某个基类版本，或者用来向读者说明调用的是类的哪个版本。

2 它已经在 QActionGroup 中了，但是仍然将其添加到 QMenu 中。

3 这将会给每个 QAction 在可停靠的窗件中添加一个可见按钮。

4 没有连接每个动作的信号，而是针对包含所有这些动作的 ActionGroup 进行了一次连接。

将每个单独的 QAction triggered()信号连接到独立的槽中也是可能的。示例 10.5 中，我们将相关的 QAction 一起连接到一个 QActionGroup 中。如果这个群组中的任意一个成员得到了触发，QActionGroup 就会发射一个单独的信号 triggered(QAction *)，这样就可以按照一种统一的方式来处理一组动作。该信号带有一个指向所触发的特定动作的指针，以便可以选择对应的响应。

示例 10.5　src/widgets/menus/study.cpp

```
[ . . . . ]


// Factory function for creating QActions initialized in a uniform way
QAction* Study::addChoice(QString name, QString text) {
    QAction* retval = new QAction(text, this);
    retval->setObjectName(name);
    retval->setEnabled(false);
    retval->setCheckable(true);
    actionGroup->addAction(retval);                          1
    return retval;
}
```

1 把每一个动作都添加到 QActionGroup 中，因此只需将信号连接到一个槽上。

在创建之后，每个 QAction 都会通过 addAction()被添加到其他三个对象中。

1. 一个 QActionGroup，用于信号处理。
2. 一个 QMenu，它是 QMenuBar 中三个可能的下拉菜单之一。
3. 一个 QToolBar，这里被渲染成一个按钮。

为了让这个例子更为有趣，在每两个菜单选择项之间创建了一些逻辑依赖关系，使其与各种电影的情节一致。这种逻辑关系在示例 10.6 中的 actionEvent()函数中进行了描述。

示例 10.6　src/widgets/menus/study.cpp

```
[ . . . . ]

void Study::actionEvent(QAction* act) {
   QString name = act->objectName();
   QString msg = QString();

   if (act == useTheForce ) {
       studyWithObiWan->setEnabled(true);
       fightObiWan->setEnabled(true);
       useTheDarkSide->setEnabled(true);
   }
   if (act == useTheDarkSide) {
       studyWithYoda->setEnabled(false);
       fightYoda->setEnabled(true);
       studyWithEmperor->setEnabled(true);
       fightEmperor->setEnabled(true);
```

```
        fightDarthVader->setEnabled(true);
    }


    if (act == studyWithObiWan) {
        fightObiWan->setEnabled(true);
        fightDarthVader->setEnabled(true);
        studyWithYoda->setEnabled(true);
    }
[ . . . . ]

    if (act == fightObiWan ) {
        if (studyWithEmperor->isChecked()) {
             msg = "You are victorious!";
        }
        else {
            msg = "You lose.";
            act->setChecked(false);
            studyWithYoda->setEnabled(false);
        }
    }
[ . . . . ]

    if (msg != QString()) {
       QMessageBox::information(this, "Result", msg, "ok");
    }
}
```

因为所有的动作都在一个 QActionGroup 中，可以只把一个 triggered(QAction*) 信号连接到 actionEvent() 槽上。

初始化时，所有菜单选项中只有一个选项是启用的。当用户从可用选项中进行选择时，其他的选项会变成启用的或者禁用的，如图 10.3 所示。值得注意的是，按钮和菜单中的选项之间存在一致关系。单击启用的按钮将会使得相应的菜单变成选中状态。QAction 存储了该状态(启用的/禁用的)，而 QMenu 和 QToolBar 提供了 QAction 的视图。

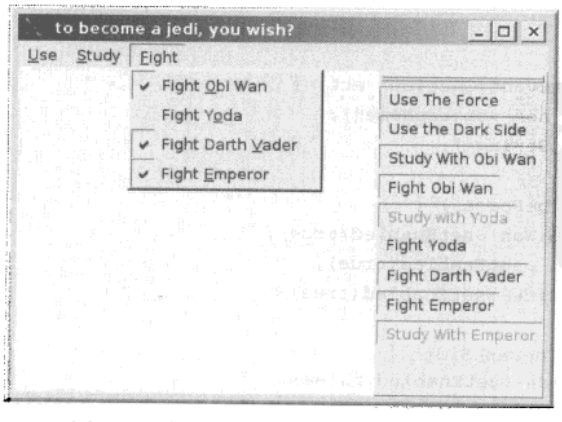

图 10.3 菜单和工具栏中可被选中的动作

10.1.2　练习：CardGame GUI

编写一个 21 点游戏(blackjack game)，其中包含下列动作：

- New game(开始新游戏)
- Deal new hand(新发一手牌)
- Shuffle deck(洗牌)
- Hit me(要另一手牌)
- Stay(计算手中的牌)
- Quit(退出游戏)

借助菜单栏和工具栏，如图 10.4 中应用程序的主窗口所示，这些动作应当是可以使用的。接下来介绍这个游戏的规则。

游戏开始时，会分别给玩家和庄家分发一手牌。每手牌刚开始都只有两张。玩家先玩，可通过单击“Hit Me”动作向其手中添加零次或者多次牌。每单击一次，就添加一张牌。如果玩家不再需要任何牌，那么可以单击“Stay”来触发相应的动作。

图 10.4　21 点游戏

游戏的目标是获得不大于 21 点的最大点数。

为了计算手中牌的点数，需将每一张花牌（J, Q 和 K）算作 10 点，而 A 可以算作 1 点或者 11 点，可根据哪种值最好而决定。其他牌的值与其牌面值相等。如果手中有一张 A 外加一张 J，则 A 应算作 11，因为这样的总点数就是 21。但是，如果手中有一张 8 外加一张 7，而手中又增加了一个 A，那么将 A 算作 1 更好一些。

如果玩家的点数大于 21 点，那么玩家在这一局就算“失败”(busted)(输)，这一局也就结束了。

如果玩家手中有 5 张牌，其总点数没有超过 21 点，那么玩家就会赢得这一局。

在玩家赢、输或者暂停之后，庄家可以根据需要单击多次，目的是使总点数大于 18。当达到这个状态时，庄家必须暂停，这一局也就结束了。如果玩家的点数更加接近 21 点，却没有超过 21 点，那么玩家就赢得此局。如果两者的点数相等，那么庄家赢。

一局结束后，玩家只能选择“Deal Hand”、“New Game”或者“Quit”(“Hit Me”和“Stay”会被禁用)。

在玩家选择“Deal Hand”之后，在当前一局完成之前应会禁用该选项。

跟踪庄家和玩家各自赢的局数，开始时分别初始化为 0，每赢一局就对应地为赢家加 1。在扑克牌的上方显示这两个数字。

如果用户没有选择“Shuffle Deck”或者牌桌还有扑克牌，需不断地发牌而不能重新洗牌。

可以尝试复用或者扩展 6.9.1 节中开发的 `CardDeck` 类及其相关类。通过给每张牌显示一个图片来为自己的游戏添加图形化表示，大致如在 9.5 节和 9.6 节中显示的那样。

为每个 `QAction` 提供一个下拉菜单和一个工具栏。确保 Hit Me 和 Stay 按钮只有在游戏开始之后才能使用。

在窗口顶部用一个只读的 QSpinBox 来显示还剩多少张牌。

新游戏应该将胜负次数归零并将扑克牌归位。

设计建议

尝试将模型类与视图类区分开来，而尽量不要将 GUI 添加到模型类中。总是尽量将模型和视图之间的代码分开是一种良好的编程风格，这样也会让你的项目从头至尾都变得更易于管理。

图 10.5 给出了一种可能的设计方案。

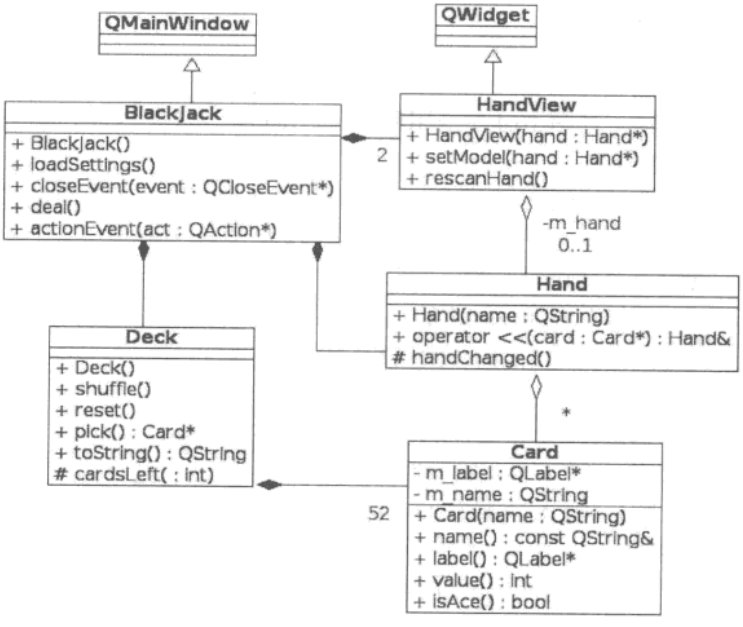

图 10.5 解决方案中的一种设计

10.2 区域和 QDockWidget

任何 QMainWindow 的派生类都有一些停靠窗口区域，分别位于中央窗件的四条边上，如图 10.6 所示。这四个区域可以用来向中央窗件添加二级窗口，也可以称为停靠窗口(dock window)。

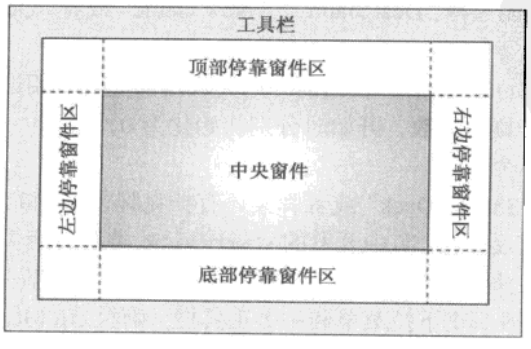

图 10.6 QMainWindow 的四个停靠区域

QDockWidget 可以看作是另外一个窗件的封装。它有一个标题栏、一个可以包含其他窗件的内容区域。根据属性设置值的不同，终端用户可以将 QDockWidget 拖离（以便让它“浮动”）、改变大小、关闭、拖到不同的位置，或者是将其停靠到相同或者不同的停靠窗件区域上。让数个停靠窗件同时占用同一个停靠区域是没有问题的。

QMainWindow 在中央窗件和 QDockWidget 之间恰当地创建了数个可以滑动的 QSplitter。有两个主要的 QMainWindow 函数可以用来管理停靠的窗口区域。

1. setCentralWidget(QWidget *)，此函数的作用是确立中央窗件。
2. addDockWidget(Qt::DockWidgetAreas, QDockWidget *)，此函数的作用是把给定的 QDockWidget 添加到指定的窗口区域。

在集成开发环境中，这些停靠的窗口非常重要，因为在不同的情况下往往需要不同的工具或者视图。每个停靠窗口都是一个可使用停靠机制轻易地将视图拖进主窗口中的窗件，如图 10.7 所示。

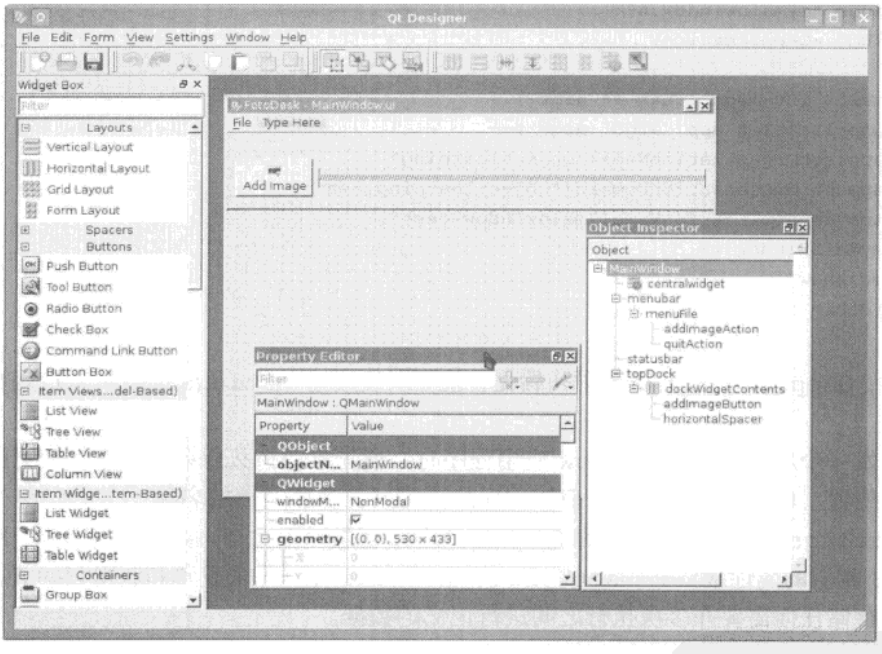

图 10.7　包含几个 QDockWindow 的设计师界面

正如大多数 Qt 应用程序一样，用于设计和构建 GUI 的工具——设计师，也大量使用了停靠窗口。设计师有一个窗件编辑器（中央窗件）和一些用于工具中的可停靠视图，例如：

- Widget Box（窗件工具箱）
- Object Inspector（对象检查器）
- Property Editor（属性编辑器）
- Signal/Slot Editor（信号/槽编辑器）
- Action Editor（动作编辑器）
- Resource Editor（资源编辑器）

这些可停靠窗件并非都是需要同时显示的，因此会有一个视图(View)菜单来允许对它们进行选择或者取消选择。QMainWindow::createPopupMenu()函数可以返回这个菜单，从而可以从上下文菜单中使用它们，也可以将其添加到工具栏或者下拉菜单中。

10.3 QSettings：保存和恢复应用程序的状态

大多数桌面应用程序都有让用户对设置进行配置的方式。而且，所有的设置、喜好和可选项都是需要是持久不变的。实现这一目标的机制都包含在 QSettings 中。

QSettings 在首次使用之前必须用应用程序的名称、组织名和组织域进行初始化。这样可以防止一个应用程序的设置值不小心覆盖了另一个应用程序的设置值。然而，如果 QApplication 自身设置了这些信息，如在示例 10.7 中给出的那样，则 QSettings 的默认构造函数将会使用这些值。

示例 10.7 src/widgets/mainwindow/mainwindow-main.cpp

```
#include "mymainwindow.h"
#include <QApplication>

int main( int argc, char ** argv ) {
    QApplication app( argc, argv );
    app.setOrganizationName("objectlearning");
    app.setOrganizationDomain("objectlearning.net");
    app.setApplicationName("mainwindow-test");
    MyMainWindow mw;
    mw.show();
    return app.exec();
}
```

现在，QApplication 得到了初始化，就可以使用 QSettings 的默认构造函数创建一些实例。

当开发一个新的 QMainWindow 应用程序时，最先希望持久保存的设置值很可能是窗口的大小和位置。也可能希望保存由应用程序最近打开的那些文档的名称。

QSettings 会管理键/值对的永久映射关系。它是一个 QObject，并且会使用一些与 QObject 相似的属性接口——setValue()和 value()——来设置和获得它的值。这个类可以用来存储任何需要在多次执行之间进行记忆的数据。

QSettings 需要一个组织名和一个应用程序名，但当使用默认的构造函数时，QSettings 会从 QApplication 中获得这些值。每个名称组合都会定义一个唯一的永久映射，这使得不会与其他命名的 Qt 应用程序产生冲突。

Monostate 模式

允许多个实例共享相同状态的类，可以看成是 Monostate 模式的一种实现。拥有相同组织/应用程序名称的两个 QSettings 实例，可以用来访问同一个永久映射数据。这简化了应用程序从不同源文件访问公共设置值的过程。

QSettings 是 Monostate 模式的一种实现。

对于 QSettings 数据永久存储的具体机制与实现方法相关且非常灵活。数据存储中的

一些可能实现甚至包括了 Win32 的注册项(Windows 平台)和$HOME/.settings 目录(Linux 平台)。更多详情，可以参阅 Qt 的 QSettings API 文档。

QMainWindow::saveState()函数返回一个包含有主窗口工具条和停靠窗件信息的 QByteArray。为了达到这一目的，使用了每个子窗件的 objectName 属性，因此确保每个名称都不相同是非常重要的。saveState()有一个可选的 int versionNumber 参数。这个 QSettings 对象用关键字符串"state"来存储 QByteArray。

QMainWindow::restoreState()带一个 QByteArray，大概是由 saveState()创建的，它使用其中包含的信息来将工具体和停靠的窗件放入一个之前的镜像中。该函数也有一个可选的 versionNumber 参数。在读取这些设置值后会创建其对象，如示例 10.8 所给出的那样。

示例 10.8　src/widgets/mainwindow/mymainwindow.cpp

```
[ . . . . ]

void MyMainWindow::readSettings() {
    QSettings settings;
    QPoint pos = settings.value("pos", QPoint(200, 200)).toPoint();
    QSize size = settings.value("size", QSize(400, 400)).toSize();
    QByteArray state = settings.value("state", QByteArray()).toByteArray();
    restoreState(state);
    resize(size);
    move(pos);
}
```

应当在用户打算退出应用程序后、窗口关闭前写入设置值。使用事件处理器会更为恰当一些，以便在窗件响应之前可由自己来处理这些事件。示例 10.9 给出了一个用于 closeEvent()的事件处理器，用来向 QSettings 存储这些信息。

示例 10.9　src/widgets/mainwindow/mymainwindow.cpp

```
[ . . . . ]

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


void MyMainWindow::writeSettings() {
    /* Save postion/size of main window */
    QSettings settings;
    settings.setValue("pos", pos());
    settings.setValue("size", size());
    settings.setValue("state", saveState());
}
```

10.4 剪贴板和数据传输操作

某些时候，将数据从一个地方取出并“发送”到另一个地方是很有必要的。一种方式是使用剪贴板的“剪切并粘贴”，另一种方式是“拖动并放下”。对于这两种操作，用于数据传输的类是相同的。

每个 Qt 应用程序都可以使用 qApp->clipboard()访问系统的剪贴板。剪贴板会保存带类型的数据(文本、图片、URL 或者自定义数据)。要往剪贴板中放置数据，可以创建一个 QMimeData，以一定的方式对数据进行编码，并且调用 QClipBoard->setMimeData()。

- setText()用来保存文本。
- setHtml()用来保存富文本。
- setImageData()用于图片。
- setUrls()用于 URL 或者文件名的列表。

示例 10.10 中，将系统剪贴板变化的信号和 MainWindow 的 clipboardChanged()槽进行了连接。无论何时，只要任意应用程序往剪贴板上复制了任何东西，都会触发该信号。可以运行这个例子并看看当从 QTextEdit 或者其他应用程序复制数据时可以使用哪种数据类型。

示例 10.10 src/clipboard/mainwindow.cpp

```
[ . . . . ]
MainWindow::MainWindow(QWidget* parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    connect (qApp->clipboard(), SIGNAL(changed(QClipboard::Mode)),
             this, SLOT(clipboardChanged(QClipboard::Mode)));
}

void MainWindow::clipboardChanged(QClipboard::Mode) {
    QStringList sl;
    QClipboard *cb = qApp->clipboard();
    const QMimeData *md = cb->mimeData();
    sl << "Formats: " + md->formats().join(",");
    foreach (QString format, md->formats()) {
        QByteArray ba = md->data(format);
        sl << "    " + format + ": " + QString(ba);
    }
    ui->clipboardContentsEdit->setText(sl.join("\n"));
}
[ . . . . ]
```

图 10.8 给出了从 QTextEdit 复制文本时发生的变化。这种情况下，剪贴板数据以三种方式进行编码：普通文本、HTML 和 OASIS 开放文档格式。选择何种格式取决于在粘贴或者放下对象时所需的数据类型。

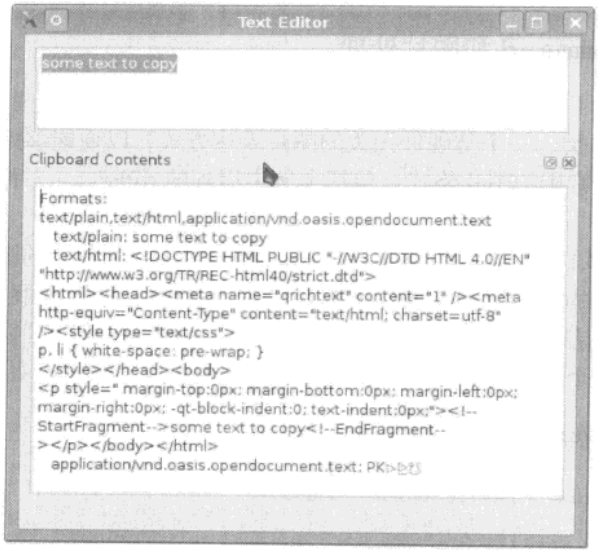

图 10.8　系统的剪贴板演示

10.5　命令模式

在软件中，可能会提供不同的撤销动作(undo)来降低用户的焦虑感并鼓励其尝试新功能。与确认对话框相比，这可能是一种更好的选择。Qt 提供了一些基本的 undo 功能，以使编写带 undo 功能的应用程序更为简单。

命令模式

命令模式，如文献[Gamma95]描述的那样，把操作封装为具有公共执行接口的对象。这样使得将操作放置在一个队列之中、日志维护操作以及取消已执行操作的影响成为可能。该模式完美的实现还可能提供一种处理错误或者异常情况的公用位置。

Qt 类中的 `QUndoCommand`，`QRunnable` 和 `QAction`，可认为在一定程度上实现了这一模式。

利用 Qt 实现命令模式非常简单：

- 可以创建一些命令并在适当的容器(例如，`QQueue`)内对它们进行排队。
- 通过把 `QUndoCommand` 放到 `QUndoStack` 上即可得到执行。
- 如果需要同时执行各个命令，则可以从 `QRunnable` 派生它们，并使用 `QtConcurrent::Run()` 在线程池中对它们进行调度[①]。
- 或许会把一些命令序列化到文件并在随后再次对它们进行调用(也有可能是在另一台机器上)，以用来实现批处理或者分布式执行[②]。

① 17.2 节会讨论线程。

② 17.4 节会讨论序列化模式(Serializer Pattern)。

10.5.1 QUndoCommand 和图片处理

下面的例子显示了 QUndoCommand 的用法，这个程序用到了一些图片处理的操作[①]。这个例子使用 QImage 类，一个与硬件无关的图片表达方式，以便能够处理每个单独的像素。QImage 支持多种最常用的图片格式，包括本例中用到的 JPEG[②]，一种对摄影图像进行有损压缩的系统。

首先，如示例 10.11 所示，从 QUndoCommand 派生一系列典型的图片处理操作。第一个操作通过对每个像素的红、绿和蓝成分进行双倍乘法运算而修正其颜色。第二个操作会根据用户给定的参数，或者水平或者竖直地对图片的另一半进行镜像而替换一半图片。每个操作的构造函数都带有一个源 QImage 的引用，并用同样的尺寸和格式初始化一个空的 QImage。

示例 10.11 src/undo-demo/image-manip.h

```
[ . . . . ]
class AdjustColors : public QUndoCommand {
public:
   AdjustColors(QImage& img, double radj, double gadj, double badj)
     : m_Image(img), m_Saved(img.size(), img.format()), m_RedAdj(radj),
     m_GreenAdj(gadj), m_BlueAdj(badj)   {setText("adjust colors"); }
   virtual void undo();
   virtual void redo();
private:
    QImage& m_Image;
    QImage m_Saved;
    double m_RedAdj, m_GreenAdj, m_BlueAdj;
    void adjust(double radj, double gadj, double badj);

};

class MirrorPixels : public QUndoCommand {
public:
   virtual void undo();
   virtual void redo();
[ . . . . ]
```

这两个操作都会在修改原始图片的任何像素之前生成一个它的备份。示例 10.12 给出了这些操作类中的一个实现，AdjustColors。它的构造函数会通过遍历每个 QImage 的所有像素并对每个像素都调用 pixel() 函数。pixel() 函数会返回颜色的 ARGB 四元组，这是一个无符号、8 字节的 int 值，格式为 AARRGGBB，其中每个字节都表示颜色的一个分量。我们使用 qRed()，qGreen() 和 qBlue() 函数对这个四元组（已由 typedef QRgb 给定）进行操作，以提取独立的三原色值，它们都在 0 到 255 之间[③]。然后它会用修正后的红、绿、蓝的值替换该像素。需要记住的是，颜色的修正操作是用 int 值乘以双精度浮点数后再将乘积赋值给 int 变量的，这就产生了截断(truncation)。换句话说，该调整操作是不能通过执行逆乘操作进行复原的。

① 这个例子的灵感来自 Guzdial 和 Ericson 所做的工作[Guzdial07]以及他们的 MediaComp 项目。
② 参见http://www.jpeg.org。
③ 第四个数值对保存 alpha 分量的值，它用来表示像素的透明度。该值的默认值是 ff(255)，表示不透明。

undo()函数会返回已存储的图像备份，redo()函数则会调用像素修改功能。

示例 10.12　src/undo-demo/image-manip.cpp

```
[ . . . . ]

void AdjustColors::adjust(double radj, double gadj, double badj) {
   int h(m_Image.height()), w(m_Image.width());
   int r, g, b;
   QRgb oldpix, newpix;
   m_Saved = m_Image.copy(QRect()); // save a copy of entire image
   for(int y = 0; y < h; ++y) {
      for(int x = 0; x < w; ++x) {
         oldpix = m_Image.pixel(x,y);
         r = qRed(oldpix) * radj;
         g = qGreen(oldpix) * gadj;
         b = qBlue(oldpix) * badj;
         newpix = qRgb(r,g,b);
         m_Image.setPixel(x,y,newpix);
      }
   }
}

void AdjustColors::redo() {
   qDebug() << "AdjustColors::redo()";
   adjust(m_RedAdj, m_GreenAdj, m_BlueAdj);
}

void AdjustColors::undo() {
    qDebug() << "AdjustColors::undo()";
    m_Image = m_Saved.copy(QRect());
}
```

我们使用 QtCreator 来设计 GUI。在转换成 QPixmap 之后，QImage 会显示在屏幕上的 QLabel 中。图 10.9 给出了程序处理之前的一幅照片。

图 10.9　原始照片

图 10.10 给出了这幅照片在经 AdjustColors 和 MirrorPixels 操作处理之后的不成功的结果。

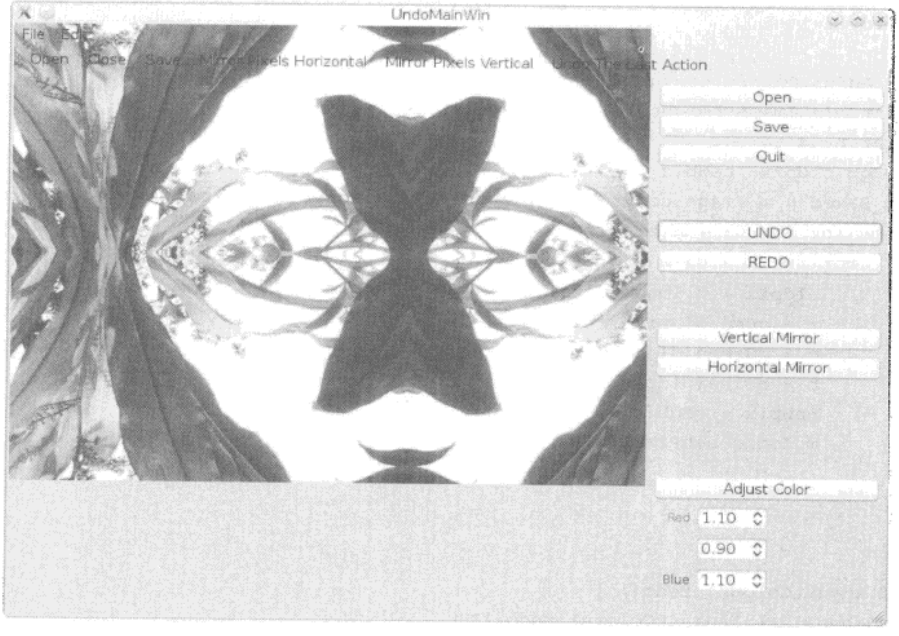

图 10.10 不成功的操作

UndoMainWin 类派生自 QMainWindow 并用到了 QUndoStack。默认情况下,QtCreator 会把 Ui 类作为指针成员嵌入到 UndoMainWin 中。示例 10.13 中,当从设计师中在窗件和动作上使用 Go to slot 特性时,私有槽会先从 QtCreator 生成的部分代码片段开始。

示例 10.13 src/undo-demo/undomainwin.h

```
#ifndef UNDOMAINWIN_H
#define UNDOMAINWIN_H

#include <QMainWindow>
#include <QUndoStack>

class QWidget;
class QLabel;
class QImage;
class QEvent;
namespace Ui {
    class UndoMainWin;
}

class UndoMainWin : public QMainWindow {
    Q_OBJECT
 public:
    explicit UndoMainWin(QWidget* parent = 0);
    ~UndoMainWin();
```

```
    public slots:
        void displayImage(const QImage& img);

    private:
        Ui::UndoMainWin* ui;
        QLabel* m_ImageDisplay;
        QImage m_Image;
        QUndoStack m_Stack;

    private slots:
        void on_redoButton_clicked();
        void on_openButton_clicked();
        void on_actionAdjust_Color_triggered();
        void on_actionUndo_The_Last_Action_triggered();
        void on_actionHorizontal_Mirror_triggered();
        void on_actionVertical_Mirror_triggered();
        void on_actionQuit_triggered();
        void on_actionSave_triggered();
        void on_actionClose_triggered();
        void on_saveButton_clicked();
        void on_quitButton_clicked();
        void on_adjustColorButton_clicked();
        void on_undoButton_clicked();
        void on_verticalMirrorButton_clicked();
        void on_horizontalMirrorButton_clicked();
        void on_actionOpen_triggered();
    };

    #endif // UNDOMAINWIN_H
```

示例 10.14 中可以看到 QtCreator 用来把各部分连接到一起的实现风格。还要注意的是，示例 10.13 中也给出了一个紧凑的私有槽完整列表。

QImage 在像素处理上得到了优化。QPixmap 使用了视频存储器(video memory)，它也是需要在屏幕上显示图片的多种窗件要用到的类。正如之前提到的，可以把 QImage 转换成 QPixmap 并将其在 QLabel 进行显示。

示例 10.14　src/undo-demo/undomainwin.h

```
[ . . . . ]
#include "image-manip.h"
#include "ui_undomainwin.h"
#include "undomainwin.h"

UndoMainWin::UndoMainWin(QWidget *parent)
: QMainWindow(parent), ui(new Ui::UndoMainWin),
  m_ImageDisplay(new QLabel(this)), m_Image(QImage()) {
  ui->setupUi(this);
  m_ImageDisplay->setMinimumSize(640,480);
}

UndoMainWin::~UndoMainWin() {
  delete ui;
```

```
}

void UndoMainWin::displayImage(const QImage &img) {
    m_ImageDisplay->setPixmap(QPixmap::fromImage(img));
}

void UndoMainWin::on_actionOpen_triggered() {
    m_Image.load(QFileDialog::getOpenFileName());
    displayImage(m_Image);
}

void UndoMainWin::on_horizontalMirrorButton_clicked() {
    MirrorPixels* mp = new MirrorPixels(m_Image, true);
    m_Stack.push(mp);
    displayImage(m_Image);
}

void UndoMainWin::on_adjustColorButton_clicked() {
    double radj(ui->redSpin->value()), gadj(ui->greenSpin->value()),
    badj(ui->blueSpin->value());
    AdjustColors* ac = new AdjustColors(m_Image, radj, gadj, badj);
    m_Stack.push(ac);
    displayImage(m_Image);
}
[ . . . . ]
```

1 既不是 QObject 也不是子对象，必须显式地将其删除。

10.5.1.1 练习：QUndoCommand 和图片处理

向示例 10.11 中添加一些其他的可撤销的图片处理操作。以下是一些值得尝试的做法。

1. 单色摄影术(monochrome)——将一幅三色图片转换成一幅灰度级单色图片。灰度是通过把所有三种颜色分量都设置为同一值而产生的。遗憾的是，如果只是简单地把图片中的每个像素的颜色都用三种颜色的平均值进行简单替换，那么所得的图片的总体效果将显得偏暗。产生可接受的灰度级图片的标准方法是基于纠正这样一个事实：蓝色是一种比红色更“深”一些的颜色。可以通过对这三种颜色中的每种颜色使用通用的权重因子的方法来调整每个像素[①]，如下所示。

   ```
   redVal *= 0.30; greenVal *= 0.59; blueVal *=0.11;
   ```

   然后，可以计算该像素的亮度值。亮度(luminance，或者称为强度，intensity)是一个等于三种颜色值进行加权平均后的 int 值。这种情况下，因为已经对它们进行了加权处理，因此，

   ```
   luminance = redVal + greenVal + blueVal
   ```

   最后，用该亮度值替换每一个像素的三种颜色值。
2. 底片——把一幅带三种颜色的图片转换成其负片值。要实现这一点，只需简单地把每一种颜色值 v 用 255 - v 进行替换即可。

① 例如，在 http://tinyurl.com/ydpjvgk 中就讨论了亮度值。

3. 打乱颜色——对于每一个像素，交换这些颜色的值，以便让红色值可以得到蓝色的初始值，绿色值可以得到红色的初始值，而蓝色值可以得到绿色的初始值。
4. 三重色——对于每个像素，计算它的颜色强度值 ci(其三种颜色的平均值)。如果 ci 低于 85，把它的红色和蓝色值降为 0。如果 ci 大于等于 85 但低于 170，那么把它的蓝色和绿色的值降为 0。如果 ci 是 170 或者更高，那么把它的红色和绿色的值降为 0。
5. 曝光边界——对于每个像素，把它的颜色强度值与它下方的像素的强度值进行比较。如果差值的绝对值超过阈值(以参数的形式提供)，就把它的颜色设置成黑色(三种颜色的所有值都设置成 0)；否则，把它的值设置成白色(三种颜色的所有值都设置成 255)。

10.6　tr()和国际化

如果所编写的程序可能会被翻译成另外一种语言(国际化)，那么 Qt Linguist 和 Qt 的翻译工具已经为解决这个问题提供了一种解决方案，它们提供了如何组织和何处放置翻译后字符串的方法。为了准备进行代码翻译，需要使用 QObject::tr()函数来包围应用程序中任何需要翻译的字符串。当将其用作非静态函数时，它会将 QMetaObject 所提供的对象的类名称，作为要翻译字符串分组的“上下文”。

tr()函数可以起到两个作用：

1. 它使得 Qt 的 lupdate 工具可以提取所有可翻译的字符串文字。
2. 如果翻译文件可用，并且该语言已被选中，该函数就会返回翻译后的字符串。

如果没有可以使用的翻译文件，那么 tr()函数返回原始字符串。

注意

确保每个可翻译的字符串都完全在 tr()函数内部且在编译时可被进行提取是非常重要的。对于带参数的字符串，可以使用 QString::arg()函数来将参数放置在翻译过的字符串中。例如

```
statusBar()->message(tr("%1 of %2 complete. progress: %3%")
                     .arg(processed).arg(total).arg(percent));
```

这样一来，翻译过程中就可以将参数按照不同的顺序进行放置以应对语言导致文字/思想的顺序发生改变。

下列工具可用于翻译过程。

1. lupdate——扫描查找设计师的.ui 文件和 C++文件中可供翻译的字符串。生成一个.ts 文件。
2. linguist——编辑.ts 文件并让用户可以进行翻译工作。
3. lrelease——读取.ts 文件并生成.qm 文件，它们可用来由应用程序载入其翻译。

有关如何使用这些工具的详情，可以参阅 linguist 参考手册[①]。

① 参见 http://doc.qt.nokia.com/latest/internationalization.html。

10.7 练习：主窗口和动作

1. 编写一个文本编辑器应用程序，以 QTextEdit 作为其中央窗件。
   - 在窗口标题栏中显示文件名并说明是否有需要保存的变化。
   - File 菜单：添加用于 Open、Save as 和 Quit 的动作。
   - Help 菜单：添加用于 About 和 About Qt 的动作。
   - 如果有需要保存的变化，退出时询问用户确认退出。
2. 编写一个允许用户从磁盘选择并打开文本（或者是富文本）文件的应用程序，可通过滚动来查看其内容。滚动视图应当每次至少显示 10 行文本。
   该应用程序也应当允许用户搜索文件中的一个字符串。如果找到了该字符串，含有字符串的那一行应当显示在滚动视图中，以便可以让用户在文件中看到它的内容。如果该字符串没有找到，应当在状态栏中显示一个适当的消息。
   用户应当可以单击按钮来查找下一个出现的地方或者回到上一次出现字符串的地方。也应当有 Close 按钮，可从显示中移除文件并请求用户选择另一个文件或者退出。
   图 10.11 是一种可能的解决方案的屏幕截图，其中的两个菜单包含了除 Clear Search（只是用来清空搜索的文本）之外的全部动作的副本。

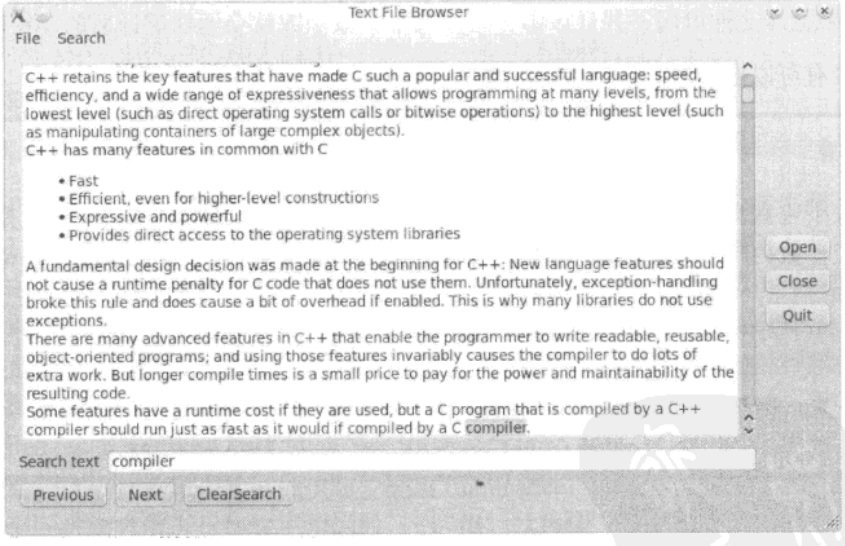

图 10.11 文本文件浏览器

3. QTextEdit 提供了用于 undo 和 redo 的方法。研究这些特性并讨论哪些操作可以进行 undo 和 redo。

10.8 复习题

1. QMainWindow 的主要特点是什么？
2. 如何在 QMainWindow 中安装 QMenuBar？

3. 可以采用什么方法来保存并随后再恢复一个 GUI 应用程序中窗件的大小、位置和排列次序？
4. 什么是中央窗件？
5. 如何让一个窗件变成中央窗件？
6. 停靠窗件是做什么用的？
7. 在一个应用程序中，可以有多少个停靠窗件？
8. 如何使用停靠窗件？
9. 什么是动作？
10. 如何使用动作？
11. 什么是动作群组？为什么愿意在应用程序中使用动作群组？
12. 如何让那些不懂英语的人也可以使用你的应用程序？