第 13 章　模型和视图

模型-视图设计框架提供了将底层数据类型集(模型)与呈现给用户的 GUI 类型集(视图)进行分离的工具和技术。模型通常用来组织具有平面表格结构或者层级树形结构的数据。这一章中将给出如何在 Qt 中使用模型类来呈现多种不同类型数据的方法。

在先前的几个示例中，我们看到了那些努力将表现数据的模型类与呈现用户界面的视图代码进行清晰划分的代码。加强这种分离有几个重要的原因。

首先，将模型和视图分离可减少复杂性。模型和视图代码有完全不同的维护准则——变化是由完全不同的因素驱动的——因此当它们保持分离时将更易于维护。此外，将模型与视图分离使得维护使用同一数据的若干个不同的但保持一致的视图成为可能。那些可复用于设计良好的模型类的专业视图类的数量正在不断地增长。

许多 GUI 工具箱都提供有列表、表格，还有树形视图，但是需要开发者将数据存储在视图中。Qt 有从相应的视图类派生出来的部件，如图 13.1 所示。对于没有使用过模型-视图框架的开发者来说，这些部件类或许比与之相对应的视图类容易学习一些。尽管如此，将数据存储在这些部件中导致了在用户界面和底层的数据结构之间产生了很强的依赖性。这些依赖导致在其他类型数据中或者在其他应用程序中复用这些部件相当困难。它也使得维护使用相同数据的多个视图的一致性非常困难。因此，易用性和便利性(尤其在 Qt 设计师中)的代价是灵活性和复印性的降低。

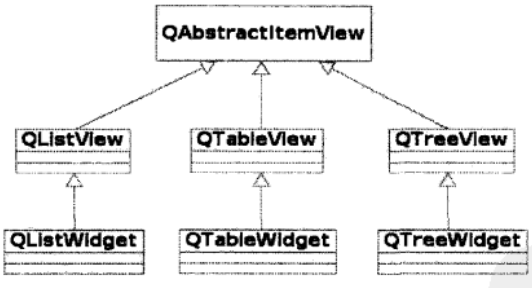

图 13.1　窗件和视图

13.1　模型-视图-控制器(MVC)

控制器代码管理在事件、模型和视图之间的相互交互。通常，工厂方法、委托以及创建和销毁代码都属于控制器的范畴。在 Qt 框架中，许多控制器机制可以在委托中找到。委托控制视图中单个项的渲染和编辑。视图提供了足以满足大多数场景的默认委托。尽管如此，如果有必要，我们可以通过从 `QAbstractItemModel` 派生一个自定义模型来优化默认委托渲染视图项的方式。

MVC：一个经典的设计模式

“四人组”(The Gang of Four)[Gamma95]提供了一个关于 MVC 的简要但精准的描述：

MVC 由三类对象组成。模型是应用程序对象，视图是它的屏幕展示，控制器定义了用户界面对用户输入的反应行为。在 MVC 概念之前，用户界面设计趋向于将这些对象归并在一起。MVC 将它们分拆开来以增强灵活性和复用性。

数据和角色

当获取和设置数据时，有一个可选的 role 参数允许指定 `Qt::ItemDataRole` 中的某个特定角色，该角色用于视图从模型中获取数据。一些角色指定了一般意义上的数据值，诸如 `Qt::DisplayRole`(默认)、`Qt::EditRole`(用于编辑的 `QVariant` 类型数据)以及 `Qt::ToolTipRole`(在工具提示中显示的 `QString` 类型数据)。其他角色用于描述外观，如 `Qt::FontRole`，使得默认委托可以指定一个特殊的 `QFont` 对象，或者 `Qt::TextAlignmentRole`，使得默认委托指定一个特殊的 `Qt::AlignmentFlag` 对象。`Qt::DecorationRole` 用于在视图中修饰数据值的图标。通常，应该在这个角色中使用一个 `QColor`，`QIcon` 或者 `QPixmap` 类型的数据值。`Qt::UserRole` 及其以上角色可以用于自定义数据值。可以将它们看成是表格模型中附加的数据列。

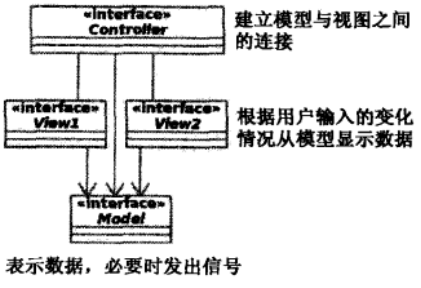

图 13.2 模型-视图-控制器类

如图 13.2 所示，模型-视图-控制器框架使用了多个设计模式，以支持多个视图使用相同数据的应用开发。它显示了模型代码（负责维护数据）、视图代码（负责以不同方式显示所有或部分数据）和控制器代码(负责处理影响到数据以及模型的事件，比如委托）要放置在分离的类中。这种分离使得在添加或删除视图和控制器时模型不需要做任何改变。它使得多个视图能够保持同步并且与模型保持一致，即使数据同时在不同的视图中被交互编辑。允许模型或视图均可被替换，使代码复用得以最大化。

控制器类的主要目标是封装控制器代码。一个复杂的应用可能在不同的组件或层级中具有多个控制器。

在 Qt 中，不同的控制器类的基类是 `QAbstractItemDelegate`。那些连接信号与槽的 connect 语句也可以认为是控制器代码。如你所见，将控制器代码放在模型和视图类之外能够获得额外的设计益处。

13.2 Qt 模型和视图

Qt 中包含有一个模型/视图框架，用于维护数据的组织管理和向用户的呈现方式之间的分离。三个最常用的视图类（列表、树和表格）都是默认提供的。另外，它还提供了抽象的和具体的数据模型，这些数据模型可被扩展和自定义以保存不同类型的数据。在应用中将一个模型以不同的方式同时呈现的情况比比皆是。

视图是获取、修改和呈现数据的对象。图 13.3 展示了 Qt 模型-视图框架中的四种视图。

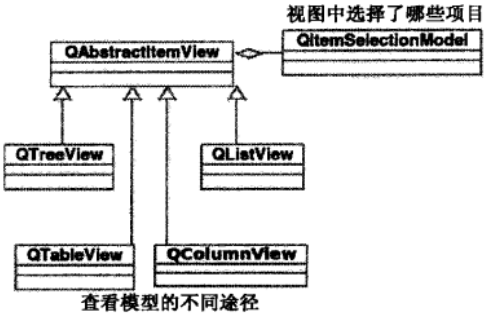

图 13.3　Qt 视图类

QAbstractItemModel 定义了视图(还有委托)访问数据的一个标准接口。模型中存储供显示和操作(例如排序、编辑、保存、获取、转换等)的具体数据。通过信号和槽，它们将数据的变化通知给所有相关联的视图。每个视图对象都有一个指向模型对象的指针。视图对象会频繁访问项模型的方法以获取或设置数据，或者做各种其他操作。图 13.4 展示了设计用来用各种视图类紧密配合的模型类。

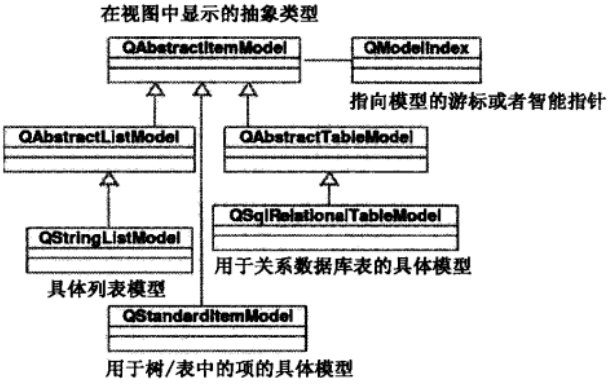

图 13.4　Qt 模型类

选择模型是用来描述在模型中被视图选中的那些数据项的对象。QModelIndex 的行为就像一个游标或智能指针，提供了统一的方式来遍历模型中的列表、树或表格项。在 setModel()方法调用之后，每当模型发生变化时视图会自动更新(假设模型是正确的)。

有两种方式来实现数据模型，每种方式都有各自的特性。

1. 重新实现 QAbstractItemModel 的被动接口，包括数据呈现。
2. 复用一个通用用途的具体的数据模型，比如 QStandardItemModel，填充其数据。

QAbstractItemModel 中的那些接口在实现上提供了更多的灵活性。使用为特定的访问模式或数据分布方式优化过的数据结构也是可行的。

第二种方式中复用了 QStandardItem(Model)类，使得可以用类似于使用 QListWidget，QTableWidget 和 QTreeWidget 的形式写树形或基于项的代码。

视图

QAbstractItemView 为这三个不同的模型类型的通用特性提供了接口：

- 各种布局的列表。
- 表格，或许带有交互元素。
- 表征父-子结构的树。

模型索引

模型中的每个数据项都用一个模型索引来表示。模型索引为视图和委托提供了在不知道其底层数据结构的情况下间接访问模型中数据项的方法。只有模型需要知道如何直接访问这些数据。QModelIndex 类提供了一个用于索引和访问派生自 QAbstractItemModel 的模型类中数据的接口。在列表、表格和树这几种模式下它工作得相当好。每一个索引都有一个指针指向创建它的模型，在具有层次关系的数据结构中(比如树)还可能有一个父项索引。QModelIndex 对待模型数据仿佛它们被安排在一个具有行和列索引的二维数组中一样，无论底层拥有这些数据是何种数据结构。

QModelIndex 对象由模型创建，可以被模型、视图或委托代码用于定位数据模型中的特定项。QModelIndex 对象具有很短的生命周期，可能在刚刚创建后就变成无效的状态，因此它们应该被立即使用而后丢弃。

如果使用一个在若干指令操作前已经存在的 QModelIndex，那么应该先调用 QModelIndex::isValid()。QPersistentModelIndex 对象具有更长的生命周期，但是在使用前仍应该先调用 isValid()方法来进行检查。

13.2.1 QFileSystemModel

QFileSystemModel 可以以列表、表格或树形视图呈现。图 13.5 展示了一个在 QTreeView 中显示的 QFileSystemModel 对象。

QFileSystemModel 早已将数据准备好了，所以可以简单地创建一个 QFileSystemModel 对象，创建一个视图，然后调用 view->setModel(model)。示例 13.1 展示了一个可能是最简单的 Qt 模型-视图示例。

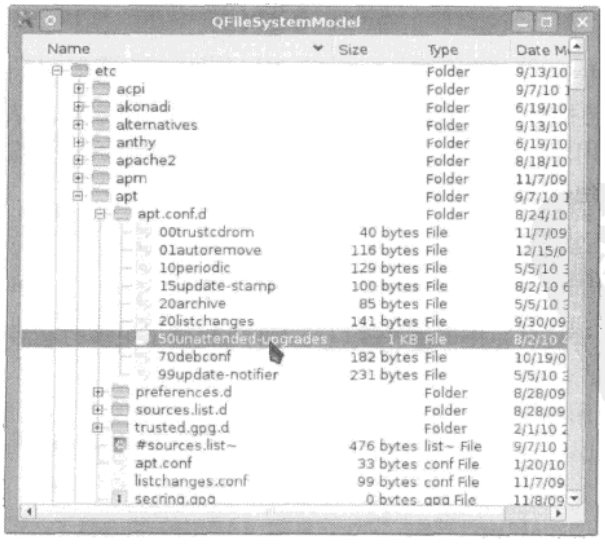

图 13.5 QTreeView 中的 QFileSystemModel

示例 13.1　src/modelview/filesystem/main.cpp

```
#include <QtGui>
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QFileSystemModel model;
    model.setRootPath("/");
    QTreeView tree;
    tree.setModel(&model);
    tree.setSortingEnabled(true);                                    1
    tree.header()->setResizeMode(QHeaderView::ResizeToContents);
    tree.resize(640, 480);
    tree.show();
    return app.exec();
}
```

1 使表头(HeaderView)的排序按钮可用。

通过设置 HeaderView 的 resizeMode 属性，表格或树中的列无论窗体大小是多少，都能够进行宽度调整。这些类是构建一个文件浏览部件的基本组成部分。

13.2.2　多重视图

示例 13.2 是一个用于创建可以以树、表格或简单的列表形式进行查看的 QStandardItemModel 的函数。这个函数也演示了 QStandardItem 类的使用。

示例 13.2　src/modelview/multiview/createModel.cpp

```
#include <QtGui>

QStandardItemModel* createModel(QObject* parent, int rows,
                                int cols, int childNodes) {
    QStandardItemModel*
        model = new QStandardItemModel(rows, cols, parent);
    for( int r=0; r<rows; r++ )
        for( int c=0; c<cols; c++)  {
            QStandardItem* item = new QStandardItem(
                QString("Row:%0, Column:%1").arg(r).arg(c) );
            if( c == 0 )                                             1
                for( int i=0; i<childNodes; i++ ) {
                    QStandardItem* child = new QStandardItem(
                                                QString("Item %0").arg(i) );
                    item->appendRow( child );
                }
            model->setItem(r, c, item);
        }
    model->setHorizontalHeaderItem( 0, new QStandardItem( "Name" ));
    model->setHorizontalHeaderItem( 1, new QStandardItem( "Value" ) );
    return model;
}
```

1 给第一列元素添加子节点。

示例 13.3 中所示的主程序创建了四个不同的视图：QListView，QTableView，QTreeView 和 QColumnView。注意，QTableView 和 QListView 并不显示子节点，另外，

QColumnView 和 QListView 也不显示表格模型中的非首列。QColumnView 在右边显示当前选中节点下的树子节点，这与 MacOS X Finder 中显示一个选中目录下的文件的方式相似。

所有的视图共享同一个模型，所以在一个单元格中编辑后立即会在其他视图中显示出来。另外，所有的视图共享一个选择模型，所以选择状态在四个视图中也是同步的。

示例 13.3 src/modelview/multiview/multiview.cpp

```
[ . . . . ]

#include "createModel.h"

int main( int argc, char** argv ) {
  QApplication app( argc, argv );
  QStandardItemModel* model = createModel(&app);
  QSplitter vsplitter(Qt::Vertical);
  QSplitter hsplitter;                                          1

  QListView list;
  QTableView table;
  QTreeView tree;
  QColumnView columnView;
  [ . . . . ]

  list.setModel( model );
  table.setModel( model );
  tree.setModel( model );                                       2
  columnView.setModel (model);
  [ . . . . ]

  list.setSelectionModel( tree.selectionModel() );
  table.setSelectionModel( tree.selectionModel() );             3
  columnView.setSelectionModel (tree.selectionModel());
  table.setSelectionBehavior( QAbstractItemView::SelectRows );
  table.setSelectionMode( QAbstractItemView::SingleSelection );
```

1 默认的，子项水平布局。

2 共享同一个模型。

3 公共选择模型。

当执行这段代码时，你会看到如图 13.6 所示的窗口。需要注意的是，在视图中选择一项时会导致在其他视图中该项也会被选中。因为我们使用的是具体的 QStandardItemModel，模型项对每个视图来说都是可编辑的。此外，从任何一个视图产生的变化都会自动影响到其他视图，从而确保每个视图与模型是统一的。

根据 QAbstractItemView::EditTriggers 的设置情况，可以通过 F2 键、双击鼠标键或者仅仅进入到某个单元格来触发编辑状态。你或许注意到了在本地 windows 环境中熟悉的其他快捷键(剪切、复制、粘贴、Ctrl+光标键等)在视图和编辑区域中同样可以使用。

对于这个应用来说，我们使用了 QSplitter 部件。QSplitter 具有一些布局方面的特性，但是所管理的窗件是它的子部件。分隔栏部件允许用户在运行时通过拖动子窗件之间的分隔条来调整子窗件所占据空间的大小。示例 13.4 包含了建立分隔部件的代码。

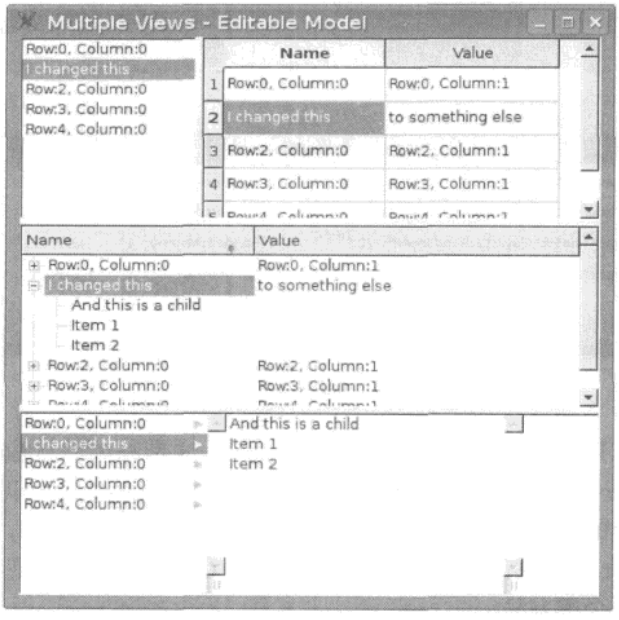

图 13.6　多重视图，同一个模型

示例 13.4　src/modelview/multiview/multiview.cpp

```
[ . . . . ]

  hsplitter.addWidget( &list );
  hsplitter.addWidget( &table );
  vsplitter.addWidget( &hsplitter );
  vsplitter.addWidget ( &tree );
  vsplitter.addWidget ( &columnView );

  vsplitter.setGeometry(300, 300, 500, 500);
  vsplitter.setWindowTitle("Multiple Views - Editable Model");
```

13.2.3　委托类

如图 13.7 所示，委托在模型和视图之间提供了另一层间接关联，它提升了自定义的可行性。

委托类通常派生于 QAbstractItemDelegate，为 Qt 的模型-视图框架中添加若干控制器特性。委托类能够提供一个工厂方法，以使得视图类能够创建编辑器以及虚函数 getter 和 setter，从模型中获取数据或者保存数据到模型中。它还能够提供一个 paint() 虚方法以自定义视图中项的显示。委托还能够设置为应用于整个 QAbstractItemView 或者仅仅应用于一列。

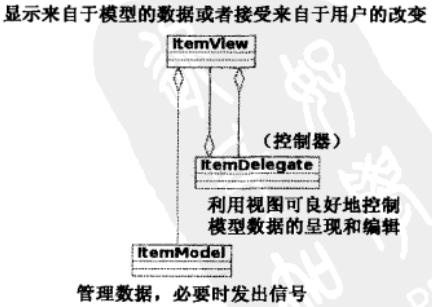

图 13.7　模型、视图和委托

图 13.8 展示了一个来自于$QTDIR/examples/modelview/stardelegate 的 StarDelegate 示例的修改版。

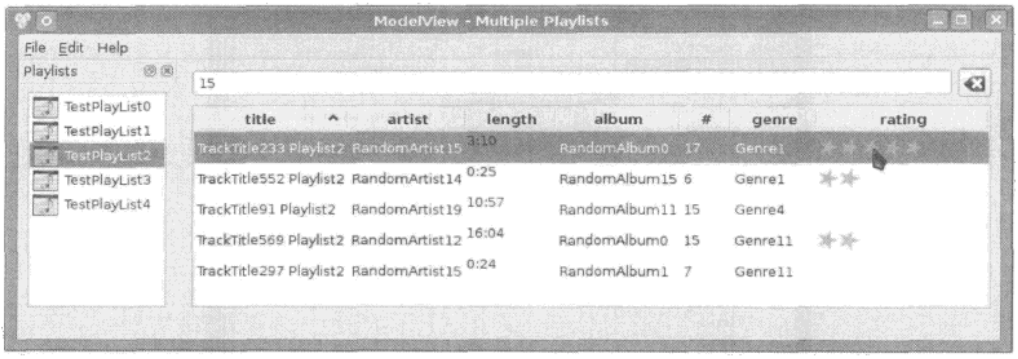

图 13.8 StarDelegate 示例

示例 13.5 中扩展了 QStyledItemDelegate。这是一个默认应用于 QListView，QTableView 和 QTreeView 中的具体类。它提供了一个 QLineEdit 以编辑 QString 类型属性，以及其他适合的编辑器以用于类型 boolean，QDate，QTime，int 或 double。对自定义的 StarRating 值，该自定义委托展示了要在表格中获得“星号”而不是简单的整数值，则必须重写一个虚方法。

示例 13.5 src/modelview/playlists/stardelegate.h

```
#ifndef STARDELEGATE_H
#define STARDELEGATE_H

#include <QStyledItemDelegate>
#include <QStyleOptionViewItem>
class StarDelegate : public QStyledItemDelegate {
    Q_OBJECT
public:
    typedef QStyledItemDelegate SUPER;
    StarDelegate(QObject* parent=0) : SUPER(parent) {};
    QWidget* createEditor(QWidget* parent,
                          const QStyleOptionViewItem& option,
                          const QModelIndex& index) const;
    void paint(QPainter* painter,
               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;
};

#endif // STARDELEGATE_H
```

通过重写 paint()方法，委托为视图中的项如何显示提供了全面的控制，如示例 13.6 所示。

示例 13.6　src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]

void StarDelegate::
    paint(QPainter* painter,
          const QStyleOptionViewItem& option,
          const QModelIndex& index) const {
    QString field = index.model()->headerData(index.column(),
                                              Qt::Horizontal).toString();
    if (field == "length") {
        QVariant var = index.data(Qt::DisplayRole);
        Q_ASSERT(var.canConvert(QVariant::Time));
        QTime time = var.toTime();
        QString str = time.toString("m:ss");
        painter->drawText(option.rect, str, QTextOption());
        // can't use drawDisplay with QStyledItemDelegate:
        // drawDisplay(painter, option, option.rect, str);
        return;
    }
    if (field != "rating") {
        SUPER::paint(painter, option, index);
        return;
    }
    QVariant variantData = index.data(Qt::DisplayRole);
    StarRating starRating = variantData.value<StarRating>();
    if (option.state & QStyle::State_Selected)
        painter->fillRect(option.rect, option.palette.highlight());
    starRating.paint(painter, option.rect, option.palette,
                     StarRating::ReadOnly);
}
```

另外，委托还能在用户触发一个项的编辑状态时，确定应显示何种类型的部件。为此，可以如示例 13.7 所示重载 `createEditor()`，或者提供一个自定义的 `QItemEditorFactory`。

示例 13.7　src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]

QWidget* StarDelegate::
    createEditor(QWidget* parent,
                 const QStyleOptionViewItem& option,
                 const QModelIndex& index) const {
    QString field = index.model()->headerData(index.column(),
                                  Qt::Horizontal).toString();
    if (field == "rating")  {
            return new StarEditor(parent);
    }
    if (field == "length") {
        return new TimeDisplay(parent);
    }
    return SUPER::createEditor(parent, option, index);
}v
```

什么是编辑触发器

有各种方式来“触发”视图使其进入编辑模式。在大多数桌面平台中，F2 键是“平台编辑”键。在手持设备上，它可能是一个手势，如两次轻触(double-tap)或一个特殊的按钮。请查看 QAbstractItemView::setEditTriggers(EditTriggers triggers)的 API 文档 http://doc.qt.nokia.com/latest/qabstractitemview.html#editTriggers-prop。

当触发一个编辑请求时，我们想要看到这样一个编辑器，其初始化值来自于你想查看或修改的模型。这是由 setEditorData()方法完成的，如示例 13.8 所示。

示例 13.8 src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]

void StarDelegate::
    setEditorData(QWidget* editor,
                  const QModelIndex& index) const {
    QVariant val = index.data(Qt::EditRole);
    StarEditor* starEditor = qobject_cast<StarEditor*>(editor);          1
    if (starEditor != 0) {
        StarRating sr = qVariantValue<StarRating>(val);                  2
        starEditor->setStarRating(sr);
        return;
    }

    TimeDisplay* timeDisplay = qobject_cast<TimeDisplay*>(editor);       3
    if (timeDisplay != 0) {
        QTime t = val.toTime();
        timeDisplay->setTime(t);
        return;
    }
    SUPER::setEditorData(editor, index);                                 4
    return;
}
```

1 动态类型检查。

2 从 QVariant 中抽取用户定义类型值。

3 动态类型检查。

4 让基类来处理其他类型。

当用户完成编辑后，如示例 13.9 所示会调用 setModelData()方法以将数据写回到 QAbstractItemModel。

示例 13.9 src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]

void StarDelegate::
    setModelData(QWidget* editor, QAbstractItemModel* model,
                 const QModelIndex& index) const {
    StarEditor* starEditor = qobject_cast<StarEditor*>(editor);
```

```
    if (starEditor != 0) {
        StarRating r = starEditor->starRating();
        QVariant v;
        v.setValue<StarRating>(r);
        model->setData(index, v, Qt::EditRole);
        return;
    }
    TimeDisplay* td = qobject_cast<TimeDisplay*>(editor);
    if (td != 0) {
        QTime t = td->time();
        model->setData(index, QVariant(t));
        return;
    }
    SUPER::setModelData(editor, model, index);
    return;
}
```

13.3　表格模型

下一个示例如图 13.9 所示，是一个使用表格模型的能够显示和编辑动作和对应快捷键的视图。为了演示 Qt 模型-视图类中所支持的不同数据角色的使用和显示，我们获取了一个 QAction 对象列表并将它显示在一个表格中。每一个动作都可以有一个图标、工具提示、状态提示以及其他用户数据。这直接对应到了四种可用数据角色。

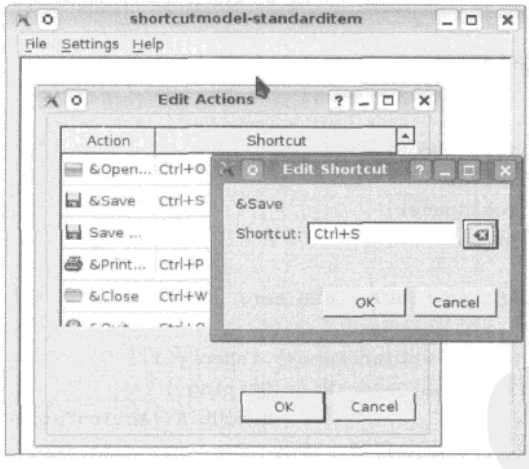

图 13.9　快捷键编辑器

13.3.1　标准模型与抽象模型的比较

当开发者刚开始熟悉 QStandardItem 时，他们有时会将其应用到一些或许并非是最佳选择的场景中。尽管 QStandardItemModel 使得无须从一个抽象基类派生即可轻易建立一个模型，如果你关注到 QStandardItem 中的数据与其他内存数据无法保持同步，或者用它创建标准项的初始化过程太繁琐，这就表明应该采用从 QAbstractItemModel 直接或间接派生的方式来建立模型。

示例 13.10 是一个基于 QStandardItem 类的快捷键表格模型。

示例 13.10 src/modelview/shortcutmodel-standarditem/actiontableeditor.h

```
[ . . . . ]
class ActionTableEditor : public QDialog {
    Q_OBJECT
public:
    ActionTableEditor(QWidget* parent = 0);
    ~ActionTableEditor();
protected slots:
    void on_m_tableView_activated(const QModelIndex& idx=QModelIndex());
    QList<QStandardItem*> createActionRow(QAction* a);
protected:
    void populateTable();
    void changeEvent(QEvent* e);
private:
    QList<QAction*> m_actions;
    QStandardItemModel* m_model;
    Ui_ActionTableEditor* m_ui;
};
[ . . . . ]
```

因为这是一个采用 Qt 界面设计师创建的窗口，所以自动生成的部件的创建和初始化代码看起来像示例 13.11 所示的样子。

示例 13.11 src/modelview/shortcutmodel-standarditem/actiontableeditor_ui.h

```
[ . . . . ]
class Ui_ActionTableEditor
{
public:
  QVBoxLayout *verticalLayout;
  QTableView *m_tableView;
  QSpacerItem *verticalSpacer;
  QDialogButtonBox *m_buttonBox;

  void setupUi(QDialog *ActionTableEditor)
  {
    if (ActionTableEditor->objectName().isEmpty())
      ActionTableEditor->setObjectName(QString::
                                       fromUtf8("ActionTableEditor"));
    ActionTableEditor->resize(348, 302);
    verticalLayout = new QVBoxLayout(ActionTableEditor);
    verticalLayout->setObjectName(QString::fromUtf8("verticalLayout"));
    m_tableView = new QTableView(ActionTableEditor);
    m_tableView->setObjectName(QString::fromUtf8("m_tableView"));
    verticalLayout->addWidget(m_tableView);
[ . . . . ]
```

示例 13.12 展示了如何在 QStandardItemModel 中创建数据行，每一行是一个 QAction 对象。

示例 13.12　src/modelview/shortcutmodel-standarditem/actiontableeditor.cpp

```
[ . . . . ]

QList<QStandardItem*> ActionTableEditor::
createActionRow(QAction* a) {
    QList<QStandardItem*> row;
    QStandardItem* actionItem = new QStandardItem(a->text());
    QStandardItem* shortcutItem =
        new QStandardItem(a->shortcut().toString());                    1
    actionItem->setIcon(a->icon());                                     2

    actionItem->setToolTip(a->toolTip());                               3
    actionItem->setFlags(Qt::ItemIsSelectable | Qt::ItemIsEnabled);     4
    shortcutItem->setFlags(Qt::ItemIsSelectable| Qt::ItemIsEnabled);
    shortcutItem->setIcon(a->icon());                                   5
    row << actionItem << shortcutItem;
    return row;
}

void ActionTableEditor::populateTable() {
    foreach (QWidget* w, qApp->topLevelWidgets())                       6
        foreach (QAction* a, w->findChildren<QAction*>()) {             7
            if (a->children().size() > 0) continue;                     8
            if (a->text().size() > 0) m_actions << a;
        }

    int rows = m_actions.size();
    m_model = new QStandardItemModel(this);
    m_model->setColumnCount(2);
    m_model->setHeaderData(0, Qt::Horizontal, QString("Action"),
                           Qt::DisplayRole);
    m_model->setHeaderData(1, Qt::Horizontal, QString("Shortcut"),
                           Qt::DisplayRole);
    QHeaderView* hv = m_ui->m_tableView->horizontalHeader();
    m_ui->m_tableView->
       setSelectionBehavior(QAbstractItemView::SelectRows);
    m_ui->m_tableView->
       setSelectionMode(QAbstractItemView::NoSelection);
    hv->setResizeMode(QHeaderView::ResizeToContents);
    hv->setStretchLastSection(true);
    m_ui->m_tableView->verticalHeader()->hide();
    for (int row=0; row < rows; ++row ) {
        m_model->appendRow(createActionRow(m_actions[row]));
    }
    m_ui->m_tableView->setModel(m_model);                               9
}
```

1 从 QAction 复制数据到 QStandardItem。

2 复制更多的数据。

3 复制更多的数据。

4 只读模型，没有 Qt::ItemIsEditable 选项。

5 复制更多的数据。
6 所有顶层部件。
7 所有可被找到的 QAction 对象。
8 跳过群组动作。
9 将视图和模型关联起来。

QStandardItem 有它自己的属性，所以我们从每个 QAction 对象中复制数据值到两个相关项中。当工作于大型数据模型时，这种类型的复制工作会有严重影响性能和内存开销。其主要问题就是创建模型时它有着显著的开销，当完成它以后就会抛弃它。

这个示例并没有使用视图的编辑特性以修改模型中的数据。这需要写一个委托来提供自定义编辑器部件，并将作为一个练习留给读者(见 13.6 节的练习 4)。相反，当用户激活一行时，将弹出了一个如示例 13.13 所示的对话框。当对话框返回 Accepted 时，QAction 对象的快捷键将完全跳过这个模型被直接设置。下次再显示这个快捷键表格时，模型必须从动作列表重新生成，否则我们需要正确的处理变化。

示例 13.13 src/modelview/shortcutmodel-standarditem/actiontableeditor.cpp

```
[ . . . . ]

void ActionTableEditor::
on_m_tableView_activated(const QModelIndex& idx) {
    int row = idx.row();
    QAction* action = m_actions.at(row);
    ActionEditorDialog aed(action);
    int result = aed.exec();                                    1
    if (result ==  QDialog::Accepted) {                         2
        action->setShortcut(aed.keySequence());                 3
        m_ui->m_tableView->reset();
    }
}
```

1 弹出模式对话框以供编辑动作的快捷键。
2 这将是检查重复的/不明确的绑定的好地方。
3 跳过模型直接设置 QAction 属性。

一种更好的方法

示例 13.14 是一个扩展自 QAbstractTableModel 的表格模型，它重新实现 data()和 flags()这两个纯虚方法以提供对模型数据的访问。模型是一个对已在内存中存在的 QAction 对象列表的代理，这意味着不再需要在模型数据中进行数据复制。

示例 13.14 src/libs/actioneditor/actiontablemodel.h

```
[ . . . . ]
class ACTIONEDITOR_EXPORT ActionTableModel : public QAbstractTableModel {
    Q_OBJECT
public:
    explicit ActionTableModel(QList<QAction*> actions, QObject* parent=0);
    int rowCount(const QModelIndex&  = QModelIndex()) const {
        return m_actions.size();
```

```
    }
    int columnCount(const QModelIndex& = QModelIndex()) const {
        return m_columns;
    }
    QAction* action(int row) const;
    QVariant headerData(int section, Qt::Orientation orientation,          1
        int role) const;
    QVariant data(const QModelIndex& index, int role) const;              2
    Qt::ItemFlags flags(const QModelIndex& index) const;                  3
    bool setData(const QModelIndex& index, const QVariant& value,
        int role = Qt::EditRole);                                         4

protected:
    QList<QAction*> m_actions;
    int m_columns;
};
[ . . . . ]
```

1 可选重写。

2 必须重写。

3 必须重写。

4 对可编辑模型是必须的。

示例 13.15 展示了 data() 的实现。注意，对 QAction 对象的许多不同属性，这里都有一个相应的数据角色。可以将角色(尤其是用户角色)看成是一个数据附加列。

示例 13.15　src/libs/actioneditor/actiontablemodel.cpp

```
[ . . . . ]

QVariant ActionTableModel::
data(const QModelIndex& index, int role) const {
    int row = index.row();
    if (row >= m_actions.size()) return QVariant();
    int col = index.column();
    if (col >= columnCount()) return QVariant();
    if (role == Qt::DecorationRole)
        if (col == 0)
            return m_actions[row]->icon();

    if (role == Qt::ToolTipRole) {
        return m_actions[row]->toolTip();
    }
    if (role == Qt::StatusTipRole) {
        return m_actions[row]->statusTip();
    }
    if (role == Qt::DisplayRole) {
        if (col == 1) return m_actions[row]->shortcut();
        if (col == 2) return m_actions[row]->parent()->objectName();
        else return m_actions[row]->text();
    }
    return QVariant();
}
```

从它无须创建/复制任何数据的意义上讲，ActionTableModel 是轻量级的。只有视图请求时它才会呈现数据给视图。这意味着有可能在数据模型底层实现一个松散的数据结构。这还意味着模型可以从另一个数据源以一种懒惰的方式获取数据，如同 QSqlTableModel 和 QFileSystemModel 那样。

13.3.2 可编辑模型

对于可编辑模型，必须重写 flags()和 setData()。如果想就地编辑(在真实视图中)，就需要在 flags()函数中返回 Qt::ItemIsEditable 标识。因为在某项被单击时依然会弹出一个 ActionEditorDialog，而不需要就地编辑，所以示例 13.16 中简单地返回了 Qt::itemIsEnabled。

示例 13.16 src/libs/actioneditor/actiontablemodel.cpp

```
[ . . . . ]

Qt::ItemFlags ActionTableModel::
flags(const QModelIndex& index) const {
    if (index.isValid()) return Qt::ItemIsEnabled;
    else return 0;
}
```

示例 13.17 展示了在 setData()中如何在实际设置新值之前检查不明确的快捷键。数据修改之后，发射 dataChanged()信号是很重要的，这样还在显示旧数据的视图可以知道应该从模型中获取新数据了。

示例 13.17 src/libs/actioneditor/actiontablemodel.cpp

```
[ . . . . ]

bool ActionTableModel::
setData(const QModelIndex& index, const QVariant& value, int role) {
    if (role != Qt::EditRole) return false;
    int row = index.row();
    if ((row < 0) | (row >= m_actions.size())) return false;
    QString str = value.toString();
    QKeySequence ks(str);
    QAction* previousAction = 0;

    if (ks != QKeySequence() ) foreach (QAction* act, m_actions) {
        if (act->shortcut() == ks) {
            previousAction = act;
            break;
        }
    }
    if (previousAction != 0) {
        QString error = tr("%1 is already bound to %2.").
                arg(ks.toString()).arg(previousAction->text());
        bool answer = QMessageBox::question(0, error,
                    tr("%1\n Remove previous binding?").arg(error),
                    QMessageBox::Yes, QMessageBox::No);
```

```
        if (!answer) return false;
        previousAction->setShortcut(QKeySequence());
    }
    m_actions[row]->setShortcut(ks);
    QModelIndex changedIdx = createIndex(row, 1);                    1
    emit dataChanged(changedIdx, changedIdx);                        2
    return true;
}
```

1 列 1(第二列)显示快捷键。

2 视图需要它以便知道什么时间什么内容要更新。

要支持数据行的插入/删除，有类似的信号 `rowsInserted()`和 `rowsRemoved()`存在，它们必须从实现函数 `insertRows()`/`removeRows()`内发射。

在一个或多个快捷键修改后，将它们保存到 `QSettings` 对象中。示例 13.18 展示了如何对需要保存的 `QAction` 对象进行持续跟踪。

示例 13.18　src/libs/actioneditor/actiontableeditor.cpp

```
[ . . . . ]

void ActionTableEditor::
on_m_tableView_activated(const QModelIndex& idx) {
    int row = idx.row();
    QAction* action = m_model->action(row);
    ActionEditorDialog aed(action);

    int result = aed.exec();
    if (result ==  QDialog::Accepted) {
        QKeySequence ks = aed.keySequence();
        m_model->setData(idx, ks.toString());
        m_changedActions << action;
    }
}
```

示例 13.19 展示了在用户在对话框中选择 Accepted 后如何保存这些快捷键到 `QSettings` 对象中。

示例 13.19　src/libs/actioneditor/actiontableeditor.cpp

```
[ . . . . ]

void ActionTableEditor::accept() {
    QSettings s;
    s.beginGroup("shortcut");
    foreach (QAction* act, m_changedActions) {
        s.setValue(act->text(), act->shortcut() );
    }
    s.endGroup();
    QDialog::accept();
}
```

13.3.3 排序和过滤

不要劳神去查找用于创建图 13.10 中的 QLineEdit、清除按钮或二者之间的 connect 代码。它们都是在 Qt 设计师中定义的，并且由 uic 自动生成。

这得感谢 QSortFilterProxyModel，用少于 5 行的代码就可以为一个已存在的模型增加排序/过滤功能。图 13.11 展示了该代理是如何出现在视图和模型之间的。

图 13.10 过滤后的表格视图

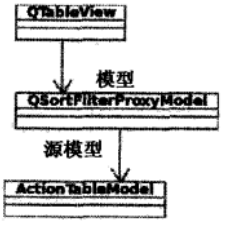

图 13.11 排序过滤代理

示例 13.20 展示了在前面的动作表格示例中设置排序过滤代理需要什么。

示例 13.20 src/libs/actioneditor/actiontableeditor.cpp

```
[ . . . . ]

void ActionTableEditor::setupSortFilter() {
    m_sortFilterProxy = new QSortFilterProxyModel(this);
    m_sortFilterProxy->setSourceModel(m_model);                    1
    m_ui->m_tableView->setModel(m_sortFilterProxy);                2
    m_sortFilterProxy->setFilterKeyColumn(-1);                     3

}
void ActionTableEditor::on_m_filterField_textChanged               4
    (const QString& newText) {
    m_sortFilterProxy->setFilterFixedString(newText);              5
}
```

1 设置 SortFilterProxy 的源模型为 ActionTableModel。

2 表格视图的模型设置为代理模型而非 ActionTableModel。

3 所有数据列启用过滤。

4 自动连接的槽。

5 修改过滤字符串。

filterField_textChanged 是一个自动连接的槽，每当 QLineEdit 对象 filterField 的 textChanged 信号发射时会被调用。

13.4　树模型

在 QTreeView(具有父-子关系的项)中显示树形数据，有以下几个选项：

1. QAbstractItemModel 是一个可用于 QTreeView，QListView 或 QTableView 的通用抽象模型。
2. QStandardItemModel 是在示例 13.2 中使用过的可存储 QStandardItem 数据项的具体类，使得填充一个树形节点具体模型十分便利。
3. QTreeWidgetItem 不是一个模型类，但是它能用在从 QTreeView 派生而来的 QTreeWidget 部件中构造树形结构。

> **WidgetItem 类**
>
> QTreeWidget 和 QTreeWidgetItem 类主要用在 Qt 设计师中填充视图项。其 API 与 Qt 3 中的做法相似，它们仅被推荐应用于简单类型的数据以及单个视图中。这是因为使用基于部件/项的类，将模型与视图进行分离，或者当数据变化时多个视图自动更新都是不可能的。

QStandardItemModel 和 QTreeWidgetItem 类都是可被实例化或者进行扩展的树形节点。单个对象以树的形式连接起来，与 QObject 对象的子对象(参见 8.2 节)或者 QDomNode 节点(参见 15.3 节)相似。事实上，这些类都是组合模式的实现。

图 13.12 是下一个示例的截图，它给出当前内存中组成该应用程序用户界面的对象。

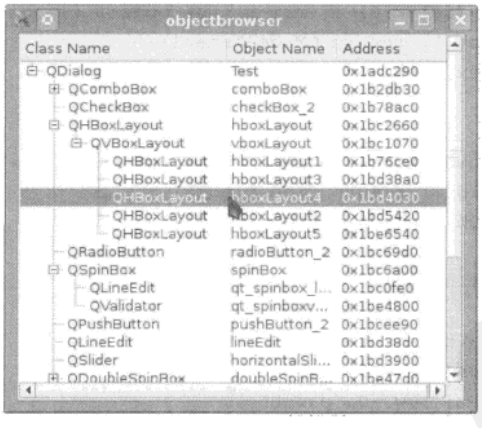

图 13.12　ObjectBrowser 树

这个应用程序的类定义在示例 13.21 中。ObjectBrowserModel 是一个扩展自 QAbstractItemModel 的具体树模型。它实现了一个只读对象浏览树所需的所有方法。

示例 13.21　src/modelview/objectbrowser/ObjectBrowserModel.h

```
[ . . . . ]
#include <QAbstractItemModel>
class ObjectBrowserModel :public QAbstractItemModel {
 public:
```

```
    explicit ObjectBrowserModel (QObject* rootObject);
    int columnCount ( const QModelIndex& parent = QModelIndex() ) const;
    int rowCount ( const QModelIndex& parent = QModelIndex() ) const;
    QVariant data ( const QModelIndex& index,
                    int role = Qt::DisplayRole ) const;
    QVariant headerData(int section, Qt::Orientation,
                        int role = Qt::DisplayRole) const;
    QModelIndex index ( int row, int column,
                        const QModelIndex& parent = QModelIndex()) const;
    QModelIndex parent ( const QModelIndex& index ) const;

 protected:
    QList<QObject*> children( QObject* parent ) const;
    QString label( const QObject* widget, int column ) const;
    QObject* qObject( const QModelIndex& ) const;
 private:
    QObject *rootItem;
};
[ . . . . ]
```

为了使树视图能够在 QAbstractItemModel 的父-子层次节点中上下移动，需要实现两个表格视图不需要的方法：index()和 parent()。示例 13.22 给出了它们的实现。

示例 13.22 src/modelview/objectbrowser/ObjectBrowserModel.cpp

```
[ . . . . ]

QModelIndex ObjectBrowserModel::
index(int row, int col, const QModelIndex& parent) const {
    if ((row < 0) || (col < 0) || row >= rowCount() ||
    col >= columnCount()) return QModelIndex();
    return createIndex( row, col, qObject(parent) );                    1
}

QModelIndex ObjectBrowserModel::parent( const QModelIndex& index ) const {
    if (!index.isValid()) return QModelIndex();
    QObject* obj = qObject(index)->parent();                            2
    if ( obj == 0 )
        return QModelIndex();

    QObject* parent = obj->parent();
    int row = children( parent ).indexOf( obj );
    return createIndex( row, 0, parent );
}

QObject* ObjectBrowserModel::
qObject(const QModelIndex& index) const {                               3
    if ( index.isValid() ) {
        QObject* parent = reinterpret_cast<QObject*>( index.internalPointer() );
        return children(parent)[index.row()];                           4
    }
    return 0;                                                           5
}
```

1 在索引中存储一个 internalPointer。
2 qObject()返回索引的直接行子项，但需要的是该索引的父项 QObject 指针，它存储在 index.internalPointer()。
3 索引的 internalPointer，父对象 QObject 指针。父项的直接子项，即 row()。
4 当前索引。
5 根索引。

index()也可以被想像成是“子索引”(childIndex)，因为它用于查找树中子项所对应的 QModelIndex，而 parent()用于在另一个方向上计算一步。在只应用于表格视图的模型中实现这些方法并不重要，但是如果希望在树形视图中查看这个模型的话，index()和 parent()就是必须的了。

13.4.1　Trolltech 模型测试工具

在模型中实现的方法会被一些尚未测试过的视图调用，或许用于尚未尝试过的情形。因为数据值常常是由用户驱动的，使用“模型测试”工具来测试模型或许很有帮助，它能快速地寻找通常的实现错误并且指出如何修复它们。

图 13.13 展示了在 QtCreator 调试器中使用 ModelTest 工具运行示例 13.21 中的 ObjectBrowser 时会发生的情况。错误断言和异常终止可能在栈踪迹中隐藏很深，所以需要一个调试器来查看全部栈踪迹。通常在终止行之前会有代码注释指明在该处正在执行什么样的测试。如示例 13.23 所示，它所包含的文件 Readme.txt 里面有使用 ModelTest 的简要指令说明。

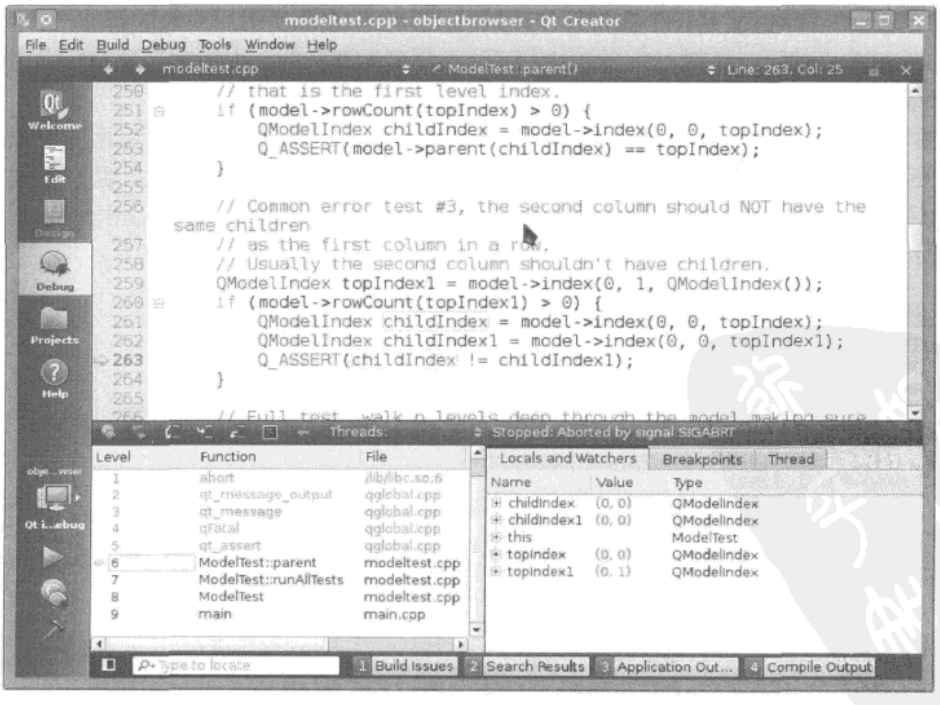

图 13.13　调试器中的 ModelTest

示例 13.23 src/libs/modeltest/readme.txt

应依照下面的指示来使用模型测试。

1) 像下面这样使用 include()命令将 pri 文件添加到工程文件.pro 中:

```
include(../path/to/dir/modeltest.pri)
```

2) 然后在源文件中包含 modeltest.h，并用模型初始化 ModelTest，这样测试才能在模型生存期内始终存在。例如:

```
#include <modeltest.h>

QDirModel *model = new QDirModel(this);
new ModelTest(model, this);
```

3) 这样就可以了。当测试找到问题时它会发出断言。

Modeltest.cpp 包含了一些关于如何修复该测试所找到问题的提示。

13.5 智能指针

尽管 C++不支持垃圾回收，但 C++对象的自动内存管理还是可以通过好几种方式来实现，主要是通过智能指针使用以及引用计数。Qt 提供了许多不同的智能指针类型，以适用于不同的用途。

一个重写了指针解引用操作 operator*()和 operator->()的类被称为智能指针。这使得类实例的行为就像它是一个内置指针一样。这样的类几乎总是模板类，因此定义时必须在模板参数中提供引用类型。最常见的能找到这些重写操作算子的地方是在迭代器以及智能指针中。使它们变得智能的通常是在构造、析构以及赋值中的自定义行为。

QScopedPointer 是一个在指针作用域结束后自动删除所引用对象的智能指针。它类似于 std::auto_ptr。复制 QScopedPointer 是毫无意义的，因为它会导致所引用的对象重复删除。指针的作用域明确地表明了所引用对象的生存期和所属。

类似于 QScopedPointer，QSharedPointer 是一个自动删除它所引用的对象的智能指针，但是它允许被复制，而且 QSharedPointer 会保持一个引用计数。共享的堆对象只有在最后一个指向它的智能指针销毁时才会被删除。示例 13.24 的 DataObjectTableModel 中使用了 QSharedPointer。

示例 13.24 src/libs/dataobjects/dataobjecttablemodel.h

```
[ . . . . ]

class DOBJS_EXPORT DataObjectTableModel : public QAbstractTableModel {
    Q_OBJECT
  public:
    explicit DataObjectTableModel(DataObject* headerModel = 0,
                                  QObject* parent=0);
    virtual bool insertRecord(QSharedPointer<DataObject> newRecord,
                              int position = -1,
                              const QModelIndex& = QModelIndex());
    QStringList toStringList() const;

    QString toString() const;
```

```
    virtual int fieldIndex(const QString& fieldName) const;
    virtual ~DataObjectTableModel();
[ . . . . ]

  public slots:
    void clear();
    void rowChanged(const QString& fileName);

  protected:
    QList<QSharedPointer<DataObject> > m_Data;
    QList<bool> m_isEditable;
    QStringList m_Headers;
    DataObject* m_Original;
    void extractHeaders(DataObject* hmodel);
  public:
    DataObjectTableModel& operator<<(QSharedPointer<DataObject> newObj) {
        insertRecord(newObj);
        return *this;
    }
};
```

可以通过智能指针间接地调用 DataObject 的 property()和 setProperty()方法，就像普通指针那样，使用 operator->的情形见示例 13.25。

示例 13.25　src/libs/dataobjects/dataobjecttablemodel.cpp

```
[ . . . . ]

QVariant DataObjectTableModel::
data(const QModelIndex& index, int role) const {
    if (!index.isValid())
        return QVariant();
    int row(index.row()), col(index.column());
    if (row >= rowCount()) return QVariant();
    QSharedPointer<DataObject> lineItem(m_Data.at(row));
    if (lineItem.isNull()) {
        qDebug() << "lineitem=0:" << index;
        return QVariant();
    }
    if (role == Qt::UserRole || role == Qt::ToolTipRole)
        return lineItem->objectName();
    else if (role == DisplayRole || role == EditRole) {
        return lineItem->property(m_Headers.at(col));
    } else
        return QVariant();
}


bool DataObjectTableModel::
setData(const QModelIndex& index, const QVariant& value, int role) {
    if (index.isValid() && role == EditRole) {
        int row(index.row()), col(index.column());
        QSharedPointer<DataObject> lineItem(m_Data.at(row));
```

```
            lineItem->setProperty(m_Headers.at(col), value);
            emit dataChanged(index, index);
            return true;
        }
        return false;
    }
```

如果每一行是由一个可在多个 DataObjectTableModel 中存在的 DataObject 对象所代表，通过使用带引用计数的指针，该表格在其拥有指向该对象的最后一个智能指针时可以清除这些 DataObject 对象。

13.6 练习：模型和视图

1. 写一个文件系统浏览器，具有地址栏、上一级按钮以及可选的可在其他浏览器中找到的其他常规按钮和特性。使用 QFileSystemModel 以及至少两个用 QSplitter 分隔开的视图类。一个视图要使用 QTableView。如图 13.14 所示，可以选择一个 Windows Explorer 样式树放于表格侧边，或者带 QColumnView 和表格的 Mac OS X 风格的浏览器。要使得能通过树/列视图、工具栏或者上一级按钮来选择一个目录，这样任意一种方式都能更新表格内容以反映新选择的目录。

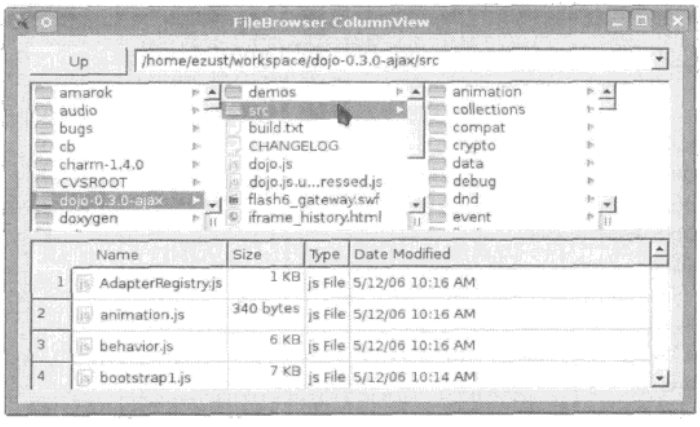

图 13.14 ColumnView 文件浏览器

2. 扩展 QAbstractTableModel 并且定义一个 PlayListModel，它应当表示一个 MetaDataValue(或者 MetaDataObject)对象列表。可以选择基于音轨或视频来设计。生成并显示测试数据，可以使用真实多媒体文件和 MetaDataLoader 或者使用自己的测试数据/工厂方法。实现加载/保存播放列表到磁盘的动作。
3. 重新温一下 11.6 节中的程序，实现一个显示好友列表或符号之间双向关系的界面，该界面应当显示两个 QListView 或者 QListWidget，就像图 13.15 所示那样。

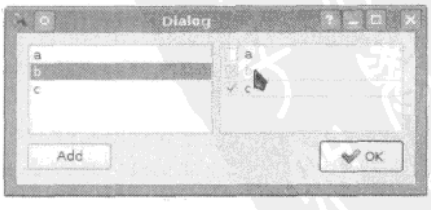

图 13.15 好友列表

- 两个列表都要显示可用符合集。
- 单击 Add 按钮，在两个列表中都添加一个新符号。
- 当左边的符号被选中时，右边列表中的好友显示为选中，陌生人置为未选中。
- 选中/取消右边的复选框，则添加/删除两个人之间的关系。
- 如果联系人与左侧选中的是同一个，则不允许用户取消选择它。始终将其显示为选中。

4. 重写快捷键编辑器示例来使用就地编辑。需要为表格视图写一个委托类，它提供一个自定义编辑器以供用户用来修改快捷键。

13.7　复习题

1. 什么是模型类？什么是视图类？在它们之间应保持什么样的关系？
2. Qt 提供了什么样的工具来与模型和视图共同工作？
3. 什么是 MVC？
4. 什么是控制器代码？哪些 Qt 类是控制器类？
5. 什么是委托？它们在哪里能发现？
6. 与委托相关的东西，角色有什么作用？
7. 如何判断一个 `QListView` 中的项是否被选中？
8. 如果想遍历一个 `QAbstractItemModel` 中的项，应该使用哪个类？
9. 有两套分层结构的类可用于存储和显示树形数据：`*Widget/Item` 和`*ItemModel/View`。使用其中一个而不是另一个的缘由是什么？
10. 为什么愿意使用 `QStandardItemModel` 而不是 `QAbstractItemModel`？反过来呢？