第 9 章　窗件和设计师

这一章将简要介绍 Qt 库中的图形用户界面(GUI)的构成模块，即窗件(widget)，并会包含一些如何使用窗件的简单示例。将用 QtCreator 和设计师(Designer)探索各个窗件、窗件的特性以及它们与用户代码的结合方式。

窗件就是一个 QWidget 派生类的对象，它能够在屏幕上显示。QWidget 的基本结构形式如图 9.1 所示。

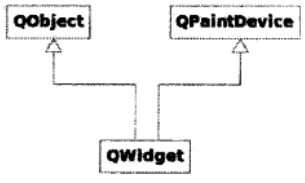


图 9.1　QWidget 的继承层次

QWidget 是一个采用了多重继承(参见 22.3 节)的类。首先，QWidget 是一个 QObject，因此它可以有父对象、信号、槽以及可受管理的子对象。同时，QWidget 也是一个 QPaintDevice，这个类是所有可在屏幕上进行“绘制”的对象的基类。

QWidget 与其子对象交互的方式非常有趣。没有父对象的窗件称为窗口(window)。如果一个窗件是另外一个窗件的父对象，那么子窗件的边界将完全置于父窗件的边界内部[①]。所有被包含的窗件将按照布局规则(参见 9.6 节)进行显示。

QWidget 可以接收来自窗口系统内各种实体的信号(例如，鼠标、键盘、计数器、其他进程，等等)，通过对这些信息的响应，QWidget 可以处理各种事件。QWidget 能够将其自身的矩形图像绘制在屏幕上。同样也可以从屏幕上移除自身而并不妨碍当前显示在屏幕上的所有其他事物。

典型的桌面 GUI 应用程序可以包含许多(即使达到数百个也不稀奇)不同的 QWidget 派生类对象，它们会根据父-子关系进行配置，并根据具体应用程序的布局进行排列。

QWidget 被认为是所有 GUI 类中最为简单的，这是因为它看上去就像一个空盒子。但是，该类本身非常复杂，它包含了数百个函数。当复用 QWidget 及其子类时，其实就已经站在了巨人的肩膀上，这是因为 QWidget 是基于几个 Qt 代码层创建的，而取决于你所处的平台(Linux 中的 X11，MacOS 中的 Cocoa 以及 Windows 上的 Win32)，这些代码又在不同的本地系统代码层次之上。

9.1　窗件的分类

Qt 窗件可以按照几种方式进行划分，如此就可以更轻松地查找那些想要使用的类。而较为复杂的 QWidget 可能会被划入不止一个的子类中。这一节将简要介绍一些在开始学习 GUI 编程时可能需要用到的类。

QWidget 可以分为四类基本窗件。按钮窗件(button widget)的 Windows 风格的显示效果如图 9.2 所示。输入窗件(input widget)的 Plastique 风格的显示效果如图 9.3 所示。

---

① 这一规则存在一些例外。例如，悬浮的 QDockWidget 或者 QDialog 就可以位于父窗口部件的边界之外但仍会在父窗口部件的“前面”。10.2 节将讨论 QDockWidget。

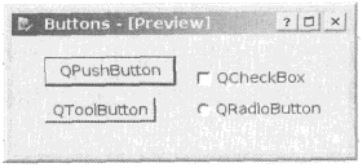


图 9.2 按钮窗件

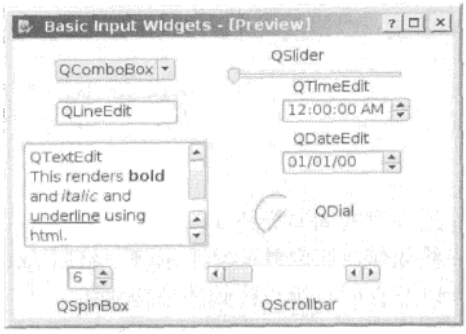


图 9.3 输入窗件

显示窗件(display widget)是不可交互的，如 QLabel，QProgressBar 和 QPixmap。容器窗件(container widget)，如 QMainWindow，QFrame，QToolBar，QTabWidget 和 QStackedWidget，可以包含其他窗件。

- 上述各类窗件可用作构建模块来创建其他更为复杂的窗件，比如 QFileDialog，QInputDialog 和 QErrorMessage。
- 能够显示数据集合的视图，诸如 QListView，QTreeView，QColumnView 和 QTableView，这四类的显示效果如图 9.4 所示。

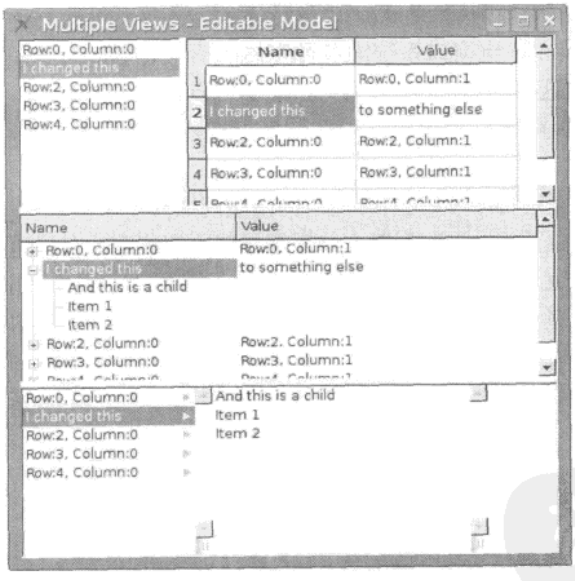


图 9.4 一个模型的四种视图

此外，还有一些 Qt 类没有任何图形化的显示(所以它们都不是窗件)，但是会在 GUI 开发中使用到。这些类主要包括如下几种。

- Qt 数据类型——QPoint，QSize，QColor，QImage 和 QPixmap 是在处理图形对象时经常用到的类型。
- 布局——这些类能够动态地管理窗件的布局。其中有些是常用的特殊布局，包括 QHBoxLayout，QVBoxLayout，QGridLayout，QFormLayout 等。

- 模型——QAbstractItemModel 及其各个派生类，如 QAbstractListModel 和 QAbstractTableModel，外加一些已有的可继承实体类，如 QSqlQueryModel 和 QFileSystemModel，都是 Qt 模型/视图框架中的一部分，该框架内置将一个模型和其他不同视图相连接的机理，以便对一个组件的修改可以自动变换到其他组件上。
- 控制器类——QApplication 和 QAction 两者都是管理 GUI 应用程序控制流的对象。QItemDelegate 用作模型和视图之间的控制器。

如果希望更详细地了解不同风格的窗件，可参考 *The Qt Widget Gallery*[①]，其中包含了各种屏幕截图和源代码，并以不同的风格渲染窗件。

9.2　设计师简介

设计师(Designer)是用于应用程序构图和编辑的一个图形化程序，它拥有拖放式的接口和大量节省编程时间和编程精力的一系列特性。设计师的设计阶段的输出是一个称为 *classname*.ui 的 XML 文件，其中的 *classname* 通常是由设计阶段的开始部分确定的。

在 QtCreator 中，无论何时打开一个 .ui 文件，设计师都会以“设计模式”的形式包含到 QtCreator 中。只要有相应合适的集成包存在，设计师也可作为嵌入型应用程序而集成到其他集成开发环境(IDE)中去（例如，Eclipse, Microsoft Developer Studio, Xcode 等）。

classname.ui 文件可以描述一个包含子对象、布局和各个内部连接的设计师窗件。如 9.7 节所述，XML 文件会被翻译成 C++头文件，但本节仅介绍用设计师来包含 GUI 的用法。

可以从窗件工具箱（Widget Box）将窗件拖放到中央的窗件编辑器（Widget Editor）处。拖放了一些窗件后，就可以在对象检查器（Object Inspector）中选择其中的任意窗件，并在属性编辑器（Property Editor）中查看它的各个属性，如图 9.5 所示。

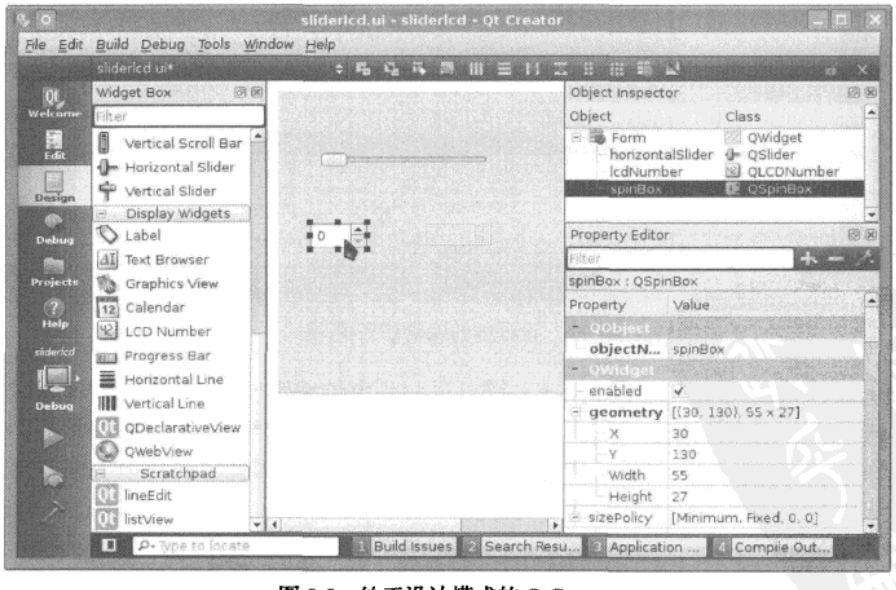


图 9.5　处于设计模式的 QtCreator

① 参见http://doc.trolltech.com/4.6/gallery.html。

在属性编辑器中可以修改任意的属性，而这些修改将会使 `classname.ui` 文件内发生相应的变化。如果编辑的属性会改变窗件的外观，那么在窗件编辑器中可以即刻看到所做的修改。

绝大多数的对象可以从窗件编辑器或者对象检查器中进行重命名，或者是通过单击鼠标左键来选中该窗件然后再按下 F2 键进行重命名，或者是通过在该窗件上双击来重命名。如果愿意，也可以设置属性编辑器上的 *objectName*。然后，需要重点注意的是，输入的各个窗件名是大小写敏感的，因为输入的那些名称将还会成为由用户界面编译器(User Interface Compiler，UIC)生成的类的数据成员名。

通过切换到编辑信号/槽(Edit Signals/Slots，快捷键 F4)模式，可以在各窗件之间拖放各个连接，如图 9.6 所示。在进入编辑信号/槽模式时，窗件工具箱会无法使用，此时就可以从一个带信号的窗件上拖动连接到另一个带槽的窗件上(在窗件编辑器上)。

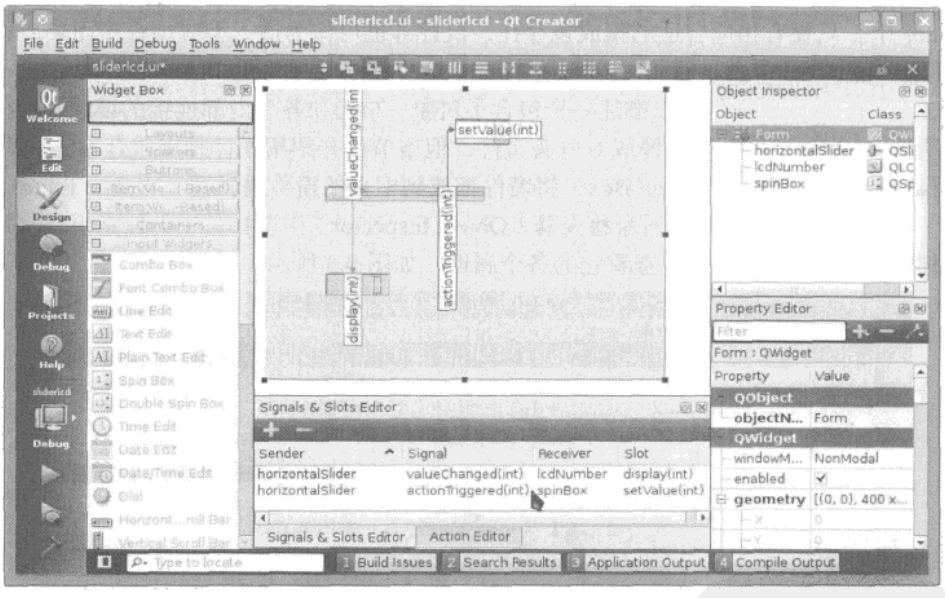


图 9.6 设计师的编辑信号/槽模式

在新的连接构造完成后，通过按 F3 键可以回到窗件编辑模式。通常可以让信号/槽编辑器变成可停靠的方式来观察或者编辑已有的各个连接(如图 9.6 靠近底部的位置所示)，而无须考虑窗件编辑器是窗件编辑模式、连接模式还是伙伴(buddy)编辑模式。利用预览窗件(Preview Widget，快捷键为 Ctrl + Alt + R)模式，就可以对刚刚构建的各个连接进行动态行为预览，如图 9.8 所示。

尽管到现在为止还没有编写任何代码，但是可以立即通过设计师对新创建的 ui 文件的动态行为进行预览。9.7 节描述了如何将 ui 文件和自己(处理数据)的类集成。

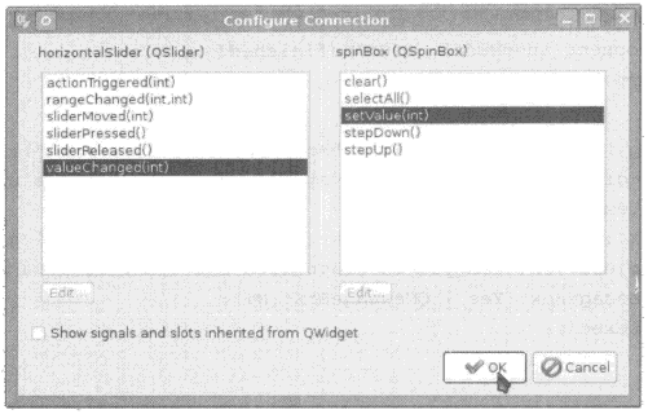


图 9.7　配置连接对话框

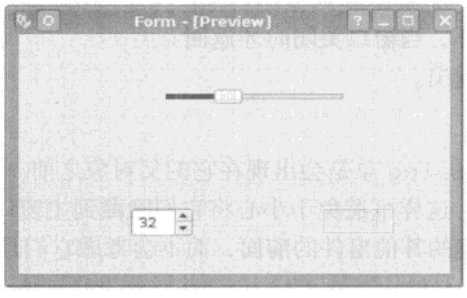


图 9.8　窗件预览

9.3　对话框

QDialog 是 Qt 所有对话框的基类。对话框窗口通常用于和用户进行简单交互。对话框窗口可以是模态(modal)对话框也可以是非模态(nonmodal)对话框。当程序调用静态的便利函数“QMessageBox::”或者“QFileDialog::”时，弹出的对话框就是模态对话框。当模态对话框显示在屏幕上时，它会冻结同一应用程序中的其他所有可见窗口的输入功能。用户解除模态对话框后，与应用程序的常规交互才可以继续下去。QDialog::exec()是将模态对话框放到屏幕上的另一种方式。当用户完成了所需的响应后，对话框就可以返回数据(例如，字符串或者数值)，也可以返回对话框代码(QDialog::Accepted 或者 QDialog::Rejected)。

可以像 QWidget 一样通过 show()显示一个 QDialog。在此情况下，对话框是非模态的，用户也就可以与应用程序的其他窗口继续交互。示例 9.1 给出了模态对话框和用 show()弹出的非模态对话框的区别。

示例 9.1　src/widgets/dialogs/modal/main.cpp

```
[ . . . . ]
#include <QtGui>
int main (int argc, char* argv[]) {
    QApplication app(argc, argv);
    QProgressDialog nonModal;
    nonModal.setWindowTitle("Non Modal Parent Dialog");
```

```
    nonModal.show();                                                    1
    nonModal.connect(&nonModal, SIGNAL(finished()),
            &app, SLOT(quit()));                                        2
[ . . . . ]
    QFileDialog fileDialog(&nonModal, "Modal File Child Dialog");
    // 2 modal dialogs. exec() takes over all user interactions until closed.
    fileDialog.exec();                                                  3
    QMessageBox::question(0, QObject::tr("Modal parentless Dialog"),
            QObject::tr("can you interact with the other dialogs now?"),
            QMessageBox::Yes | QMessageBox::No);
    return app.exec();                                                  4
}
[ . . . . ]
```

1 立即返回。

2 结束的情形。

3 类似进入了事件循环，当窗口关闭时才返回。

4 当非模态关闭时才退出。

父对象和子对象

一个带有父窗件的 QDialog 总是会出现在它的父对象之前。单击父对象同时会显示该对话框。对于非模态对话框，这样可避免不小心将它们隐藏到主窗口的后面。模态对话框通常会出现在由该应用程序创建的其他窗件的前面，而不会考虑它们是何种父-子关系。

输入对话框

已经预定义了一些可复用的输入对话框。如图 9.9 所示，在目录$QTDIR/examples/dialogs/standarddialogs 中，给出了其中一些最为常用的输入对话框。

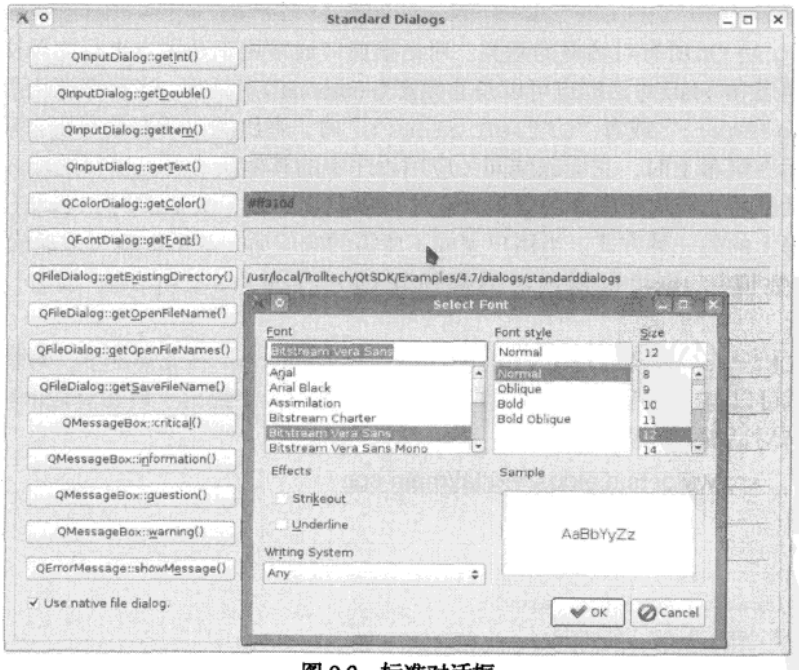


图 9.9 标准对话框

9.4　窗体的布局

有时，可能需要用一些自定义的输入域来创建自己的对话框。图 9.10 中给出了位于 QVBoxLayout 中的 QformLayout，它可从用户那里获得 Qstring，QDate 和 QColor 值。可以用一些 QLayout 型对象将窗件组织到网格中，再嵌套到行、列或者窗体中去。

无论是通过手工方式还是通过设计师，都可以通过 QFormLayout 类来简化窗体的创建过程。它会创建两列：一列用于大小固定的标签，另一列用于大小可变的输入窗件。下面给出的是 InputForm 的类定义。

示例 9.2　src/layouts/form/inputform.h

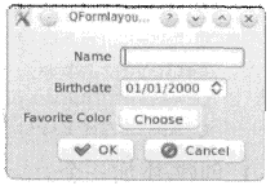


图 9.10　QFormLayout 示例

```
#ifndef INPUTFORM_H
#define INPUTFORM_H

#include <QDialog>
class QLineEdit;
class QDateEdit;
class QPushButton;
class QDialogButtonBox;

class InputForm : public QDialog {
    Q_OBJECT
public:
    explicit InputForm(QWidget* parent = 0);
    void updateUi();
protected slots:
    void accept();
    void chooseColor();
private:
    QColor m_color;
    QLineEdit* m_name;
    QDateEdit* m_birthday;
    QPushButton* m_colorButton;
    QDialogButtonBox* m_buttons;
};

#endif // INPUTFORM_H
```

示例 9.3 给出了 InputForm 构造函数中的几行代码。对 addRow() 的每个调用都会在该行代码所对应的布局中添加一个输入窗件以及它所对应的 QLabel。没有必要明确地构建每个 QLabel。

示例 9.3　src/layouts/form/inputform.cpp

```
[ . . . . ]

    m_name = new QLineEdit;
    m_birthday = new QDateEdit;
    m_birthday->setDisplayFormat("dd/MM/yyyy");
    m_colorButton = new QPushButton(tr("Choose"));
    m_colorButton->setAutoFillBackground(true);
```

```
m_buttons = new QDialogButtonBox(QDialogButtonBox::Ok |
                QDialogButtonBox::Cancel);

QVBoxLayout* vbox = new QVBoxLayout;
QFormLayout* layout = new QFormLayout;

layout->addRow(tr("Name"), m_name);                              1
layout->addRow(tr("Birthdate"), m_birthday);
layout->addRow(tr("Favorite Color"), m_colorButton);

vbox->addLayout(layout);                                         2
vbox->addWidget(m_buttons);

Q_ASSERT(vbox->parent() == 0);
Q_ASSERT(m_birthday->parent() == 0);
setLayout(vbox);                                                 3
Q_ASSERT(vbox->parent() == this);
Q_ASSERT(m_birthday->parent() == this);
```

1 创建/添加一个 QLabel 并在这一行中创建/添加一个输入窗件。

2 用来说明如何在一个布局中嵌套另一个布局。

3 对之前布局过的各窗件重新设置父对象。

可以用 QDialogButtonBox 来显示一些面向用户的标准按钮，这样可以确保它们每次都能够以同样的顺序显示出来。这样做，还有可能让这些对话框可以在不同的平台上依据风格和屏幕尺寸而显示在不同的地方。

没有必要设置窗体中不同窗件的父对象，它们都会添加到以 vbox 为根而构成的部件树的不同布局中。对 setLayout(vbox) 的调用会把 vbox 的父对象设置为宿主的 InputForm 对象上。它还会把添加到 vbox 不同子布局中的 QWidget 的父对象重定义到 vbox 的父对象(同样是宿主的 InputForm)。上面的和下面的 Q_ASSERT 声明都证实了这一点。

9.5 图标，图像和资源

使用图形化的图片可以给应用程序添加可视化效果，这一节将讲解如何在工程中构建一个含有图形化图片的应用程序或者库。

Qt 允许工程使用二进制资源，比如图像、声音、图标、某些外来字体的文字，等等。这些资源通常存储在独立的二进制文件中。在实际工程中集成二进制文件的好处在于它们可以使用不依赖于本地文件系统的路径寻址，并且可以随可执行文件进行自动部署。

Qt 提供至少两种方式来获得标准的图标。一种方式来自桌面样式的 QStyle::standardIcon()，另一种则来自插件型图标主题：QIcon::fromTheme()。但是，你或许还希望使用一些来自其他来源的额外图标。

接下来的一个例子给出了如何创建和复用一个包含图像的库——每一个都代表了一副扑克牌中的一张牌。

第一步是在一个资源集合文件中列出希望使用的二进制文件，该资源文件是一个以 .qrc 为后缀的 XML 文件。libcard2d 中使用的资源文件的片段如下。

```
<!DOCTYPE RCC>
<RCC version="1.0"><qresource>
<file alias="images/qh.png">images/qh.png</file>
<file alias="images/qd.png">images/qd.png</file>
<file alias="images/jc.png">images/jc.png</file>
<file alias="images/js.png">images/js.png</file>
[...]
</qresource>
</RCC>
```

提示

诺基亚基于 Qt 的开源集成开发环境 QtCreator，是一个 qrc 文件编辑器，如图 9.11 所示。可以创建一些新的(或者编辑已存在的)资源文件，而且通过 GUI 窗体和文件选择器来添加和修改资源，可以确保它包含那些希望的资源。假设在.pro 文件中添加了.qrc 文件，并且正好在使用 QtCreator，那么对任何可被设置资源的属性，都可以通过属性编辑器的省略号按钮来访问这些资源。

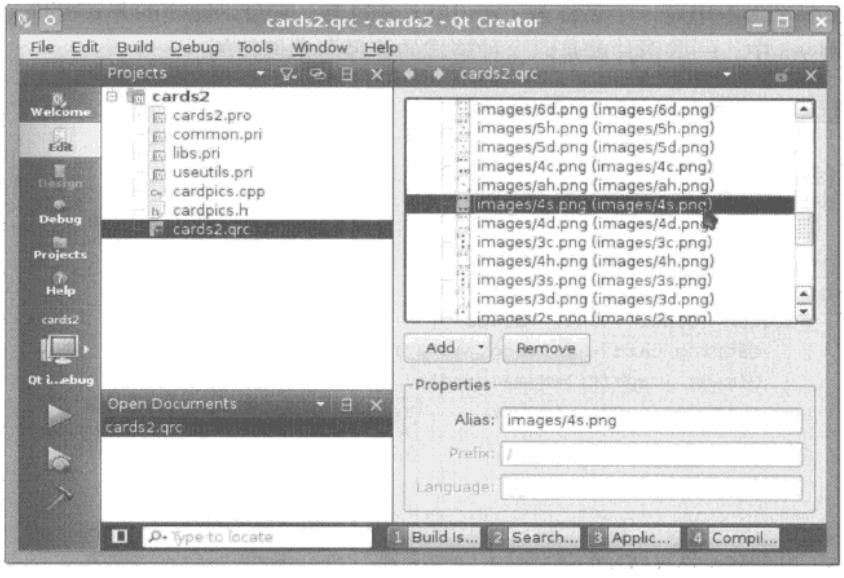


图 9.11　QtCreator 的属性编辑器

在每个含有 qresource 文件名称的工程文件中添加 RESOURCES 行，如示例 9.4 所示。

示例 9.4　src/libs/cards2/cards2.pro

```
include (../libs.pri)

TEMPLATE = lib
QT += gui

# For locating the files.
RESOURCES = cards2.qrc
SOURCES += cardpics.cpp \
    card.cpp
```

```
HEADERS += cardpics.h \
    card.h \
    cards_export.h

win32 {
        DEFINES += CARDS_DLL
}
```

在构建该工程时，rcc 会额外生成一个名称为 cards2_qrc.cpp 的文件，它含有 C++中定义的字节数组。该文件而不是原始的 card 图像文件会被编译并链接到工程的二进制文件中(可执行文件或者库文件)。DESTDIR 这一行指定了用于 libcards2 的共享对象文件将会与已构建过的其他库一起位于$CPPLIBS 中。

把所需的二进制数据文件作为资源附加到工程中会让工程更为健壮。源代码无须为资源文件使用一些不可移植的路径名。要引用一个存储成资源的文件，可以使用在.rcc 文件中指定的别名并在前面带一个前缀“:/”。于是，每个资源都好像位于一个私有虚文件系统中，其根为“:/”。然而，得到这些好处的确还是需要付出一些代价的。可执行文件会更大，程序也会需要更多的内存。

在 libcards2 中有一个名称为 CardPics 的类，它会以便捷的方式提供 QImage 型 card。示例 9.5 中，一些图片就是以这种格式的路径名来创建的。

示例 9.5 src/libs/cards2/cardpics.cpp

```
[ . . . . ]
const QString CardPics::values="23456789tjqka";
const QString CardPics::suits="cdhs";

CardPics::CardPics(QObject* parent) : QObject(parent) {
    foreach (QChar suit, suits) {
        foreach (QChar value, values) {
            QString card = QString("%1%2").arg(value).arg(suit);
            QImage image(fileName(card));
            m_images[card]= image;
        }
    }
}

QString CardPics::fileName(QString card) {
    return QString(":/images/%1.png").arg(card);                    1
}
QImage CardPics::get(QString card) const {
   return m_images.value(card.toLower(), QImage());
}
[ . . . . ]
```

1 来自资源。

有三个 Qt 类可以简化处理图片。

- QImage——用于离屏(off-screen)操作，输入输出操作，并可直接访问像素。
- QPixmap——用于在屏幕上进行绘制并优化。仅用在主线程中。
- QIcon——用于视频内存的缓冲且经常用到，但仅用在主线程中。
- QPicture——存储绘制的操作而不是实际的位图图片。

在 libcards2 中，会先从资源中创建每个 QImage 并添加到 CardPics 中，以便可以用 get() 函数进行快速访问。

9.6　窗件的布局

窗件可以像对话框一样弹出并显示在屏幕上，也可以作为一个较大窗口的一部分。无论何时，只要打算在较大的窗件中排放一些较小的窗件，就必然要用到布局。所谓布局，就是一个仅属于某一窗件的对象(即，是其子对象)。布局的唯一任务就是合理地组织其拥有的子窗件所占据的空间。

尽管每个窗件都有一个 setGeometry() 函数，也可以通过此函数来设置其大小和位置，但是在窗口应用程序中很少采用绝对大小和绝对位置，其原因就是这种方式通常会使得设计过于死板。通过使用布局，可以灵活自然地安排所有的可见空间，从而使得窗口能够成比例地改变大小、拖动或者为窗口加入滚动条。

在屏幕上安排窗件位置和顺序的主要过程，就是将屏幕空间合理地划分成几个区域，并使用 QLayout 来管理每个区域。布局可以将它们的窗件排列成如下几种形式。

- 垂直型(QVBoxLayout)
- 水平型(QHBoxLayout)
- 网格型(QGridLayout)
- 窗体型(QFormLayout)
- 栈型，任何时候都只有一个窗件可见(QStackedLayout)

可以使用 addWidget() 函数向 QLayout 添加窗件。当窗件添加到布局中时，它会成为拥有该布局的窗件的子对象。窗件永远不会成为布局的子对象。

布局不是窗件，它们也没有任何可见的表达形式。Qt 提供了一个名称为 QLayout 的抽象基类，还提供了几个特殊的 QLayout 子类：QBoxLayout(可以细分为 QHBoxLayout 和 QVBoxLayout)、QGridLayout 和 QStackedLayout。这些布局类型的每一种都拥有一组合适的函数来控制空间、大小、对齐方法以及对其窗件的访问方式。

为了能够顺利地管理其几何形状，每个 QLayout 对象都必须有一个父对象，这可以是一个 Qwidget，也可以是一个 Qlayout。可以在创建布局时通过向构造函数传递一个指向父窗件或者布局的指针来指定其父对象。当然，也可以先创建一个 QLayout 而不指定 QLayout 的父对象，这种情况下，可以稍后通过调用 QWidget ::addLayout() 来指定其父对象。

布局可以拥有子布局。通过调用 addLayout() 函数，可以将一个布局添加为另外一个布局的子布局。当然，具体的做法取决于所采用的布局类型。如果布局的父对象是一个窗件，那么该窗件将再也无法成为另外一个布局的父对象。

示例 9.6 中定义的 CardTable 类复用了 libcards2，以此来更为轻松地访问扑克牌(参见 9.5 节)的 QImage。构建 CardTable 对象之后，屏幕显示如图 9.12 所示。

示例 9.6　src/layouts/boxes/cardtable.h

```
#ifndef CARDTABLE_H
#define CARDTABLE_H
#include <cardpics.h>
#include <QWidget>
```

```
class CardTable : public QWidget {
  public:
    explicit CardTable(QWidget* parent=0);
  private:
    CardPics m_deck;
};

#endif        //  #ifndef CARDTABLE_H
```

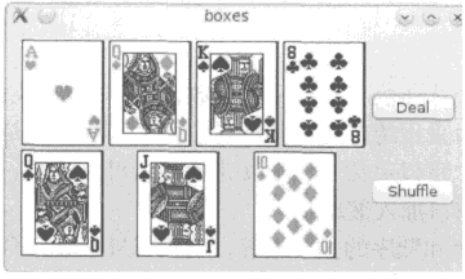


图 9.12 显示几行和几列扑克牌

示例 9.7 中实现的 CardTable 充分地利用了 QLabel 可以存放图像的事实，这个实现展示了一些简单却非常有用的布局技术。

示例 9.7 src/layouts/boxes/cardtable.cpp

```
[ . . . . ]

CardTable::CardTable(QWidget* parent)
: QWidget(parent) {

    QHBoxLayout* row = new QHBoxLayout();                    1
    row->addWidget(new Card("ah"));                          2
    row->addWidget(new Card("qd"));
    row->addWidget(new Card("ks"));
    row->addWidget(new Card("8c"));

    QVBoxLayout* rows = new QVBoxLayout();                   3
    rows->addLayout(row);                                    4

    row = new QHBoxLayout();                                 5
    row->addWidget(new Card("qs"));
    row->addWidget(new Card("js"));
    row->addWidget(new Card("td"));
    rows->addLayout(row);                                    6

    QVBoxLayout* buttons = new QVBoxLayout();                7
    buttons->addWidget(new QPushButton("Deal"));
    buttons->addWidget(new QPushButton("Shuffle"));
    QHBoxLayout* cols = new QHBoxLayout();                   8
    setLayout(cols);                                         9
    cols->addLayout(rows);                                   10
    cols->addLayout(buttons);                                11
}
[ . . . . ]
```

1 第一行。
2 父对象会通过布局得到设置，因此不必指定。
3 垂直地布局各行。
4 在垂直布局中嵌入一行。
5 第二行。
6 再次嵌套。
7 用于各个按钮的一列。
8 把它们全都放在一起。
9 这个窗件的“根布局”。
10 把这两行扑克牌都加为一列。
11 把一列按钮都加为另一列。

示例 9.8 中给出的客户代码足以在屏幕中显示该窗口。

示例 9.8　src/layouts/boxes/boxes.cpp

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

int main(int argc, char* argv[]) {
        QApplication app (argc, argv);
        CardTable ct;
        ct.show();
        return app.exec();
}
```

如果构建并运行这个例子，然后用鼠标改变窗口的大小，可以注意到，按钮的宽度会首先进行伸展来占用那些多出来的空间。但是，在各个纸牌之间和各个按钮之间也有可供伸展的空间。如果将按钮移除，就可以观察到纸牌之间的水平空间将会均匀地增长。

9.6.1　分隔，伸展和支撑

不使用 Qt 设计师时，可以使用 QLayout 类的 API 来直接指定各个窗件之间的分隔(spacer)、伸展(stretch)和支撑(strut)。

- addSpacing(int size)会向布局的末尾添加固定数量的像素。
- addStretch(int stretch = 0)会添加数目不定的像素。此函数由一个最小的数目开始，然后逐渐扩展到使用所有的可用空间。如果在同一个布局中进行多次扩展，可以用此作为一个增长因子。
- addStrut(int size)将给垂直方向施加一个最小的数值(也就是，QVBoxLayout 的宽度或者 QHBoxLayout 的高度)。

回到示例 9.7 中，我们将试图使此布局在改变大小时表现得更加自然。图 9.13 给出了在该应用程序中添加一定的伸展量和分隔量后的结果。

通常情况下，布局会试图平等对待所有的窗件。当打算让一个窗件与另一个窗件不相邻或者相互远离时，可以使用伸展和分隔功能来与常规效果加以区分。示例 9.9 给出了如何使用伸展和分隔的做法。

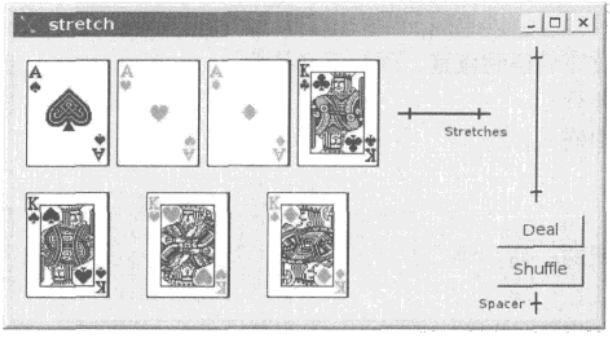


图 9.13 改进的带有伸展量和分隔量的布局

示例 9.9 src/layouts/stretch/cardtable.cpp

```
[ . . . . ]
    row = new QHBoxLayout();
    row->addWidget(new Card("td"));
    row->addWidget(new Card("js"));
    row->addWidget(new Card("kc"));
    rows->addLayout(row);
    rows->addStretch(1);                                    1
    QVBoxLayout* buttons = new QVBoxLayout();
    buttons->addStretch(1);                                 2
    buttons->addWidget(new QPushButton("Deal"));
    buttons->addWidget(new QPushButton("Shuffle"));
    buttons->addSpacing(20);                                3
    QHBoxLayout* cols = new QHBoxLayout();
    setLayout(cols);
    cols->addLayout(rows);
    cols->addLayout(buttons);
    cols->addStretch(0);                                    4
}
[ . . . . ]
```

1 用于各行的可伸展空间。

2 在列中各按钮之前的可扩展空间。

3 按钮之后的固定空间。

4 这会如何影响各按钮的尺寸呢?

如果使用示例 9.9 而不是示例 9.7 中的代码来构建和运行此应用程序,然后再次调整主窗口的大小,就可以注意到,按钮不会再增长了,而是被推到了一个角上。扑克牌之间的水平空间不再增长,但是其垂直空间仍会增加。

9.6.2 大小策略和大小提示

每个 QWidget 都有一些与屏幕尺寸相关的属性。一些窗件可以在运行时使用一个或多个方向上的额外空间。通过设置自定义窗件的 sizePolicy 值和 sizeHint 值,可以控制其默认的大小变化行为。

sizeHint 会为窗件保存推荐的 QSize 值。也就是说,该值是其第一次出现在屏幕上时的给定尺寸。QSize 定义了窗件的宽度和高度大小。还有一些成员函数可以用来完全控

制自定义窗件在运行时的大小，包括 setMinimumSize()，setMaximumSize()，setMinimumHeight()，setMaximumWidth()，setSizeHint()等。

每个 Qwidget 和 QLayout 都有一个水平方向和垂直方向的 QsizePolicy。Minimum，Maximum，Fixed，Preferred，Expanding，MinimumExpanding 和 Ignore 这些大小策略表达了窗件或者布局大小被改变时自身的意愿。默认的大小策略是 Preferred/Preferred，这表示 sizeHint 会让窗件拥有较好的大小。Ignore 表示 minimumHeight 和 minimumWidth 不会对改变尺寸的各种操作进行约束。

以固定尺寸显示的窗件将不会从可改变大小的窗件中获得任何多余的空间。因此，应当使用 Fixed 或 Preferred 型水平和垂直尺寸策略。比较而言，可滚动的窗件、容器型窗件以及文本编辑型窗件应当使用扩展型尺寸策略，以便可以向用户显示出更多的信息，因为这可能也是用户之所以使窗口变大的首要原因。多余的空间可由使用柔性尺寸策略的窗件给定或获得。

在同一布局中使用扩展型策略的窗件，根据所基于的伸展因子的不同，会以不同的比率来给定多出来的空间。

从图 9.14 可以看出，按钮通常不会占用任何方向上的多余空间。在开始的两行上，按钮使用了水平扩展尺寸策略，随后的一行显示了各个拉伸因子是如何影响可扩展型按钮的。第三行中，可以看到按钮会占用垂直方向上的全部空间并会逼迫起初的两行变得尽可能地高。

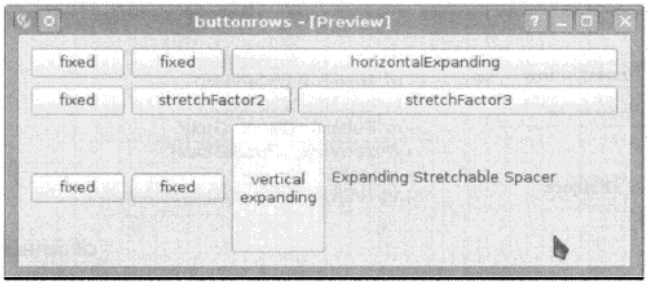


图 9.14　拉伸了的按钮

9.6.3　练习：窗件的布局

1. 在 Qt 设计师中，试着重做排放按钮的布局，使其能够拥有如图 9.14 所示的形式。
2. 在 15 拼图(或 $n^2 - 1$)游戏中含有一个 4 × 4(或 $n \times n$)的网格，其中含有 15 个从 1 到 15 编号的麻将牌和一个空白空间。只有与该空白空间相邻的牌才可以移动。

- 在 QGridLayout 中用 QPushButton 创建一个 15 拼图游戏，如图 9.15 所示。
- 在游戏开始时，各个牌会向玩家以“随机的”顺序显示出来。游戏的目的就是重新排列它们，以便可以让它们显示成升序的形式，使最小数字的牌位于左上角。
- 如果玩家成功完成了这个拼图，则弹出一个 QMessageBox，显示“获胜!”(或是其他一些更聪明的东西)。
- 添加一些按钮：
  - Shuffle（洗牌）——通过执行一个更大的合法的牌数(最大 50)来使牌随机排列。
  - Quit（退出）——终止游戏。

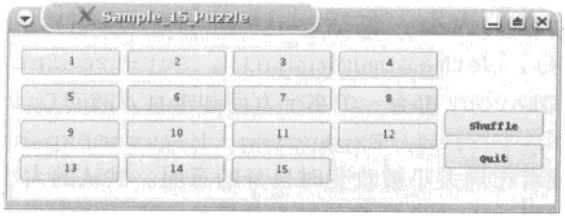


图 9.15 15 拼图的示例

提示

为了避免对每个按钮都使用信号和槽相连接，可以把各按钮群组织到一个 QButtonGroup 中，它有一个 QAbstractButton 的单参数信号。

提示

以图 9.16 所示的类开始，试着将自己的代码适当地整合到其中。“视图”类会处理 GUI，“模型”类则应当没有任何的 QWidget 代码或代码依赖，并且仅用于处理本游戏的逻辑。

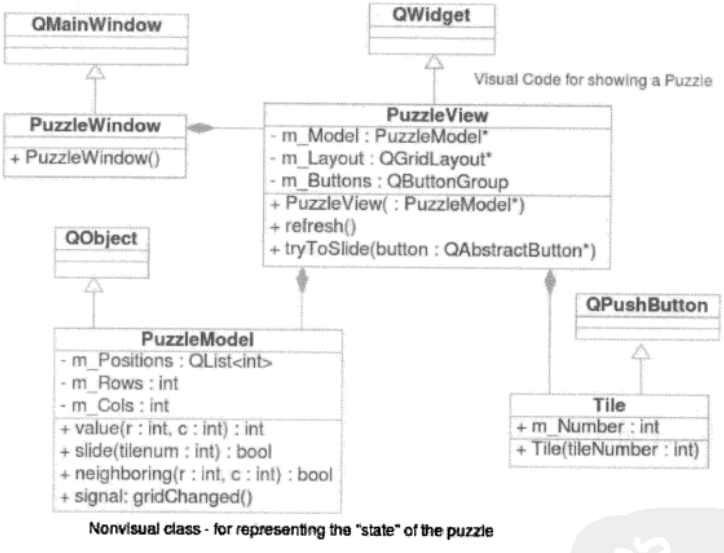


图 9.16 用于拼图的“模型—视图—控制器”设计模式

9.7 设计师和代码的集成

考虑如图 9.17 所示的 ProductForm，它是一个用于 Product 示例的窗体。QFormLayout 是一个便利的窗件，可方便地将 QLabel 和输入窗件组织成两列。

ProductForm 窗件可以接受一个新 Product 对象中来自用户的数据，可用来显示(只读)或者编辑已保存的 Product 实例中的值。根据使用模式，按钮可以有文字和角色。例如，当对用户新增一个 Product 响应时，单击 OK 按钮后应当用窗体中的数据创建一个新 Product 实例。如果单击的是 Cancel 按钮，则应当丢弃这些值。在编辑模式下单击 OK 按钮

时，窗体中的这些值应当去替换那些已保存实例的值；在消息(info)模式下单击 OK 按钮时，窗体应当消失，而 Cancel 按钮也应当隐藏起来。

使用它时，ProductForm 会有一个指向它所编辑的 Product 的引用，如图 9.18 所示。

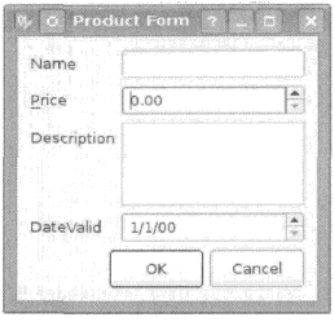


图 9.17　Product 窗体

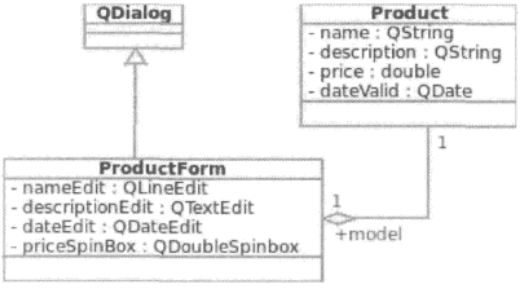


图 9.18　Product 及其窗体

使用设计师设计窗体

跟往常一样，第一步是选择一个基类名称(例如，ProductForm)并为其写出一个头文件(ProductForm.h)和一个实现文件(ProductForm.cpp)。然后，在设计师中用同样的基类名称(ProductForm.ui)创建一个窗体。设置该窗体根对象的 objectName 的名称(ProductForm)，然后启用 uic 为相应的 Ui 类(Ui_ProductForm)生成头文件。

图 9.19 给出了 uic 如何将通过设计师产生的带 .ui 扩展名的特殊 XML 文件当作输入，并生成可以包含进自己代码中的头文件。

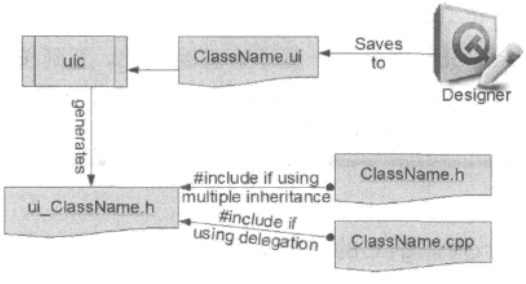


图 9.19　从设计师到代码

示例 9.10 给出了生成的一小段代码片段，它既定义了一个 Ui_ProductForm 类又定义了一个 Ui::ProductForm 派生类。在 C++代码中使用哪一个类由用户决定。正如所看到的，有数据成员会指向设计师窗体中的每一个窗件。每个成员的名称都来自设计师窗体中所设定的对象名，在生成的代码中，赋予了成员名的完全控制权。

示例 9.10　src/designer/productform/ui_ProductForm.h

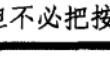
```
[ . . . . ]
class Ui_ProductForm
{
public:
    QVBoxLayout *verticalLayout;
    QFormLayout *formLayout;
```

```
    QLabel *label;
    QDoubleSpinBox *priceSpinbox;
    QLabel *label_2;
    QLabel *label_3;
    QLineEdit *nameLineEdit;
    QLabel *label_4;
    QTextEdit *descriptionEdit;
    QDateEdit *dateEdit;
    QSpacerItem *verticalSpacer;
    QDialogButtonBox *buttonBox;

    void setupUi(QDialog *ProductForm)
[ . . . . ]
};
namespace Ui {
    class ProductForm: public Ui_ProductForm {};
} // namespace Ui
[ . . . . ]
```

什么是 Ui 类?

Ui 类是一个类，仅含有由 uic 工具自动生成的代码。



设计师一开始会把一些来自 QdialogButtonBox 的信号与 QDialog 的 accept()和 reject()槽相连接。从可停靠的信号和槽编辑器中可以看到这一点。这也就意味着，在 OK 按钮按下之前，后台的 Product 对象将一直不会发生变化，直到按下 OK 按钮——可能正如所期望的——reject()的基类行为会关闭对话框。你也可能会希望覆盖基类的 accept()，但不必把按钮的信号与之连接。基类版也会关闭对话框。

集成的方法

以下是将 Ui 类和基于 QWidget 的自定义窗体类相集成的三种方法。

1. 作为指针成员集成。
2. 多重(私有)继承。
3. 作为嵌入式对象集成。

推荐使用通过指针的集成方法(也是默认的)，因为这样可使修改 Ui 文件而不造成带有 ProductForm 头文件的二元破坏(binary breakage)成为可能。示例 9.11 给出了通过指针成员进行集成的示例。ProductForm.h 使用了一个前置类声明而没有直接包含 Ui_ProductForm.h。

示例 9.11 src/designer/delegation/productform.h

```
[ . . . . ]
#include <QDialog>
class Product;
class Ui_ProductForm;
```

```
class QWidget;
class QAbstractButton;
class ProductForm : public QDialog {
        Q_OBJECT
public:
    explicit ProductForm(Product* product = 0, QWidget* parent=0); 1
    void setModel(Product* model);

public slots:
    void accept();
    void commit();
    void update();

private:
    Ui_ProductForm *m_ui;
    Product* m_model;
};
[ . . . . ]
```

1 明确标示以避免在各指针间出现隐式转换！

只有其实现文件，如示例 9.12 所示，需依赖于 `uic` 生成的头文件。例如，这样可使在把其放入库中时变得更为简单。

示例 9.12　src/designer/delegation/productform.cpp

```
#include <QtGui>
#include "productform.h"
#include "ui_ProductForm.h"
#include "product.h"

ProductForm::ProductForm(Product* product, QWidget* parent)
: QDialog(parent), m_ui(new Ui::ProductForm),  m_model(product) {
    m_ui->setupUi(this);                                          1
    update();
}

void ProductForm::setModel(Product* p) {
    m_model =p;
}

void ProductForm::accept() {
    commit();
    QDialog::accept();                                            2
}

void ProductForm::commit() {
    if (m_model == 0) return;
    qDebug() << "commit()";
    m_model->setName(m_ui->nameLineEdit->text());
    QTextDocument* doc = m_ui->descriptionEdit->document();
    m_model->setDescription(doc->toPlainText());
    m_model->setDateAdded(m_ui->dateEdit->date());
    m_model->setPrice(m_ui->priceSpinbox->value());
```

```
}

void ProductForm::update() {
    if (m_model ==0) return;
    qDebug() << "update()";
    m_ui->nameLineEdit->setText(m_model->name());
    m_ui->priceSpinbox->setValue(m_model->price());
    m_ui->dateEdit->setDate(m_model->dateAdded());
    m_ui->descriptionEdit->setText(m_model->description());
}
```

1 让 Ui 对象和来自.ui 文件中设置的适当值的实例一起工作。

2 关闭对话框。

9.7.1 QtCreator 和设计师的集成

QtCreator 用一种简便的方式将集成开发环境和设计师集成一体，在它能理清 Ui 文件和类文件对应关系的地方，会为这些类生成集成代码和各个槽的函数体。通过菜单 QtCreator File->New->Qt->Designer Form Class，可以一次创建所有新的.ui 文件、头文件和相应的类文件。图 9.20 给出了它所支持的三种代码生成方式。

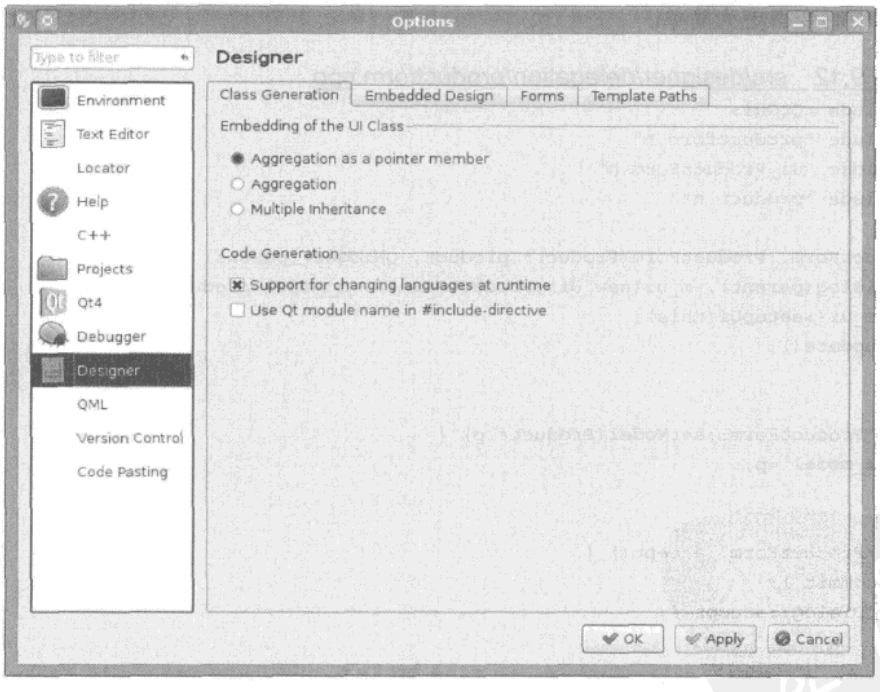


图 9.20 QtCreator 中设计师的代码生成选项

对于需要相互连接的各个子对象(例如，QMainWindow 的各个 GUI 组件)，QtCreator 提供了一种非常方便的特性，如图 9.21 所示。可以在 Widget 编辑器中发射信号的任意组件(例如，Save 按钮)上或者在 Action 编辑器中的任意动作(例如，actionSave)上右键单击，并选择 Go to slot 即可。

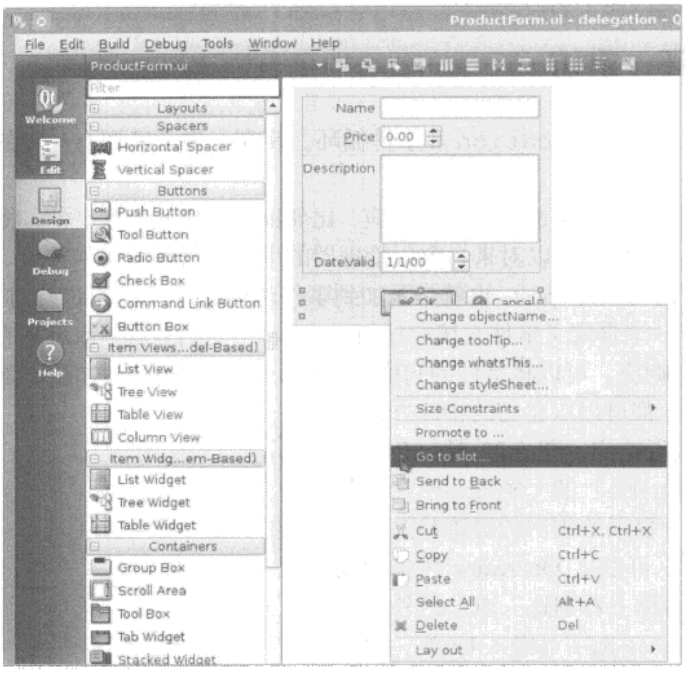


图 9.21　QtCreator 中的 Go to Slot 选项

接下来，从弹出的清单中选择一个信号，例如 `triggered()`，QtCreator 会生成一个私有槽——头文件中的一个原型和实现文件中的一个代码片段——如果还没有一个带有生成名称的槽，例如 `on_actionOpen_triggered()`。通过添加必要的操作细节，即可完成该新槽的定义。所选择的信号在运行时会自动连接到刚定义的槽上。在实现文件(`.cpp`)中，将无法找到连接语句。相反，这些连接是由 `QMetaObject::connectSlotsByName()` 函数完成的。

10.5 节讨论了一个在 QtCreator 中完全使用这些工具编写的应用程序。

9.8　练习：输入窗体

1. 实现 `OrderForm` 类中为创建 `Order` 对象输入对话框窗体所缺少的方法。在 `src/handouts/forms/manual` 中给出了一些开始的东西。对话框看起来会与图 9.22 类似。
   代码放在 `orderform.cpp` 中。用户应当单击 OK 按钮来给 `Order` 发送数据，或者单击 Cancel 按钮来取消该对话框并且不给 `Order` 发送任何数据。
   `totalPrice` 域应当是只读的，且计算值要基于 `quantity` 和 `unitPrice` 的值。
2. 接下来，用设计师创建同样的输入对话框，并使用委托或者多重继承的方式将其与你的代码相集成，如 9.7 节所述。

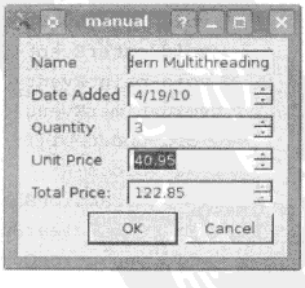


图 9.22　Order 窗体对话框

9.9 事件循环：重访

8.3 节中讲解过 QApplication 和事件循环。现在，你已经拥有一定的 GUI 编程经验，可以更深入地探讨事件了。

QWidget 根据用户产生的动作进行响应，比如鼠标事件、键盘事件，将 QEvent 发送给其他的 QObject。窗件还可以对来自窗口管理器的事件进行响应，如重绘、更改大小或者关闭事件等。事件可进行投递(post，也就是添加到事件队列中)，也可以进行过滤和排列优先级。此外，也有可能将一个自定义事件直接向任何 QObject 进行投递，或者是有选择地对可以投递的其他对象进行响应。可以将事件的句柄委托给一个特殊定义的对象处理程序。

每个 QWidget 都可以以自己的特殊方式对键盘和鼠标事件进行响应。图 9.23 是一个名称为 KeySequenceLabel 的 QWidget 在捕获 QKeySequence 并显示给用户的屏幕截图。

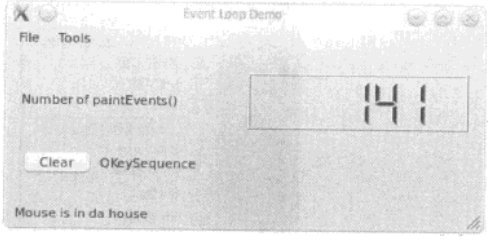


图 9.23 KeySequenceLabel 窗件

示例 9.13 中，可以从 QtCreator 用来构成 GUI 的私有槽的名称上看到这一点。另外，选中 Multiple Inheritance 选项可嵌入 Ui 类，因而 KeySequenceLabel 类会从 QMainWindow 类和设计师生成的 Ui 类中派生出来。

示例 9.13 src/eventloop/keysequencelabel.h

```
[ . . . . ]
#include "ui_keysequencelabel.h"
#include <QList>
#include <QPair>
class QObjectBrowserAction;

class KeySequenceLabel : public QMainWindow, private Ui::KeySequenceLabel {
    Q_OBJECT
 public:
    explicit KeySequenceLabel(QWidget* parent = 0);
 protected:                                                    1
    void changeEvent(QEvent* e);
    void keyPressEvent(QKeyEvent*);
    void leaveEvent(QEvent*);
    void enterEvent(QEvent*);
    void paintEvent(QPaintEvent*);
    void timerEvent(QTimerEvent*);                             2
    void updateUi();
 private slots:
    void on_actionShow_ObjectBrowser_triggered(bool checked);
    void on_m_clearButton_clicked();
    void on_actionQuit_triggered();
 private:
    QObjectBrowserAction* m_browserAction;
```

```
    QList<QPair<int, int> > m_keys;
    int m_paints;
};
[ . . . . ]
```

1 重载 QWidget 事件处理程序。

2 重载 QObject 事件处理程序。

在这个类的实现中包含了一些有意思的东西。

在对话框中有一个 QLabel，用来显示 QKeySequence。QKeySequence 类封装了一个序列，它可存储多达四次的键击，一般用作 QAction 调用的快捷方式。在 KeySequenceLabel 窗件中重载了基类的 keyPressEvent()处理程序。示例 9.14 中，可以发现，键击事件在子窗件看到它们之前已被捕捉并保存到一个列表中。在 updateUi()内，这些事件会先转换成一个 QkeySequence，然后再转换成一个 QString，该字符串通过 setText()送到 QLabel 显示。

示例 9.14　src/eventloop/keysequencelabel.cpp

```
[ . . . . ]

void KeySequenceLabel::keyPressEvent(QKeyEvent* evt) {
    bool doNothing = false;

    if (evt->key() == 0) doNothing = true;
    if (m_keys.size() > 3) doNothing = true;
    if (doNothing) {
        QMainWindow::keyPressEvent(evt);                    1
        return;
    }
    QPair<int, int> pair = QPair<int, int>(evt->modifiers(), evt->key());
    m_keys << pair;
    evt->accept();
    updateUi();
}

void KeySequenceLabel::updateUi() {
    if (!m_keys.isEmpty()) {
        int keys[4] = {0,0,0,0};
        for (int i=0; i<m_keys.size(); ++i) {
            QPair<int, int> pair = m_keys[i];
            keys[i] = pair.first | pair.second;
        }
        QKeySequence seq = QKeySequence(keys[0], keys[1], keys[2], keys[3]);
        m_label->setText(seq.toString());
    }
    else m_label->clear();
}
```

1 QWidget 的基类处理程序对弹出窗口的 ESC 响应。

示例9.15中定义了一些鼠标事件处理程序，用来说明鼠标指针是何时进入或者离开该窗件的。

示例9.15 src/eventloop/keysequencelabel.cpp

```
[ . . . . ]

void KeySequenceLabel::enterEvent(QEvent* evt) {

    statusBar()->showMessage(tr("Mouse is in da house"));
    evt->accept();

}

void KeySequenceLabel::leaveEvent(QEvent* evt) {
    statusBar()->showMessage(tr("Mouse has left the building"));
    evt->accept();
}
```

示例9.16中，构造函数使用QObject内置的定时器并调用了QObject::startTimer()，它会每2 s就产生一个计时事件。可以来这样处理timerEvent()：每2 s使用当前绘制事件被执行的次数来更新一个QLCDNumber。例如，每当LCDNumber的值发生改变时，都会间接造成该窗件的重绘。可以运行一下这个应用程序，试着对窗件进行移动和改变大小，看看它是如何影响绘制事件次数的。

示例9.16 src/eventloop/keysequencelabel.cpp

```
[ . . . . ]

KeySequenceLabel::KeySequenceLabel(QWidget* parent) :
    QMainWindow(parent), m_browserAction(new QObjectBrowserAction(this)) {

    setupUi(this);
    startTimer(2000);                                          1
    m_paints = 0;
}

void KeySequenceLabel::timerEvent(QTimerEvent*) {
    m_lcdNumber->display(m_paints);
}

void KeySequenceLabel::paintEvent(QPaintEvent* evt) {
    ++m_paints;
    QMainWindow::paintEvent(evt);
}
```

1 每2 s发生一次定时器事件。

到目前为止，所有与GUI活动有关的讨论都是关于事件和事件处理程序的。现在要讨论该示例中一个更为有趣的特点。从Tools菜单可以选择Show ObjectBrowser，即可看到如图9.24所示的窗件。

QObjectBrowser是一个开源的代码调试工具，在程序执行过程中，可以对QObject

的属性、信号和槽进行图形化显示[①]。使用它的一种简单方法是将 QObjectBrowserAction 直接添加到 QMainWindow 的菜单上。

在 src/libs 目录中，已含有一个对该工具进行简单修改的版本。

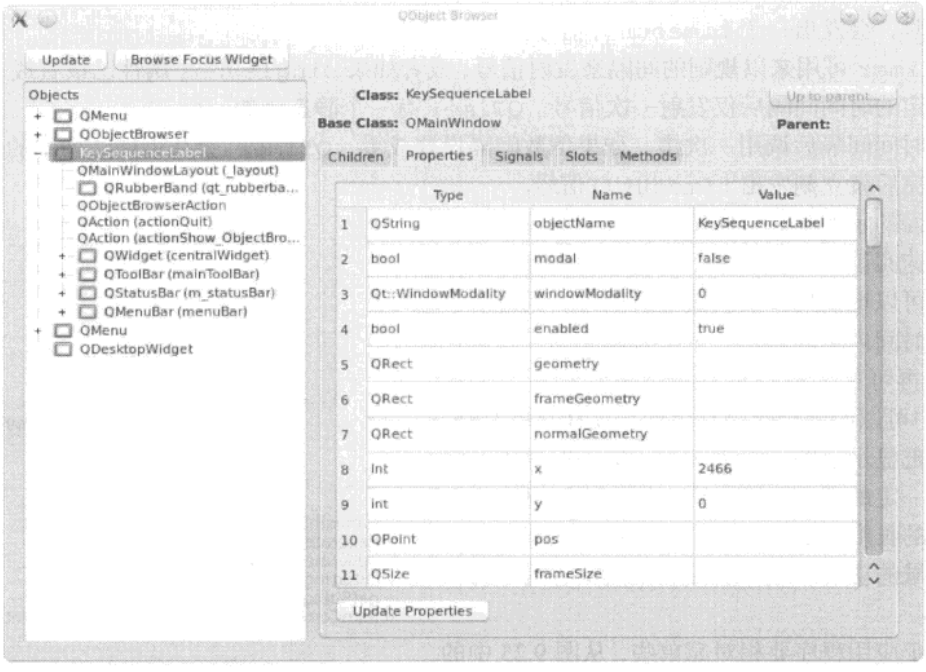


图 9.24　QObjectBrowser 窗件

自定义事件与信号和槽的比较

一般情况下，从语义学的角度看，可以将信号看成是来自对象的更为高级的消息，事件则可以认为是来自用户或系统的更为底层的消息。这两者的机理都体现了 Publish/Subscribe（发布/订阅）模式。

定义一些自定义的事件或者信号和槽还是可能的。利用自定义事件，可以用 QObject::postEvent()对任意的 QObject 进行发布，并可以用 installEventFilter()订阅任意 QObject 的有意向的事件。

绝大多数情况下，如果可以选择，信号—槽连接推荐用于自定义的 QEvent，因为信号和槽要更为安全一些、高级一些，且在消息传递方面更具柔性机制。信号和槽更具柔性机制，是因为发射状态不需要目的对象，而信号可用于多个目标槽中。

信号和槽也需要依靠事件循环，如果打算将线程中对象的信号连接到另一个线程中多个对象的槽上，就会立即注意到这一点。信号可以（以阻塞或者非阻塞的方式）传递给同一个线程或（自身拥有事件循环的）另一个线程中的槽。

① 由 Magland 开发。从其最初发布的地方可以跟踪这个工具开发过程，也可以在 http://tinyurl.com/2a6qkpy 中看到社区的一些建议和作者的回复。

9.9.1 QTimer

上一节中给出了一个使用 QObject 内置定时器的例子。Qt 还有一个 QTimer 类，它为定时器提供一种高级接口。QTimer 对象是一个倒数计时器，以毫秒级时间间隔启动。当其到达零时，会发出一个 timeout()信号。

QTimer 可用来以规则的间隔来发射信号，或者如果 singleShot 属性已设置成 true，则在给定的时间间隔后仅发射一次信号。QTimer 有一个静态函数 singleShot()，可以在给定的时间间隔后调用一次槽。如果倒数间隔设置成零，QTimer 会在事件队列中的全部事件处理完后就立刻发出 timeout()信号。

下面的示例是一个使用 QTimer 编写的训练用户快速阅读的应用程序。该程序的设计者们还声称它可以增加用户的阅读理解能力。其思想就是一段时间内简单显示一个字符串序列，让用户在其不再可见之前尽可能快地输入每一个字符串。用户可以明确给定字符串的长度和显示时间。程序会把显示过的字符串和敲入的字符串进行对比并以一定的形式记下得分。用户可以逐步增加字符串的长度并缩短字符串的显示时间来增强对这一领域的认知，进而过渡成阅读文字的一种高级能力。

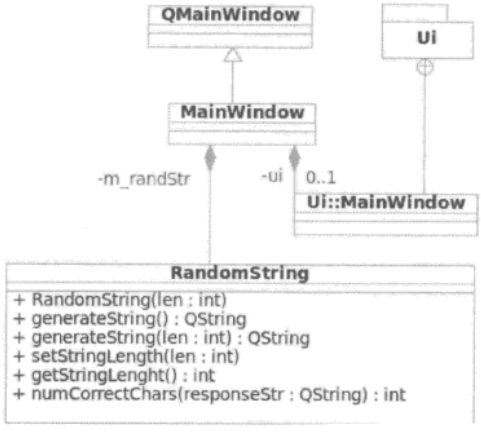


图 9.25 速度—读者的 UML 框图

这个应用程序要相对简单些，从图 9.25 中的 UML 框图即可看出。

示例 9.17 中给出了 MainWindow 的类定义。可以看到它的若干个槽使用了让 QMetaObject :: connectSlotsByName (MainWindow)可以正常工作的命名惯例。这些名称是由 QtCreator 基于这种命名惯例生成的——这就是为什么在代码中看不到它们的连接语句的原因。

示例 9.17 src/timer/speed-reader/mainwindow.h

```
[ . . . . ]
class MainWindow : public QMainWindow {
    Q_OBJECT
public:
    explicit MainWindow(QWidget* parent = 0);
    ~MainWindow();
protected:
    void changeEvent(QEvent* e);
    void processTrial();
private:
    Ui::MainWindow* ui;
private slots:
    void on_nextButton_clicked();
    void on_responseString_returnPressed();
    void on_startButton_clicked();
    void on_lengthSlider_valueChanged(int value);
```

```
    void on_exposureSlider_valueChanged(int value);
    void timerDisplayRandStr();
private:
    int m_expInterval;
    RandomString m_randStr;
    int m_trials;
    int m_correctChars;
    int m_totTrials;
    int m_totCorrectChars;
};
[ . . . . ]
```

这个应用程序使用 QTimer 的静态函数 singleShot() 来控制每个随机字符串的显示时间。在选中的滚动时间间隔耗尽后，singleShot() 会向 timerDisplayRandStr() 槽发射一个 timeout() 信号。如示例 9.18 所示，processTrial() 给出了在没有显式的连接语句的情况下是如何创建连接的。

示例 9.18　src/timer/speed-reader/mainwindow.cpp

```
[ . . . . ]
void MainWindow::processTrial() {
    //clear response text editor
    ui->responseString->setText("");
    //display the random string
    ui->targetString->setText(m_randStr.generateString());
    ui->responseString->setEnabled(false);
    ui->nextButton->setEnabled(false);
    //count the number of trials
    m_trials++;
    m_totTrials++;
    ui->nextButton->setText(QString("String %1").arg(m_trials));
    //begin exposure
    QTimer::singleShot(m_expInterval, this, SLOT(timerDisplayRandStr()));
}


void MainWindow::timerDisplayRandStr() {
    ui->targetString->setText(QString(""));
    //enable the response line editor and next button
    ui->responseString->setEnabled(true);
    ui->responseString->setFocus();
    ui->nextButton->setEnabled(true);
}

[ . . . . ]
```

图 9.26 给出了在选中显示时间和字符串长度后程序的运行截图。

9.9.1.1　练习：QTimer

1. 修改速度—读者应用程序(从示例 9.17 开始)，以便让敲入响应字符串的时间量可限制在用户所给定的时间量之内。需要在用户界面中添加另一个输入窗件。
2. 让用户在工作时可以用一些语言单词而不是随机的字符串。对于这个问题，可自由使

用 src/handouts/canadian-english-small 文件，其中含有英语中的一些特殊方言单词。在本书的 dist 目录中可以下载到 src 压缩包。应该如何控制这些单词字符串的长度？

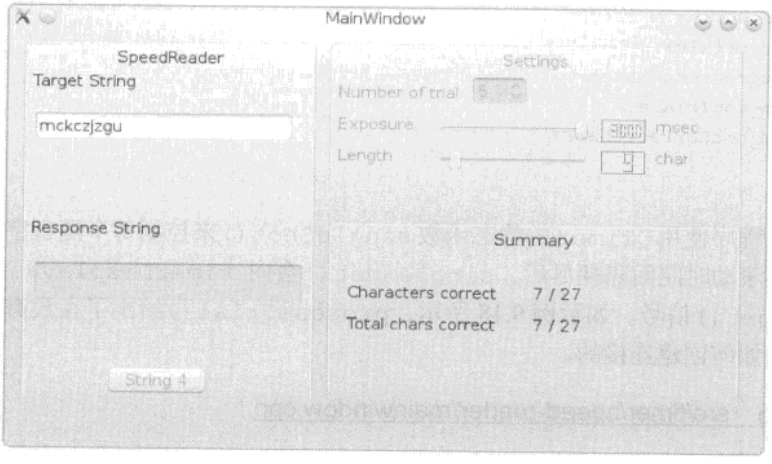


图 9.26 速度—读者屏幕截图

9.10 绘制事件和画图

一个窗件应当在其 paintEvent() 方法中执行适当的绘制操作。这是 QWidget 中唯一可以创建 QPainter(this) 的地方。

正如所看到的，有下面的几个原因使得 QPaintEvent 可以被发送到 QWidget 上。

1. 窗件是隐藏的，然后又显露了出来。
2. 窗件改变了大小或者进行了重新排布。
3. 调用了 update() 或者 repaint()。

示例 9.19 中定义了一个重载了 paintEvent() 的自定义 QWidget。

示例 9.19 src/widgets/life/lifewidget.h

```
[ . . . . ]
class LifeWidget : public QWidget
{
    Q_OBJECT
public:
    explicit LifeWidget(QWidget* parent = 0);
    ~LifeWidget();
    QSize sizeHint() const;
    void paintEvent(QPaintEvent* evt);
public slots:
    void setImage(const QImage& image);
private:
    QImage m_image;
    QSize m_size;
};
[ . . . . ]
```

1 自定义绘制事件。

为 QWidget 获得一个 QPainter 的步骤如下。

1. 创建一个 QPainter(this)。
2. 使用 QPainter 的 API，在 QWidget 上进行绘制。

示例 9.20 给出了一个带有离屏 QImage 的 paintEvent()例子，并直接将其绘制到 QWidget 上。

示例 9.20　src/widgets/life/lifewidget.cpp

```
[ . . . . ]

void LifeWidget::paintEvent(QPaintEvent* evt) {
    QPainter painter(this);                                    1
    if (!m_image.isNull())
        painter.drawImage(QPoint(0,0), m_image);
}
```

1 绝大多数 paintEvent 的第一行。

这个程序基于在康威的《游戏人生》(Conway's Game of Life)[①]中描述的规则，绘制了连续数代的人口地图。图 9.27 给出了一代人口的截屏。17.2.3 节中对其进行并行化时还会再次用到这个游戏。

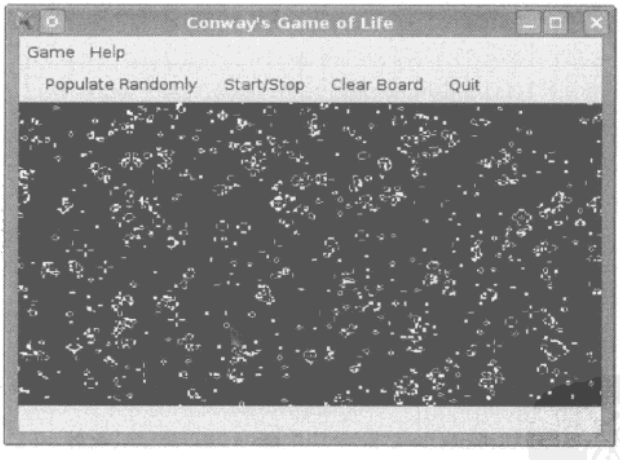


图 9.27　康威的《游戏人生》

通常情况下，不会直接调用 paintEvent()，但可以同步或者异步地将其调度到调用序列中。repaint()在 paintEvent()得到调用后才会返回。update()在 QPaintEvent 被放进事件队列后会立即返回。示例 9.21 是把图片缩放到正确的尺寸并且对其加以保存，在调用 update()之前，确保 LifeWidget 显示的是不久时间内的新图片。

① 参见http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life。

示例 9.21 src/widgets/life/lifewidget.cpp

```
[ . . . . ]

void LifeWidget::setImage(const QImage& image) {
    m_size = image.size();
    m_image = image.scaled(size());
    update();                                                    1
}
```

1 异步——立刻返回。

9.11 复习题

1. 列举所有 QWidget 都有的 6 件事物。
2. 什么是对话框？使用对话框的场合有哪些？
3. 什么是 QLayout？其目的是什么？列举一个具体的 QLayout 类的例子。
4. 窗件可以是布局的子对象吗？
5. 布局可以是窗件的子对象吗？
6. 一个布局可以是另一个布局的子对象吗？
7. 在资源文件中列举图像的优点是什么？
8. 分隔(spacer)和伸展(stretch)的区别是什么？