1、MVC 由三种对象组成。Model 是应用程序对象，View 是它的屏幕表示，Controller 定义了用户界面如何对用户输入进行响应。在 MVC 之前，用户界面设计倾向于三者揉合在一起，MVC 对它们进行了解耦，提高了灵活性与重用性。

2、Qt 中的 model/view 模型主要的类有三种：

- 容器类：QList、QMap
- 模型类：QStringListModel、QDirModel
- 视图类： QListView、QTableView、QTreeView
- 便利类：QListWidget、QTableWidget、QTreeWidget

使用前，一般需要将容器类和模型类绑定在一起，如 QStringListModel 的 setStringList 函数；模型类和视图类绑定在一起，如 QListVier 的 setModel 函数。

3、面向对象编程和面向过程编程不一样，将面向过程的代码打包起来封装成一个 API，就成了面向对象。开发者要做的就是给对象传送数据。

4、自定义模型

- 自定义模型时，对于只读表模型，必须重新实现三个函数：rowCount()、columnCount()和 data()。
- 实现自定义模型可以通过 QAbstractItemModel 类继承，也可以通过 QAbstractListModel 类和 QAbstractTableModel 类继承实现列表模型或表格模型。
- 实现自定义的 view，可继承自 QAbstractItemView 类，对所需要的纯虚函数进行重定义与实现，对 QAbstractItemView 类中的纯虚函数，在子类中必须进行重定义，但不一定要实现，可根据需要选择实现。
- 在表格中嵌入各种不同控件，通过表格的控件对编辑的内容进行限定。通常情况下，这种在表格中插入控件的方式，控件始终显示。当表格中控件数较多时，将影响表格美观，此时，可利用 Delegate 的方式实现同样的效果，控件只有在需要编辑数据项时才会显示，从而解决了上述遇到的问题。

5、model 中的项可以作为各种角色来使用，这允许为不同的环境提供不同的数据。也就是同一个 model，可以根据不同的环境设置角色，以提供不同的数据。模型可以针对不同的组件（或者组件的不同部分，比如按钮的提示以及显示的文本等）提供不同的数据。

6、视口是由任意矩形指定的物理坐标，窗口则是该矩形的逻辑坐标表示。窗口是视口就是简单的线性映射关系。这样，即使视口坐标发生变化，窗口坐标依然不变，就可以根据窗口坐标求出视口坐标。例如，把视口坐标原点（width() - side) / 2, (height() - side) / 2）映射成窗口的坐标原点(-50,-50)，视口坐标点（（width() - side) / 2+side，(height() - side) / 2+side）映射成窗口(50,50)，然后求出线性关系，依照线性关系求出每个坐标点逻辑坐标。矩阵变换只出现在 QPaint 及相关类中。由于窗口坐标是固定不变的，所以无论物理坐标（视口坐标）怎么变化，依照线性关系依然可以求出物理坐标。通常让视口和“窗口”保持相同的纵横比，以防止变形是一个好主意。（视口物理，窗口逻辑）

```
    painter.setViewport((width() - side) / 2， (height() - side) / 2，side，side);
    //设置视口
    painter.setWindow(-50, -50, 100, 100);                    //按给定
    的矩形设置窗口
    draw(&painter);
```

使用时使用 setViewport 和 setWindow 设置视口坐标和窗口坐标，就可以使用窗口坐标绘制出可以线性变换的界面了。

7、整个绘图系统基于 QPainter、QPainterDevice 和 QPaintEngine 三个类。

上面的示意图告诉我们，Qt 的绘图系统实际上是，使用 QPainter 在 QPainterDevice 上进行绘制，它们之间使用 QPaintEngine 进行通讯（也就是翻译 QPainter 的指令）。

8、Qt 的绘制界面在某些时候可以重载重绘事件来完成。

9、注意，声明一个类时，类中各种控件实际上并未定义，只是声明，所以在控制函数里必须分配存储空间。如声明中有这样一句：

```
  QLabel *helloworld;
```

则以下两种定义是错的：

```
  QLabel helloworld = new QLabel(tr("Hello World!!"));
```

这种定义出错是因为前面多了个声明 QLabel；

```
 Qlabel->settext(tr(“Hello World”));
```

这种出错是因为根本没有分配存储空间就直接使用了。

下面这种定义才是正确的：

```
  helloworld = new QLabel(tr("Hello World!!"));
```

11、一个子控件要想捕获键盘事件，该控件必须要有焦点，只有具有焦点的控件能捕获键盘事件。

12、给窗口设置背景时，可以通过调色板 QPalette 的 setBrush 函数和 setPalette 函数完成。如：

```
setAutoFillBackground(true);
QPixmap background(":/images/background.jpg");
QPalette palette;
palette.setBrush(backgroundRole(),
QBrush(background.scaled(600,400,Qt::KeepAspectRatio)));
setPalette(palette);
```

13、通过给 QLabel、QPushButton 的 setPixmap 函数可以给控件添加背景图片，这样可以添加自定义按钮等，如：

```
Qlabel snakelab = new QLabel(this);
snakelab->setMinimumSize(600,300);
snakelab->setMaximumSize(600,300);
QPixmap snakepixmap(":/images/snakepix.png");
snakelab->setPixmap(snakepixmap);
snakelab->setAlignment(Qt::AlignRight) ;
```

14、对于 QWidget，QPainter 只能用在 paintEvent() 函数中。

15、在信号和槽的 connect 函数当中，sender 和 receiver 都是 const 型的指针，使用的时候必须指定 const，否则会出错，如下面就是正确的，parentWidget()函数指向父对象的 const 型指针。

```
connect(this,SIGNAL(scoreSig(QString)), parentWidget()
,SLOT(setScoreNum(QString)));
```

16、当需要子对象和父对象一起关闭的时候，可以利用重新实现 closeEvent 函数并调用 parentWidget()函数：

```
void games::closeEvent(QCloseEvent *event)
{
   event->accept();
   parentWidget()->close();
}
```

17、QConicalGradient 类用于与 QBrush 组合，以指定锥形渐变填充。

18、在实际应用中，我们经常需要让应用程序只有一个实例，再打开新的文档或者页面时，只是替换现在的窗口或者新打开一个标签，而不是重新启动一次应用程序。Qt 中是否可以做到这样呢，答案是肯定的，因为 Qt 本身可以直接调用系统 API，肯定可以做到，但是我们希望找到一个跨平台的通用的解决方案。

这就要用到 Qt 的 QLocalSocket，QLocalServer 类了，这两个类从接口上看和网络通信 socket 没有区别，但是它并不是真正的网络 API，只是模仿了而已。这两个类在 Unix/Linux 系统上采用 Unix 域 socket 实现，而在 Windows 上则采用有名管道（named pipe）来实现。

既然是网络 API，那么思路就很简单了，应用程序启动时首先会去连一个服务器（这里通过应用程序的名字来标识，就像网络端口一样），如果连接失败，那么则自己是第一个实例，就创建这么一个服务器，否则将启动参数发送到服务器，然后自动退出，而服务器会在收到通知以后进行处理。

详见

19、新建快捷键使用 QShortcut 类和 QKeySequence 类：

```
new QShortcut(QKeySequence(Qt::Key_F6), this, SLOT(slotSwapFocus()));
```

20、Q_INIT_RESOURCE()：初始化.qrc 文件中指定的资源。

21、简而言之，框架就是制定一套规范或者规则，程序员在该规范或者规则（思想）下编写代码。

22、Qt 提供了 QFile 类用于进行文件操作。QFile 类提供了读写文件的接口，可以读写文本文件、二进制文件和 Qt 资源文件。处理文本文件和二进制文件，可以使用 QTextStream 类和 QDataStream 类。处理临时文件可以使用 QTemporaryFile，获取文件信息可以使用 QFileInfo，处理目录可以使用 QDir，监视文件和目录变化可以使用 QFileSystemWatcher。

23、读写文本文件的方法通常有两种：一种是直接使用传统的 QFile 类方法；另一种是利用更为方便的 QTextStream 类方法。

24、QDataStream 类提供了将二进制文件串行化的功能，用于实现 C++基本数据类型，如 char、short、int 和 char *等的串行化。

25、QDir 类具有存取目录结构和内容的能力，使用它可以操作目录、存取目录或文件信息、操作底层文件系统，而且还可以存取 Qt 的资源文件。Qt 使用“/”作为通用的目录分隔符和 URL 分隔符。

26、QDir 可以使用相对路径或绝对路径指向一个文件。isRelative()和 isAbsolute()函数可以判断 QDir 对象使用的是相对路径还是绝对路径。

27、QFileinfo 类提供了对文件进行操作时获得的文件相关属性信息，包括文件名、文件大小、创建时间、最后修改时间、最后访问时间及一些文件是否为目录、文件或符号和读写属性等。

28、在 Qt 中可以使用 QFileSystemWatcher 类监视文件和目录的改变。使用 addPath()函数监视指定的文件和目录，如需要监视多个目录，则可以使用 addPaths()函数加入监视。若要移除不需要监视的目录，则可以使用 removePath()和 removePaths()函数。当监视的文件被修改或删除时，产生一个 fileChanged()信号。如果所监视的目录被改变或删除，将产生 directoryChanged()信号。

29、运行 QHostInfo、QNetworkInterface、QNetworkAddressEntry 可获得本机的网络信息。

30、UDP 编程模型：Qt 通过 QUdpSocket 类实现 UDP 协议的编程。

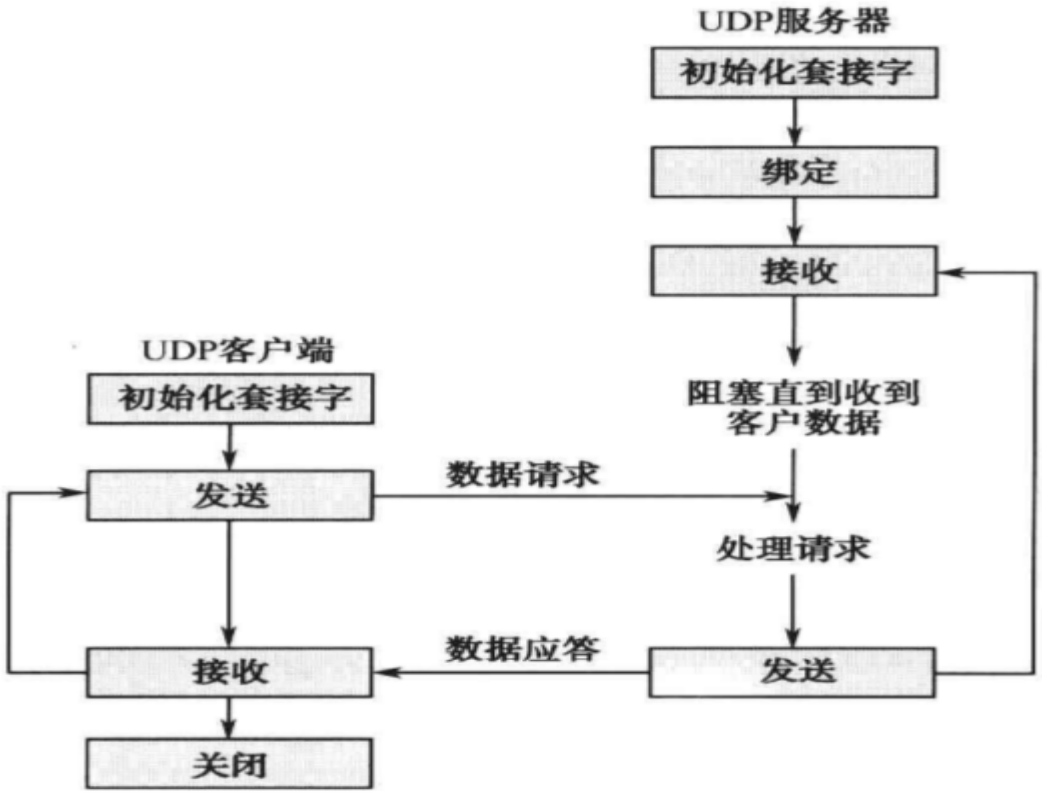


图 10.5　UDP 客户端与服务器间的交互时序

31、TCP 编程模型：Qt 中通过 QTcpSocket 类和 QTcpServer 类实现 TCP 协议的编程。

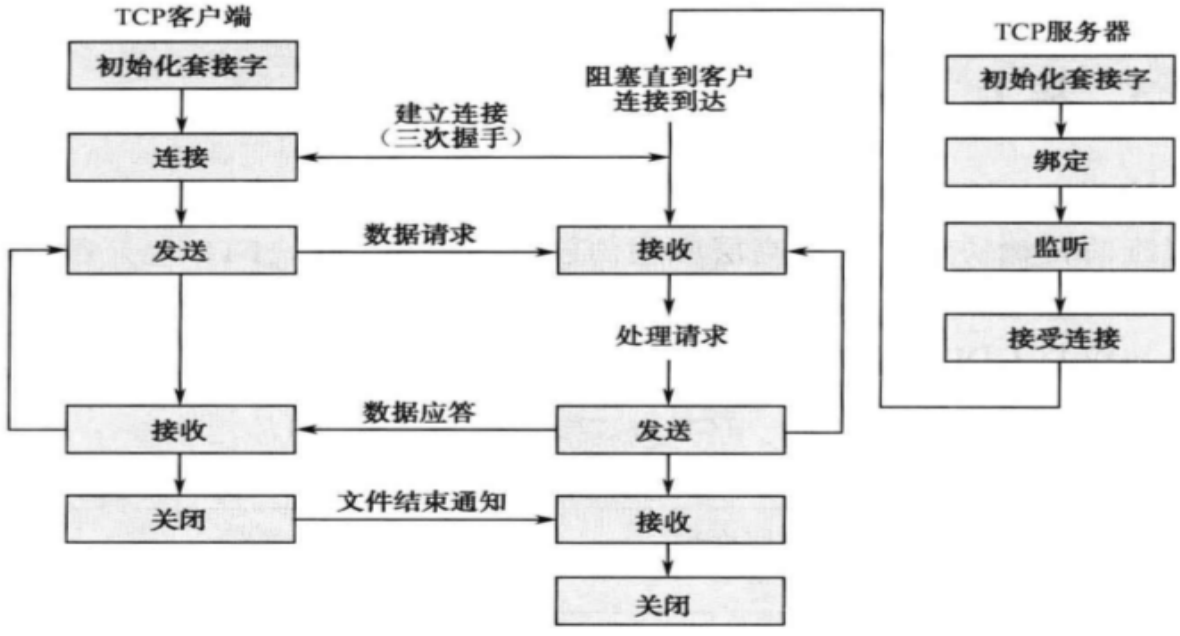


图 10.10　TCP 客户端与服务器间的交互时序

32、QTestlib 框架，是一种针对基于 Qt 编写的程序或库的单元测试工具。QTestlib 提供了单元测试框架的基本功能，并提供了针对 GUI 测试的扩展功能。

C++GUI Qt4 编程

1、Q_PROPERTY()是一个宏，用来在一个类中声明一个属性 property，由于该宏是 qt 特有的，需要用 moc 进行编译，故必须继承于 QObject 类。

- 必须有一个 read 函数。它用来读取属性值。因此用 Const 限定。它的返回值类型必须为属性类型或者属性类型的引用或者指针。不能是其他类型例如：QWidget::hasFocus()。
- 有一个可选的 write 函数。它用来设置属性值，它的返回值必须为 void 型，而起必须要含有一个参数。例如:QWidget::setEnabled()。
- 一个 reset 函数能够把 property 设置成其默认状态，它也是可选的。复位功能必须返回 void，并且不带参数。
- 一个 NOTIFY 信号是可选的。如果定义，它提供了一个信号这个信号在值发生改变时会自动被触发。

2、默认情况下，只有当用户按住一个键不放的时候，会产生 moseMoveEvent 事件。

3、每一个窗口部件都会配备一个调色板，由它来确定做什么事应该使用什么颜色。一个窗口部件的调色板由三个颜色组构成：激活组（Active）、非激活组（Inactive）和不可用组（Disabled）。应该使用哪一个颜色取决于该窗口部件的当前状态:

- Active 颜色组可用于当前激活窗口中的那些窗口部件
- Inactive 颜色组可用于其他窗口中的那些窗口部件
- Disabled 颜色组可用于任意窗口中的那些不可用窗口部件

  QWidget::palette()函数可以返回窗口部件的调色板。

4、在 QT 设计师中使用自定义窗口部件之前，必须让 QT 设计师感觉到它们的存在。有两种方法可以完成这一任务：改进法和插件法。

- 改进法是最为快捷和简单的方法。这种方法包括：选择一个内置的 QT 窗口部件，但该窗口部件要和我们自定义的窗口部件具有相类似的应用程序接口，并在 QT 设计师的自定义窗口部件对话框中填写一些与这个窗口部件相关的信息。
- 插件法需要创建一个插件库，QT 设计师可以在运行时加载这个库，并且可以利用该库创建窗口部件的实例。如对 QDesignerCustomWidgetInterface 进行子类化。

5、Q_INTERFACES 宏告诉 Qt 重新实现哪些接口类。这是编写插件时使用的。

```
Q_INTERFACES(QDesignerCustomWidgetInterface);
```

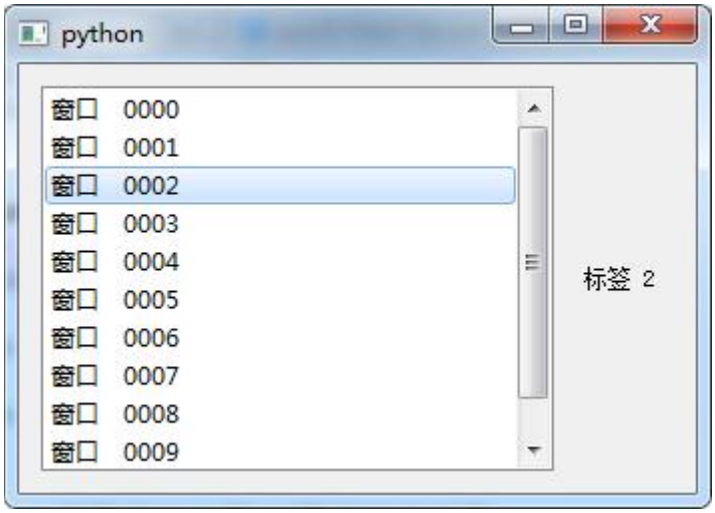
6、双缓冲是一种图形用户界面编程技术，它包括把一个窗口部件渲染到一个脱屏像素映射中以及把这个像素映射复制到显示器上。

注意：渲染是指用软件从模型生成图像的过程。

7、对于需要具有一个图形处理或者图形测绘窗口部件的真正应用程序来说，最好还是使用那些可以获取的第三方窗口部件，而不是创建一个自定义窗口部件。

8、最重要的三种布局管理器是：QHBoxLayout、QVBoxLayout 和 QGridLayout。QGridLayout 的工作基于一个二维单元格。在这个布局中，左上角的位置为（0，0），依此类推。

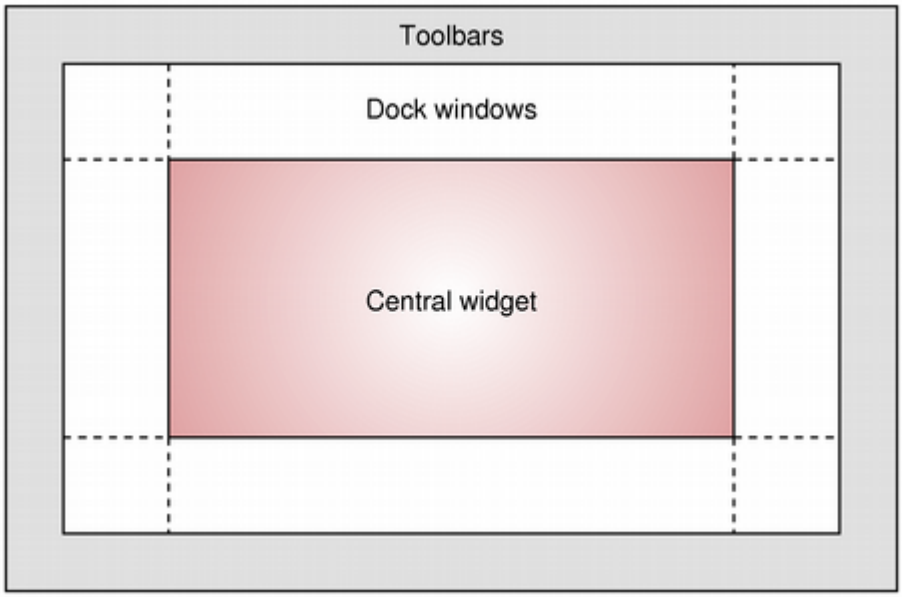
9、QStackedLayout 类可以对一组子窗口部件进行摆放，或者对它们进行分页，而且一次只显示其中一个，而把其他的子窗口部件或者隐藏或者分页。



10、QSplitter 就是一个可以包含一些其他窗口部件的窗口部件。在切分窗口中的这些窗口部件会通过切分条而分隔开来。QSplitter 中的子窗口部件将会自动按照创建时的顺序一个换一个地放在一起，并以切分窗口拖动条来分隔相信窗口部件。

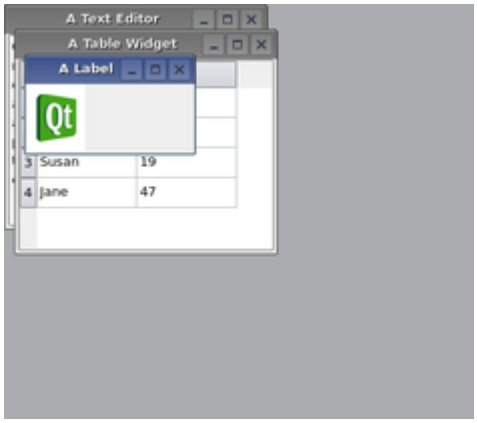
11、QScrollArea 类提供了一个可以滚动的视口和两个滚动条。如果想给一个窗口部件添加一个滚动条，则可以使用一个 QScrollArea 类来实现。QScrollArea 的使用方法，就是以我们想要添加滚动条的窗口部件为参数调用 setWidget()。默认情况下，只有在视口的大小小于窗口部件的大小时，才会把滚动条显示出来。但是通过设置滚动条的策略，可以强制滚动条总是可见的。

12、停靠窗口（dock window）是指一些可以停靠在 QMainWindow 中或是浮动为独立窗口。QMainWindow 提供了 4 个停靠窗口区域：分别在中央窗口部件的上部、下部、左侧和右侧。在 QT 中，各个停靠窗口都是 QDockWidget 的实例。每一个可停靠控件都有一个标题条。用户可以拖动标题栏把窗口从一个可停靠区域移动到另一个可停靠区域。拖动标题栏把窗口移动到不能停靠的区域，窗口就浮动为一个独立的窗口。自由浮动的窗口总是在主窗口的上面。用户点击标题栏上的关闭按钮可以关闭 QDockWidget。调用 QDockWidget::setFeatures()能够设置以上这些属性。



注意：可以使用 QMainWindow::saveState()和 QMainWindow::restoreState()来保存、恢复停靠窗口和工具栏的几何形状和状态。

13、在主窗口的中央区域能够提供多个文档的那些应用程序就称为多文档界面应用程序，或者称为 MDI 应用程序。在 QT 中，通过把 QMdiArea 类作为中央窗口部件，并且通过让每一个文档窗口都成为这个 QMdiArea 的子窗口部件，就可以创建一个多文档界面应用程序了。



14、QStackedLayout、QSplitter、QScrollArea、QDockWidget、QMdiArea。

第 7 章 事件处理

1. 在 QT 中，事件就是 QEvent 子类的一个实例。通过重新实现 keyPressEvent()和 keyReleaseEvent()，就可以处理键盘事件了。
2. QT 中定时器的使用方法：

- 重载 timerEvent(QTimerEvent *)函数，然后再在类的构造函数中设置时间间隔

```
StartTimer(50);                    //单位为毫秒
```

- 在类的构造函数中设定如下：

```
QTimer *timer=new Qtimer(this);
connect(timer,SIGNAL(timeout()),this,SLOT(timeoutslot()));    //timeoutslot()为自定义槽
    timer->start(1000);
```

3. Qt 创建了 QEvent 事件对象之后，会调用 QObject 的 event()函数处理事件的分发。QObject 有一个 eventFilter()函数，重写这个函数，用于建立事件过滤器，过滤拦截某些事件。重写之后调用 QObject::installEventFilter()函数安装这个事件过滤器。

第 8 章 二维图形

1. 反走样是图形学中的重要概念，用以防止通常所说的“锯齿”现象的出现。很多系统的绘图 API 里面都内置了有关反走样的算法，不过由于性能问题，默认一般是关闭的，Qt 也不例外。打开反走样：

```
painter.setRenderHint(QPainter::Antialiasing, true);
```

2. 要在绘图设备（一般是窗口部件）上绘图，只需创建一个 QPainter，再将指针传到该设备中。使用 QPainter 的 draw……()函数，可以绘制各种各样的形状。三个主要的设置是画笔、画刷和字体：

- 画笔用来画线和边缘。使用 QPen 的 setPen()函数设置。
- 画刷用来填充几何图案。使用 QBrush 的 setBrush()函数设置。
- 字体用来绘制文字。使用 QFont 的 setFont 来设置。

3. Qt 支持三种类型的渐变，分别是线性渐变(QLinearGradient)、辐射渐变(QRadialGradient)、锥形渐变(QConicalGradient)：

- 线性渐变由两个控制点定义，连接这两点的线上设置一系列的颜色断点。
- 辐射渐变由一个中心点、半径、一个焦点，以及颜色断点控制。中心点和半径定义一

个圆。颜色从焦点向外扩散，焦点可以是中心点或者圆内的其他点。

- 锥形渐变由一个中心点和一个角度定义，颜色从 x 轴正向偏转一个角度开始，按给定颜色断点旋转扩散。

4. 逻辑坐标、设备坐标和物理坐标：

- 设备坐标和物理坐标应该是一回事；
- “视口”表示物理坐标指定一个任意的矩形，“窗口”用逻辑坐标描述相同的矩形；（视口物理、窗口逻辑）
- 窗口中的每个点都对应一个设备，逻辑坐标实际上是相对坐标。

例如：你用鼠标在屏幕上的某个位置点一下，他会有一个设备坐标，你拉动 VScroll 或 HScroll，再在相同的位置点一下，它还是产生那个设备坐标，而这时它的逻辑坐标已经变了。因为相对于可见部分来说，他们是同一个点。而相对于 Document 来说，他们不是同一个点。

5. 暂时跳过二维图形章节

第 9 章 拖放

1. 拖放操作包括两个截然不同的动作：拖动和放下。QT 的窗口部件可以作为拖动点、放下点或者同时作为拖动点和放下点。
2. 当用户把一个对象拖动到这个窗口部件上时，就会调用 dragEnterEvent()。如果对这个事件调用 acceptProposedAction()，就表明用户可以在这个窗口部件上拖放对象。默认情况下，窗口部件是不接受拖动的。如果希望用户拖动的是文件，而不是其他类型，可以检查拖动的 MIME 类型。
3. 当用户在窗口部件上放下一个对象时，就会调用 dropEvent()。
4. 曼哈顿距离就是两点之间的距离（按照勾股定理进行计算而来），也就是这个向量的长度。startDragDistance()返回执行拖放时系统推荐的最短距离。如果曼哈顿距离大于 startDragDistance()的返回值，才执行拖放，这是为了防止手抖误操作。
5. 拖放操作传送的数据在一个 QMimeData 对象里。这是些数据通过 setMimeData()函数按以下方式指定：

```
QDrag *drag = new QDrag(this);
QMimeData *mimeData = new QMimeData;
mimeData->setText(commentEdit->toPlainText());
drag->setMimeData(mimeData);
```

QDrag::setPixmap()调用则可以在拖放发生时使图标随光标移动。

6. 在 QT 中，使用 QDrag::exec()开启拖放操作，直到用户放下或取消此次拖动操作才会停止。
7. 如果想拖动自定义数据，则必须选择如下三种方式之一：

- 将自定义数据作为 QByteArray 对象，使用 QMimeData::setData()函数作为二进制数据

存储到 QMimeData 中，然后使用 QMimeData::data()读取。

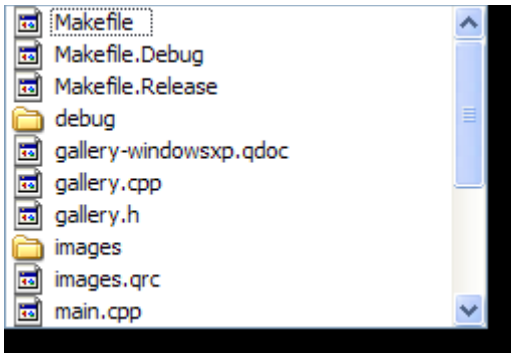
- 继承 QMimeData，重写其中的 formats()和 retrieveData()函数操作自定义数据。
- 如果拖放操作仅仅发生在同一个应用程序，可以直接继承 QMimeData，然后使用任意合适的数据结构进行存储。

这三种选择各有千秋：第一种方法不需要继承任何类，但是有一些局限：即是拖放不会发生，我们也必须将自定义的数据对象转换成 QByteArray 对象，在一定程度上，这会降低程序性能；另外，如果你希望支持很多种拖放的数据，那么每种类型的数据都必须使用一个 QMimeData 类，这可能会导致类爆炸。后两种实现方式则不会有这些问题，或者说是能够减小这种问题，并且能够让我们有完全的控制权。

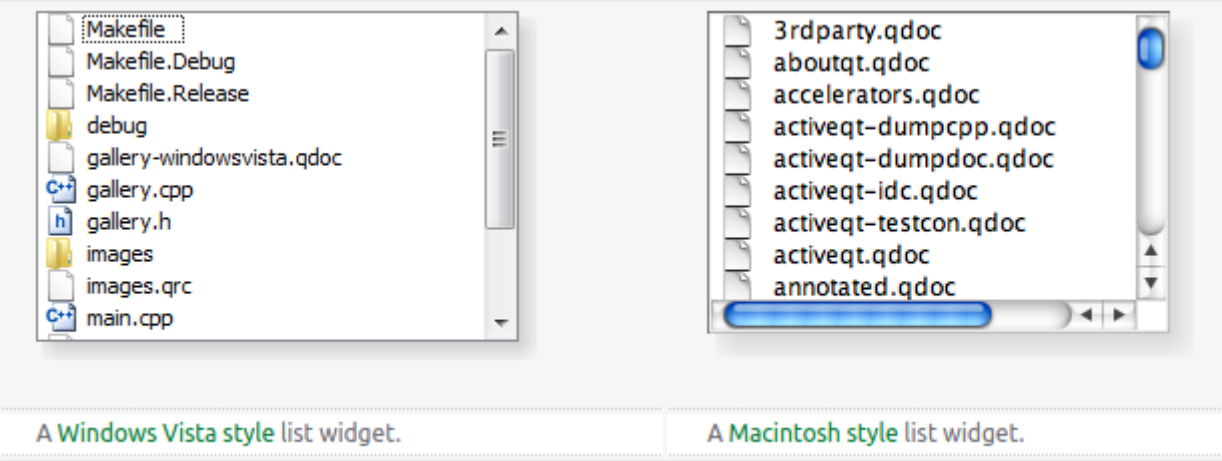
8. 当编写自己的类时，可以通过 QApplication::clipboard()访问剪贴板，它会返回一个指向应用程序 QCliboard 对象的指针。处理系统剪贴板是很容易的：调用 setText()、setImage()或者 setPixmap()把数据放到剪贴板中，并且调用 text()、image()和 pixmap()来重新获得数据即可。
9. QClipboard 提供的数据类型很少，如果需要，我们可以继承 QMimeData 类，通过调用 setMimeData()函数让剪贴板能够支持我们自己的数据类型。

第 **10** 章 项视图类

1. 在 MVC 中，模型负责获取需要显示的数据，并且存储这些数据的修改。每种数据类型都有它自己对应的模型，但是这些模型提供一个相同的 API，用于隐藏内部实现。视图用于将模型数据显示给用户。对于数量很大的数据，或许只显示一小部分，这样就能很好的提高性能。控制器是模型和视图之间的媒介，将用户的动作解析成对数据的操作，比如查找数据或者修改数据，然后转发给模型执行，最后再将模型中需要被显示的数据直接转发给视图进行显示。
2. 一般地，视图将数据向用户进行展示并且处理通用的输入。但是，对于某些特殊要求（比如这里的要求必须输入数字），则交予委托完成。
3. QT 使用类似于 MVC 架构的模型视图架构，不同的是，QT 使用的不是控制器，而是使用了一种稍微有些不同的抽象：委托(delegate)。委托用于对项的如何显示和如何编辑提供精细控制。
4. 使用 QT 的项视图中的简便类通常要比定义一个自定义模型简单得多，并且如果我们不需要由区分模型和视力所带来的好处时，这种方法也是比较合适的。
5. QListView 类提供了一个列表或图标视图上的模型，它是模型/视图类之一，是 Qt 的模型/视图框架的一部分。



6. QListWidget 提供类似 QListView 的一个列表视图，采用了经典的基于项目的界面添加和删除项目。 QListWidget 使用内部模型来在管理列表中每个 QListWidgetItem。



7. QListWidgetItem 和 QListWidget 一起使用，QListWidgetItem 表示在 QListWidget 的单个项目，每个项目可容纳几条信息，并会适当地显示它们。

```
QMapIterator<int, QString> i(symbolMap);                    // 定义一个 JAVA 类型的迭代器
 while (i.hasNext()) {
      i.next();
      QListWidgetItem *item = new QListWidgetItem(i.value(), listWidget);
                                                  // 定义 listWidget 的项目
      item->setIcon(iconForSymbol(i.value()));
      item->setData(Qt::UserRole, i.key());
}
```

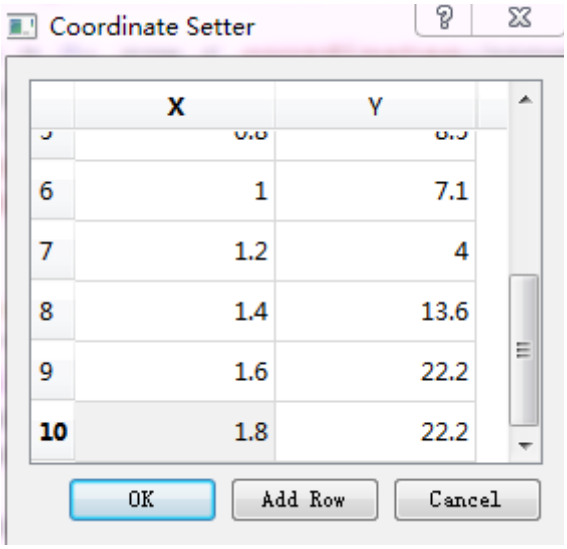
8. QMap 是一个模板类，它提供了一个基于跳跃列表字典。QMap<key,T>是 Qt 的通用容器类之一。它存储键值对，并提供了一个键关联的值的快速查找。QMap 和 QHash 提供非常相似的功能。区别是：QHash 提供比 QMap 更快的查找速度。

```
QMap<int, QString> symbolMap;
symbolMap.insert(132, QObject::tr("Data"));
symbolMap.insert(135, QObject::tr("Decision"));
```

9. QList 是一个提供列表的模板类，是 Qt 的通用容器类之一。它存储值的列表，并提供基于索引的快速访问，以及快速的插入和删除。

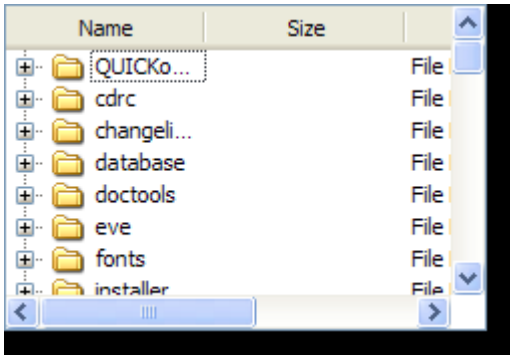
```
QList<QPointF> coordinates;
coordinates << QPointF(0.0, 0.9)
            << QPointF(0.2, 11.0)
```

10. QTableWidget 类提供基于项目的表视图。QTableWidget 中的每一项都使用一个 QTableWidgetItem 表示。setHorizontalHeaderLabels 用于设置表头。




```
QTableWidget tableWidget = new QTableWidget(0, 2);
tableWidget->setHorizontalHeaderLabels(          // 设置表头
        QStringList() << tr("X") << tr("Y"));
```

11. QTreeWidget 类提供了使用树模型的树视图，QTreeWidget 中的每一项都使用一个 QTreeWidgetItem 表示。



12. 在模型中，每一个数据都有一个模型索引和一套属性，称为角色，这些角色保存任意值。模型可以针对不同的组件（或者组件的不同部分，比如按钮的提示以及显示的文本等）提供不同的数据。

13. 自定义模型时，对于只读表模型，必须重新实现三个函数：
rowCount()、columnCount()和 data()。其中：

- rowCount()：返回行数。
- columnCount()：返回列数。
- data()：返回给定角色给定索引下的数据。

14. QVariant 类是一个最为普遍的 Qt 数据类型的联合。不同于其他数据类型，QVariant 包含了很多的数据类型，可以用于多种数据类型。也就是说，QVariant 类型的变量可以保存很多种类型的数据。

15. 一个 model 中的每个数据元素都有一个 model 索引。这个索引指明这个数据位于 model 的位置，比如行、列等。这就是 QModelIndex。每个数据元素还要有一组属性值，称为角色(roles)。这个属性值并不是数据的内容，而是它的属性，比如说，这个数据是用来展示数据的，还是用于显示列头的？因此，这组属性值实际上是 Qt 的一个 enum 定义的，比如，Qt::TextAlignmentRole 用于设置对齐方式，Qt::DisplayRole 用于显示数据。
16. 可编辑自定义模型：当用户编辑一个项时，就会调用 setData()函数；使用 flags()函数得到该如何对一个项进行相关操作（例如，是否可以编辑）。所以，需要重新实现 setData()函数和 flags()函数。
17. 委托用来渲染和编辑视图中不同的项。
18. 自定义委托时，可以使用 QItemDelegate 作为基类，所以可以从默认的委托实现中获益。如果想从头开始做，则可以使用 QAbstractItemDelegate 作为基类。为了提供一个可编辑的数据委托，必须实现 createEditor()、setEditorData()和 setModelData。
19. 继承 QStyledItemDelegate 需要实现以下几个函数：

- createEditor()：返回一个组件。该组件会被作为用户编辑数据时所使用的编辑器，从模型中接受数据，返回用户修改的数据。
- setEditorData()：提供上述组件在显示时所需要的默认值。
- updateEditorGeometry()：确保上述组件作为编辑器时能够完整地显示出来。
- setModelData()：返回给模型用户修改过的数据。

下面依次看看各函数的实现示例：

```
QWidget *SpinBoxDelegate::createEditor(QWidget *parent,
                                       const QStyleOptionViewItem & /* option */,
                                       const QModelIndex & /* index */) const
{
    QSpinBox *editor = new QSpinBox(parent);
    editor->setMinimum(0);
    editor->setMaximum(100);
    return editor;
}
```

在 createEditor()函数中，parent 参数会作为新的编辑器的父组件。

```
void SpinBoxDelegate::setEditorData(QWidget *editor,
                                    const QModelIndex &index) const
{
    int value = index.model()->data(index, Qt::EditRole).toInt();
    QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
    spinBox->setValue(value);
}
```

setEditorData()函数从模型中获取需要编辑的数据（具有 Qt::EditRole 角色）。由于我们知道它就是一个整型，因此可以放心地调用 toInt()函数。editor 就是所生成的编辑器实例，我们将其强制转换成 QSpinBox 实例，设置其数据作为默认值。

```
    void SpinBoxDelegate::setModelData(QWidget *editor,
                                       QAbstractItemModel *model,
                                       const QModelIndex &index) const
{
    QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
    spinBox->interpretText();
    int value = spinBox->value();
    model->setData(index, value, Qt::EditRole);
}
```

在用户编辑完数据后，委托会调用 setModelData()函数将新的数据保存到模型中。因此，在这里我们首先获取 QSpinBox 实例，得到用户输入值，然后设置到模型相应的位置。标准的 QStyledItemDelegate 类会在完成编辑时发出 closeEditor()信号，视图会保证编辑器已经关闭，但是并不会销毁，因此需要另外对内存进行管理。由于我们的处理很简单，无需发出 closeEditor()信号，但是在复杂的实现中，记得可以在这里发出这个信号。针对数据的任何操作都必须提交给 QAbstractItemModel，这使得委托独立于特定的视图。当然，在真实应用中，我们需要检测用户的输入是否合法，是否能够存入模型。

```
    void SpinBoxDelegate::updateEditorGeometry(QWidget *editor,
                                               const QStyleOptionViewItem &option,
                                               const QModelIndex &index) const
{
    editor->setGeometry(option.rect);
}
```

最后，由于我们的编辑器只有一个数字输入框，所以只是简单将这个输入框的大小设置为单元格的大小（由 option.rect 提供）。如果是复杂的编辑器，我们需要根据单元格参数（由 option 提供）、数据（由 index 提供）结合编辑器（由 editor 提供）计算编辑器的显示位置和大小。

现在，我们的委托已经编写完毕。接下来需要将这个委托设置为 QListView 所使用的委托：

```
    listView->setItemDelegate(new SpinBoxDelegate(listView));
```

第 **11** 章 容器类

1. 隐含共享，或者称为写时复制，是一个能够把整个窗口作为不需要太多运行成本的值来传递的优化过程。
2. 向量 QVector 和普通 C++数组的区别在于：向量知道自己的大小并且可以被重新定义大小。在向量末尾添加额外的项是非常快速有效的，而在微量前面或者中间插入项则是比较耗时的。如果事先知道需要多少项，则在定义向量时，就可以初始化向量的大小，并使用[]操作符为它的项赋值。否则，可以稍后重新定义向量的大小，或者在向量的末端增加项。

```
QVector<QString> stringVector;
QVector<QString> vector(200);
```

3. 遍历向量的项的方式之一就是使用[]操作符和 count()函数

```
for (int i = 0; i < vector.count(); ++i) {
    sum+=vect[i];
```

4. QT 还提供了 QLinkedList<T>，这是一种把项存储到内存中不相邻位置的数据结构。
5. 链表不提供[]操作符，所以必须使用迭代器来遍历项。QList<T>连续容器是一个“数组列表”，结合了单一类中 QVector<T>和 QLinkList<T>的最重要的优点。除非我们想在一个极大的列表中执行插入或者要求列表中的元素都必须占据连续的内存地址，否则 QList<T>通常是最合适的多用途容器类。
6. QStringList 类是被广泛应用于 QT 应用编程接口的 QList<QString>的子类。
7. Qstack<T>是一个可以提供 push()、pop()和 top()的向量。QQueue<T>是一个可以提供 enqueue()、dequeue()和 head()的列表。
8. 对于目前所读过的所有容器类，值类型 T 可以是一个与 int、double、指针类型、具有默认构造函数的类、复制构造函数或者赋值操作符相似的类。
9. QT 提供两类迭代器用于遍历存储在容器中的项：Java 风格的迭代器和 STL 风格的迭代器。JAVA 风格的迭代器易于使用，STL 风格的迭代器则可以结合 QT 和 STL 的一般算法而具有更加强大的功能。
10. 对于每个容器类，都有两种 Java 风格的迭代器类型：只读迭代器和读写迭代器。只读迭代器类有 QVectorIterator<T>、QLinkedListIterator<T>和 QListIterator<T>。相应的读写迭代器则在其名字中都含有“Mutable”字样（如 Q MutableVectorIterator<T>）
11. 当使用迭代器时必须首先牢记的是：它们本身并不是直接指向项的，而是能够定位在第一项之前、最后一项之后或者两项之间。
12. 除了 Java 风格的迭代器，每一个连续容器类 C<T>都有两个 STL 风格的迭代器类型 C<T>::iterator 和 C<T>::const_iterator。这两者的区别在于 const_iterator 不允许修改数据。
13. 利用只读 Java 风格的迭代器，不必复制容器。这个迭代器在后台自动生成一个副本，以确保问题遍历首先返回的函数的数据。这是由于采用了名为隐含共享的最优化过程。
14. QT 提供了最后一种在连续容器中遍历项的方式——foreach 循环。在循环的每一次迭代中，迭代变量都被设置为一个新项，从容器中的第一项开始向前迭代。
15. 关联容器可以保存任意多个具有相同类型的项，且它们由一个键索引。Qt 提供两个主要的关联容器：QMap<K,T>和 QHash<k,T>。
16. Qmap<K,T>是一个以升序键顺序存储键值对的数据结构，K 类型必须提供 operator<()，因为 QMap<K,T>要使用这个操作符以升序顺序存储项。QHash<K,T>是一个在哈希表中存储键值对的数据结构。除了对存储在容器类中所有值类型的一般要求，QHash<K,T>中 K 的类型还需要提供一个 operator==()，并需要一个能够为键返回哈希值的全局 qHash()函数的支持。

17. 除了 QHash<K,T>外，QT 还提供了一个用来高速缓存与键相关联的对象的 QCache<K,T>类以及仅仅存储键的 QSet<K>容器。

18. 通用算法：

- qFind()算法在容器类中查找一个特定的值。
- qBinaryFind()算法执行的搜索操作与 qFind()算法相似，其区别在于 qBinaryFind()算法假设项目都是以升序的顺序存储的，并且使用了快速二分搜索而不是 qFind()算法的线性搜索。
- qFill()算法采用一个特定的值组装一个容器。
- qCopy()算法将一个容器类的值复制到另一个容器中。
- qCopy()算法将一个容器类的值复制到另一个容器类中。
- qSort()算法则以升序排序容器类中的项。
- qDeleteAll()算法对每一个存储在容器类中的指针调用 delete。
- qSwap()算法可以交换两个变量的值。

19. 字符串 QString、字节数组 QByteArray 和变量 QVariant：

- toInt()、toLongLong()、toDouble()函数完成从字符串到数字的逆转换。
- 如果仅仅想要检查字符串是否是某个字符（串）开始或者结束，则可以使用 startWith()和 endWith()函数。

第 12 章 输入输出

1. QT 通过 QIODevice 这个封装能够读写字节块“设备”的提取器来实现输入输出，其子类如下：

QFile	在本地文件系统和嵌入式资源中存取文件
QTemporaryFile	创建或访问本地文件系统中的临时文件
QBuffer	从一个 QByteArrary 中读数据或将数据写入到一个 QByteArray 中
QProcess	运行外部程序并处理进程间的通信
QTcpSocket	使用 TCP 协议传输数据流
QUdpSocket	通过网络发送或接收 UDP 数据流
QSslSocket	利用 SSL/TLS 在网络上传输加密的数据流

其中，QProcess,QTcpSocket,QUdpSocket,QSslSocket 是顺序存储设备，只能顺序访问，而 QFile,QTemporaryFile, QBuffer 是随机访问文件，可随机访问，可使用 QIODevice::seek()来重定位文件指针。

2. Qt 也提供了两个更高级的流类，可用于向任何 QIODevice 设备中读或写数据。其中 QDataStream 用于读写二进制数据，而 QTextStream 用于读写文本数据。

3. Qt 还提供了 QDir 和 QFileInfo 类，它们分别用于处理目录地址和提供内部文件信息。

4. Qt 中载入和保存二进制数据的最简单方式是通过实例化一个 QFile 打开文件，然后通过 QDataStream 对象保存它。
5. 用于读取数据的 QDataStream 版本与用于写入数据的 QDataStream 版本一样，必须保证这一点。QDataStream 可以处理多种 C++和 Q t 类型，还可以通过重载<<和>>操作符为用户的自定义类型增加支持。为自定义数据类型提供流操作符有几个好处。其中之一是允许流输出使用自定义类型的容器类。
6. 总之，一共有三种处理 QDataStream 版本的策略：硬编码的版本号、明确地写入并读取版本号以及根据应用程序版本号使用不同的硬编码版本号。
7. 如果想一次读取或者写入一个文件，可以完全不全 QDataStream 而使用 QIODevice 的 write()和 readAll()函数。
8. Qt 提供了 QTextStream 类读写纯文本文件以及如 HTML、XML 和源代码等其他文件格式。写入文本数据非常容易，但读取文本却是一个挑战。因为文本数据（与使用 QDataStream 写入的二进制数据不同）从根本上就是含糊不确定的。

```
Out << “Denmark” << “Norway”;
in >> str1 >> str2;
```

实际上，str1 获得整个词“DenmarkNoorway”，而 str2 什么也没有得到。使用 QDataStream 则不会发生这个问题，因为它在字符串数据前面保存了每个字符串的长度。

9. QDir 类提供了一种与平台无关的遍历目录并获得有关文件信息的方法。
10. QFileInfo 类可以访问文件的属性，比如文件的大小、权限、属主和时间戳，等等。
11. 除了在 Windows 操作系统上认可“\”，QDir 在所有平台上都把“/”认作是目录分隔符。在把路径呈现给用户的时候，可以调用 QDir::convertSeqarators()，这个静态函数把斜线转换为针对具体平台的正确的分隔符。
12. 我们使用 QDir::currentPath()将路径初始化为当前目录。作为备用选择，也可以使用 QDir::homePath()将它初始化为用户的主目录。
13. 利用 Qt 还可以在应用程序的可执行文件中嵌入二进制数据或者文本。这可以通过 Qt 的资源系统来实现的。与文件系统中的普通文件一样，嵌入的文件也可以通过 QFile 读取。
14. 通过下面一行代码加到.pro 文件中来告诉 qmake 包括专门的规则以运行 rcc

```
RESOURCES = myesourcefile.qrc
```

15. 在应用程序中，资源是通过:/路径前缀识别的。
16. QProcess 类允许我们执行外部程序并且和它们进行交互。这个类是异步工作的，且它在后台完成它的工作，这样用户界面就可以始终保持响应。
17. on_objectName_signalName()槽一般用于 Qt 设计师做的界面。
18. setupUi()调用不仅创建并布置所有的窗体部件，还为 on_objectName_signalName()槽建立了信号-槽的连接。

19. QProcess 的另一个用处是：还可以启动其他的用户图形界面应用程序。然而，如果目标是建立应用程序之间的关联，而不是简单地从一个应用程序中调用运行另一个，则最好采用 Qt 的网络类，或其在 Windows 下的 ActiveQt 扩展程序，让应用程序之间能够更好地实现直接通信。而如果想启动用户喜欢的网页浏览器或者电子邮件客户端程序，仅仅需要调用 QDektopServices::openUrl()。

第 **13** 章 数据库

1. 为了执行 SQL 查询，首先必须建立与数据库的连接。
2. 建立数据库连接之后，QSqlQuery 执行底层数据库支持的任何 SQL 语句。
3. QSqlQuery 提供了一些可以遍历结果集的函数：first()、last()、previous()和 seek()。这些函数都很方便，但对于某些数据库，它们可能会比 next()更慢或者更加耗费内存。在操作一个大数据集时，为了便于优化，可以在调用 exec()之前调用 QSqlQuery::setForwardOnly(true)，然后只使用 next()遍历结果集。
4. 如果需要插入多条记录，或者想避免将数值转换成为字符串，可以使用 prepare()来指定一个包含占位符的查询，然后赋值绑定想插入的数值。在 exec()调用之后，可以调用 bindValue()或者 addBindValue()来赋值绑定新值，然后再次调用 exec()并利用这些新值执行查询。
5. QsqlDatabase::database()函数返回一个表示在 creatConnection()中创建的连接的 QsqlDatabase 对象。
6. 一些数据库不支持事务处理。对于这类数据库，transaction()、commit()和 rsollback()几个函数就什么也不做。可以使用 hasFeature()对数据库相关的 QSqlDriver 进行测试，看看这个数据库是不是支持事务处理。
7. 使用 QSqlDriver::handle()和 QSqlResult::handle()函数，还可以读取低级数据库驱动程序句柄和查询结果集的低级句柄。
8. 如果想创建多个连接，可以把数据库名作为第二个参数传递给 addDatabase()。

```
    QSqlDatabase db = QsqlDatabase::addDatabase("QPSQL","OTHER");      // 创建数据库对象
```

9. 可以通过把数据库名传递给 QSqlDatabase::database()得到指向 QSqlDatabase 对象的指针。

```
QSqlDatabase db = QsqlDatabase::database("QPSQL");
```

10. 如果想一次执行多个事务处理，多重连接是很有用的，因为每个连接只能处理一个有效的事务处理。当使用多个数据库连接时，还可以有一个命令的连接，而且如果没有具体指定的话，QSqlQuery 就会使用这个未命名的连接。
11. 除了 QSqlQuery 之外，Qt 提供了 QSqlTableModel 类作为一个高级界面接口，让我们不必使用原始的 SQL 语句来执行大多数常用的 SQL 操作。这个类可以用来独立处理数据库而不涉及任何的图形用户界面，它也可以用作 QListView 或 QTableView 的数据源。

12. 利用 QSqlTableModel::record()获得某一给定的记录，或者利用 value()读取单独的字段。QSqlRecord::value()函数既可以接收字段名也可以接收字段索引。
13. 为了在数据库表中插入记录，可调用 insertRow()来创建一个新的空行（记录），然后使用 setData 设置每一个列（字段）的值。
14. 在调用 submitAll()之后，记录可能会被移动到一个不同的行位置，这取决于表是如何排序的。SQL 模型与标准模型之间最大的区别在于：对于 SQL 模型，必须调用 submitAll()以将发生的更改写入数据库。
15. 主键和外键：

- 关系型数据库中的一条记录中有若干个属性，若其中某一个属性组(注意是组)能唯一标识一条记录，该属性组就可以成为一个主键。比如，学生表(学号、姓名、性别、班级)，其中每个学生的学号是唯一的，学号就是一个主键；课程表(课程编号、课程名、学分)，其中课程编号是唯一的，课程编号就是一个主键。
- 外键用于与另一张表的关联。成绩表中的学号不是成绩表的主键，但它和学生表中的学号相对应，并且学生表中的学号是学生表的主键，则称成绩表中的学号是学生表的外键。

第 **14** 章 多线程

1. 在 QT 应用程序中提供多线程是非常简单的：只需要子类化 QThread 并且重新实现它的 run()函数就可以了。线程要实现的功能在 run()函数里。使用 QThread::start()来开启一个线程。
2. 对于多线程应用程序，一个最基本要求就是能实现几个线程的同步执行。QT 提供了以下这几个用于同步的类：QMutex、QReadWriteLock、QSemaphore 和 QWaitCondition。
3. QMutex 类提供了一种保护一个变量或者一段代码的方法，这样就可以每次只让一个线程读取它。
4. 使用互斥量的一个问题在于：每次只能有一个线程可以访问同一变量。在程序中，可能会有许多线程同时尝试访问读取某一变量（不是修改该变量），此时，互斥量可能就可能会成为一个严重的性能瓶颈。在这种情况下，可以使用 QReadWriteLock，它是一个同步类，允许同时执行多个读取访问而不会影响性能。
5. QSemaphore 是互斥量的另外一种泛化表示形式，但与读写锁定不同，信号量可以用于保护一定数量的相同资源。
6. 主线程是唯一允许创建 QApplication 或者 QCoreApplication 对象，并且可以对创建的对象调用 exec()的线程。调用 exec()之后，这个线程或者等待一个事件，或者处理一个事件。在次线程和主线程之间进行通信的一个解决方案是在线程之间使用信号-槽的连接。通常情况下，信号和槽机制可以同步操作，这就意味着发射信号的时候，使用直接函数即可立即调用连接到一个信号上的多个槽。
7. 当函数可以同时被不同的线程安全地调用时，就称其为“线程安全的”。若将这个概念

推广，当一个类的所有函数都可以同时被不同的线程调用，并且它们之间互不干涉，即使是在操作同一个对象的时候也互不妨碍，我们就把这个类称为线程安全的。

8. 如果类的不同实例可以同时用于不同的线程那么这个类是可重入的。

第 **15** 章 网络

1. Qt 提供了 QFtp 和 QHttp 两个类与 FTP 和 HTTP 协议配合使用。
2. 使用 FTP 和 HTTP 下载文件时需要先新建一个本地文件，再下载覆盖本地文件。
3. QFtp 类在 Qt 中实现了 FTP 协议的客户端程序，它提供了非常多的函数来执行多数的 FTP 操作，同时还可以执行任意的 FTP 指令。QFtp 类是异步工作的，若调用一个像 get()或者 put()这样的函数，它会立即返回并且仅在控制权回到 Qt 的事件循环时才发生数据传输。这样就确保了在执行 FTP 指令时，用户界面可以保持响应。
4. QHttp 类在 Qt 中实现了 HTTP 协议的客户端程序，它提供了非常多的函数来执行多数的 HTTP 操作，包括 get()和 post()，并且还提供了一个改善任意 HTTP 请求指令的方式。QHttp 类是异步工作的，若调用一个像 get()或者 put()这样的函数，它会立即返回并且仅在控制权回到 Qt 的事件循环时才发生数据传输。这样就确保了在执行 FTP 指令时，用户界面可以保持响应。
5. QTcpSocket 和 QTcpServer 类可以用来实现 TCP 客户端和服务器。TCP 是一个传输协议，它构成了包括 FTP 和 HTTP 等很多应用程序层的因特网协议基础，它也可以用于定制用户自己的协议。

其他

1. pro 文件相关内容：

```
#-------------------------------------------------
#
# Project created by QtCreator 2014-04-19T17:42:15
#
#-------------------------------------------------

QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = HelloWorld
TEMPLATE = app


SOURCES += main.cpp\
        mainwindow.cpp

HEADERS  += mainwindow.h
```

```
TRANSLATIONS = HelloWorld_zh_CN.ts
```

- QT　　+= core gui：指明我们项目需要 QtGui 和 QtCore 两个模块。QT 是 qmake 支持的一个内建变量。
- greaterThan(QT_MAJOR_VERSION, 4): QT += widgets：如果 Qt 版本大于 4 时，给项目添加 widgets 模块。
- TARGET = HelloWorld：设定项目的目标文件名，如果不设置，目标文件名会被自动设置为跟项目文件一样的名称。
- TEMPLATE = app：TEMPLATE 变量用来设定项目的构建类型。如果没有在项目文件中设定这个变量，qmake 会默认按应用来构建你的项目。
- SOURCES += main.cpp mainwindow.cpp：包含哪些源文件，多个源文件之间可以用空格隔开。
- HEADERS += mainwindow.h：指定头文件
- 使用 INCLUDEPATH 变量指定第三方库的头文件或自己积累的类库。

```
INCLUDEPATH += e:/protobuf-2.0.3/src \
    e:/media/vlc-1.1.11/sdk/include \
e:/google/protobuf
```

- 有时项目会引入第三方库，或者为了模块化开发而把大的项目拆分，把一些模块编译成动态库进行复用。在 linux 平台下，你可以使用“-L”指定一个库，使用“-l”指定一个具体的库。
- 在 Qt 项目文件中，我们可以使用 DEFINES 变量来定义一个宏。

```
    DEFINES += LIVE_TEST
    DEFINES += DEBUG_CONNECTION
DEFINES += TEST_STUB
```

2. 伸展因子决定了窗体尺寸发生改变时，控件发生改变的比例。

```
m_layout->setColumnStretch(0,3);
m_layout->setColumnStretch(1,1);
```

窗体比例为 3：1。

3. Qt 样式表和 CSS 几乎完全相同，以下列方法使用：

(1) 建立文本文件，写入样式表内容，更改文件后缀名为 qss；

```
QPushButton{
    border:2px solid gray;
    border-radius: 10px;
}
QPushButton:hover{
    color:white;
    background:red;
}
```

(2) 在工程中新建资源文件*.qrc，将 qss 文件加入资源文件 qrc 中，此处注意 prefix 最好为"/"，否则在调用 qss 文件时会找不到文件；

(3) 通过传入路径\文件名的方式创建一个 QFile 对象，以 readonly 的方式打开，然后 readAll，最后 qApp->setStyleSheet 就可以使 qss 生效。

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();
    QFile styleFile(":/resource/sty/style.qss");
    styleFile.open(QIODevice::ReadOnly);
    QString setStyleSheet(styleFile.readAll());;
    a.setStyleSheet(setStyleSheet);

    return a.exec();
}
```

4. 注意类方法和对象方法的区别：类方法并没有对象，对象方法是类对象的方法。使用下列方法调用类方法：

```
 ClassName::function()
```

5. QCoreApplication 为没有 UI 文件的 QT 应用提供一个事件循环。

```
QCoreApplication::setApplicationName("Player Example");
QCoreApplication::setOrganizationName("QtProject");
```

6. QWidget、QMainWindow、QDialog 和 QFrame 的区别：

- 在 Qt 中所有的类都有一个共同的基类 QObject ， QWidget 直接继承与 QPaintDevice 类，QDialog、QMainWindow、QFrame 直接继承 QWidget 类。
- QWidget 类是所有用户界面对象的基类。
- QMainWindow 类提供一个菜单条、锚接窗口（如工具栏）和一个状态条的主应用程序窗口。主窗口通常用在提供一个大的中央窗口部件以及周围菜单、工具条和一个状态条。
- QDialog 类是对话框窗口的基类。对话框窗口是主要用于短时期任务以及用户进行简要通讯的顶级窗口。QDialog 可以是模态对话框也可以是非模态对话框。
- QFrame 类是有框架的窗口部件的基类。

7. 使用 QPainter 的 rotate 函数旋转角度时，以原点为中心，旋转 QPainter 的角度，而不是窗口的角度，也就是说，如果 QPainter 要写一个字，则写好字后，以原点为中心对字进行旋转。