第一章 入门

1.1 第一个 Qt 程序

不可免俗的，从最简单的基本窗口产生开始介绍，窗口标题就叫作 First Qt!!好了，请新增一个目录
hello，并在当中使用任一编辑器来编辑一个 hello.cpp 的档案，内容如下：
hello.cpp

```
#include <QApplication>
#include <QLabel>
int main(int argc, char *argv[])
{
    QApplication
    app(argc, argv);
    QLabel *label = new QLabel("Hello!World! Orz...");
    label->setWindowTitle("First Qt!");
    label->resize(200, 50);
    label->show();
    return app.exec();
}
```

要使用 Qt 的组件，必须含入（include）相对应的定义档案，程序的第一行含入了 QApplication 与 QLabel 标头档案(header file)，稍后才可以使用 QApplication 与 QLabel 两个组件的定义类别。
每个 Qt 窗口程序，都必须有且只能有一个 QApplication 对象，它管理了整个应用程序所需的资源，QLabel 是 Qt 的图型组件之一，继承自 QWidget，Widget 这个名称来自 Window Gadget，表示可视的使用者接口组件，可接受使用者的动作操作，文字画面、按钮、滚动条、工具列甚至容器（Container）等都是一种 Widget。

C++程序从 main 开始，再来进行 Qt 组件的初始化动作，在第一行中：

```
QApplication app(argc, argv);
```

QApplication 负责程序的初始、结束及处理事件（event）的循环等，并提供基本的窗口外观，这个外观与系统的桌面环境有关，例如标题列的样式、窗口外观、系统功能键等，在不同的操作系统桌面环境下，会有各自不同的外观，QApplication 对象接受命令列自变量作为它的自变量，像是如果您没有设定窗口标题，且会使用执行文件的名称作为窗口标题名称，可以使用的自变量与其作用，可以查询 Qt 在线文件关于 QApplication 类别的说明。

接着建立 QLabel 组件，它用以显示所指定的文字（在这边指定了"Hello!World! Orz..."），setWindowTitle()用以设定窗口标题，如果不设定标题，则会自动以程序的文件名称作为标题，resize()方法用以设定组件的长、宽，单位是像素（Pixel），Qt 的组件预设是不可视的，所以要使用 show()方法将之显示出来。

如果您曾经写过 Qt3，可以发现在 Qt4 中，不用指定 MainWidget 了，您可以设定 Widget 的 parent/child 关系，没有指定 parent 的 Widget，就是一个独立的独立窗口（window），要显示独立的 window，就直接呼叫其 show()方法，parent/child 的设定实例，可以参考 使用 Signal 与 Slot（使用拉杆改变 LCD 数字）。

在最后一行,呼叫了 QApplication 的 exec()方法,这将程序的控制权交给了 QApplication，exec()方法会提供一个事件处理循环，窗口显示之后会不断倾听（listen）事件，像是键盘、鼠标等动作所引发的事件，撰写好程序存盘之后，就可以开始进行 make 的动作，必须先产生 Makefile，Qt 提供了 qmake 程序来协助建立 Makefile，它可以自动根据目前目录下档案产生*.pro 的专案档（project file）：

qmake -project
然后根据项目档产生 Makefile：
qmake
接下来就可以进行 make：
make
make 完成之后，可以在 debug 目录下找到产生的 hello.exe 档案，直接执行：
hello

如果您要可以在 Windows 下直接 double click 就执行程序,记得系统环境变量中要设定 PATH 包括 Qt 安装目录下的 bin 目录，执行时的参考画面如下所示：

如果要建构 release 版本，则使用 make 时指定-f 与 Makefile 名称，例如：
make -f Makefile.Release
则您可以在 release 数据夹下看到建构好的档案。
QLabel 支持 HTML 卷标，如果您把程序改为以下的内容：

```
#include <QApplication>
#include <QLabel>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel  *label=new  QLabel("<h1><font  color=blue>Hello!World!</font><font color=red>Orz...</font></h1>");
    label->setWindowTitle("First Qt!");
```

```
    label->resize(200, 50);
    label->show();
    return app.exec();
}
```

重新建置之后，执行结果将显示如下：

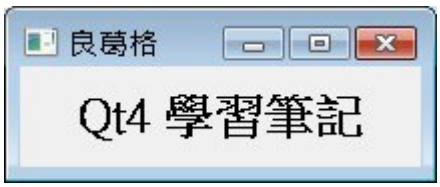
1.2 简单的显示中文（使用 Unicode 转换）

如果您的作业环境是中文环境，并想让 Qt 窗口程序显示中文，最简单的方法就是使用 Unicode 转换，这必须使用到 QTextCodec 类别的方法。

QTextCodec 提供静态的 codecForName()方法，可以指定国际化文字名称，以正体（繁体）中文来说是指定 Big5-ETen，codecForName ()方法会传回 QTextCodec 实例，您使用实例的 toUnicode()方法将 Big5 码转换为 Unicode，然后当作一个 QString 使用，下面这个程序是个简单的实作：

```
#include <QApplication>
#include <QLabel>
#include <QTextCodec>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("Big5-ETen");
    QLabel *label = new QLabel;
    label->setText(codec->toUnicode("<center><h1>Qt4 学习笔记</h1></center>"));
    label->setWindowTitle(codec->toUnicode("良葛格"));
    label->resize(200, 50);
    label->show();
    return app.exec();
}
```

如果没有黑体字的部份来转换中文字码，而直接指定中文字的话，程序执行时将出现乱码。下面为程序执行时的参考画面：

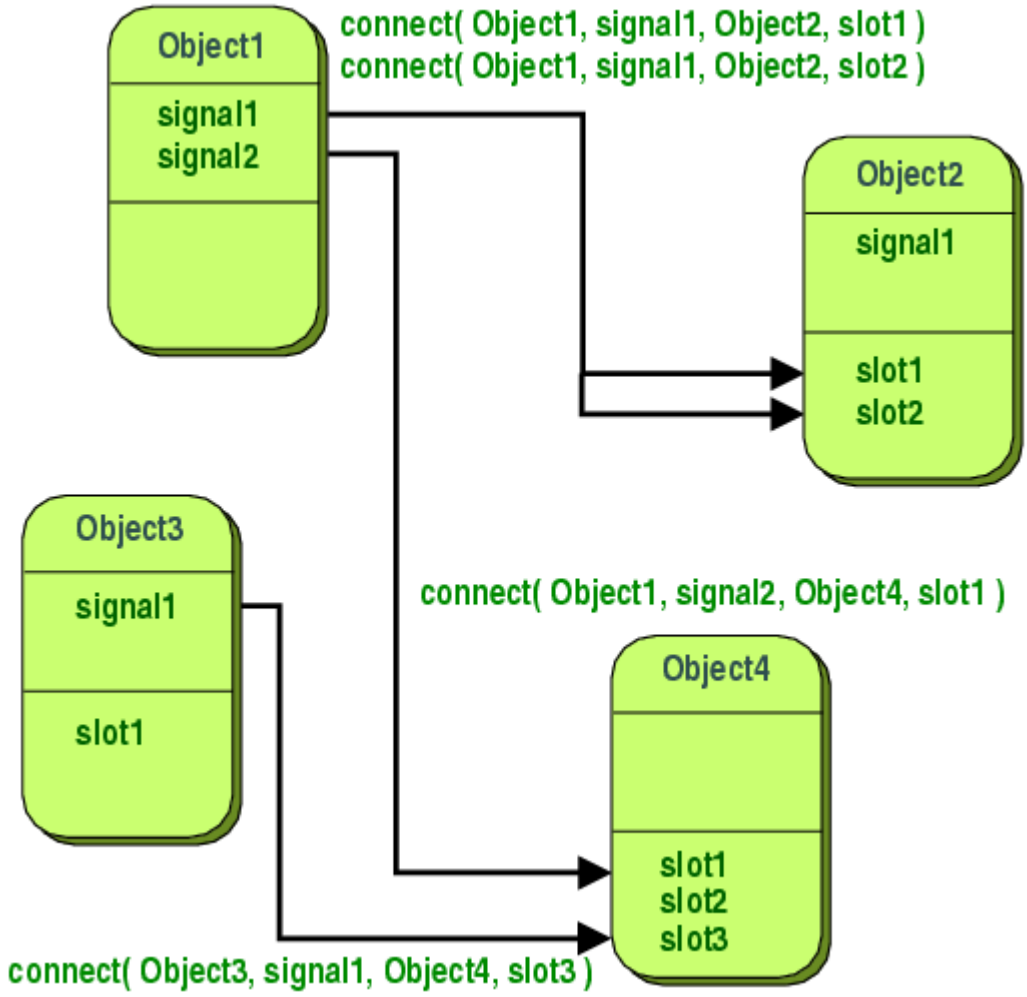


第二章 信号与曹函数

2.1 使用 Signal 与 Slot（使用按钮关闭窗口）

在 Qt 之中，当某个组件发生状态改变，而另一个组件想得知其状态改变，并作出一些相对应行为的话，可以使用 Signal 与 Slot 的机制来达到，Signal 与 Slot 机制是 Qt 与其它框架非常不同的核心特性。

在 Qt 中，您可以设计对象拥有一些 Signal，在特定的情况下，例如状态改变、使用者的操作事件发生时，发出（emit）特定的 Signal，对于某些 Signal 有兴趣的对象，可以设计一些 Slot 来接受特定的 Signal，并定义相对应的动作，Signal 与 Slot 之间，可透过 QObject 的静态方法 connect 来连结，Signal 与 Slot 之间的处理是同步的（Synchronized）。



（图片取自 Qt 官方的 Signals and Slots 文件）

使用 Signal 与 Slot，Qt 中的对象可以不用知道彼此的存在，但又可以彼此沟通，在 Qt 中的组件，预设有一些 Signal 与 Slot，例如按钮组件 QPushButton 若被按下，会发出 clicked()的 Signal，您可以将之连接（connect）至 QApplication()的 quit()这个 Slot，quit()会执行关闭 Qt 应用程序的动作。

以下的程序实作出上述的行为：

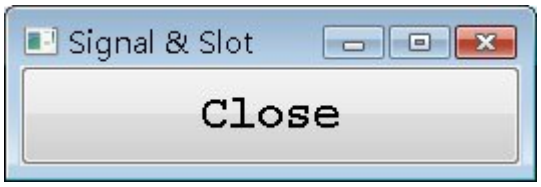
```
#include <QApplication>
#include <QPushButton>
#include <QFont>

int main(int argc, char *argv[])
```

```
{
    QApplication app(argc, argv);
    QPushButton *btn = new QPushButton("Close");
    btn->setWindowTitle("Signal & Slot");
    btn->setFont(QFont("Courier", 18, QFont::Bold));
    btn->resize(250, 50);
    QObject::connect(btn, SIGNAL(clicked()), &app, SLOT(quit()));
    btn->show();
    return app.exec();
}
```

QPushButton 是 Qt 的按钮组件，您可以设定按钮所显示的文字，或者是使用 QFont 设定按钮文字的字型，在这边设定字型为 Courier、大小 18、黑体字。

QObject 是 Qt 中许多类别的父类别，是 Qt 对象模型的核心。connect()方法的第一个参数是发出 Signal 的对象之地址，第三个参数是对 Signal 有兴趣的对象之地址。SIGNAL ()与 SLOT()为宏函式，是语法的一部份，所传入的 Signal 或 Slot 为没有参数名称的函式签名（function signature），在上面的例子中，如果按钮 btn 被按下，会发出 clicked()的 Signal，而处理的 Slot 为应用程序 app 的 quit ()函式。

可以看到，程序中 btn 与 app 并不知道彼此的存在，而是藉由 connect()连接 Signal 与 Slot，这降低了对象之间的耦合度。在建置这个程序并执行之后，若按下窗口中的按钮，将会直接关闭窗口，窗口画面如下所示：



在 Qt 的在线文件 中，可以于 API Reference 中，查询各个类别的 Signal 或 Slot 及其作用。

2.2 使用 Signal 与 Slot（使用拉杆改变 LCD 数字）

在 使用 Signal 与 Slot（使用按钮关闭窗口） 中，QPushButton 的 clicked() Signal 及 QApplication 的 quit() Slot 都不带参数，Signal 在发出时是可以带参数的，而相对应的 Slot 也可以接受参数。

以下的例子将看到有参数的 Signal 发送及 Slot 接受，并也将介绍 Qt 组件的 parent/child 关系，这个程序将建立一个 LCD 数字显示组件，以及一个拉杆组件，LCD 数字将反应目前拉杆的进度：

```
#include <QApplication>
#include <QWidget>
#include <QSlider>
#include <QLCDNumber>

int main(int argc, char *argv[])
{
```

```
    QApplication app(argc, argv);

    QWidget *parent = new QWidget;
    parent->setWindowTitle("Signal & Slot");
    parent->setMinimumSize(240, 140);
    parent->setMaximumSize(240, 140);
    QLCDNumber *lcd = new QLCDNumber(parent);
    lcd->setGeometry(70, 20, 100, 30);
    QSlider *slider = new QSlider(Qt::Horizontal, parent);
    slider->setRange(0, 99);
    slider->setValue(0);
    slider->setGeometry(70, 70, 100, 30);
    QObject::connect(slider, SIGNAL(valueChanged(int)),
                             lcd, SLOT(display(int)));
    parent->show();
    return app.exec();
}
```

在 Qt 中建立 Widget 时，要建立在 heap 区（即以 new 的方式），Qt 会自动管理 parent 下 child 的 delete，让您不用亲自管理具有 parent/child 关系的 Widget 建构与删除，这可以避免 memory leak，您要 delete 的只有那些没有 parent 的对象，如果您将对象建立在 stack 区，程序将可能会有错误发生。

QWidget 是 Qt 中所有使用者图形接口组件的父类别，可在屏幕上绘制自身，可接受鼠标、键盘等接口操作，一个 QWidget 可以指定它的 parent 为哪个组件，而这也表示 child 可显示的范围将是在 parent 之内，parent 没有显示的话，子组件也不会显示。没有指定 parent 的 QWidget 是一个独立窗口（window），例如先前所看到的几个 Qt 范例，无论是 QLabel 或 QPushButton，都没有指定 parent，它们可独立的显示在画面之中，只要呼叫其 show()方法。

在程序中建立了一个 QWidget 实例，并设定它的标题名称，以及可拉动的最大（setMaximumSize）最小（setMinimumSize）尺寸，由于都设定为 240X140 像素大小，所以这个窗口就变为不可变动大小的了，也可以只使用一个 setFixedSize()方法来设定：

```
QWidget *parent = new QWidget;
parent->setWindowTitle("Signal & Slot");
parent->setMinimumSize(240, 140);
parent->setMaximumSize(240, 140);
```

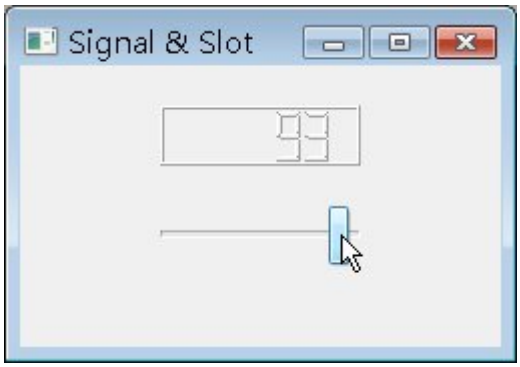
这个 QWidget 没有指定 parent，所以它是一个独立窗口，接下来的 QLCDNumber 实例建立时，指定了这个 QWidget 为它的 parent，所以 QLCDNumber 被置入了 QWidget 之中成为 child，可显示的范围限制在 parent 的边界大小之中，它在 parent 中的位置为 X: 70、Y: 20，长为 100、宽为 30（setGeometry()）：：

```
QLCDNumber *lcd = new QLCDNumber(parent);
lcd->setGeometry(70, 20, 100, 30);
```

接下来的 QSlider 实例在建立时，也指定了这个 QWidget 为它的 parent，程序中设定 QSlider 为水平拉杆（Qt:: Horizontal），可拉动的数值范围为 0 到 99（setRange()），目前拉杆光标值

为 0（setValue()），而它在 parent 中的位置为 X: 70、Y: 70，长为 100、宽为 30：
QSlider *slider = new QSlider(Qt::Horizontal, parent);
slider->setRange(0, 99);
slider->setValue(0);
slider->setGeometry(70, 70, 100, 30);

当您拉动 QSlider 的光标，造成光标值变动时会发出 valueChanged(int) Signal，参数 int 表示 Signal 带有一个整数值，在这表示 QSlider 的游标值一并被发出，QLCDNumber 的 display(int) Slot 接受 Signal 所带来的整数值，可以在 QLCDNumber 显示数字：
QObject::connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
一个程序执行时的参考画面如下所示：



2.3 自订 Signal 与 Slot

除了使用 Qt 现有组件预先定义好的 Signal 与 Slot 之外，您也可以定义自己对象的 Signal 与 Slot，方式是继承 QObject 或它的子类别（例如 QWidget），以下直接使用实际例子来说明。

在 使用 Signal 与 Slot（使用拉杆改变 LCD 数字） 中，您直接使用拉杆来变动 LCD 数字显示，假设现在您想要定义一个对象，当拉杆拉动时，必须通知该对象储存拉杆的光标值，而对象储存的光标值有变动时，LCD 数字显示也必须更新，这样的一个对象不是图形组件，它是个数据模型，用以储存与图形接口无关的数据。
您可以这么定义一个 Model 类别：
Model.h

```
#ifndef MODEL_H
#define MODEL_H
#include <QObject>

class Model : public QObject
{
    Q_OBJECT
public:
    Model() { m_value = 0; }
    int value() const { return m_value; }
public slots:
    void setValue(int);
signals:
```

```
    void valueChanged(int);
private:
    int m_value;
 };
#endif
```

只有在 MODEL_H 名称没被定义过（#ifndef）的情况下，才会编译以下的定义内容。这是个类别定义的技巧，可以避免类别定义的重复，如果类别定义重复，该段定义将不会被编译：

```
#ifndef MODEL_H
#define MODEL_H
```

再来简介一下 Qt 的 Meta-Object System，它基于以下三个部份：

(1)QObject 类别

(2)Q_OBJECT 宏

(3)Meta-Object Compiler（moc）

Qt 管理的对象必须继承 QObject 类别，以提供 Qt 对象的 Meta 讯息，若要实作 Signal 与 Slot 机制，则必须包括 Q_OBJECT 宏，moc 会处理 Qt 的 C++扩充（Meta-Object System），使用 moc 读取 C++标头档案，若发现类别定义中包括 Q_OBJECT 宏，就会产生 Qt meta-object 相关的 C++程序代码。

若您使用 qmake 来产生 Makefile，若必要时，档案中就会包括 moc 的使用，程序完成建置之后，会在 release 或 debug 目录中，找到 moc_Model.cpp，即为 moc 所提供的 C++程序代码。

在 Model 中，自订了 Signal 与 Slot，slots 与 signals 关键词其实是宏，将被展开为相关的程序代码，其中 Slot 定义 setValue(int)，将接收 Signal 传来的整数数据，如果不想接受数据的话，int 可以省去， Signal 定义 valueChanged(int)，表示将发出的 Signal 会带有一个整数：

```
public slots:
    void setValue(int);
signals:
    void valueChanged(int);
```

Signal 与 Slot 的签名是对应的，若 Signal 带有参数，则对应的 Slot 也要带有参数。

接着定义 Model.cpp：

Model.cpp

```
#include "Model.h"

void Model::setValue(int value)
{
    if (value != m_value)
    {
        m_value = value;
        emit valueChanged(m_value);
    }
}
```

Slot 只是一般的函式，可以由程序的其它部份直接呼叫，也可以连接至 Signal，若有呼叫 setValue()，程序执行到 emit 时，就会发出 valueChanged()的 Signal。

接着修改一下 使用 Signal 与 Slot（使用拉杆改变 LCD 数字）中的范例：

main.cpp

```
#include <QApplication>
#include <QWidget>
#include <QSlider>
#include <QLCDNumber>
#include "Model.h"

int main(int argc, char *argv[])
{
	QApplication app(argc, argv);

	QWidget *parent = new QWidget;
	parent->setWindowTitle("Signal & Slot");
	parent->setMinimumSize(240, 140);
	parent->setMaximumSize(240, 140);

	QLCDNumber *lcd = new QLCDNumber(parent);
	lcd->setGeometry(70, 20, 100, 30);

	QSlider *slider = new QSlider(Qt::Horizontal, parent);
	slider->setRange(0, 99);
	slider->setValue(0);
	slider->setGeometry(70, 70, 100, 30);
	Model model;
	QObject::connect(slider, SIGNAL(valueChanged(int)),
				&model, SLOT(setValue(int)));
	QObject::connect(&model, SIGNAL(valueChanged(int)),
				lcd, SLOT(display(int)));
	parent->show();
	return app.exec();
}
```

您使用 connect()连接 QSlider 的 valueChanged() Signal 及 Model 的 setValue() Slot，所以拉动拉杆时，Model 的 m_value 就会被设定为 QSlider 的光标值，而 setValue()中使用了 emit 发出 valueChanged() Signal，由于您将 Model 的 valueChanged() Signal 连接至 QLCDNumber 的 display() Slot，所以 LCD 显示数字也会改变。

一个 Signal 可以连至数个 Slot，例如这个程序的 Signal 与 Slot 连接也可以改为，由 QSlider 同时发出 Signal 给 Model 及 LCD 显示，执行结果不变：

```
QObject::connect(slider, SIGNAL(valueChanged(int)),
```

```
                    &model, SLOT(setValue(int)));
QObject::connect(slider, SIGNAL(valueChanged(int)),
                    lcd, SLOT(display(int)));
```

一个 Slot 也可以被数个 Signal 连接，例如：

```
QObject::connect(slider, SIGNAL(valueChanged(int)),
                    &model, SLOT(setValue(int)));
QObject::connect(combox, SIGNAL(valueChanged(int)),
                    &model, SLOT(setValue(int)));
```

Signal 与 Slot 的签名基本上要相同，但若 Signal 的参数多于 Slot 的参数，则额外的参数会被 Slot 忽略。如果要断开 Signal 与 Slot 的连接，则使用 disconnect()，例如：

```
QObject::disconnect(slider, SIGNAL(valueChanged(int)), &model, SLOT(setValue(int)));
```

第三章 事件处理

3.1 事件类型与处理者

当您执行 QApplication 的 exec()方法之后，应用程序会进入事件循环来倾听应用程序的事件，事件来源通常是窗口系统，例如使用者的鼠标事件或键盘事件，事件来源可以是 Qt 应用程序事件本身，例如 QTimerEvent（在 QTimer 与 QLCDNumber 中有 QTimerEvent 的范例），事件来源也可以是使用者自定义的事件，透过 QApplicaiton 的 sendEvent()或 postEvent()来发送。

当事件发生时，Qt 为之建立事件实例，QEvent 是 Qt 中所有事件的基础类别，Qt 所建立的事件实例为 QEvent 的子类别实例，并将之传送给 QObject 子类别实例的 event()函式，event()这个函式本身通常不直接处理事件，而是基于所传送的事件类型，分派给处理特定类型的事件处理者（Event Handler），这在 事件接受与否、event() 方法 中有进一步说明。

QEvent 是 Qt 中所有事件的基础类别，最常见的事件类型皆为其子类别，像是鼠标事件的 QMouseEvent、键盘事件的 QKeyEvent、缩放事件的 QResizeEvent 等，这些子类别事件皆加入其特定的函式，像是鼠标事件的 x()、y()函式指出发生鼠标事件时，鼠标光标的 x、y 坐标，键盘事件的 key()函式可以取得目前所按下的按键常数。

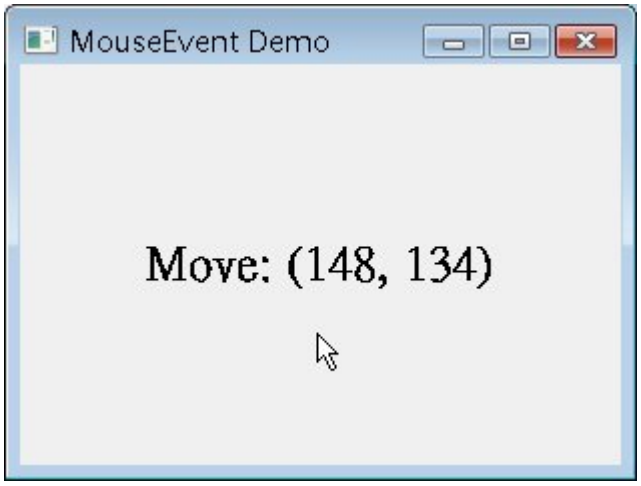
以图型组件来说，通常您会继承 QWidget 或其子类别，并重新定义事件处理者，也就是事件处理函式，QWidget 定义了像是 keyPressEvent() 、 keyReleaseEvent() 、mouseDoubleClickEvent()、mouseMoveEvent ()、mousePressEvent()、mouseReleaseEvent()等事件处理函式，并接受 QEvent 的特定子类别实例作为自变量，您只要根据想要处理的事件重新定义对应的函式即可进行事件处理。

以下则是个简单的事件处理示范，继承了 QLabel 并重新定义了相关的事件处理者，当鼠标移动、按下或放开时，显示鼠标光标的所在位置：

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QMouseEvent>
```

```
class EventLabel : public QLabel
{
protected:
    void mouseMoveEvent(QMouseEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
};

void EventLabel::mouseMoveEvent(QMouseEvent *event)
{
    QString msg;
    msg.sprintf("<center><h1>Move: (%d, %d)</h1></center>",
                event->x(), event->y());
    this->setText(msg);
}

void EventLabel::mousePressEvent(QMouseEvent *event)
{
    QString msg;
    msg.sprintf("<center><h1>Press: (%d, %d)</h1></center>",
                event->x(), event->y());
    this->setText(msg);
}

void EventLabel::mouseReleaseEvent(QMouseEvent *event)
{
    QString msg;
    msg.sprintf("<center><h1>Release: (%d, %d)</h1></center>",
                event->x(), event->y());
    this->setText(msg);
}

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    EventLabel *label = new EventLabel;
    label->setWindowTitle("MouseEvent Demo");
    label->resize(300, 200);
    label->show();
    return app.exec();
}
```

执行时的参考画面如下所示：



Qt 的事件跟 Signal、Slot 机制是不同的。Signal 与 Slot 的机制是同步的（Synchronous），Signal 是由对象发出的，使用 QObject 的 connect()连接对象上定义的 Slot 来立即处理。Qt 的事件可以是异步的（Asynchronous）的，Q 使用一个事件队列来维护，新的事件产生时基本上会被排到队列的尾端，前一个事件处理完成，再从队列的前端取出下一个队列来处理，必要的时候，Qt 的事件也可以是同步的，而事件还可以使用 事件过滤器 进行过滤处理。

3.2 事件接受与否、event() 方法

不同类型的事件，都有对应的事件处理函式，它们接受 QEvent 的特定子类别实例作为自变量，像是下例中 mousePressEvent()事件处理函式上的 QMouseEvent，您可以针对事件的某些状况作特定处理，而其它未处理的状况，则呼叫父类别对应的的事件处理函式，让父类别预先定义的事件处理可以完成：

```
void CustomLabel::mousePressEvent(QMouseEvent *event) {
     if (event->button() == Qt::LeftButton) {
         // 处理左键按下
         // ....
     } else {
         // 由父类别所定义的事件处理函式来事件
         QLabel::mousePressEvent(event);
     }
}
void CustomLabel::mouseReleaseEvent(QMouseEvent *event) {
    // 鼠标放开事件处理....
}
```

事实上，每个可传播的事件都有 accept()与 igore()两个方法，用以告知 Qt 应用程序，这个事件处理者是否接受或忽略此一事件，如果事件处理者中呼叫事件的 accept()，则事件不会再进一步传播，若呼叫了 ignore()，则 Qt 应用程序会尝试寻找另一个事件的接受者，您可以藉由 isAccepted()方法得知事件是否被接受。

一般来说，除了 QCloseEvent 之外，很少直接呼叫 accept()或 ignore()，如果您接受事件，则在事件处理者当中实作对事件的处理（如上例的 if 陈述句），如果您不接受事件，则直接呼叫父类别的事件实作（如上例的 else 陈述句），对于 QWidget 来说，预设的实作是：

```
void QWidget::keyPressEvent(QKeyEvent *event) {
    event->ignore();
}
```

由于 QWidget 预设的实作是呼叫 ignore()，这让事件可以向父组件传播。

QCloseEvent 则建议直接呼叫 accept()与 ignore()，accept()方法会继续关闭的操作，ignore()则会取消关闭的操作：

```
void MainWindow::closeEvent(QCloseEvent *event) {
    if (continueToClose()) {
        event->accept();
    } else {
        event->ignore();
    }
}
```

QObject 的 event()方法通常用于分派事件，但在某些情况下，您希望在事件分派给其它事件处理者之前，先行作一些处理，则可以重新定义 event()方法，例如在窗口程序中，Tab 键按下时希望其将焦点移至下一个图型组件，而不是直接让目前焦点的图形组件直接处理 Tab 键，则您可以在继承 QWidget 子类别时，重新定义其 event()方法，例如：

```
bool CustomWidget::event(QEvent *event) {
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if (keyEvent->key() == Qt::Key_Tab) {
            // 处理 Tab 键
            return true;
        }
    }

    return QWidget::event(event);
}
```

在执行时期想要知道所取得之 QEvent 类型，可以使用 QEvent 的 type()方法取得常数值，并与 QEvent::Type 作比对。
事件若顺利处理完毕，则要传回 true，表示这个事件被接受并处理，QApplication 可以继续事件队列中的下一个事件处理，若传回 false，则 QApplication 尝试寻找下一个可以处理事件的方法。您不用呼叫事件的 accept()或 ignore(),这也没有意义,accept()或 ignore()是用来在特定的事件处理者之间作沟通，而 event()的 true 或 false，是用来告知 QApplication 的 notify()方法是否处理下一事件，以 QWidget 的 event() 实作来说，它是根据事件的 isAccepted()来判断该传回 true 或 false：

```
bool QWidget::event(QEvent *event) {
```

```
    switch (e->type()) {
    case QEvent::KeyPress:
         keyPressEvent((QKeyEvent *)event);
        if (!((QKeyEvent *)event)->isAccepted())
            return false;
        break;
    case QEvent::KeyRelease:
        keyReleaseEvent((QKeyEvent *)event);
        if (!((QKeyEvent *)event)->isAccepted())
            return false;
        break;
        ...
    }
    return true;
}
```

另一个重新定义 event()的情况是自订 QCustomEvent 子类型时，您可以将之分派给其它函式或直接在 event()中处理，例如：

```
bool CustomWidget::event(QEvent *event) {
    if (event->type() == MyCustomEventType) {
        CustomEvent *myEvent = static_cast<CustomEvent *>(event);
        // 对自订事件的处理，或呼叫其它函式来处理事件
        return true;
    }
    return QWidget::event(event);
}
```

自订事件必须是 QCustomEvent 的子类别，您也可以直接实作 customEvent()方法来处理自订事件，详可参考 自订与传送事件。

3.3 事件过滤器

Qt 将事件封装为 QEvent 实例之后，会呼叫 QObject 的 event()方法并将 QEvent 实例传送给它，在某些情况下，您希望对象在执行 event ()处理事件之前，先对一些事件进行处理或过滤，然后再决定是否呼叫 event()方法，这个时候您就可以使用事件过滤器。

以 事件接受与否、event() 方法 中所谈及的，对 QWidget 按键事件的 Tab 键处理而言，如果您的图形接口中有很多的组件，每个图型组件都要如当中的范例重新定义 event()方法，显然是非常没有效率且没什么维护性的方法。

您可以自定义一个对象继承 QObject（或其子类别），重新定义它的 eventFilter()方法，例如您自定义了一个 FilterObject，您希望 Tab 键可以用来将焦点转移至下一个子组件：

```
bool FilterObject::eventFilter(QObject *object, QEvent *event)
{
    if(event->type() == QEvent::KeyPress) {
```

```
            QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
            if (keyEvent->key() == Qt::Key_Tab) {
                // 处理 Tab 键
                return true;
            }
        }
        return false;
}
```

eventFilter()的 object 参数表示事件发生的来源对象，eventFilter()若传回 false，则安装该事件过滤器的对象之 event()就会继续执行，若传回 true，则安装该事件过滤器的对象之 event()方法就不会被执行，由此进行事件的拦截处理。

要为指定的对象安装事件过滤器，可以使用对象的 installEventFilter()方法，例如：

```
QLineEdit *nameEdit = new QLineEdit;
QLineEdit *addressEdit = new QLineEdit;
...
FilterObject filter = new FilterObject;
...
nameEdit->installEventFilter(filter);
addressEdit->installEventFilter(filter);
....
```

您也可以将事件过滤器安装在 QApplication，在任何的事件发生后呼叫每个对象的 event()方法之前，会先经过事件过滤器，这给您更多控制应用程序事件的能力。

Qt 的事件循环与 sendEvent()方法会呼叫 QCoreApplication（QApplication 的父类别）的 notify()以分派事件，如果您想要完全控制 Qt 应用程序的事件，则可以重新定义 notify()方法。

到这边，可以看出 Qt 事件处理的五个层次：重新定义事件处理者、重新定义 event()方法、为个别对象安装事件过滤器、为 QApplication 安装事件过滤器，重新定义 QCoreApplication 的 notify()方法。

3.4 自订与传送事件

您可以自订事件类型，最简单的方式，是透过 QEvent::Type 指定事件类型的常数值，在建构 QCustomEvent 时作为建构自变量并透过 postEvent()传送事件，例如：

```
const QEvent::Type MyEvent = (QEvent::Type) 9393;
...
QApplication::postEvent(object, new QCustomEvent(MyEvent));
```

自订事件必须定义事件号码（Event number），自定义的事件号码必须大于 QEvent::Type 的列举值，通常 1024 以下的值是保留给 Qt 预先定义的事件类型来使用。object 是事件的接受者，使用 postEvent()方法时，事件必须以 new 的方式建立，在事件处理完毕之后，会

自动将事件 delete，postEvent()会将事件放至事件队列的尾端，然后立即返回。若要强迫 Qt 马上处理先前 postEvent()排到队列的事件，则可以使用 sendPostedEvents()。

您可以使用 sendEvent()方法，事件会立即送至接受者，sendEvent()方法的事件不会被 delete，所以通常建立在堆栈（Stack）区，例如：

```
CustomEvent event("Event Message");
QApplication::sendEvent(object, &event);
```

自订的事件类型必须是 QEvent 的子类别，通常继承 QCustomEvent 类别，建立自订事件类别可以获得更多的型态安全（Type safety）。

要处理自订事件，可以重新定义 customEvent()方法，例如：

```
void CustomWidget::customEvent(QCustomEvent *event) {
    CustomEvent *customEvent = static_cast<CustomEvent *>(event);
    ....
}
```

或是重新定义 event()方法，将自订事件分派给其它函式或直接在 event()中处理，例如：

```
bool CustomWidget::event(QEvent *event) {
    if (event->type() == MyCustomEventType) {
        CustomEvent *myEvent = static_cast<CustomEvent *>(event);
        // 对自订事件的处理，或呼叫其它函式来处理事件
        return true;
    }
    return QWidget::event(event);
}
```

第四章 基本版面配置

4.1QHBoxLayout 与 QVBoxLayout 版面配置

设计窗口程序的人都知道，在窗口程序中最麻烦也最难的就是版面配置，每次都为了组件的位置摆放在伤脑筋，像是

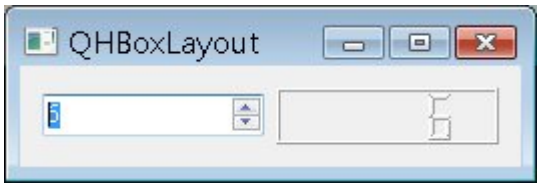
使用 Signal 与 Slot（使用拉杆改变 LCD 数字）的范例中，会使用 setGeometry()来设定 Widget 于 parent 中的 XY 位置与长宽，但这样在您窗口缩放时，当中的组件位置并不会适当的自我调整大小、位置（或像是字号自动调整之类的），以配合窗口缩放展现适当的观感。

所以窗口程序的解决方案都会提供一些现成的版面配置方式，让您可以不必自行配置组件位置，以下直接看例子，使用 QHBoxLayout 进行组件的版面配置，这可以让您以水平的方式来摆放组件：

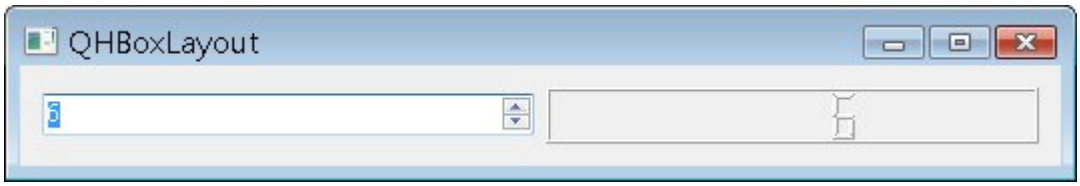
```
#include <QApplication>
#include <QWidget>
```

```
#include <QSpinBox>
#include <QLCDNumber>
#include <QHBoxLayout>

int main(int argc, char *argv[])
{
        QApplication app(argc, argv);

        QWidget *window = new QWidget;
        window->setWindowTitle("QHBoxLayout");
        window->resize(250, 50);
        QLCDNumber *lcd = new QLCDNumber;
        QSpinBox *spinBox = new QSpinBox;
        spinBox->setRange(0, 99);
        QObject::connect(spinBox, SIGNAL(valueChanged(int)),
                                lcd, SLOT(display(int)));
        QHBoxLayout *layout = new QHBoxLayout;
        layout->addWidget(spinBox);
        layout->addWidget(lcd);
        window->setLayout(layout);
        window->show();
        return app.exec();
}
```

这个程序中，没有自行设定组件的 parent/child 关系，也没有设定组件的大小、位置，而直接使用 QHBoxLayout 将组件加入，这会把 QHBoxLayout 及其管理的组件设成程序中 window 的子组件，并依 QHBoxLayout 版面配置策略自动水平配置组件，一个执行的的画面如下所示



组件会自动填满窗口，如果您拉动窗口，则当中的组件也会适当的变动大小：

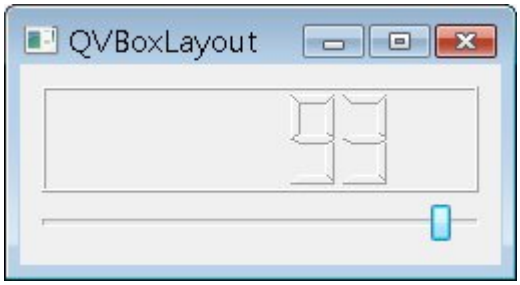


QHBoxLayout 中组件的加入顺序，就是水平配置由左至右显示的顺序，至于 QVBoxLayout 的使用方式则与 QHBoxLayout 类似，以使用 Signal 与 Slot（使用拉杆改变 LCD 数字）中的例子来说，可以改用 QVBoxLayout 来配置组件位置：

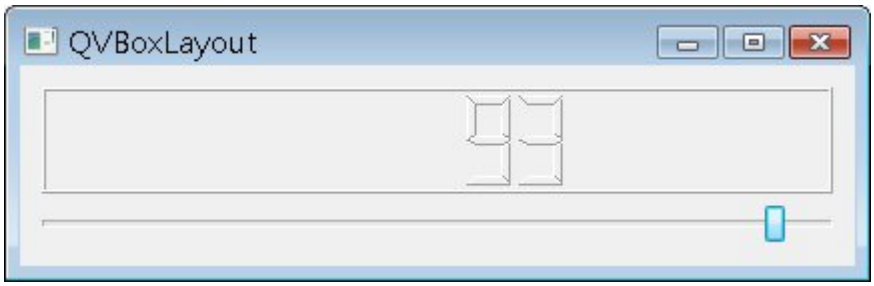
```
#include <QApplication>
#include <QWidget>
#include <QSlider>
#include <QLCDNumber>
```

```
#include <QVBoxLayout>

int main(int argc, char *argv[])
{
        QApplication app(argc, argv);

        QWidget *window = new QWidget;
        window->setWindowTitle("QVBoxLayout");
        window->resize(240, 100);
        QLCDNumber *lcd = new QLCDNumber;
        QSlider *slider = new QSlider(Qt::Horizontal);
        slider->setRange(0, 99);
        slider->setValue(0);
        QObject::connect(slider, SIGNAL(valueChanged(int)),
                                 lcd, SLOT(display(int)));
        QVBoxLayout *layout = new QVBoxLayout(window);
        layout->addWidget(lcd);
        layout->addWidget(slider);
        window->show();
        return app.exec();
}
```

程序中可以看到，在建立版面配置对象时，也可以直接指定要实施版面配置的对象。执行的画面如下所示：



拉动时的画面如下所示：



4.2QGridLayout 版面配置

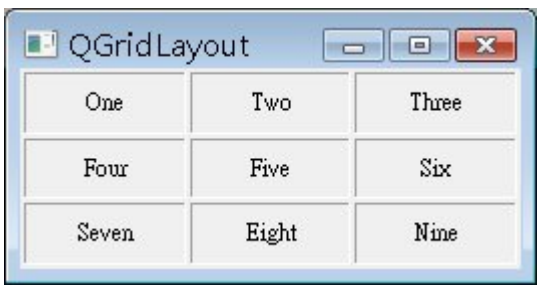
QGridLayout 版面配置会如同棋盘般排列 Widget，先来看个简单的例子：

```
#include <QApplication>
#include <QWidget>
```

```
#include <QGridLayout>
#include <QLabel>

int main(int argc, char *argv[])
{
        QApplication app(argc, argv);

        const QString texts[] = {"One", "Two", "Three",
                                 "Four", "Five", "Six",
                                 "Seven", "Eight", "Nine"};
        QWidget *window = new QWidget;
        window->setWindowTitle("QGridLayout");
        window->resize(250, 100);
        QGridLayout *layout = new QGridLayout;
        layout->setSpacing(2);
        layout->setMargin(2);

        for(int i = 0, k = 0; i < 3; i++, k = k + 3) {
              for(int j = 0; j < 3; j++) {
                    QLabel *label = new QLabel(texts[k + j]);
                    label->setFrameStyle(QFrame::Panel + QFrame::Sunken);
                    label->setMinimumSize(55, 0);
                    label->setAlignment(Qt::AlignCenter);
                    layout->addWidget(label, i, j);
              }
        }
        window->setLayout(layout);
        window->show();
        return app.exec();
}
```

在版面配置的程序代码部份，setSpacing()与 setMargin()设定每一个 Grid 的空间与彼此之间的边界，再来利用循环将 QLabel 加入 QGridLayout，程序中设定了 QLabel 的最小尺寸、显示位向与样式，注意在使用 addWidget()方法时，可以指定您要将组件加入至哪一列(Row)哪一行（Column)。

程序执行时的参考画面如下：

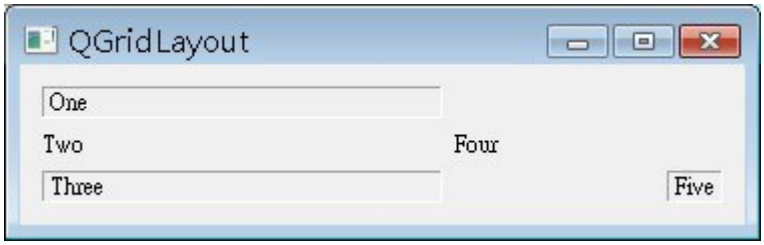


QGridLayout 中不一定要每个格子依序填满，只要指定想填入的位置，没有指定的部份会自动空下，以下的程序是个简单的示范：

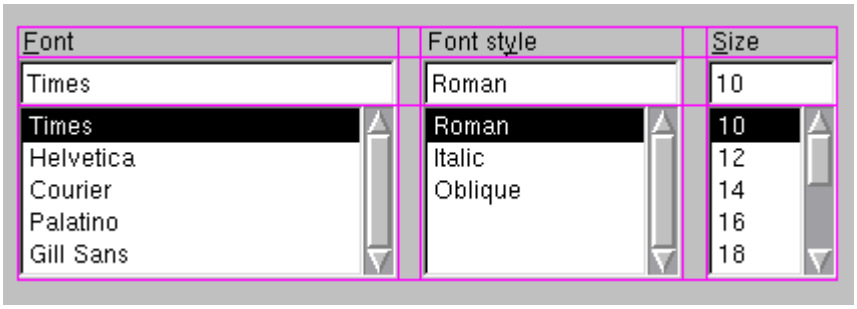
```
#include <QApplication>
#include <QWidget>
#include <QGridLayout>
#include <QLabel>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;

    QGridLayout *layout = new QGridLayout;
    layout->setColumnMinimumWidth(0, 200);
    layout->setColumnMinimumWidth(1, 100);

    QLabel *label1 = new QLabel("One");
    label1->setFrameStyle(QFrame::Panel + QFrame::Sunken);
    QLabel *label3 = new QLabel("Three");
    label3->setFrameStyle(QFrame::Panel + QFrame::Sunken);
    QLabel *label5 = new QLabel("Five");
    label5->setFrameStyle(QFrame::Panel + QFrame::Sunken);

    layout->addWidget(label1, 0, 0);
    layout->addWidget(new QLabel("Two"), 1, 0);
    layout->addWidget(label3, 2, 0);
    layout->addWidget(new QLabel("Four"), 1, 1);
    layout->addWidget(label5, 2, 2);
    window->setLayout(layout);
    window->show();
    return app.exec();
}
```

您也可以使用 setColumnMinimumWidth() 来设定指定行的最小宽度，或使用 setRowMinimumHeight()来设定指定列的最小高度，行列的索引都是由 0 开始，与二维数组的指定方式相同。

程序的执行画面如下所示：



利用指定行列置放组件的特性，您就可以进行如下的画面设计：



（图片取自 Qt 官方的 QGridLayout 类别说明）

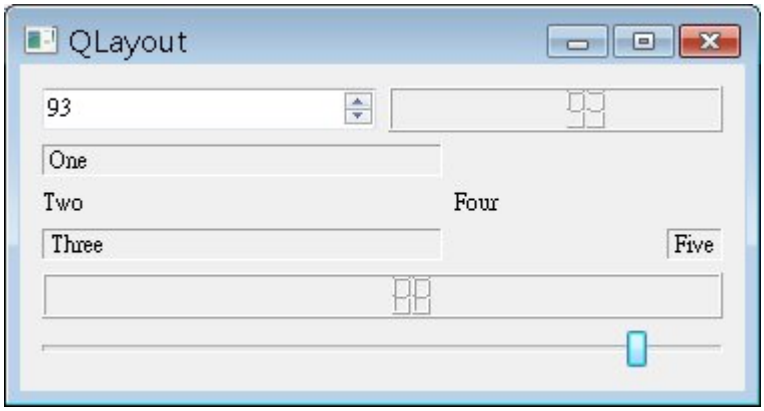
4.3 较复杂的版面配置

版面配置管理员有个 addLayout()方法，可以让您将另一个版面配置实例加入某个版面配置之中，利用版面配置管理的组合，您可以制作出更复杂的版面配置，以下组合 QHBoxLayout 与 QVBoxLayout 版面配置 及 QGridLayout 版面配置 作为实际示范：

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QSpinBox>
#include <QSlider>
#include <QLCDNumber>
#include <QHBoxLayout>
#include <QVBoxLayout>
#include <QGridLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("QLayout");

    // 组合版面配置
    QHBoxLayout *hBoxLayout = new QHBoxLayout;

    QGridLayout *gridLayout = new QGridLayout;
    gridLayout->setColumnMinimumWidth(0, 200);
    gridLayout->setColumnMinimumWidth(1, 100);

    QVBoxLayout *vBoxLayout = new QVBoxLayout;
    vBoxLayout->addLayout(hBoxLayout);
    vBoxLayout->addLayout(gridLayout);

    window->setLayout(vBoxLayout);
```

```
    // 使用 QHBoxLayout 配置
    QLCDNumber *lcd1 = new QLCDNumber;
    QSpinBox *spinBox = new QSpinBox;
    spinBox->setRange(0, 99);
    QObject::connect(spinBox, SIGNAL(valueChanged(int)),
                             lcd1, SLOT(display(int)));

    hBoxLayout->addWidget(spinBox);
    hBoxLayout->addWidget(lcd1);

    // 使用 QGridLayout 配置
    QLabel *label1 = new QLabel("One");
    label1->setFrameStyle(QFrame::Panel + QFrame::Sunken);
    QLabel *label3 = new QLabel("Three");
    label3->setFrameStyle(QFrame::Panel + QFrame::Sunken);
    QLabel *label5 = new QLabel("Five");
    label5->setFrameStyle(QFrame::Panel + QFrame::Sunken);

    gridLayout->addWidget(label1, 0, 0);
    gridLayout->addWidget(new QLabel("Two"), 1, 0);
    gridLayout->addWidget(label3, 2, 0);
    gridLayout->addWidget(new QLabel("Four"), 1, 1);
    gridLayout->addWidget(label5, 2, 2);

    // 使用 QVBoxLayout 配置
    QLCDNumber *lcd2 = new QLCDNumber;
    QSlider *slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99);
    slider->setValue(0);
    QObject::connect(slider, SIGNAL(valueChanged(int)),
                             lcd2, SLOT(display(int)));

    vBoxLayout->addWidget(lcd2);
    vBoxLayout->addWidget(slider);

    // 使用 QVBoxLayout 配置，当中包括 QHBoxLayout 与 QGridLayout 配置好的组件
    window->show();

    return app.exec();
}
```

组合完的版面配置结果如下：



在上面的范例中，程序的主流程中充斥着组件建立与版面配置的流程，建议可以利用自订组件的方式，让整个程序的流程更清楚一些，例如如下撰写程序：

ComboLCD.h

```
#ifndef COMBOLCD_H
#define COMBOLCD_H

#include <QWidget>

class ComboLCD : public QWidget
{
public:
    ComboLCD(QWidget *parent = 0);
};
#endif
```

QComboLCD.cpp

```
#include <QWidget>
#include <QLCDNumber>
#include <QSpinBox>
#include <QHBoxLayout>
#include "ComboLCD.h"

ComboLCD::ComboLCD(QWidget *parent) : QWidget(parent)
{
    QLCDNumber *lcd = new QLCDNumber;
    QSpinBox *spinBox = new QSpinBox;
    spinBox->setRange(0, 99);
    QObject::connect(spinBox, SIGNAL(valueChanged(int)),
                     lcd, SLOT(display(int)));
    QHBoxLayout *hBoxLayout = new QHBoxLayout;
    hBoxLayout->addWidget(spinBox);
    hBoxLayout->addWidget(lcd);
    this->setLayout(hBoxLayout);
```

```
}
```

GridLabel.h

```
#ifndef GRIDLABEL_H
#define GRIDLABEL_H

#include <QWidget>

class QGridLabel : public QWidget
{
public:
    QGridLabel(QWidget *parent = 0);
};

#endif
```

GridLabel.cpp

```
#include <QWidget>
#include <QLabel>
#include <QGridLayout>
#include "GridLabel.h"

GridLabel::GridLabel(QWidget *parent) : QWidget(parent) {
    QGridLayout *gridLayout = new QGridLayout;
    gridLayout->setColumnMinimumWidth(0, 200);
    gridLayout->setColumnMinimumWidth(1, 100);
    QLabel *label = new QLabel("One");
    label->setFrameStyle(QFrame::Panel + QFrame::Sunken);
    gridLayout->addWidget(label, 0, 0);
    gridLayout->addWidget(new QLabel("Two"), 1, 0);
    label = new QLabel("Three");
    label->setFrameStyle(QFrame::Panel + QFrame::Sunken);
    gridLayout->addWidget(label, 2, 0);
    gridLayout->addWidget(new QLabel("Four"), 1, 1);
    label = new QLabel("Five");
    label->setFrameStyle(QFrame::Panel + QFrame::Sunken);
    gridLayout->addWidget(label, 2, 2);
    this->setLayout(gridLayout);
}
```

main.cpp

```
#include <QApplication>
#include <QWidget>
#include <QLCDNumber>
#include <QSlider>
```

```
#include <QVBoxLayout>
#include "ComboLCD.h"
#include "GridLabel.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("QLayout");

    ComboLCD *comboLCD = new ComboLCD;
    GridLabel *gridLabel = new GridLabel;
    QLCDNumber *lcd = new QLCDNumber;
    QSlider *slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99);
    slider->setValue(0);
    QObject::connect(slider, SIGNAL(valueChanged(int)),
                     lcd, SLOT(display(int)));

    QVBoxLayout *vBoxLayout = new QVBoxLayout;
    vBoxLayout->addWidget(comboLCD);
    vBoxLayout->addWidget(gridLabel);
    vBoxLayout->addWidget(lcd);
    vBoxLayout->addWidget(slider);

    window->setLayout(vBoxLayout);
    window->show();
    return app.exec();
}
```

把一些组件配置以自订组件的方式包装起来，程序的主流程变得较为清晰，这个程序的执行画面跟上图是相同的。

4.4 自订版面配置管理员（Layout Manager）

您可以继承 QLayout 来自订版面配置管理员，您要有一个储存 QLayoutItem 的容器，例如 QList，每个 QLayoutItem 代表一个被加入版面配置的 Widget 组件。并且您要重新定义 QLayout 的几个方法：

(1)addItem()
定义组件如何加入 Layout 之中，通常是定义组件如何加入容器（例如 QList）。
(2)count()
加入 Layout 的组件个数。
(3)setGeometry()

定义组件实际的位置与大小配置方式。
(4)sizeHint()
设定 Layout 的偏好尺寸（preferred size)。
(5)itemAt()
定义如何根据索引取得 QLayoutItem。
(6)takeAt()
定义如何根据索引取得并从容器中移除 QLayoutItem。

除了以上几个关于版面配置的方法必须实作之外,如果不想让您的整个版面缩小至会覆盖当中的组件的话，建议也实作 minimumSize()，定义您的整个版面配置最小尺寸。

也可以看是否重新定义 hasHeightForWidth()，传回 true 或 false，表示是否根据组件的宽度来设定版面配置的高度，如果 hasHeightForWidth() 传回 true 就会呼叫 heightForWidth()，您可以重新定义它，这让整个版面有足够的高度来显示所有的组件。

另外，重新定义 expandingDirections()，如果必要的话，决定是否可使用比 sizeHint() 更大的额外空间，预设是 Qt:: Vertical | Qt::Horizontal，表示必要的话，可以往水平或垂直方向自动扩展 Layout 空间，或传回 0 表示不扩展。

关于自订 Layout 的程序代码示范，可以参考 Qt 在线文件的范例 Flow Layout 或 Border Layout，当中有完整的程序代码示范。
当您自订 Widget 时，如果使用版面管理员来配置 child 组件，则版面管理员会自动帮您配置 Widget 相关的版面属性，如果您没有使用版面管理员，而是自订 child 组件的版面配置，则最好重新定义 QWidget 的相关方法以定义其在 parent 组件中的版面配置,像是 sizeHint() 以设定自订 Widgt 的偏好尺寸，重新定义 QWidget 的 minimunSize()以设定 Widget 的最小尺寸等。
定义版面配置，基本上不是件易事，可以的话，尽量使用 Qt 现有的版面配置管理，如果有需要自订版面配置，建议参考一下 Qt 在线文件的 Layout Management。

第五章 常用图型组件

5.1 按钮选项

5.11QPushButton

QPushButton 继承自 QAbstractButton（再继承自 QWidget)，主要提供窗口的按钮外观及行为，在 使用 Signal 与 Slot（使用按钮关闭窗口） 已经看过 QPushButton 的基本运用，接下来的这个简单范例，将示范几个 QPushButton 的外观设定：

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QMenu>
#include <QVBoxLayout>

int main(int argc, char *argv[])
```

```
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("QPushButton");
    window->resize(250, 50);

    QPushButton *btn1 = new QPushButton("Normal Button");

    QPushButton *btn2 = new QPushButton("Toggle Button");
    btn2->setCheckable(TRUE);
    btn2->setChecked(TRUE);

    QPushButton *btn3 = new QPushButton("Flat Button");
    btn3->setFlat(TRUE);

    QPushButton *btn4 = new QPushButton( "Popup Button");
    QMenu *menu = new QMenu;
    menu->addAction("Open Item");
    menu->addAction("Save Item");
    menu->addMenu("More Item");
    menu->addSeparator();
    menu->addAction("Close Item");
    btn4->setMenu(menu);

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(btn1);
    layout->addWidget(btn2);
    layout->addWidget(btn3);
    layout->addWidget(btn4);
    layout->addStretch(1);

    window->setLayout(layout);
    window->show();

    return app.exec();
}
```

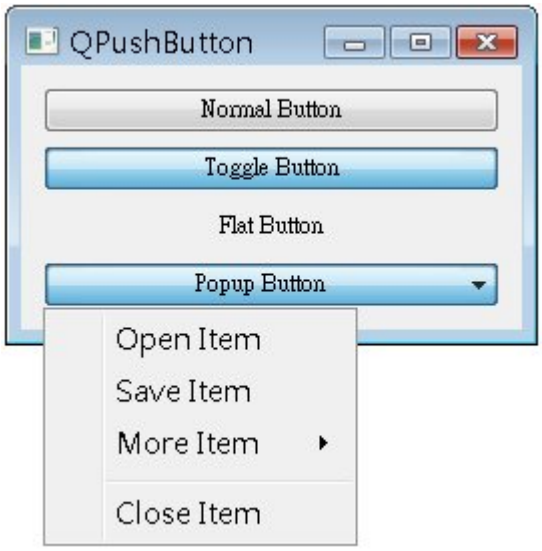
btn1 为最基本的 QPushButton 建构，btn2 使用 setCheckable()方法设定按钮为可停驻的 Toggle Button，并使用 setChecked()设定其预设为停驻状态，btn3 使用 setFlat()设定按钮为没有浮起框线的平坦按钮，只有在按下时才会显示 QPushButton 被按下的外观。

QPushButton 可以设定按下时出现下拉选单，设定选单时使用的是 QMenu，每个选单下的选项为 QAction 的实例，使用 QMenu 的 addAction()将 QAction 的实例加入可以成为一个选项，若要加入子选单，则使用 addMenu()加入，addAction()与 addMenu()都可以直接指定字符串，

将自动产生 QAction 与 QMenu。addSeparator()则在选单中加入分隔线。

要设定 QPushButton 按下后出现选单，则使用其 setMenu()方法。程序中使用 QVBoxLayout 的 addStretch()加入了拉伸系数（stretch factor），每个 QWidget 在使用 addWidget()加入版面配置时，也可以指定一个拉伸系数，不指定的话拉伸系数预设为 0，拉伸系数是当您在拉伸版面时，每个组件消耗多余空间的相对比例，在这个范例中，由于 QWidget 加入时拉伸系数预设为 0，所以最后使用 addStretch(1)时表示剩余的空间将全部作为空白的空间。

下图为程序示范画面之一，Toggle Button 为停驻状态，Flat Button 没有按下时是平坦状态，而按下第四个按钮出现了选单。



下图为按下 Flat Button 时的画面示范：

5.12QCheckBox 与 QRadioButton

复选框（CheckBox）是可以进行选项复选的组件，单选钮（RadioButton）是只能进行选项单选的组件，在 Qt 中分别使用 QCheckBox 与 QRadioButton 来负责，QRadioButton 必须使用 QGroupBox 来加以群组，来表示哪些选项为一个群组，彼此互斥，同时间只能选择一个。

以下是个简单的组件配置示范：

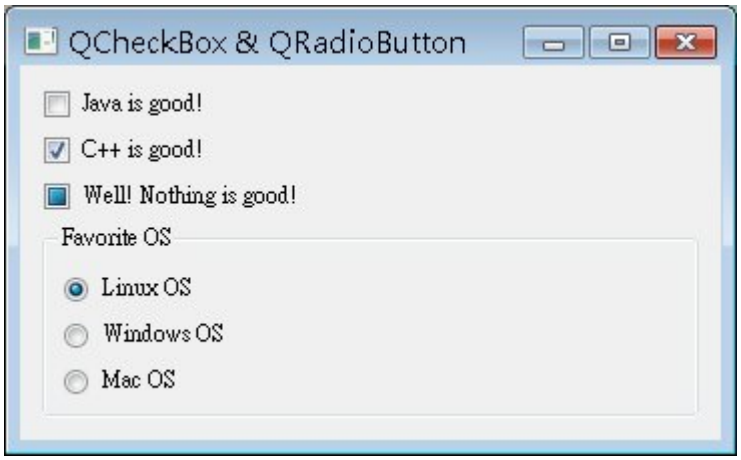
```
#include <QApplication>
#include <QWidget>
#include <QButtonGroup>
```

```
#include <QCheckBox>
#include <QRadioButton>
#include <QGroupBox>
#include <QVBoxLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("QCheckBox & QRadioButton");
    window->resize(350, 100);

    QVBoxLayout *windowLayout = new QVBoxLayout;

    // 复选框
    QCheckBox *check1 = new QCheckBox;
    check1->setChecked(true);
    check1->setText("Java is good!");
    QCheckBox *check2 = new QCheckBox("C++ is good!");
    QCheckBox *check3 = new QCheckBox("Well! Nothing is good!");
    check3->setTristate(true);

    windowLayout->addWidget(check1);
    windowLayout->addWidget(check2);
    windowLayout->addWidget(check3);

    // 单选钮
    QGroupBox *box = new QGroupBox("Favorite OS");
    QRadioButton *radio1 = new QRadioButton("Linux OS");
    QRadioButton *radio2 = new QRadioButton("Windows OS");
    QRadioButton *radio3 = new QRadioButton("Mac OS");
    QVBoxLayout *radioLayout = new QVBoxLayout;
    radioLayout->addWidget(radio1);
    radioLayout->addWidget(radio2);
    radioLayout->addWidget(radio3);
    // 三个单选钮为一个群组
    box->setLayout(radioLayout);

    windowLayout->addWidget(box);

    window->setLayout(windowLayout);
    window->show();
```

```
    return app.exec();
}
```

QCheckBox 除了预设的核取与非核取状态之外，还可以第三个状态，这可以使用 setTristate() 方法来设定，而单选钮的配置部份，首先将单选钮组件加入 QVBoxLayout 中作版面配置，然后设定其为 QGroupBox 的版面管理，如此一来， QVBoxLayout 中的三个单选钮就是在同一个群组之中。

下图为执行的参考画面，其中第三个复选框为第三个选中状态，预设的核取与非核取状态可以使用 isChecked()来得知，第三个选中状态可以使用 isTristate()来得知：



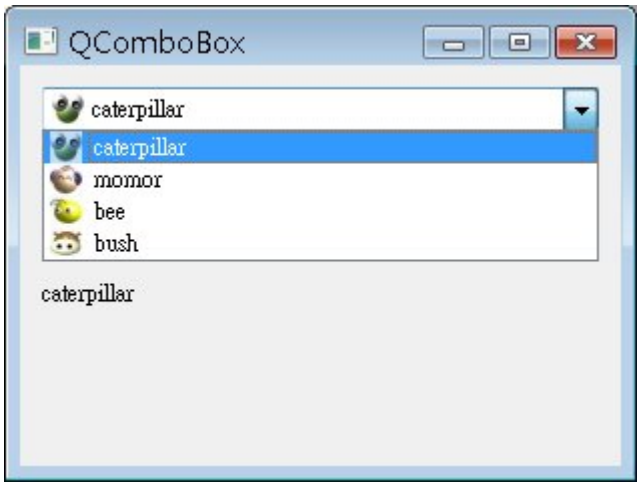
5.13QComboBox

QComboxBox 可以建立下拉选单，以供使用者选取项目，以下直接看个简单的示范，程序中包括下拉选单，选择选项之后会改变 QLabel 的文字内容：

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QComboBox>
#include <QVBoxLayout>
#include <QIcon>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("QComboBox");
    window->resize(300, 200);

    QComboBox *combo = new QComboBox;
    combo->setEditable(true);
    combo->insertItem(0, QIcon( "caterpillar_head.jpg" ), "caterpillar");
    combo->insertItem(1, QIcon( "momor_head.jpg" ), "momor");
    combo->insertItem(2, QIcon( "bush_head.jpg" ), "bush");
```

```
    combo->insertItem(3, QIcon( "bee_head.jpg" ), "bee");

    QLabel *label = new QLabel("QComboBox");

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(combo);
    layout->addWidget(label);

    QObject::connect(combo, SIGNAL(activated(const QString &)),
                             label, SLOT(setText(const QString &)));

    window->setLayout(layout);
    window->show();

    return app.exec();
}
```

QComboBox 的 setEditable()方法可设定下拉选单的选项是否可编辑，使用 insertItem()插入选项时，可以使用 QIcon 设定图示，当您选择下拉选单的某个项目时，会发出 activated()的 Signal，QString 的部份即为选项文字，这边将之连接至 QLabel 的 setText()，以改变 QLabel 的文字，一个程序执行的画面如下：



5.2 对话框

5.21QInputDialog 与 QMessageBox

在程序中常出现一些简单的对话或消息框，在这边介绍的 QInputDialog 与 QMessageBox 算是最常见的类型，它们的使用非常简单，首先看个简单的程序代码片段：

```
bool isOK;
QString text = QInputDialog::getText(parent, "Input Dialog",
                        "Please input your comment",
                          QLineEdit::Normal, "your comment", &isOK);
if(isOK) {
```

```
    QMessageBox::information(parent, "Information",
                "Your comment is: <b>" + text + "</b>",
                QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);
}
```

在这个程序片中，QInputDialog 使用 getText()方法来取得使用者输入字符串，然后使用 QMessageBox 来显示输入的字符串；QInputDialog 若取得输入，会把 isOK 中设定为 true，由此可判断使用者是否有输入，QMessageBox 可以使用一些基本的 html 语法来设定文字的显示，消息框将显示 Yes 与 No 两个按钮，在组合时的列举值是 StandButton 列举（enum）值，可以参考在线文件的表格 进行对照。最后一个参数设定预设按钮是 Yes，至于每个字符串的设定各是何作用，直接看执行结果比较清楚，以下显示的是输入方块：

以下是按下 OK 后显示消息框：

以下再介绍一些对话与消息框的样式。设定默认值为 0，下界为 0，上界为 100，递增值为 1 的整数输入对话框 ：

```
int input = QInputDialog::getInteger(parent, "Input Dialog",
                "Enter an integer", 0, 0, 100, 1, &isOK);
```

设定默认值为 0，下界为 0，上界为 10，小数字数为 2 位的浮点数输入对话框：

```
double input = QInputDialog::getDouble(parent, "Input Dialog",
                "Enter a double", 0.0, 0.0, 10.0, 2, &isOK);
```

一个警示消息框：

```
QMessageBox::warning(parent, "Warning",
                     "Oh! <b>Big Warning!</b>",
                     QMessageBox::Yes, QMessageBox::Yes);
```

一个禁止消息框：

```
QMessageBox::critical(parent, "Critical",
                      "Oh! <b>Red Critical!</b>",
                      QMessageBox::Ok, QMessageBox::Ok);
```

一个「关于..」消息框，常用于程序简介：

```
QMessageBox::about(parent, "About",
                      "Qt4 Gossip: <b>http://caterpillar.onlyfun.net</b>");
```

消息框的传回值是根据您按下的按钮，由左而右依序传回 StandButton 列举（enum）值，可以参考在线文件的表格 进行对照。

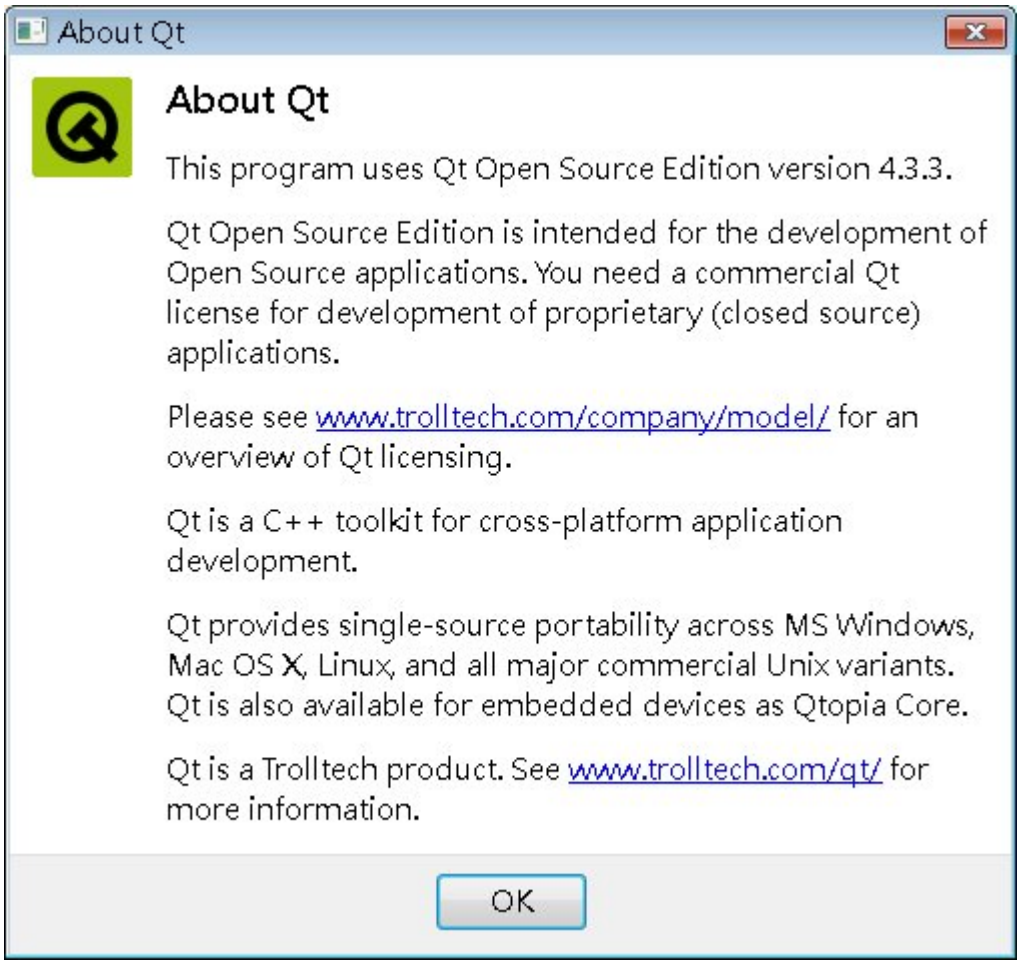
您也可以自订消息框的图标、按钮等选项，以下是一个简单的示范：

```
QMessageBox message(QMessageBox::NoIcon, "Gossip",
"Show Qt?", QMessageBox::Yes | QMessageBox::No , parent);
```

```
message.setIconPixmap(QPixmap("caterpillar.png"));
if(message.exec() == QMessageBox::Yes) {
    QMessageBox::aboutQt(parent, "About Qt");
}
```

这个程序设定前两个按钮分别为 Yes 与 No 显示与功能，程序中先不设定 ICON，而使用 setIconPixmap()设定自制的图文件为图示，QPixmap 支持的图档包括 BMP、GIF、JPG、PNG 等格式（可在 QPixmap 文件中查询）：

如果按下的是 Yes 钮，则显示 Qt 版权等相关讯息，这是 aboutQt 所作的事，以以下为执行的画面：



5.22QColorDialog 与 QFontDialog

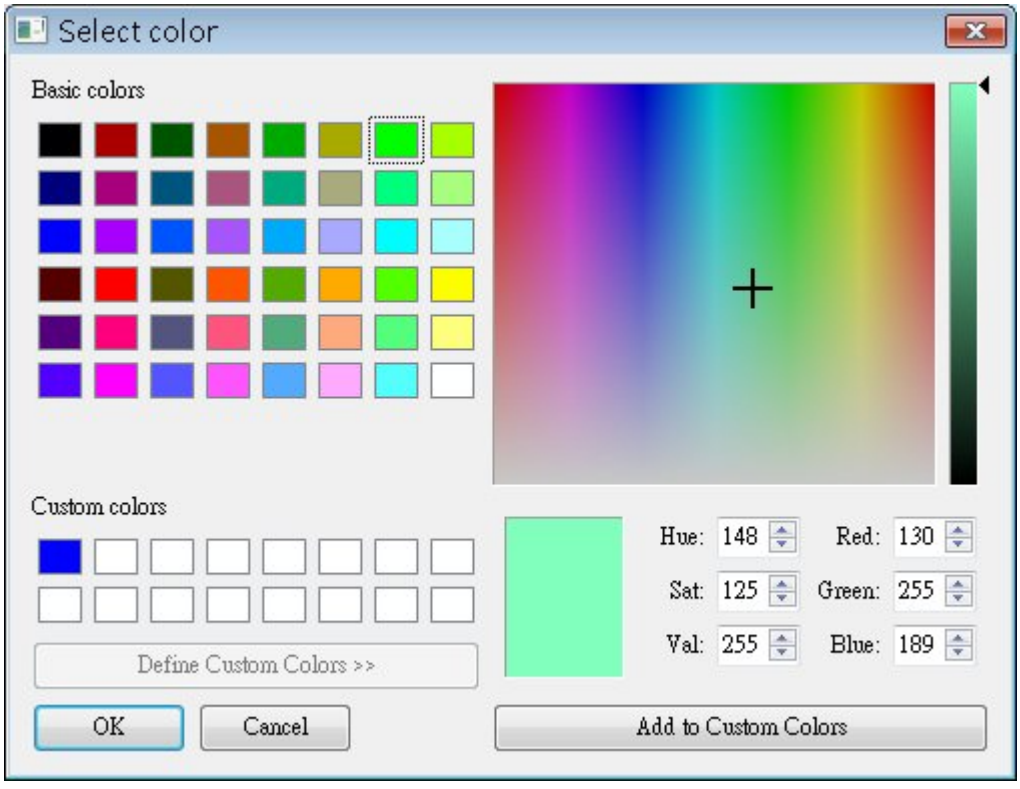
QColorDialog 类别可以显示一个颜色选取的对话框，使用者选取颜色之后，会传回一个 QColor 对象，可以藉由这个 QColor 对象来取得所选取颜色的 RGB 值。

下面这个程序片段是 QColorDialog 的简单的示范，在选取颜色之后，使用 QMessageBox 来显示选取颜色的 RGB 值：

```
QColorDialog::setCustomColor(0, QRgb(0x0000FF));
QColor color = QColorDialog::getColor(QColor(0, 255, 0));
QString text;

if(color.isValid()) {
      text.sprintf("R: %d G: %d B: %d",
                        color.red(), color.green(), color.blue());
      QMessageBox::information(0, "Selected color",
            text, QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);
}
text.sprintf("Available custom colors: %d", QColorDialog::customCount());
QMessageBox::information(0, "Get Selected Color",
            text, QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);
```

setCustomColor()方法设定颜色选取方块中，自订色彩中出现时预设的选取颜色；getColor()这个方法除了取回设定的颜色之外，也会将颜色选取方块的值预设为指定的值；customCount()方法可以取回自订色彩的最大个数，预设是 16 个待定义色彩。

在上例中也可以看到，QString 可以使用类似 C 语言的 printf()函式用法，也就是使用 sprintf()来设定文字格式。下图为颜色选取方块执行的画面：



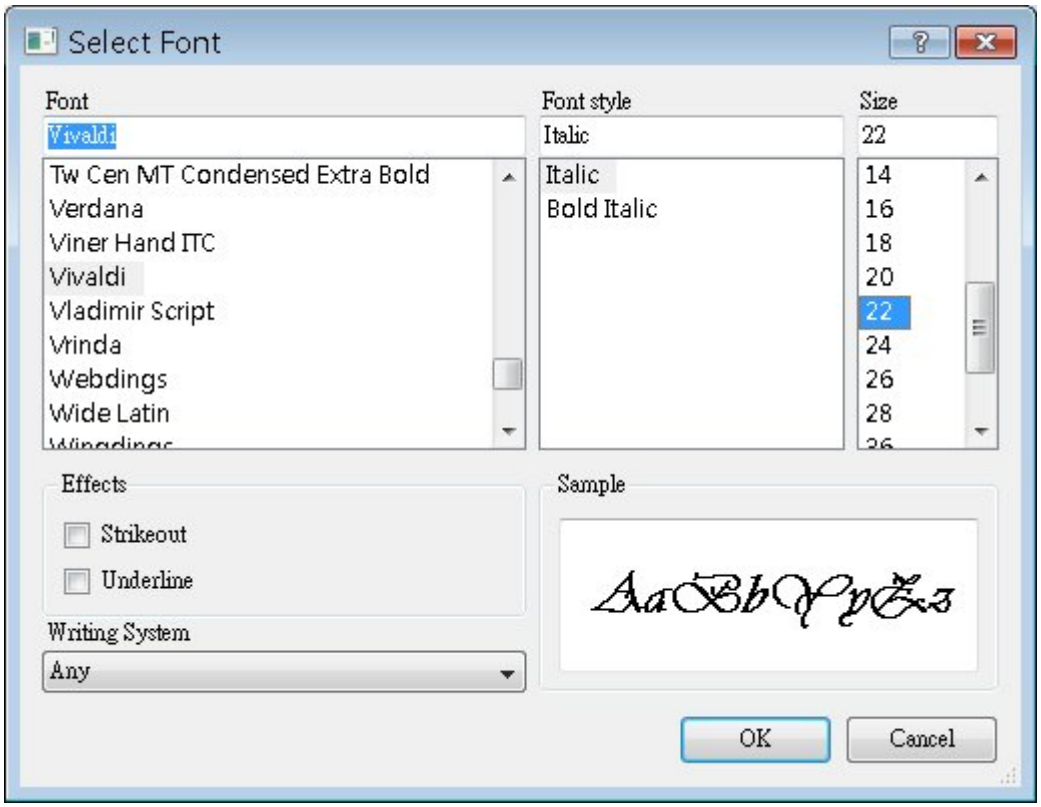
下图为显示的消息框，可以看到 QString 的文字已使用 sprintf()进行格式化：

使用 QFontDialog 会出现一个字型选择的对话框，它可以让使用者选择所要的字型样式，然后传回 QFont 对象，当中包括了所选择的样式信息，可以直接使用这个对象来设定文字的字型。

下面这个程序可以让使用者使用字型选择的对话框，设定窗口中的 QLabel 对象之字型：

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QFontDialog>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    bool isOk;
    QLabel *label = new QLabel("<center>HelloWorld!</center>");
    label->setWindowTitle("FontDialog");
    label->resize(250, 100);
    label->setFont(QFont( "Times", 18, QFont::Bold ));
    label->show();
    QFont font = QFontDialog::getFont(&isOk, QFont("Times", 18, QFont::Bold), label);
    if(isOk) {
        label->setFont(font);
    }
    return app.exec();
}
```

程序相当的简单，QLabel 对象的 setFont()方法使用 QFont 对象来设定显示的字型，以下是 QFontDialog 的执行画面：



下图为设定所选定字型的 QLabel 画面：

5.23QFileDialog

在窗口程序中开启档案或另存盘案的动作，会使用档案对话框来让使用者方便的选取或决定文件名称，在 Qt 中，这则是由 QFileDialog 类别负责，在这边将示范一些简单的使用方式。

QFileDialog 最简单的使用方法就是利用它所提供的静态方法，例如下面这个程序会显示一个开启档案的对话框，选取档案后显示所选取的档案路径与名称：

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QFileDialog>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
```

```
    QLabel *label = new QLabel("<center>FileDialog</center>");
    label->setWindowTitle("FileDialog");
    label->resize(500, 100);

    label->setFont(QFont( "Times", 18, QFont::Bold ));
    label->show();
    QString fileName = QFileDialog::getOpenFileName(label, "Open Image",
                        "C:\\", "Image Files (*.png *.xpm *.jpg)");
    if(fileName != NULL) {
        label->setText("<center>" + fileName + "</center>");
    }
    return app.exec();
}
```

getOpenFileName()方法会显示一个档案开启的对话框，如果要显示储存档案的对话框，就使用 getSaveFileName()方法，在参数的指定上，"C:\\"指定开启时的工作目录，而"Image files (*.png *.xpm *.jpg)"指定开启档案时的扩展名过滤，如果还要使用其它的过滤方式，可以使用分号，如下所示：

"Image files (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"

下图为 Qt 的档案对话框执行画面：

也可以新增对象的方式来使用 QFileDialog 类别，这可以设定更多的选项，下面这个程序示范一些常用的方法：

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QFileDialog>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QLabel *label = new QLabel("<center>FileDialog</center>");
    label->setWindowTitle("FileDialog");
    label->resize(500, 100);

    label->setFont(QFont( "Times", 18, QFont::Bold ));
    label->show();

    QFileDialog* dialog = new QFileDialog(label);

    dialog->setDirectory("C:\\");  // 设定开始目录
    dialog->setFileMode(QFileDialog::ExistingFile);  // 可选取已存在的档案
    dialog->setFilter("Image files (*.png *.xpm *.jpg)");  // 扩展名过滤
    dialog->setViewMode(QFileDialog::Detail);  // 显示详细信息

    if (dialog->exec() == QDialog::Accepted) {
        QStringList fileNames = dialog->selectedFiles();
        QStringListIterator iterator(fileNames);
        while(iterator.hasNext()) {
            label->setText("<center>" + iterator.next() + "</center>");
        }
    }
    return app.exec();
}
```

setDirectory()设定对话框第一个显示的目录，setFileMode()设定使用者可以选择的档案类型，ExistingFile 表示可选取已存在的档案，AnyFile 则表示您可以选择任何档案，即使档案不存在（像是在另存新档时指定一个新的档名时使用），Directory 表示可以选取目录，DirectoryOnly 表示只可以选取目录， ExistingFiles 表示可以进行档案多选。

setFilter()即使设定档名过滤，setViewMode()用来设定检视的细节，Details 显示详细信息，而 List 则只显示文件名与图标，selectedFiles()会传回所选取的档案清单，以 QStringList 传回（继承自 QList），程序中使用的是 Java 风格的迭代方式，您也可以使用索引风格：

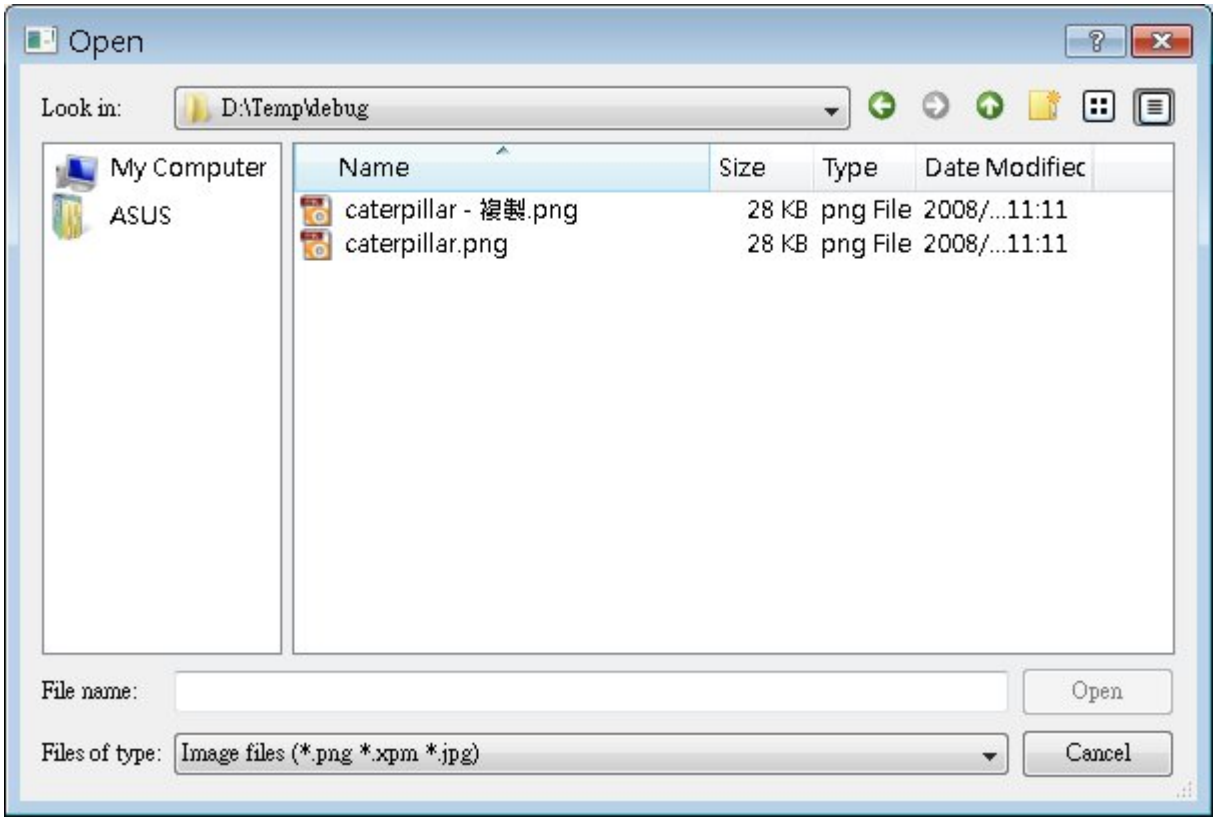
```
QStringList fileNames = dialog->selectedFiles();
for (int i = 0; i < fileNames.size(); i++) {
```

```
    label->setText("<center>" + fileNames.at(i) + "</center>");
}
```

或 STL 风格的迭代器：

```
QStringList fileNames = dialog->selectedFiles();
QStringList::const_iterator iterator;
for (iterator = fileNames.constBegin(); iterator != fileNames.constEnd();
     iterator++) {
    label->setText("<center>" + (*iterator) + "</center>");
}
```

下图为程序执行时的一个画面：



5.24 自订对话框（Dialog）

QDialog 提供了一些基础，让您可以继承 QDialog 来自订对话框。

QDialog 的 show()方法可以显示对话框，如果您使用 setModal()并设定为 true，则 show()会以独占(Modal)模式显示对话，使用者必须响应对话框，才可以继续窗口操作，如果 setModal()为 false，则 show()显示的对话框为非独占（Modelless）模式。QDialog 的 exec()方法则会忽略 setModal()的设定，直接将对话框以独占模式显示。

使用 exec()方法独占模式显示对话框时，程序流程会停止在该处，当对话框结束，exec()方法会传回执行结果，您可以使用 accept()方法关闭对话框，exec()会执行完成并传回 QDialog::Accepted，或使用 reject()方法关闭对话框，exec()会执行完成并传回 QDialog::Rejected，或者是使用 done()方法并指定整数值，关闭对话框之后 exec()会传回指定的整数值。

无论是 exec()、accept()、reject()或 done()，它们都是 QDialog 的 Slot，所以您可以在继承 QDialog 之后，自定义一些组件，将组件的 Signal 连接至 QDialog 的这些 Slot，例如自定义 QPushButton 为 OK 按钮，将其 clicked()的 Signal 连接至 QDialog 的 accept() Slot，如此使用者按下 OK 按钮时，exec()方法就会传回 QDialog::Accepted。

当您呼叫 QDialog 的 accept ()、reject()时，QDialog 会分别发出 Signal accepted()、rejected()，而呼叫 done() 并设定其整数为 QDialog:: Accepted 时，会发出 Signal accepted()，呼叫 done()并设定其整数为 QDialog::Rejected 时，会发出 Signal rejected()。

无论是呼叫 accept()、reject()或 done()时， finished()的 Signal 都会被发出，并带有所设定的结果值。
您也可以使用 setResult()设定独占对话框的传回值，使用 result()取得独占对话框的传回值。

5.3 文字字段

5.31QLineEdit

QLineEdit 提供一个文字输入字段，可以输入文字或数字，我们可以对输入作验证，或是设定为一般显示、密码显示等等，以下的程序是个简单的设定示范：

```
#include <QApplication>
#include <QLabel>
#include <QLineEdit>
#include <QIntValidator>
#include <QGridLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("QLineEdit");

    QLabel *nameLabel = new QLabel("Name");
    QLabel *pwdLabel = new QLabel("Password");
    QLabel *luckyLabel = new QLabel("Lucky Number");

    QLineEdit *nameLine = new QLineEdit;
    QLineEdit *pwdLine = new QLineEdit;
```

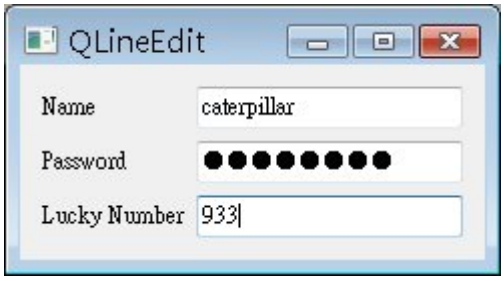
```
    pwdLine->setEchoMode(QLineEdit::Password);
    QLineEdit *luckyLine = new QLineEdit;
    luckyLine->setValidator(new QIntValidator(luckyLine));

    QGridLayout *layout = new QGridLayout;

    layout->addWidget(nameLabel, 0, 0);
    layout->addWidget(nameLine, 0, 1);
    layout->addWidget(pwdLabel, 1, 0);
    layout->addWidget(pwdLine, 1, 1);
    layout->addWidget(luckyLabel, 2, 0);
    layout->addWidget(luckyLine, 2, 1);

    window->setLayout(layout);
    window->show();

    return app.exec();
}
```

setEchoMode()可以设定输入文字的显示方式，有一般显示 （QLineEdit::Normal)、密码显示（QLineEdit::Password）与不响应文字输入（QLineEdit::NoEcho)，密码显示会使用屏蔽字符（像是*）来响应使用者的输入。

setValidator()设定是否对字段的输入进行验证，QIntValidator 用于设定整数的验证方式，也可以设定其它的验证器，像是 QDoubleValidator 用于浮点数的验证。

QLineEdit 还可以设定文字对齐方式，有置左对齐（Qt::AlignLeft)、置中对齐（Qt::AlignCenter)与置右对齐(Qt::AlignRight)等设定方式，也可以使用 setReadOnly()设定 QLineEdit 的字段是否可编辑。

下图为执行的画面参考：



5.32QTextEdit

这个程序基本上只是综合了之前所介绍过的几个组件，像是 QVBoxLayout、QPushButton、QFileDialog 等，以及 QTextEdit 组件来进行文本文件的读取与显示，在 Qt 中并不只有图形化的窗口组件，一些 I/O、绘图、网络等 API 在 Qt 也有提供，在这个例子中将会使用到的是 QFile 与 QTextStream，程序中简单的使用这两个类别，就可以达到开启档案读取的动作。

首先编辑 TxtReader.h：

TxtReader.h

```
#ifndef TXT_READER_H
#define TXT_READER_H

#include <QWidget>

class QTextEdit;
class QPushButton;

class TxtReader : public QWidget
{
    Q_OBJECT
public:
    TxtReader(QWidget *parent = 0);
protected:
    QTextEdit *txtEdit;
    QPushButton *openBtn;
protected slots:
    void readTxtFile();
};

#endif
```

在这个标头档中，定义了一个 Slot，当按下按钮时，将连接这个 Slot 来进行档案读取的动作，接下来编辑 TxtReader.cpp：

TxtReader.cpp

```
#include "TxtReader.h"

#include <QVBoxLayout>
#include <QTextEdit>
#include <QPushButton>
#include <QFont>
#include <QFileDialog>
#include <QFile>
#include <QTextStream>

TxtReader::TxtReader(QWidget *parent) : QWidget(parent)
{
    txtEdit = new QTextEdit;
    txtEdit->setFont(QFont( "Courier", 12, QFont::Bold ));

    openBtn = new QPushButton("Open Text File");
```

```
    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(txtEdit);
    layout->addWidget(openBtn);

    this->setLayout(layout);
    this->resize(350, 200);

    connect(openBtn, SIGNAL(clicked()),
            this, SLOT(readTxtFile()));
}

void TxtReader::readTxtFile()
{
    QString fileName = QFileDialog::getOpenFileName(this, "Open Text File",
                    "C:\\", "Text Files (*.txt *.java *.c *.cpp);;All files (*.*)");
    QFile file(fileName);
    if(file.open(QIODevice::ReadOnly)) {
        this->setWindowTitle(fileName);
        QTextStream stream(&file);
        txtEdit->setText(stream.readAll());
    }
    file.close();
}
```

程序中配置 QTextEdit 与 QPushButton，按下按钮后 clicked()的 Signal 会连接至自定义的 readTxtFile()，在这当中使用了 QFileDialog 供使用者选取档案，而后使用传回的文件名称来用 QFile 进行开档，由于将读取的是文本文件，使用 QTextStream 辅助，可以使用其 readAll()方法一次读进档案中所有的文字，最后记得使用 QFile 的 close()关闭档案。

QTextEdit 类别可以用来编辑文字，基本上不限于纯文字的编辑，还可以编辑字型、颜色等；setText()指定 QTextEdit 的显示文字内容。

接下来编辑主程序：

main.cpp

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

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    TxtReader *txtReader = new TxtReader;
    txtReader->show();
    return app.exec();
```

}
下图为执行时的参考画面：

5.4 清单组件

5.41QListWidget 与 QListWidgetItem

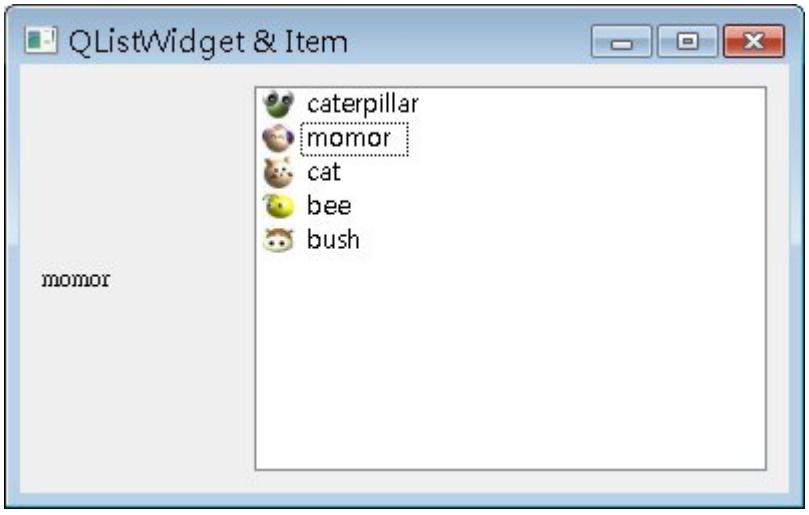
QListWidget 可以显示一个项目清单，清单中每个项目是 QListWidgetItem 的实例，每个项目可以设定文字与图像，以供使用者进行项目的选择，以下来示范一个简单的例子：

```
#include <QApplication>
#include <QHBoxLayout>
#include <QLabel>
#include <QListWidget>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("QListWidget & Item");

    QLabel *label = new QLabel;
    label->setFixedWidth (100);

    QListWidget *listWidget = new QListWidget;
    listWidget->insertItem(0, new QListWidgetItem(
                    QIcon("caterpillar_head.jpg"), "caterpillar"));
    listWidget->insertItem(1, new QListWidgetItem(
                    QIcon("momor_head.jpg"), "momor"));
    listWidget->insertItem(2, new QListWidgetItem(
```

```
                    QIcon("bush_head.jpg"), "bush"));
    listWidget->insertItem(3, new QListWidgetItem(
                    QIcon("bee_head.jpg"), "bee"));
    listWidget->insertItem(4, new QListWidgetItem(
                    QIcon("cat_head.jpg"), "cat"));

    QObject::connect(listWidget, SIGNAL(currentTextChanged (const QString &)),
                     label, SLOT(setText(const QString &)));

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(label);
    layout->addWidget(listWidget);

    window->setLayout(layout);
    window->show();

    return app.exec();
}
```

程序中设定了两个组件，QLabel 与 QListWidget，并连接 QListWidget 的 currentTextChanged()及 setText()，当您选择清单中的项目时，QLabel 会显示目前清单中所选中的项目。QListWidgetItem 还可以使用 setCheckState()方法设定是否具备复选框。

下图为执行时的画面：



您也可以设定 QListWidget 的 setViewMode()，它继承自 QListView，可以设定 QListView::ListMode、 QListView::IconMode，设定为 QListView::IconMode 可以以大图标显示，例如：

```
listWidget->setViewMode(QListView::IconMode);
```

设定为大图示的画面如下所示：

5.42QTreeWidget 与 QTreeWidgetItem

QTreeWidget 类别提供树状的列示组件，可以显示多栏与树状结构，它与 QTreeWidgetItem 一同使用，使用 QTreeWidget 时字段标题题或是字段名称是使用 QStringList 来设定，例如：

```
// 设定字段名称
QStringList columnTitle;
columnTitle.append("Name");
columnTitle.append("Size");
treeWidget->setHeaderLabels(columnTitle);
```

这个程序片段会设定两栏的字段标题，分别为 Name 与 Size 名称，QTreeWidget 要设定一个顶层的 QTreeWidgetItem：

```
QTreeWidgetItem *dir = new QTreeWidgetItem(fileColumn);
dir->setIcon(0, QIcon("caterpillar_head.jpg"));
dir->setCheckState(0, Qt::Checked); // 设定显示可核取的方块
treeWidget->addTopLevelItem(dir);
```

setCheckState()方法设定 QTreeWidgetItem 出现可核取的方块。QTreeWidget 中每个树状子节点则为每个 QTreeWidgetItem 的子组件，例如：

```
QStringList fileColumn;
fileColumn.append(fileInfo.fileName());

QTreeWidgetItem *child = new QTreeWidgetItem(fileColumn);
child->setIcon(0, QIcon("caterpillar_head.jpg"));
```

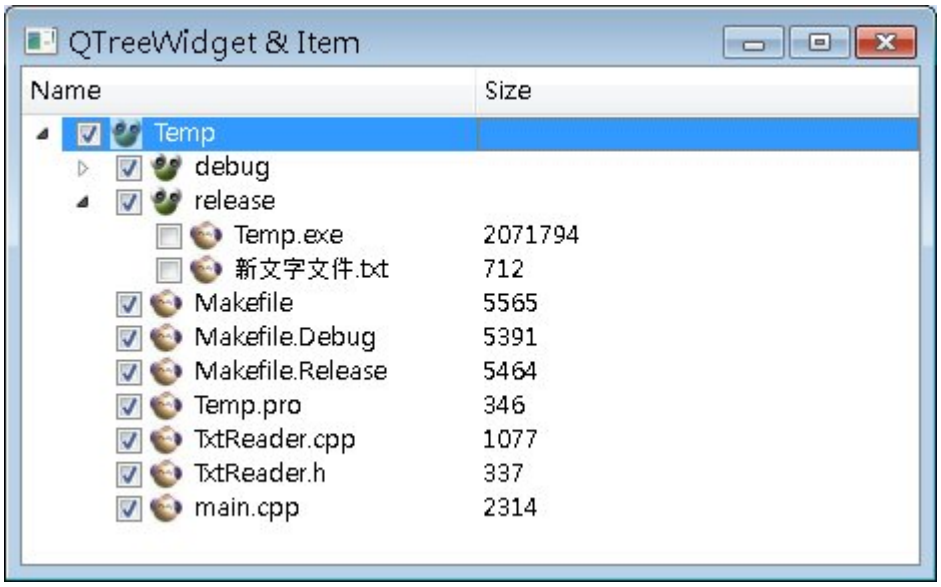
```
parentWidgetItem->addChild(child);
```

以上为 QTreeWidget 与 QTreeWidgetItem 的基本使用方式。下面这个程序将结合之后会介绍的 QFileInfo 与 QDir 类别，使用递归查询出指定目录下的所有目录与档案，但不包括隐藏文件与符号连结，查询的结果将分为目录与档案，并使用 QTreeWidget 类别的树状结构加以显示，目前您只要先注意 QTreeWidget 与 QTreeWidgetItem 的使用方式，QFileInfo 类别与 QDir 类别之后将会介绍：

```
#include <QApplication>
#include <QTreeWidget>
#include <QTreeWidgetItem>
#include <QStringList>
#include <QFile>
#include <QFileInfo>
#include <QDir>

void listFile(QTreeWidgetItem *, QFileInfo &);

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QTreeWidget *treeWidget = new QTreeWidget;
    treeWidget->setWindowTitle("QTreeWidget & Item");
    treeWidget->resize(400, 250);

    // 设定字段名称
    QStringList columnTitle;
    columnTitle.append("Name");
    columnTitle.append("Size");
    treeWidget->setHeaderLabels(columnTitle);

    // 查询的目录
    QFileInfo fileInfo("D:\\Temp");
    QStringList fileColumn;
    fileColumn.append(fileInfo.fileName());

    QTreeWidgetItem *dir = new QTreeWidgetItem(fileColumn);
    dir->setIcon(0, QIcon("caterpillar_head.jpg"));
    dir->setCheckState(0, Qt::Checked); // 设定可核取的方块
    treeWidget->addTopLevelItem(dir);

    // 查询目录
    listFile(dir, fileInfo);
```

```
    treeWidget->show();

    return app.exec();
}

void listFile(QTreeWidgetItem *parentWidgetItem, QFileInfo &parent)
{
    QDir dir;
    dir.setPath(parent.filePath());
    dir.setFilter(QDir::Files | QDir::Dirs | QDir::NoSymLinks);
    dir.setSorting(QDir::DirsFirst | QDir::Name);

    const QFileInfoList fileList = dir.entryInfoList();

    for (int i = 0; i < fileList.size(); i++) {
        QFileInfo fileInfo = fileList.at(i);
        QStringList fileColumn;
        fileColumn.append(fileInfo.fileName());
        if (fileInfo.fileName() == "." || fileInfo.fileName() == ".." ); // nothing
        else if(fileInfo.isDir()) {
            QTreeWidgetItem *child = new QTreeWidgetItem(fileColumn);
            child->setIcon(0, QIcon("caterpillar_head.jpg"));
            child->setCheckState(0, Qt::Checked);
            parentWidgetItem->addChild(child);
            // 查询子目录
            listFile(child, fileInfo);
        }
        else {
            fileColumn.append(QString::number(fileInfo.size()));
            QTreeWidgetItem *child = new QTreeWidgetItem(fileColumn);
            child->setIcon(0, QIcon("momor_head.jpg"));
            child->setCheckState(0, Qt::Checked);
            parentWidgetItem->addChild(child);
        }
    }
}
```

程序中直接设定显示 D:\Temp 下的所有目录与档案，一个执行的结果画面如下所示：



5.43QTableWidget 与 QTableWidgetItem

QTableWidget 可以显示一个表格组件，表格中每个储存格则为一个 QTableWidgetItem 的实例，QTableWidgetItem 要安插至表格中哪个储存格，则是依索引的指定来决定。

下面的程序代码为简单的 QTableWidget 与 QTableWidgetItem 的示范：

```
#include <QApplication>
#include <QTableWidget>
#include <QHBoxLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QTableWidget *tableWidget = new QTableWidget;
    tableWidget->setWindowTitle("QTableWidget & Item");
    tableWidget->resize(350, 200);
    tableWidget->setRowCount(10);
    tableWidget->setColumnCount(5);

    QStringList header;
    header.append("Month");
    header.append("Description");
    tableWidget->setHorizontalHeaderLabels(header);

    tableWidget->setItem(0, 0, new QTableWidgetItem("January"));
    tableWidget->setItem(1, 0, new QTableWidgetItem("February"));
    tableWidget->setItem(2, 0, new QTableWidgetItem("March"));
```

```
    tableWidget->setItem(0, 1,
        new QTableWidgetItem(QIcon("caterpillar_head.jpg"), "caterpillar's month"));
    tableWidget->setItem(1, 1,
        new QTableWidgetItem(QIcon("momor_head.jpg"), "momor's month"));
    tableWidget->setItem(2, 1,
        new QTableWidgetItem(QIcon("bush_head.jpg"), "bush's month"));

    tableWidget->show();

    return app.exec();
}
```

使用 setItem()时必须指定储存格索引值，索引为列（row）行（column），皆从 0 开始，最左上角即为索引（0, 0）位置。QTableWidgetItem 也可以设置图片或核取状态（setCheckState()）等。

下图为程序执行时的画面：

5.44Model 与 View 类别

QListWidget、QTreeWidget、QTableWidget，它们分别是 QListView、QTreeView、QTableView 的子类别，而这些父类别又都继承自 QAbstractItemView 类别， QAbstractItemView 负责 Model/View 设计中 View 的角色，而 QAbstractItemModel 则南 Model/View 设计中 Model 的角色。

Model/View 设计中，View 负责画面的呈现，而 Model 则是与画面无关的数据模型，一个画面表现出数据模型的呈现方式，数据模型可以被多个不同性质的画面呈现，例如表格、清单、长条图等，如果数据有变动，画面会同时依照数据模型而变动。

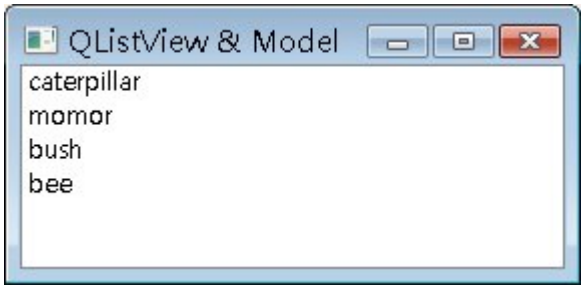
以最简单的例子来说明 View 类别的使用：

```
#include <QApplication>
#include <QStringList>
#include <QAbstractItemModel>
```

```
#include <QStringListModel>
#include <QListView>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QStringList numbers;
    numbers << "caterpillar" << "momor" << "bush" << "bee";
    QAbstractItemModel *model = new QStringListModel(numbers);

    QListView *view = new QListView;
    view->setWindowTitle("QListView & Model");
    view->setModel(model);

    view->show();

    return app.exec();
}
```

QStringListModel 為 QAbstractItemModel 的子类别，上面的程序当中，使用 QStringListModel 包装 QStringList 数据,以作为 View 的数据模型,在这边使用 QListView 作为画面，并使用其 setModel()方法设定数据模型。

下图为执行时的参考画面：

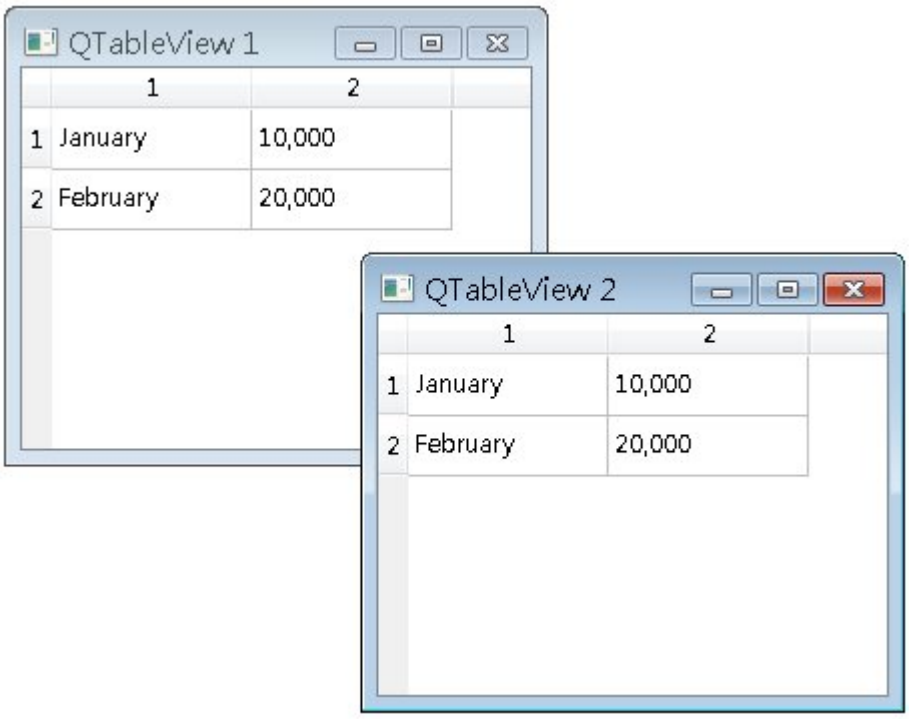


Qt 有提供 QAbstractItemModel 的几个实作，像是 QStringListModel、QSqlQueryModel、QStandardItemModel 等，下面这个程序使用 QTableView 与 QStandItemModel 制作表格，数据模型将由两个表格画面共享，当您改变其中一个表格画面的数据时，数据模型的数据会变动，而另一个表格画面的数据也会自动更新：

```
#include <QApplication>
#include <QStandardItemModel>
#include <QStandardItem>
#include <QTableView>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
```

```
    QStandardItemModel *model = new QStandardItemModel;
    model->setItem(0, 0, new QStandardItem("January"));
    model->setItem(1, 0, new QStandardItem("February"));
    model->setItem(0, 1, new QStandardItem("10,000"));
    model->setItem(1, 1, new QStandardItem("20,000"));

    QTableView *view1 = new QTableView;
    view1->setModel(model);

    QTableView *view2 = new QTableView;
    view2->setModel(model);

    view1->show();
    view2->show();

    return app.exec();
}
```

下图为执行时的参考画面：

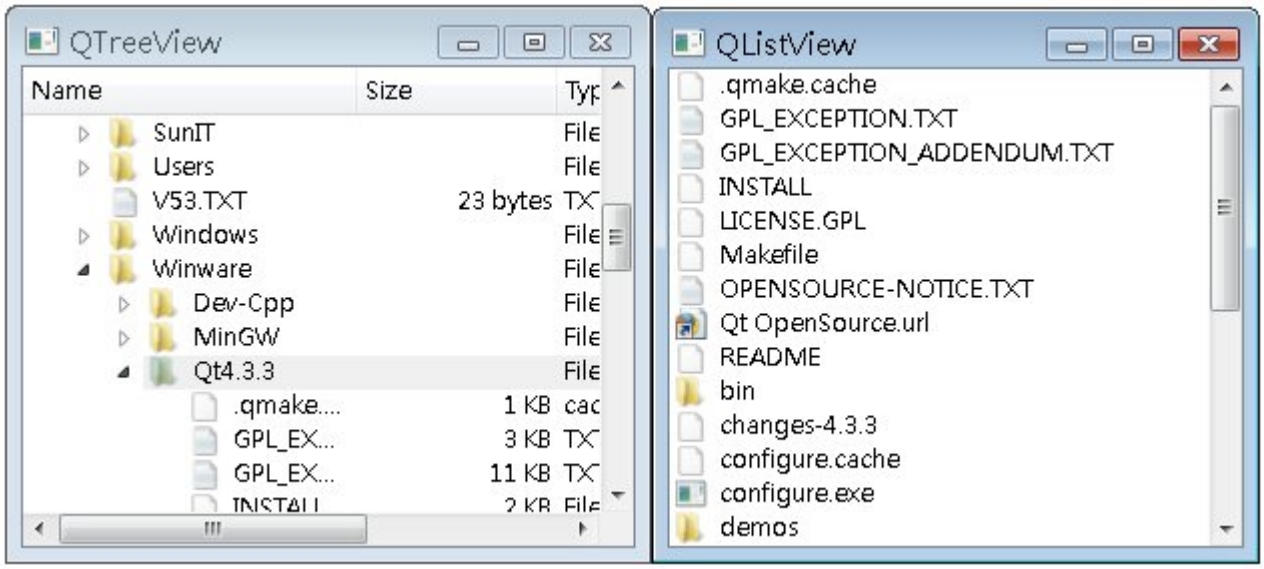


QDirModel 提供本机档案系统的档案信息快取，下面这个程序使用 QDirModel、QTreeView 与 QListview，制作类似档案总管的功能，您可以在 QTreeView 中点选项目，如果该项目是数据夹，则在 QListView 中显示该数据夹中的档案与子资料夹：

```
#include <QApplication>
#include <QDirModel>
#include <QTreeView>
#include <QListview>
```

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QDirModel *treeModel = new QDirModel;
    QTreeView *tree = new QTreeView;
    tree->setModel(treeModel);

    QListView *list = new QListView;
    list->setModel(treeModel);
    list->setRootIndex(treeModel->index("C://"));

    QObject::connect(tree, SIGNAL(clicked(const QModelIndex&)),
                     list, SLOT(setRootIndex(const QModelIndex&)));

    tree->setWindowTitle("QTreeView");
    tree->show();
    list->setWindowTitle("QListView");
    list->show();

    return app.exec();
}
```

下图为执行时的参考画面：



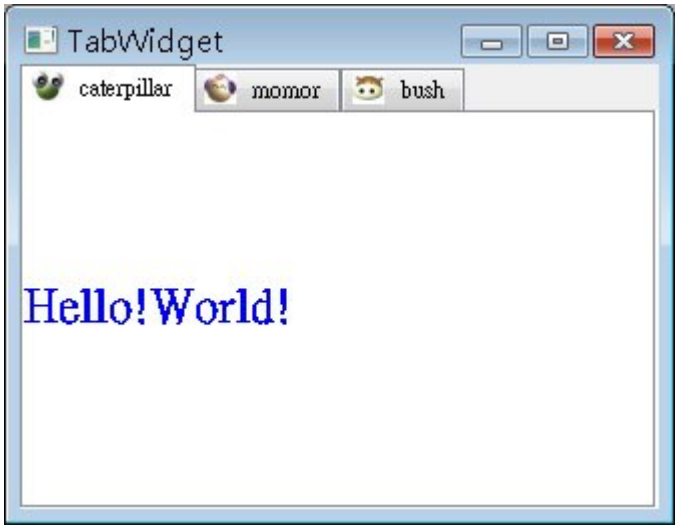
5.5 版面组件

5.51QTabWidget

在版面配置上，可以会使用 QTabWidget 来作功能页的分类，它提供多个显示页，可以藉由上方的标签来选择所要的功能页面，下面的程序简单的示范如何将组件加入 QTabWidget 成为一个标签页。

```
#include <QApplication>
#include <QTabWidget>
#include <QLabel>
#include <QIcon>
#include <QPushButton>
#include <QTextEdit>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QTabWidget *tabWidget = new QTabWidget;
    tabWidget->setWindowTitle("TabWidget");

    tabWidget->addTab(
      new QLabel("<h1><font color=blue>Hello!World!</font></h1>"),
      QIcon("caterpillar_head.jpg"), "caterpillar");

    tabWidget->addTab(
      new QPushButton("Push XD"),
      QIcon("momor_head.jpg"), "momor");

    tabWidget->addTab(
      new QTextEdit,
      QIcon("bush_head.jpg"), "bush");

    tabWidget->show();

    return app.exec();
}
```

程序中设定了三个卷标页，每个卷标页中使用 addTab()简单的加入一个 Widget，下图为执行时的画面：

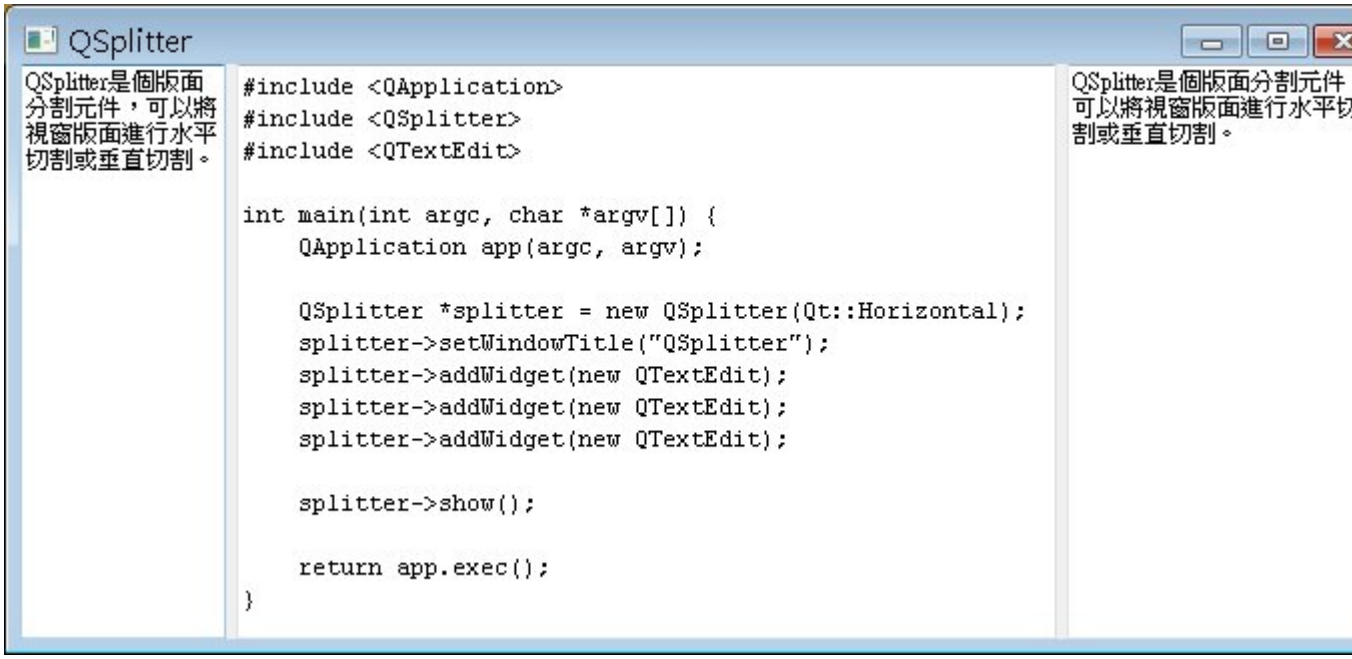


5.52QSplitter

QSplitter 是个版面分割组件，可以将窗口版面进行水平切割或垂直切割，一个最简单的范例如下所示：

```
#include <QApplication>
#include <QSplitter>
#include <QTextEdit>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QSplitter *splitter = new QSplitter(Qt::Horizontal);
    splitter->setWindowTitle("QSplitter");
    splitter->addWidget(new QTextEdit);
    splitter->addWidget(new QTextEdit);
    splitter->addWidget(new QTextEdit);

    splitter->show();

    return app.exec();
}
```

直接来看执行画面：



利用 QSplitter 的嵌套，可以组合出更复杂的画面切割方式，例如：

```
#include <QApplication>
#include <QSplitter>
#include <QTextEdit>
```

```
#include <QListWidget>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QListWidget *listWidget = new QListWidget;
    listWidget->insertItem(0, new QListWidgetItem(
                    QIcon("caterpillar_head.jpg"), "caterpillar"));
    listWidget->insertItem(1, new QListWidgetItem(
                    QIcon("momor_head.jpg"), "momor"));
    listWidget->insertItem(2, new QListWidgetItem(
                    QIcon("bush_head.jpg"), "bush"));
    listWidget->insertItem(3, new QListWidgetItem(
                    QIcon("bee_head.jpg"), "bee"));
    QSplitter *splitter1 = new QSplitter(Qt::Horizontal);
    splitter1->setWindowTitle("QSplitter");

    QSplitter *splitter2 = new QSplitter(Qt::Vertical);
    splitter1->addWidget(listWidget);
    splitter1->addWidget(splitter2);

    splitter2->addWidget(
        new QLabel("<h1><font color=blue>Hello!World!</font></h1>"));
    splitter2->addWidget(new QTextEdit);
    splitter1->show();
    return app.exec();
}
```

这个程序先将画面进行水平切割，然后在右边的切割中再进行垂直切割，所完成的画面切割如下所示：

5.53QStackedLayout

QStackedLayout 可以让您将组件分成一层一层的堆栈，每一层组件有索引，可以指定索引来表示该显示哪一层的组件，您也可以直接使用 QStackedWidget，它继承自 QWidget，内建的版面配置是 QStackedLayout。

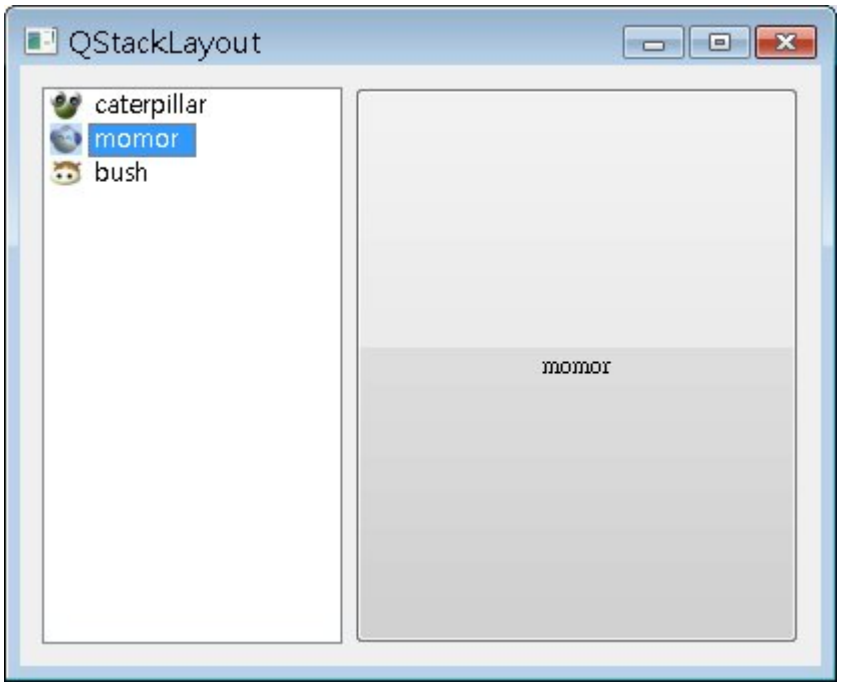
下面的程序使用 QListWidget 在窗口左边提供选项，在窗口右边使用 QStackedLayout 放置三层组件，使用者在左边选取右边要显示哪一层组件：

```
include <QApplication>
#include <QStackedLayout>
#include <QListWidget>
#include <QIcon>
#include <QLabel>
#include <QPushButton>
#include <QTextEdit>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("QStackLayout");
    window->resize(400, 300);

    QHBoxLayout *hBoxLayout = new QHBoxLayout;

    QListWidget *listWidget = new QListWidget;
    listWidget->setFixedWidth(150);
    listWidget->insertItem(0, new QListWidgetItem(
                    QIcon("caterpillar_head.jpg"), "caterpillar"));
    listWidget->insertItem(1, new QListWidgetItem(
                    QIcon("momor_head.jpg"), "momor"));
    listWidget->insertItem(2, new QListWidgetItem(
                    QIcon("bush_head.jpg"), "bush"));

    hBoxLayout->addWidget(listWidget);

    QStackedLayout *stackedLayout = new QStackedLayout;
    hBoxLayout->addLayout(stackedLayout);

    stackedLayout->addWidget(
        new QLabel("<h1><font color=blue>caterpillar</font></h1>"));
    stackedLayout->addWidget(new QPushButton("momor"));
```

```
    stackedLayout->addWidget(new QTextEdit);

    QObject::connect(listWidget, SIGNAL(currentRowChanged(int)),
                     stackedLayout, SLOT(setCurrentIndex(int)));

    window->setLayout(hBoxLayout);
    window->show();

    return app.exec();
}
```

程序中将 QListWidget 的 currentRowChanged() Signal 连接至 setCurrentIndex() Slot，currentRowChanged()会传送目前选取的选项索引，setCurrentIndex()根据所传送的索引值设定目前 QStackedLayout 要显示哪一层组件。

程序执行时的画面如下所示：



5.54QScrollArea

有些组件预设并没有滚动条，例如 QLabel，当窗口或父组件无法显示其大小时，只会显示部份区域，但不会出现滚动条，如果您希望这类组件可以出现滚动条，则可以使用 QScrollArea。

QScrollArea 包括三个部份，一个是水平滚动条、一个垂直滚动条、一个 view port，view port 为水平滚动条及垂直滚动条中的显示区域。

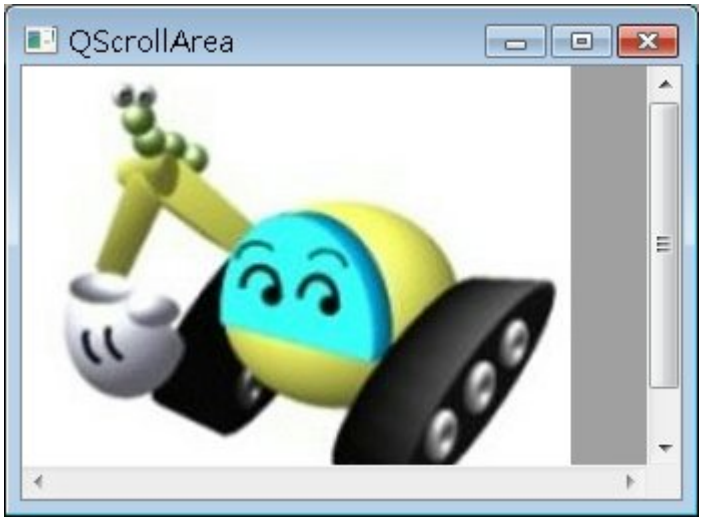
在下面的这个范例程序中，使用 QLabel 并使用 setPixmap()设定其显示图片，在这边使用 QScrollArea 的 setWidget()将 QLabel 设定为 view port 中的显示组件，并使用

setHorizontalScrollBarPolicy() 与 setVerticalScrollBarPolicy () 设定总是显示滚动条（Qt::ScrollBarAlwaysOn）：

```
#include <QApplication>
#include <QLabel>
#include <QScrollArea>
#include <QPixmap>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QLabel *label = new QLabel;
    label->setPixmap(QPixmap("caterpillar.jpg"));

    QScrollArea *scrollArea = new QScrollArea;
    scrollArea->setWindowTitle("QScrollArea");

    scrollArea->setWidget(label);
    scrollArea->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
    scrollArea->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
    scrollArea->viewport()->setBackgroundRole(QPalette::Red);

    scrollArea->show();

    return app.exec();
}
```

您可以使用 QScrollArea 的 viewport()方法取得水平、垂直滚动条间的 view port，在这边使用其 setBackgroundRole()方法，设定 view port 所显示组件外的区域显示为灰色。

下图为执行时的参考画面，可以看到水平与垂直滚动条，而 QLabel 显示图片加上灰色区域即为 view port 区域：



5.6 其它组件

5.61QScrollBar

QScrollArea 提供您方便为某个组件在画面不足以显示组件时，直接加上滚动条辅助显示，而 QScrollBar 则可以让您单独建立水平或垂直滚动条功能，以自订滚动条的控制行为。

例如以下的程序，将自行建立水平与垂直滚动条，以控制画面中的 QLabel 位置：

ScrollImage.h

```
#ifndef SCROLLIMAGE_H
#define SCROLLIMAGE_H

#include <QWidget>

class QLabel;

class ScrollImage : public QWidget {
    Q_OBJECT

public:
    ScrollImage(const QPixmap &, QWidget *parent = 0);

public slots:
    void setX(int);
    void setY(int);

private:
    QLabel *label;
};

#endif
```

ScrollImage 为自订组件，继承自 QWidget，并自订了两个 Slot，以接受水平与垂直滚动条的数值 Signal，其实作如下所示：

ScrollImage.cpp

```
#include "ScrollImage.h"
#include <QScrollBar>
#include <QLabel>
#include <QPixmap>

ScrollImage::ScrollImage(
                    const QPixmap &pixmap, QWidget *parent) : QWidget(parent) {
    this->resize(pixmap.width() * 10, pixmap.height() * 10);
```

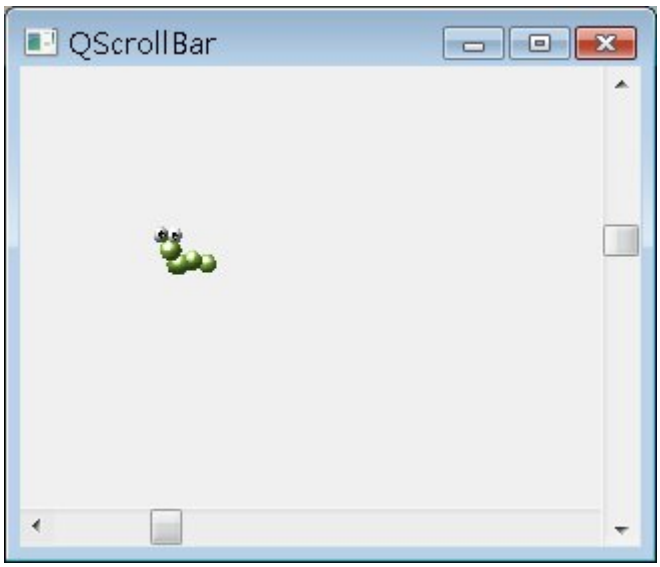
```
    label = new QLabel(this);
    label->setPixmap(pixmap);
    label->setGeometry(20, 20, pixmap.width(), pixmap.height());

    // 水平滚动条
    QScrollBar *hScrollBar = new QScrollBar(Qt::Horizontal, this);
    hScrollBar->setGeometry(0, this->height() - 20, this->width(), 20);
    hScrollBar->setMaximum(this->width() - pixmap.width() - 20);

    connect(hScrollBar, SIGNAL(valueChanged(int)),
            this, SLOT(setX(int)));

    // 垂直滚动条
    QScrollBar *vScrollBar = new QScrollBar(Qt::Vertical, this);
    vScrollBar->setGeometry(this->width() - 20, 0, 20, this->height());
    vScrollBar->setMaximum(this->height() - pixmap.height() - 30);

    connect(vScrollBar, SIGNAL(valueChanged(int)),
            this, SLOT(setY(int)));
}

void ScrollImage::setX(int x) {
    label->setGeometry(20 + x, label->y(), label->width(), label->height());
}

void ScrollImage::setY(int y) {
    label->setGeometry(label->x(), 20 + y, label->width(), label->height());
}
```

来撰写一个主程序进行程序执行与测试：

main.cpp

```
#include "ScrollImage.h"
#include <QApplication>
#include <QPixmap>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    ScrollImage *scrollImage =
        new ScrollImage(QPixmap("caterpillar_small.gif"));
    scrollImage->setWindowTitle("QScrollBar");
    scrollImage->show();

    return app.exec();
```

}
下图为执行时的参考画面，拉动滚动条时，图片会跟着移动：



5.62QTimer 与 QLCDNumber

Qt 中的 QTimer 会定时发出 QTimerEvent 事件，您可以设定一个接受 QTimerEvent 的函式来接收这个事件，一旦接收到 QTimerEvent 事件，函式的内容就会被执行，例如接下来的范例，将设计一个数字时钟，可以显示目前时间与日期。

数字时钟的显示将使用 QLCDNumber，这边直接继承 QLCDNumber 来实作：

DigitalClock.h

```
#ifndef DIGITALCLOCK_H
#define DIGITALCLOCK_H

#include <QLCDNumber>

class DigitalClock : public QLCDNumber {

public:
    DigitalClock(QWidget *parent=0);

protected:
    void timerEvent(QTimerEvent *);

private:
    bool isColon;
};

#endif
```

在 DigitalClock 中，重新定义了 timerEvent()来接受 TimerEvent，以定时接受 QTimer 的事件并进行时间的撷取与显示更新，DigitalClock 实作如下：

DigitalClock.cpp

```
#include "DigitalClock.h"

#include <QDateTime>

DigitalClock::DigitalClock(QWidget *parent) : QLCDNumber(parent) {
    this->isColon = false;
    this->setFrameStyle(QFrame::Panel | QFrame::Raised);
    this->setNumDigits(11);
    QObject::startTimer(500);
}

void DigitalClock::timerEvent(QTimerEvent *e) {
    isColon = !isColon;

    QString timeString = QTime::currentTime().toString().left(5);

    QDate date = QDate::currentDate();
    QString dateString;
    dateString.sprintf( " %2d-%2d", date.month(), date.day());

    QString displayString =    timeString + dateString;

    if (!isColon) {
        displayString[2] = ' ';
    }

    display(displayString);
}
```

QTimer 的启动是使用 QObject::startTimer(500)，如果想停止 QTimer，则使用 QTimer::killTimer()，500 的单位是毫秒，这边每 0.5 秒发出一次事件，每次 timerEvent()接受到 QTimerEvent 事件时，会撷取目前的时间与日期，每 0.5 秒处理一次是为了处理冒号的显示，这是由 timerEvent()中 if 判断式来负责。

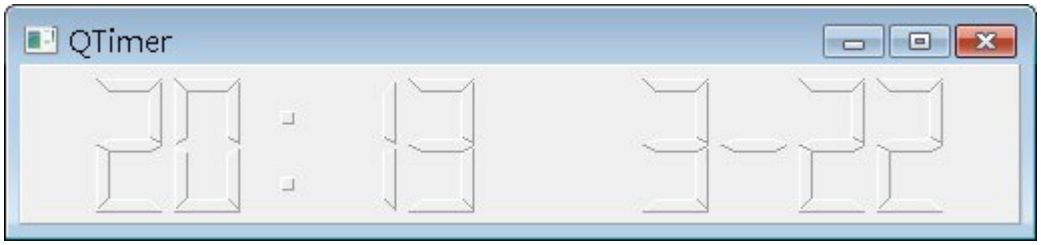
撰写一个主程序来看看这个组件是否运作正常：

main.cpp

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

int main(int argc, char *argv[]) {
```

```
    QApplication app(argc, argv);

    DigitalClock *digitalClock = new DigitalClock;

    digitalClock->resize(500, 80);
    digitalClock->setWindowTitle("QTimer");

    digitalClock->show();

    return app.exec();
}
```

执行的结果画面如下所示：



5.63QProgressBar

QProgressBar 常用来显示目前的工作进度，例如程序安装、档案复制、下载等，以下这个范例使用 QTimer 来仿真执行进度，并使用 QProgressBar 来显示执行进度。

ProgressBar.h

```
#ifndef PROGRESSBAR_H
#define PROGRESSBAR_H

#include <QProgressBar>

class ProgressBar : public QProgressBar {
    Q_OBJECT

public:
    ProgressBar(QWidget *parent=0) : QProgressBar(parent) {}

public slots:
    void stepOne();

};

#endif
```

stepOne()每执行一次，会根据目前的进度加一，如果加到 QProgressBar 最大值，就从最小值重新开始：

ProgressBar.cpp

```
#include "ProgressBar.h"

void ProgressBar::stepOne() {
	if(this->value() + 1 <= this->maximum()) {
		this->setValue(this->value() + 1);
	}
	else {
		this->setValue(this->minimum());
	}
}
```

程序中将使用 QTimer 的 timerout()来连接 stepOne()，每 500 毫秒进行一个进度：

main.cpp

```
#include "ProgressBar.h"

#include <QApplication>
#include <QTimer>

int main(int argc, char *argv[]) {
	QApplication app(argc, argv);

	ProgressBar *progressBar = new ProgressBar;
	progressBar->setWindowTitle("QProgressBar");
	progressBar->resize(250, 20);

	progressBar->setMaximum(100);
	progressBar->setMinimum(0);
	progressBar->setValue(0);

	QTimer *timer = new QTimer;
	timer->start(500);

	QObject::connect(timer, SIGNAL(timeout()), progressBar, SLOT(stepOne()));

	progressBar->show();

	return app.exec();
}
```

执行结果画面如下所示：

5.64QWizard

在应用程序安装或是使用者注册、设定时，可以提供使用者「精灵」（Wizard）进行一些选项设定与信息填写，在 Step by Step 的过程中，提示使用者完成所有必要的选项设定或信息填写，精灵可以使用 QWizard 类别来提供这个功能。

QWizard 中每一步的画面，都是一个 QWizardPage 对象，一个 QWizardPage 可以进行组件置放、设定版面管理、设定水印图片、LOGO 图片、标题、副标题等。

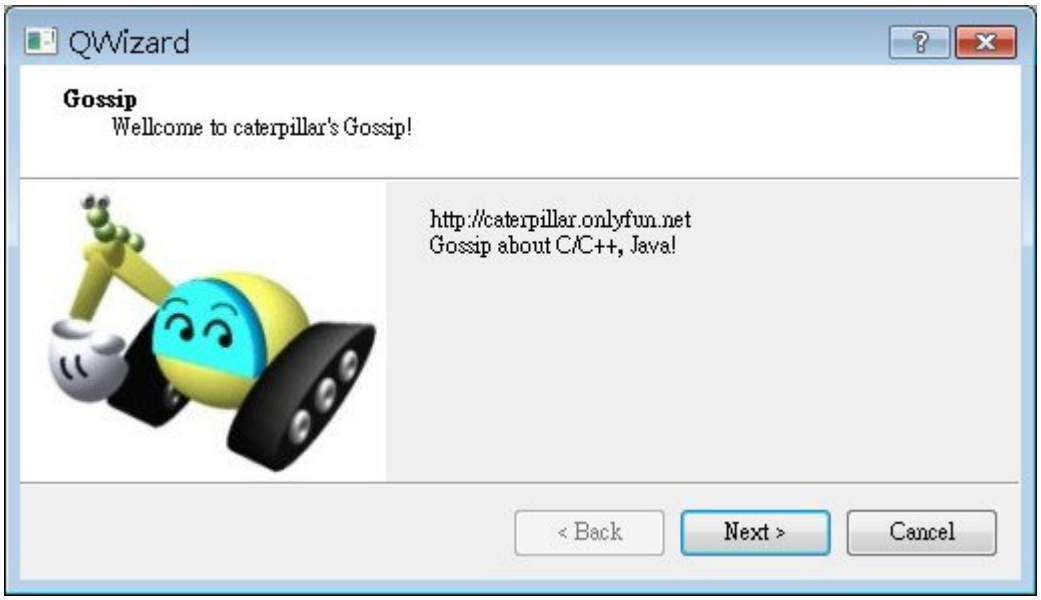
下面这个程序是个简单的示范，程序中将用 QWizard 建立有两个页面步骤的精灵：

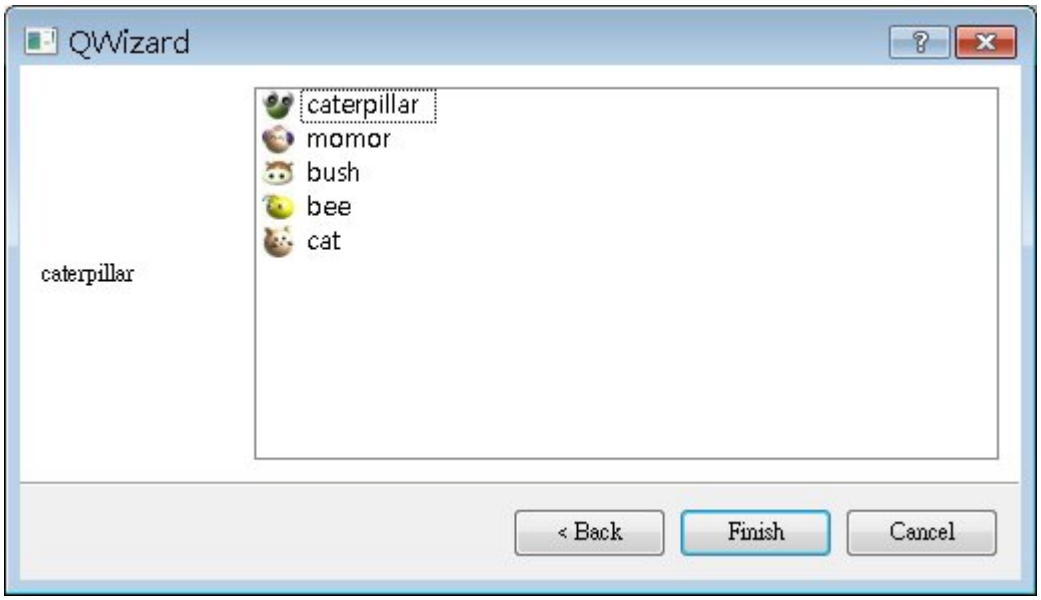
```
#include <QApplication>
#include <QLabel>
#include <QWizard>
#include <QWizardPage>
#include <QHBoxLayout>
#include <QListWidget>

QWizardPage *createWelcomePage() {
    QWizardPage *page = new QWizardPage;
    page->setTitle("Gossip");
    page->setSubTitle("Wellcome to caterpillar's Gossip!");
    page->setPixmap(QWizard::WatermarkPixmap, QPixmap("caterpillar.jpg"));

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(
      new QLabel("http://caterpillar.onlyfun.net<br>Gossip about C/C++, Java!"));
    page->setLayout(layout);

    return page;
}

QWizardPage *createListPage() {
    QWizardPage *page = new QWizardPage;

    QLabel *label = new QLabel;
    label->setFixedWidth (100);

    QListWidget *listWidget = new QListWidget;
    listWidget->insertItem(0, new QListWidgetItem(
                        QIcon("caterpillar_head.jpg"), "caterpillar"));
    listWidget->insertItem(1, new QListWidgetItem(
                        QIcon("momor_head.jpg"), "momor"));
    listWidget->insertItem(2, new QListWidgetItem(
```

```
                        QIcon("bush_head.jpg"), "bush"));
    listWidget->insertItem(3, new QListWidgetItem(
                        QIcon("bee_head.jpg"), "bee"));
    listWidget->insertItem(4, new QListWidgetItem(
                        QIcon("cat_head.jpg"), "cat"));

    QObject::connect(listWidget, SIGNAL(currentTextChanged (const QString &)),
                        label, SLOT(setText(const QString &)));

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(label);
    layout->addWidget(listWidget);

    page->setLayout(layout);

    return page;
}

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QWizard *wizard = new QWizard;
    wizard->setWindowTitle("QWizard");

    wizard->addPage(createWelcomePage());
    wizard->addPage(createListPage());

    wizard->show();

    return app.exec();
}
```

QWizardPage 的 setTitle()与 setSubTitle()分别用来设定标题与副标题，而 setPixmap ()用来设定显示在画面中的图片，图片的显示方式有水印（WatermarkPixmap）、LOGO（LogoPixmap）、横副（BannerPixmap）、背景（BackgroundPixmap）等方式，在精灵页面中的显示的位置各不相同，可以参考 Qt 在线文件中 QWizard 的说明。

程序中的精灵有两个页面，在过程中可以按「Next」进入下一个页面，如果是最后一个页面，则会显示「Finish」按钮，下图是执行的的参考画面：





5.65QMainWindow

QMainWindow 类别提供一个标准的应用程序窗口，当中可以包括选单（QMenuBar）、工具列（QToolBar）、状态列（QStatusBar）、停驻组件（QDockWidget）等组件。

直接以范例来说明如何使用 QMainWindow 类别，在当中会有一个文字编辑区、选单、工具列、状态列与一个停驻组件：

```
#include <QApplication>
#include <QMainWindow>
#include <QTextEdit>
#include <QMenuBar>
#include <QAction>
#include <QToolBar>
```

```
#include <QStatusBar>
#include <QDockWidget>
#include <QLabel>
#include <QPixmap>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QMainWindow *mainWindow = new QMainWindow;
    mainWindow->setWindowTitle("QMainWindow");

    // 文字编辑区
    QTextEdit *textEdit = new QTextEdit;
    textEdit->setFocus();

    mainWindow->setCentralWidget(textEdit);

    // 蹦现选单
    QMenu *fileMenu = new QMenu("&File");
    QAction *fileAction = new QAction("Open..", fileMenu);
    // 快捷键 Ctrl+O
    fileAction->setShortcut(Qt::CTRL + Qt::Key_O);

    fileMenu->addAction(fileAction);
    fileMenu->addAction("Save");
    fileMenu->addAction("Save as...");
    // 分隔线
    fileMenu->addSeparator();
    // 快捷键 Ctrl+X，动作连接至 QApplication 的 quit()
    fileMenu->addAction("Close", &app, SLOT(quit()), Qt::CTRL + Qt::Key_X);

    QMenu *editMenu = new QMenu("&Edit");
    editMenu->addAction("Cut");
    editMenu->addAction("Copy");
    editMenu->addAction("Paste");

    QMenu *aboutMenu = new QMenu("&About");
    aboutMenu->addAction("About");

    mainWindow->menuBar()->addMenu(fileMenu);
    mainWindow->menuBar()->addMenu(editMenu);
    mainWindow->menuBar()->addMenu(aboutMenu);

    // 工具列
```

```
    QToolBar *toolBar = new QToolBar("QToolBar");
    toolBar->addAction(QIcon("caterpillar_head.jpg"), "caterpillar");
    toolBar->addAction(QIcon("momor_head.jpg"), "momor");
    toolBar->addSeparator();
    toolBar->addAction(QIcon("bush_head.jpg"), "bush");

    mainWindow->addToolBar(toolBar);

    // 状态列
    QStatusBar *statusBar = mainWindow->statusBar();
    statusBar->showMessage("Status here...");

    // 停驻组件
    QDockWidget *dockWidget = new QDockWidget("QDockWidget");
    QLabel *label = new QLabel;
    label->setPixmap(QPixmap("caterpillar.jpg"));
    dockWidget->setWidget(label);

    mainWindow->addDockWidget(Qt::RightDockWidgetArea, dockWidget);

    mainWindow->show();

    return app.exec();
}
```

QMainWindow 的 setCentralWidget()用来设定主窗口的中央组件：

```
mainWindow->setCentralWidget(textEdit);
```

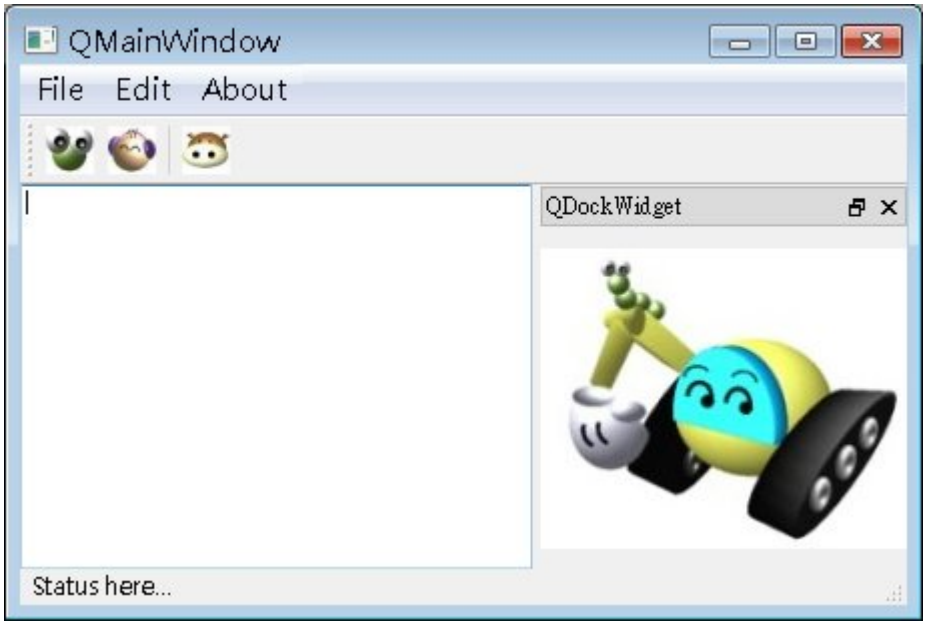
每个 QMenu 实例中的选项为 QAction 的实例，QMenu 的 addAction()可以直接使用字符串，当中会自动产生 QAction 并加入至 QMenu 中，而 addSeparator()则可以加入分隔线，比较特别的是：

```
fileMenu->addAction("Close", &app, SLOT(quit()), Qt::CTRL + Qt::Key_X);
```

这一个版本的 addAction()可以直接将 QAction 的 trigger() SIGNAL 连接至接收 SINGAL 的对象之 SLOT，并可以设定快捷键功能，在上面的程序代码中，按下选单中「Close」，则会使用 QApplication 的 quit()来关闭应用程序。

每个 QMainWindow 会具备 QMenuBar 与 QStatusBar，您可以分别使用 menuBar()与 statusBar()来取得，程序中使用 QMenuBar 的 addMenu()加入 QMenu 实例，使用 QStatusBar 的 showMessage()来设定状态列文字。

QMainWindow 的工具列，则使用 addToolBar()来加入 QToolBar 的实例，至于停驻组件的部份，则是一个类似工具列的图形组件，但您可以自行配置当中的组件与功能，Qt::RightDocuWidgetArea 设定了组件预设的停驻位置为窗口的右边，直接来看一下程序的执行画面，就可以了解何谓停驻组件，并看看 QMainWindow 配置的各组件之画面：



5.66QMdiArea

在 QMainWindow 中所示范的是 SDI（Single Document Interface）窗口接口，每个开启的文件占据一个窗口，SDI 接口主要适用所有工作都在同一个文件中进行的情况。

有的窗口程序会使用 MDI（Multiple Document Interface）界面，每个开启的文件都在同一个窗口之中成为一个子窗口，MDI 主要适用于完成一个工作，需要从多个文件来组合的情况，例如影像处理软件多使用 MDI 接口，因为影像合成通常需要多个影像文件来组合。

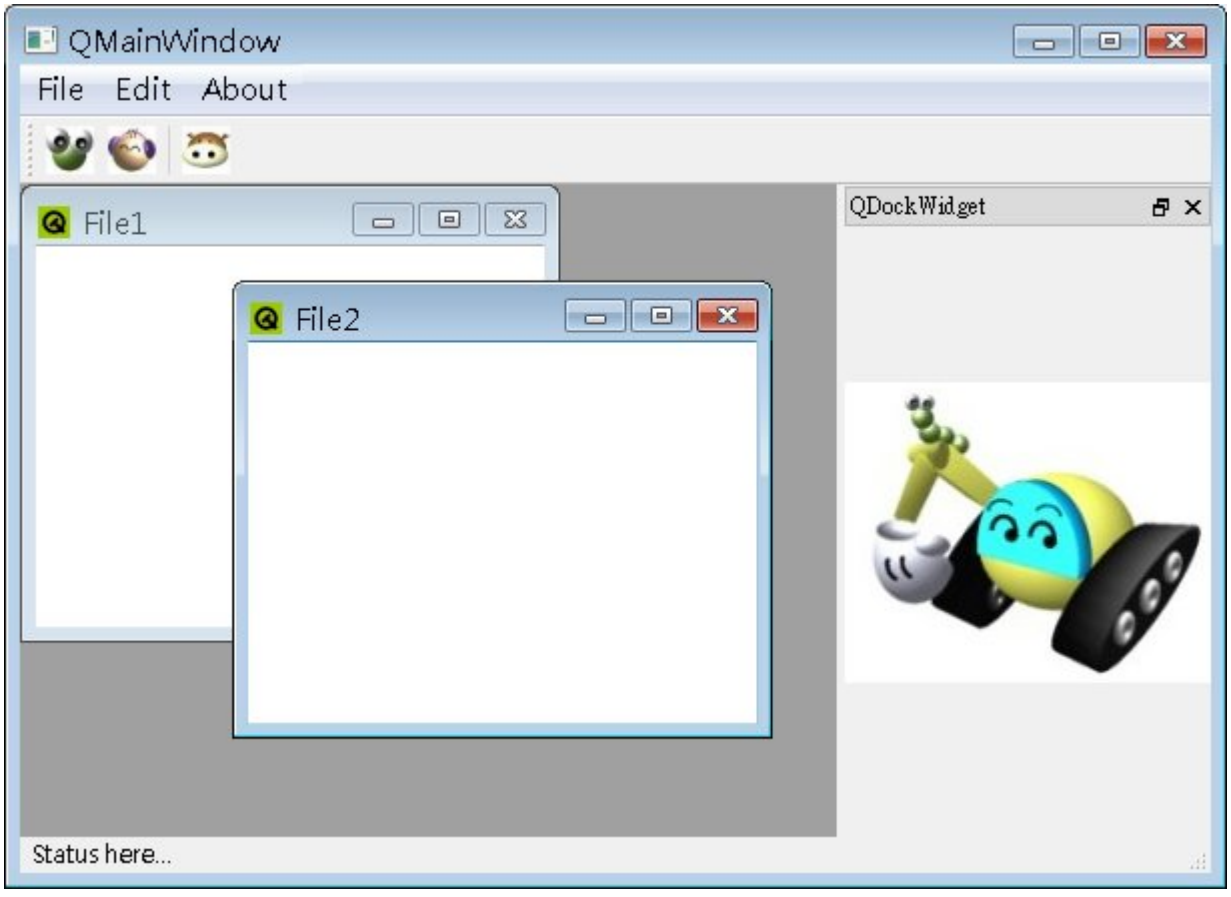
在 Qt 中要制作 MDI 接口的窗口，是使用 QMainWindow，并将其中心组件（Central Widget）设为 QMdiArea 实例，而每一个 MDI 子窗口，则使用 QMdiArea 的 addSubWindow()来加入。

下面这个程序以 QMainWindow 中的程序为基础，使用 QMdiArea 修改为 MDI 界面，当中会有两个 MDI 子窗口：

```
#include <QApplication>
#include <QMdiArea>
#include <QMainWindow>
#include <QTextEdit>
#include <QMenuBar>
#include <QAction>
#include <QToolBar>
#include <QStatusBar>
#include <QDockWidget>
#include <QLabel>
```

```
#include <QPixmap>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QMainWindow *mainWindow = new QMainWindow;
    mainWindow->setWindowTitle("QMainWindow");

    QMdiArea *mdiArea = new QMdiArea;
    mainWindow->setCentralWidget(mdiArea);

    // 文字编辑区
    QTextEdit *textEdit = new QTextEdit;
    textEdit->setWindowTitle("File1");
    mdiArea->addSubWindow(textEdit);
    textEdit = new QTextEdit;
    textEdit->setWindowTitle("File2");
    mdiArea->addSubWindow(textEdit);
    // 蹦现选单
    // 余下程序相同....
    return app.exec();
}
```

下图为执行时的参考画面：



5.67QSplashScreen

在应用程序启动时，可以显示启动画面（Splash Screen）来显示应用程序目前的启动进度，这可以使用 QSplashScreen 来达成，您可以简单的显示一个图片与讯息，或是更复制的制作进度列来显示目前应用程序的加载进度。

下面这个程序以 QMdiArea 为基础，为其加上启动画面：

```
#include <QApplication>
#include <QMdiArea>
#include <QMainWindow>
#include <QTextEdit>
#include <QMenuBar>
#include <QAction>
#include <QToolBar>
#include <QStatusBar>
#include <QDockWidget>
#include <QLabel>
#include <QPixmap>
#include <QSplashScreen>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QSplashScreen *splash = new QSplashScreen;
    splash->setPixmap(QPixmap("caterpillar.jpg"));
    splash->show();

    splash->showMessage("Starting gossip window...",
                        Qt::AlignRight | Qt::AlignTop, Qt::black);

    QMainWindow *mainWindow = new QMainWindow;
    // ... 中间相同

    mainWindow->show();

    splash->finish(mainWindow);
    delete splash;

    return app.exec();
}
```

QSplashScreen 可以使用 show()方法显示出来，showMessage()可以设定讯息显示，并可以指定讯息显示的位置，在程序中设定显示在右上角，消息正文为黑色。

您可以使用 QSplashScreen 的 close()方法关闭启动画面，若使用 finish()方法，则会在所指定的组件显示出来之后，才关闭启动画面，由于启动画面不再需要，所以最后使用 delete 将之从内存中删除以释放内存空间。

下图为启动画面撷图：

第六章 常用 API

6.1QString、容器组件

6.11QString

在 C++中有两种字符串的表现方式，一种是 C 风格的字符串，也就是最后以''\0 结尾的字符数组，另一种是 string 型态 的字符串。

在 Qt 中，提供 QString 作为字符串的表现形式，使用 16 位 Unicode 来表现 QString 中的字符，可以在字符串中包括'\0'字符，建立 QString 实例并赋予初值：

```
QString str = "caterpillar";
cout << str.length() << endl;
```

在 Qt 中会自动把"caterpillar"包装为 QString 实例，length()方法可以传回字符串的字符长度，以上例来说，将会传回"caterpillar"的长度为 11，如果要 QString 转为字符数组，也就是 char *的形式传回，则要使用 toAscii()或 toLatin1()方法传回 QByteArray，QByteArray 包括位数组，您可以 data() 方法传回 char *或使用 constData()传回 const char*，例如：

```
cout << str.toAscii().data() << endl;
```

由 QByteArray 的 data()或 constData()传回的 char*是 QByteArray 所拥有，QByteArray 使用完毕后，会负责 char*的内存回收。Qt 提供一个宏函式 qPrintable()，可以让您直接执行 toAscii().constData()，例如：

```
cout << qPrintable(str) << endl;
```

您也可以使用 QString 的 toStdString()，这会传回<iostream>的 std::string。

QString 可以使用 append()来附加字符串，也可以直接使用+、+=运算子来串接""包括的字符串，或是 QString 的实例，例如：

```
QString str = "1234";
cout << &str << endl;
str += str;
cout << qPrintable(str) << endl;
str = str + "5678";
cout << qPrintable(str) << endl;
```

您可以使用 QString 的 at()方法来传回每个字符的 QChar 对象，也可以直接使用[]运算子，使用 toAscii()或 toLatin1()可以取得 char，例如：

```
QString str = "ABCD";

for(int i = 0; i <str.length(); i++) {
    cout << str.at(i).toAscii() << endl;
}

for(int i = 0; i <str.length(); i++) {
    cout << str[i].toAscii() << endl;
}
```

可以使用=指定运算子来复制字符串：

```
QString str1 = "1234";
QString str2 = str1;
```

您可以直接使用==、!=来比较两个 QString 的字符内容是否相同，大小写是有区分的，或是使用>、>=、<、<=等比较运算，例如：

```
QString str1 = "ABC";
QString str2 = "ABC";

cout << (str1 == str2) << endl;
cout << (str1 != str2) << endl;
cout << (str1 > str2) << endl;
```

```
cout << (str1 < str2) << endl;
```

QString 有类似 C 语言 printf()函式的 sprintf()，可用于格式化字符串，例如以下将显示 "caterpillar:1975-5-26"：

```
QString str;
str.sprintf("%s:%d-%d-%d", "caterpillar", 1975, 5, 26);
cout << qPrintable(str) << endl;
```

另一个更简单的方式是使用占位字符配合 arg()方法，不用特别指定格式指定字，例如以下的显示结果与上面是相同的：

```
QString str =
    QString("%1:%2-%3-%4").arg("caterpillar").arg(1975).arg(5).arg(26);
cout << qPrintable(str) << endl;
```

要将数值转为 QString 实例，可以使用静态的 number()方法或是使用对象的 setNum()方法，例如：

```
QString str2 = QString::number(100);
```

要将字符串转为数值，可以使用 QString 的 toInt()、toDouble()等方法，如果无法顺利转换，将传回 0，例如：

```
QString str = "1975";
int number = str.toInt();
```

QString 本身还拥有许多的方法可供操作，像是 toUpper()可将字符转为大写，toLower()可将字符转为小写，replace()方法可指定置换子字符串，split()方法可以进行字符串分割并传回 QStringList，当中包括切割过后的各个 QString 对象，startsWith ()测试字符串是否以某个子字符串作为开头，endWith()测试字符串是否以某个子字符串作为结束。

关于 QString 上各种可用的方法，可以直接参考 Qt 的在线文件有关于 QString 的说明。

6.12 循序容器（QVector、QLinkedList、QList...）

QVector、 QLinkedList 与 QList 是 Qt 所提供的几个常用容器类别。QVector 将项目（item）储存在邻接的内存空间之中，提供基于索引（index-based）存取方式的容器类别。QLinkedList 以链接（Linked）的方式储存项目，提供基于迭代器（iterator- based）存取方式的容器类别。QList 提供基于索引的快速存取容器类别，内部使用指针数组，可提供

快速插入及移除项目。

首先来看看 QVector 的基本使用方式，建立一个可容纳两个元素的 QVector，并使用索引方式存取元素值：

```
QVector<double> vect(2);
vect[0] = 1.0;
vect[1] = 2.0;
for (int i = 0; i < vect.count(); ++i) {
    cout << vect[i] << endl;
}

for (int i = 0; i < vect.count(); ++i) {
    cout << vect.at(i) << endl;
}
```

要使用索引方式设定元素，必须先配置好够长的空间，否则会发生超出索引范围的错误，使用[]运算子指定索引存取的方式是比较方便，但在某些场合下，使用 at()方法会较有效率一些，这涉及 Qt 的隐式共享机制，稍后再作介绍。

您也可以使用 QVector 的 append()方法来加入元素，使用 remove()方法来移除元素，使用 insert()方法来插入元素，例如 append()的使用如下：

```
vect.append(3.0);
vect.append(4.0);
```

或者是使用<<运算子附加元素：

```
vect << 5.0 << 6.0;
```

QVector 也重载了一些其它的运算子，以及提供了一些其它可用的方法，请查询 Qt 在线文件有关于 QVector 的介绍。QVector 提供的是邻接的内存空间以存取对象，所以对于循序存取或使用索引，效率较高，但如果要插入或移除元素时，效率就会低落。QVector 的子类别 QStack 提供了 push()、pop()与 top()等方法，方便您进行堆栈结构的对象管理。

对于需要经常要在容器中插入或移除组件，您可以使用 QLinkedList 以提高存取效率，它不提供基于索引的存取方式，而是基于迭代器的存取方式，稍后会介绍迭代器的使用，以下先来看看 QList。

QList 提供的是基于索引的存取方式，其内部实作使用了指针数组，数组中每个指针指向所要储存的元素，结合了 QVector 与 QLinkedList 的优点，提供快速存取与插入、移除，其索

引存取方式或可用的方法与 QVector 是类似的，也可以使用<<运算子来附加元素，例如：

```
QList<QString> list;
list << "caterpillar" << "momor" << "bush";

for(int i = 0; i < list.size(); ++i) {
    cout << list[i].toAscii().data() << endl;
}
cout << endl;

for(int i = 0; i < list.size(); ++i) {
    cout << list.at(i).toAscii().data() << endl;
}
cout << endl;
```

QList 的子类别 QStringList 为 Qt 中应用很广的类别，可以让您储存 QString 对象，QList 的子类别 QQueue 则提供了队列结构的容器管理。

以上先列出 QVector、QLinkedList 及 QList 的使用比较：

如果想要有连续邻接的内存空间来存放组件，则使用 QVector。
如果需要真正的链接数据结构，并使用基于迭代器的存取方式，则使用 QLinkedList。
在大部份情况下，QList 可以满足快速存取、插入、移除的需求，并可提供基于索引的存取方式。

再来看看迭代器于容器类别的使用，对于容器类别，Qt 提供两种风格的迭代器：Java 风格与 STL 风格。Java 风格的迭代器使用上就如何 Java 的迭代器，使用这种迭代器对于 Java 开发人员较为容易，然而 STL 风格的迭代器则提供更有弹性的操作。

以下简单示范在 QList 上使用 Java 风格迭代器：

```
QList<QString> list;
list << "caterpillar" << "momor" << "bush";

QListIterator<QString> iterator(list);
while (iterator.hasNext()) {
    cout << iterator.next().toAscii().data() << endl;
}
```

与 Java 迭代器类似的，hasNext()测试是否有下一个元素，next()传回下一个元素，其它还有 hasPrevious()、previous()等方法可以使用。Java 风格的迭代器有两种：只读与可擦

写。QListIterator 是只读迭代器，对于可擦写迭代器，命名上会加上 Mutable，例如 QMutableListIterator,除了 next()、previous()等方法之外,还提供了 insert()、remove()等方法可以使用，例如：

```
QLinkedList<QString> list;
list << "caterpillar" << "momor" << "bush";

QMutableLinkedListIterator<QString> rwIterator(list);
while (rwIterator.hasNext()) {
    if(rwIterator.next() == "momor") {
        rwIterator.insert("bee");
        break;
    }
}

QLinkedListIterator<QString> rIterator(list);
while (rIterator.hasNext()) {
    cout << rIterator.next().toAscii().data() << endl;
}
```

上面这个程序片段，使用可擦写迭代器来于"momor"之后插入一个"bee"，之后使用只读迭代器读出数据。

您可以使用容器类别的 begin()方法传回基于 STL 的迭代器,它指向容器的第一个元素地址，end()方法则传回指向容器最后一个元素之后的地址。您可以如下使用基于 STL 的迭代器：

```
QList<QString> list;
list << "caterpillar" << "momor" << "bush";

QList<QString>::const_iterator i = list.begin();
while (i != list.end()) {
    cout << (*i).toAscii().data() << endl;
    ++i;
}
```

STL 风格的迭代器一样有两种，C<T>::const_iterator 形式的迭代器宣告为只读，则可以读取数据，不可修改数据，C<T>::const_iterator 形式的迭代器则可以修改数据，例如：

```
QList<QString> list;
list << "caterpillar" << "momor" << "bush";

QList<QString>::iterator i = list.begin();
```

```
while (i != list.end()) {
    (*i) = (*i) + ".onlyfun";
    ++i;
}
```

对于简单的循序存取，Qt 提供了 foreach 虚拟关键词（pseudo-keyword），以标准的 for 循环实作，例如您可以如下循序取出 QList 中的元素：

```
QList<QString> list;
list << "caterpillar" << "momor" << "bush";

foreach(QString str, list) {
    cout << str.toAscii().data() << endl;
}
```

接下来进一步说明前面所提及，Qt 的容器类别有个称之为隐式共享（implicit sharing）的机制，又称之为 copy on write，顾名思义，就是在数据有变动的情况下，才进行容器内数据结构对象之复制，否则容器内数据结构对象是共享的。

举个例子来说，如果您使用[]运算子方式，则会进行容器内数据结构对象之复制：

```
QList<QString> list1;
list1 << "x";
QList<QString> list2 = list1;

cout << &list1[0] << endl;
cout << &list2[0] << endl;
```

在上例中，使用了[]运算子，list1 与 list2 内部的数据结构对象经过复制，并不是共享的，所以显示出来的两个内存地址并不相同，但使用 at()时的情况则是相同的：

```
QList<QString> list1;
list1 << "x";
QList<QString> list2 = list1;

cout << &(list1.at(0)) << endl;
cout << &(list2.at(0)) << endl;
```

所以在只读的情况下，建议使用 at()方法而不是[]运算子的方法，以避免容器内部数据结构对象复制的成本，藉以获得较好的效率。

隐式共享又称之为 copy on write，是因为在容器中的数据有变动时，就不再共享内部数据结构对象，可以从下面的程序代码看出：

```
QList<QString> list1;
list1 << "x";
QList<QString> list2 = list1;

// 以下两行显示相同的内存地址
cout << &(list1.at(0)) << endl;
cout << &(list2.at(0)) << endl;

// 对 list2 作变动
list2 << "y";

// 以下两行显示不同的内存地址
cout << &(list1.at(0)) << endl;
cout << &(list2.at(0)) << endl;
```

Qt 容器的隐式共享机制之一，让您可以用较简明的方式来撰写程序，例如您可以如下撰写一个函式：

```
QList<QString> doSomething() {
    QList<QString> list;
    // ...blah..blah
    cout << &list << endl;
    return list;
}
```

然后就直接如下撰写程序以利用这个函式，但不会进行对象复制：

```
QList<QString> list = doSomething();
cout << &list << endl;
```

上面的程序代码片段，在 doSomething()中的 list 地址与呼叫 doSomething()中 list 的地址会是相同的。无论是 Java 风格或 STL 风格的迭代器，使用只读迭代器时，背后也都会使用到隐式分享机制，以增加循序读取的效率。

容器类别的值可以是基本数据型态、指针或对象，对象必须有预设建构子、复制建构子与指定运算子（C++预设的复制及指定运算亦可）。

6.13 关联容器（QMap、QHash...）

关联容器（Associative Container）为 Key/Value 匹配的容器，Qt 提供的关键之一为 QMap，可根据 Key 来快速取得相对应的 Value。

您可以使用 QMap 的 insert()方法指定 Key/Value 插入 QMap 中，使用 QMap 的 value()指定 Key 来取得对应的 Value，例如：

```
QMap<QString, QString> map;

map.insert("caterpillar", "caterpillar's message!");
map.insert("momor", "momor's message!");

cout << map.value("caterpillar").toAscii().data() << endl;
cout << map.value("momor").toAscii().data() << endl;
```

也可以直接使用[]及=来指定 Key/Value，或是使用[]根据 Key 取得 Value，使用上就如同关联数组，例如：

```
QMap<QString, QString> map;

map["caterpillar"] = "caterpillar's message!";
map["momor"] = "momor's message!";

cout << map["caterpillar"].toAscii().data() << endl;
cout << map["caterpillar"].toAscii().data() << endl;
```

使用 QMap 的[]或 value()方法指定 Key 来取得 Value 时，如果没有该 Key，则会根据 Value 类别的预设建构子产生一个新对象并传回，如果是基本数据型态则或指针则传回 0，不过 value()有另一个版本，也可以指定当 Key 不存在时，传回的默认值，例如：

```
QString message = map.value("bush", "N.A.");
```

假设 map 中原先并没有储存"bush"作为 Key 的相对应 Value，则上式将会传回"N.A"的 QString。QMap 中的 Key/Value 可以放置基本数据型态、对象、指针，对象必须有预设建构子、复制建构子与指定运算子，QMap 中的排序是根据 Key 作递增排序，所以 Key 的部份还必须提供<()运算子以判断 Key 的排序先后。

如果想要迭代 QMap 中的 Key/Value，可以使用 keys()与 values()方法，它们各传回 QList 对象，当中各包括 Key 与 Value：

```
QMap<QString, QString> map;

map["caterpillar"] = "caterpillar's message!";
map["momor"] = "momor's message!";
map["bush"] = "bush's message!";

QList<QString> keys = map.keys();
foreach(QString key, keys) {
    cout << key.toAscii().data() << " " << endl;
}
cout << endl;

QList<QString> values = map.values();
foreach(QString value, values) {
    cout << value.toAscii().data() << " " << endl;
}
cout << endl;
```

您也可以直接使用 QMapIterator 来进行迭代，直接来看个实例：

```
QMap<QString, QString> map;

map.insert("caterpillar", "caterpillar's message!");
map.insert("momor", "momor's message!");
map.insert("bush", "bush's message!");

QMapIterator<QString, QString> iterator(map);
while(iterator.hasNext()) {
    iterator.next();
    cout << iterator.key().toAscii().data() << endl;
    cout << iterator.value().toAscii().data() << endl;
}
```

QMapIterator 是只读迭代器，如果在迭代过程中想要修改，则可以使用 QMutableIterator。一般来说，Map 中的 Key/Value 是一对一，您如果在 QMap 中基于同一个 Key 插入了两个不同的 Value，则后者会覆盖前者，但 QMultiMap 让您可以在一个 Key 上指定多个 Value，它是 QMap 的子类别，可使用 values()方法传回某 Key 所对应的数个 Value，例如：

```
QMultiMap<QString, QString> map;

map.insert("caterpillar", "caterpillar's message1!");
map.insert("caterpillar", "momor's message2!");
```

```
QList<QString> values = map.values("caterpillar");
```

与 QMap 作用类似的是 QHash，它是基于杂凑表（Hash table）的关联容器，使用接口与 QMap 也几乎相同，但 QHash 是无序的，其内部杂凑表会自动依需要自动增长，您可以使用 reserve() 来保留杂凑表大小，或是 squeeze() 来缩减杂凑表大小。

QHash 中的 Key/Value 可以放置基本数据型态、对象、指针，对象必须有预设建构子、复制建构子与指定运算子（基本上，这是放至容器中的对象之基本需求），QHash 的 Key 是基于杂凑表，所以 Key 还必须提供==运算子，以及全域的 qHash()函式，以计算出 Key 的杂凑值，Qt 本身已提供许多基本的 qHash()函式，可以协助您计算 Key 的杂凑值，可以参考 Qt 的 QHash 类别说明，当中有定义的范例可以参考。

如果想要基于一个 Key 储存多个 Value，则可以使用 QHash 的子类别 QMultiHash。

如果只想要储存 Key，则可以使用 QSet。

6.14 泛型演算（Generic Algorithms）

Qt 的全域函式中，包括了一些适用 Qt 容器的泛型演算函式，像是 qFind()可以指定容器直接进行搜寻，也可以指定起始与结束迭代器地址，进行指定值搜寻，若搜寻到则传回该值的迭代器地址，若没有找到，则传回容器 end()方法的迭代器地址。qSort()可以对容器进行排序。qBinaryFind()可以以二元搜寻的方式搜寻已递增排序容器中指定的值。例如：

```
QStringList list;
list << "caterpillar" << "momor" << "bush" << "justin";

// 以下显示 momor、bush、justin
QStringList::iterator i = qFind(list.begin(), list.end(), "momor");
while(i != list.end()) {
      cout << (*i).toAscii().data() << endl;
      ++i;
}
cout << endl;

// 排序
qSort(list.begin(), list.end());

// 以下显示 momor
QStringList::iterator j = qBinaryFind(list.begin(), list.end(), "momor");
while(j != list.end()) {
         cout << (*j).toAscii().data() << endl;
      ++j;
```

```
}
```

qSort()预设使用递增排序，如果要以递减排序的方式，则可以传入 qGreater<T>()，类似的，qBinaryFind()也可以使用 qGreater<T>()指定在递减排序的容器中进行二元搜寻，例如：

```
    QStringList list;
    list << "caterpillar" << "momor" << "bush" << "justin";

     // 递增排序
    qSort(list.begin(), list.end(), qGreater<QString>());

    // 以下显示 momor、justin、caterpillar、bush
    QStringList::const_iterator iterator = list.begin();
    while(iterator != list.end()) {
        cout << (*iterator).toAscii().data() << endl;
        ++iterator;
    }


    QStringList::iterator j = qBinaryFind(list.begin(), list.end(),
                                          "caterpillar", qGreater<QString>());
    // 以下显示 caterpillar、bush
    while(j != list.end()) {
          cout << (*j).toAscii().data() << endl;
        ++j;
    }
```

您也可以自定义排序规则，例如依字符串长度进行排序：

```
bool lengthLessThan(const QString &str1, const QString &str2) {
    return str1.length() < str2.length();
}
```

然后再指定给 qSort()，例如：

```
QStringList list;
list << "caterpillar" << "momor" << "bush" << "justin";

qSort(list.begin(), list.end(), lengthLessThan);

// 以下显示 bush、momor、justin、caterpillar
QStringList::const_iterator iterator = list.begin();
```

```
while(iterator != list.end()) {
    cout << (*iterator).toAscii().data() << endl;
    ++iterator;
}
```

与 qSort()类似的 qStableSort()，则是在比较两个项目相同时，会保留两个项目原有的先后顺序。

在这边仅列出几个泛型演算的例子，更多泛型演算的函式，可以查询 Qt 的 <QtAlgorithms> - Generic Algorithms 中之介绍。

6.2 档案处理

6.21QFile

QIODevice 是 Qt 中关于输入输出的基础类别，其中关于档案写入与读取的子类别是 QFile，您可以使用 exists()测试档案是否存在，使用 size()来取得档案大小，使用 remove()来移除档案，使用 open()开启档案，使用 readLine()读取档案，使用 flush()确定写出所有的数据，使用 close()关闭档案等，如果在 QFile 离开呼叫的范围之后，QFile 也会自动关闭档案。

QFile 在开启档案的时候，可以设定开启模式（OpenMode），例如 QIODevice::ReadOnly、QIODevice:: WriteOnly、QIODevice::Append 或 QIODevice::ReadWrite 等，您可以使用 QFile 来进行档案读取，但 QFile 提供的是较低阶的接口，通常会搭配 QTextStream 或 QDataStream，在使用上较为方便，前者适用于纯文字数据的读取，后者为二进制数据的存取。

以下的程序，直接使用 QFile 来进行档案复制，程序使用 QFile 以只读方式读取指定的来源档案，以 readAll()方式读入数据为 QByteArray，以唯写方式写入指定的目的档案，将读入的 QByteArray 使用 write()写入档案，以完成复制的动作：

```
#include <QFile>
#include <QString>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    QString src(argv[1]);

    QFile srcFile(src);
    if (!srcFile.open(QIODevice::ReadOnly)) {
        cerr << "Cannot open file for reading:"
             << qPrintable(srcFile.errorString()) << endl;

        return false;
```

```
    }

    QString dest(argv[2]);

    QFile destFile(dest);
    if (!destFile.open(QIODevice::WriteOnly)) {
        cerr << "Cannot open file for writing: "
                << qPrintable(destFile.errorString()) << endl;

        return false;
    }

    QByteArray in = srcFile.readAll();

    destFile.write(in);

    return srcFile.error() == QFile::NoError
            && destFile.error() == QFile::NoError;
}
```

QFile 的 error()传回代码，表示档案的读取或写入过程中是否有误。程序执行时指定命令列自变量如下以进行档案复制：

```
qcopy caterpillar.jpg caterpillar_backup.jpg
```

6.22QTextStream

对于文字数据的读取，可以使用 QTextStream 作为方便的操作接口，它适用于 QIODevice 的子类别、QByteArray、QString 等，可以使用 << 与 >> 运算子进行数据的读取与写入，QTextStream 只是对 QIODevice、QByteArray、QString 的装饰（Decorator），提供方便的操作。

下面的程序先示范 QTextStream 于 QFile 上的运用，程序可以让您使用命令列自变量指定要读取的档案，并显示在主控台中：

```
#include <QFile>
#include <QTextSTream>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    QString src(argv[1]);

    QFile file(src);

    if (!file.open(QIODevice::ReadOnly)) {
        cerr << "Cannot open file for reading:"
```

```
                << qPrintable(file.errorString()) << endl;

        return false;
    }

    QTextStream in(&file);

    while (!in.atEnd()) {
        cout << qPrintable(in.readLine()) << endl;
    }

    return true;
}
```

程序中用 QTextStream 包装 QFile，使用 QTextStream 的 atEnd()方法测试是否取档案结束，使用 readLine()读入数据并包装为 QString 实例。

如果想要写入文字至档案，可以使用 << 运算子，例如：

```
#include <QFile>
#include <QTextSTream>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    QFile file("data.txt");

    if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
        cerr << "Cannot open file for writing:"
             << qPrintable(file.errorString()) << endl;

        return false;
    }

    QTextStream out(&file);

    out << "name\tscore" << endl;
    out << "justin\t" << 95 << endl;
    out << "momor\t" << 93 << endl;
    out << "minnie\t" << 93 << endl;

    return true;
}
```

在开启档案时，如果是想要附加文字内容至档案，记得设定开启模式为 QIODevice::Append。设定开启模式为 QIODevice::Text，这在读取文本文件时，换行字符会置换为'\n'，在写入文本文件时，换行字符会置换为平台相依字符，例如 Windows 平台的'\r\n'。

在上面的程序中可以看到，您可以直接使用 << 并搭配 C++标准函式库的 I/O 格式控制器或 I/O 格式化旗标，例如以下将显示 123456 的 16 进位的数字 3039：

```
out << hex << 12345 << endl;
```

您也可以直接使用 QTextStream 的 setIntegerBase(16)来设定相同的效果，另外还可以使用 setXXXFlags()方法来设置 showbase 等格式控制，详细设定方式，可以查询 QTextStream 类别的文件说明。

在读入档案的时候，可以使用 >> 运算子，不过要注意的是，读取时以空白为区隔，例如若使用以上的程序写入档案，在使用以下的程序片段读取时，结果 str1 会是"name"，str2 会是"score"：

```
QTextStream in(&file);
QString str1, str2;
in >> str1 >> str2;
cout << qPrintable(str1) << endl;
cout << qPrintable(str2) << endl;
```

之前说过，QTextStream 可以使用于 QIODevice、QString 等之上，例如以下的程序片段中，str 结果将储存"caterpillar.onlyfun"：

```
QString str;
QTextStream in(&str);
in << "caterpillar" << "." << "onlyfun";
cout << qPrintable(str) << endl;
```

预设上，QTextStream 会使用系统的预设编码作为读取写入文字数据时的编码，您也可以使用 setCodec()方法来设置读取写入文字时的编码，例如：

```
QTextStream stream;
stream.setCodec("UTF-8");
```

6.23QDataStream

对于纯碎的二进制数据，可以使用 QDataStream 来协助处理，可以直接处理 C++基本数据型态、还有许多 Qt 数据型态，像是 QByteArray、QString、QMap 等，可以使用 << 或 >> 运算子来进行数据输出或写入。

先使用以下的简单例子，示范一下 QDataStream 搭配 QFile 来进行档案读写：

```
#include <QFile>
```

```
#include <QDataStream>
#include <QString>
#include <QMap>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
	QFile file("data.dat");

	QMap<QString, int> map;
	map.insert("caterpillar", 95);
	map.insert("momor", 93);

	if(!file.open(QIODevice::WriteOnly)) {
		cerr << "Cannot open file for writing: "
			<< qPrintable(file.errorString()) << endl;
		return false;
	}

	QDataStream out(&file);
	// 设定 QDataStream 支持版本
	out.setVersion(QDataStream::Qt_4_3);

	// 写入资料
	out << 1 << map;
	file.close();

	if(!file.open(QIODevice::ReadOnly)) {
		cerr << "Cannot open file for reading: "
			<< qPrintable(file.errorString()) << endl;
		return false;
	}

	QDataStream in(&file);
	// 设定 QDataStream 支持版本
	in.setVersion(QDataStream::Qt_4_3);

	int num = 0;
	QMap<QString, int> inMap;

	// 读入资料
	in >> num >> inMap;

	cout << "num: " << num << endl
```

```
        << "map: <caterpillar, " << inMap.value("caterpillar") << ">" << endl
        << "map: <momor, " << inMap.value("momor") << ">" << endl;

    return true;
}
```

程序中可以看到 setVersion()方法，这设定 QDataStream 读写时的版本，因为 Qt 的对象成员等数据，会随着不同版本而可能有所不同，例如 QMap 新版中可能有一些成员属性是旧版本所没有的，使用 setVesrion()设定 Qt 支持的读写版本，告诉 QDataStream 在写入或读取时应当处理的对象数据。

程序执行时的结果如下所示：

```
num: 1
map: <caterpillar, 95>
map: <momor, 93>
```

QDataStream 也可以直接处理位数据，例如使用 readRawBytes()与 writeRawBytes()来进行原始位数据的处理。QDataStream 处理数值时，预设使用 big- endian 的方式，如果您要改变为使用 little-endian，则可以使用 setByteOrder()方法设定为 QDataStream:: LittleEndian。

如果想要 QDataStream 可以使用 << 或 >> 来支持您的自订义对象，则您需要重载 << 与 >> 运算子，告诉 QDataStream 如何储存或读取对象，例如：

```
#include <QFile>
#include <QDataStream>
#include <QString>
#include <iostream>
using namespace std;

class Dog {
public:
    Dog() { _number = 0; }

    Dog(int number, const QString &name) {
        _number = number;
        _name = name;
    }

    void setNumber(int number) { _number = number; }
    int number() const { return _number; }

    void setName(const QString &name) { _name = name; }
    QString name() const { return _name; }

private:
    int _number;
```

```
    QString _name;
};

QDataStream &operator<<(QDataStream &out, const Dog &dog) {
    out << dog.number() << dog.name();
    return out;

}

QDataStream &operator>>(QDataStream &in, Dog &dog) {
    int number = 0;
    QString name;

    in >> number >> name;

    dog.setNumber(number);
    dog.setName(name);

    return in;
}


int main(int argc, char *argv[]) {
    QFile file("data.dat");

    Dog dog1(1, "caterpillar");
    Dog dog2(2, "momor");

    if(!file.open(QIODevice::WriteOnly)) {
        cerr << "Cannot open file for writing: "
            << qPrintable(file.errorString()) << endl;
        return false;
    }

    QDataStream out(&file);
    out.setVersion(QDataStream::Qt_4_3);
    out << dog1 << dog2;
    file.close();

    if(!file.open(QIODevice::ReadOnly)) {
        cerr << "Cannot open file for reading: "
            << qPrintable(file.errorString()) << endl;
        return false;
    }
```

```
    QDataStream in(&file);
    in.setVersion(QDataStream::Qt_4_3);

    in >> dog1 >> dog2;

    cout << dog1.number() << ", " << qPrintable(dog1.name()) << endl
         << dog2.number() << ", " << qPrintable(dog2.name()) << endl;

    return true;
}
```

程序执行时的结果如下所示：

```
1, caterpillar
2, momor
```

如以上重载 << 与 >> 运算子，还有一个好处，就是可以让自定义对象支持 QList 等容器之 << 与 >> 之附加与取出，例如像以下的操作：

```
QList<Dog> list;

Dog dog1(1, "caterpillar");
Dog dog2(2, "momor");

list << dog1 << dog2;

QList<Dog>::const_iterator iterator = list.begin();

while(iterator != list.end()) {
    cout << (*iterator).number() << ", "
         << qPrintable((*iterator).name()) << endl;
    ++iterator;
}
```

6.24QFileInfo 与 QDir

QFileInfo 顾名思义，就是用来取得指定的档案之相关讯息，像是相对或绝对路径信息、隐藏属性、大小、最后更新等，为了加快存取档案的信息，QFileInfo 会快取信息，如果档案在快取信息之后，被使用者作了修正，则可以使用 refresh()更新 QFileInfo 的信息。QFileInfo 可以是符号链接(Symbol Link)、目录或档案，由 isFile()、isDir()与 isSymLink()来作判断。

QDir 则可以让您指定目录，以取得所指定目录或其中的项目信息（档案、目录等），您可以设定名称过滤、属性过滤（像是只读、档案或是目录等等）与排序，过滤与排序可以使用 setNameFilter()、setFilter()与 setSorting()方法来设定，您可以使用 entryList()来取得一个目录下的所有档案与子目录字符串名称，或是使用 entryInfoList()传回

QFileInfoList，当中包括目录中所有项目的 QFileInfo 指针。

跟目录有关的操作，也可以透过 QDir 来达成，例如使用 mkdir()方法可以建立一个新的目录，使用 rename()方法来更改目录名称，使用 rmdir()方法将一个已经存在的目录移除，使用 remove()方法来移除档案，您可以使用 exists()、 isReadable()与 isRoot()方法来测试目录。根目录的显示是由 drives()提供；在 Unix 系统下这会传回包括根目录"/"的列示；在 Windows 下则通常包括"D:/"等。 如果您想要路径的形式是基于操作系统的适当形式，使用 convertSeparetors()。

以下这个程序使用 QDir 与 QFileInfo 来查询目前目录下的项目信息：

```
#include <QDir>
#include <QFileInfo>
#include <QFileInfoList>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    QDir d;
    d.setFilter( QDir::Files | QDir::Hidden | QDir::NoSymLinks );
    d.setSorting( QDir::Size | QDir::Reversed );

    const QFileInfoList list = d.entryInfoList();
    QFileInfoList::const_iterator iterator = list.begin();

    cout << "Filename\t\tSize" << endl;
    while ( iterator != list.end() ) {
        cout << qPrintable((*iterator).fileName()) << "\t"
            << (*iterator).size() << endl;
        ++iterator;
    }

    return 0;
}
```

一个查询指定目录下所有子目录与档案的程序，可以参考 QTreeWidget 与 QTreeWidgetItem。

6.25Qt 资源系统

Qt 资源系统（Resource System）可以提供与平台无关的机制，让您把应用程序的图文件、语系文件、数据等储存于可执行档之中，避免相关的资源文件遗失的问题，Qt 资源系统是基于 qmake、rcc（resource compiler），并搭配 QFile 来使用，您必须在产生的.pro 档之中，告知资源群集文件（Resource Collection File）的位置与名称。

资源群集档的扩展名为.qrc，实际的内容为 XML 格式的档案，当中告知了这个应用程序所

要使用到的资源文件，例如您想要将 QListWidget 与 QListWidgetItem 中所使用到的图档储存在可执行档案之中，则可以撰写一个 resourcefile.qrc：

resourcefile.qrc

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/bee_head.jpg</file>
    <file>images/bush_head.jpg</file>
    <file>images/cat_head.jpg</file>
    <file>images/caterpillar_head.jpg</file>
    <file>images/momor_head.jpg</file>
</qresource>
</RCC>
```

档案的路径是相对于.qrc 档案的位置。接着您要在产生的.pro 档案中，增加一行，告知.qrc 档案的位置，例如：

```
RESOURCES = resourcefile.qrc
```

qmake 会产生出制造 qrc_resourcefile.cpp 的规则，之后使用 rcc 产生.cpp 档案，当中会将想要嵌入的相关档案，压缩并转换为代表二进制数据的 C++静态无号字符数组，如果您的.qrc 档案内容有变动，在编译时.cpp 档案也会重新产生。

如果要使用嵌入的资源，则要在路径的前端放置:/，例如：

```
QListWidget *listWidget = new QListWidget;
listWidget->insertItem(0, new QListWidgetItem(
                QIcon(":/images/caterpillar_head.jpg"), "caterpillar"));
listWidget->insertItem(1, new QListWidgetItem(
                QIcon(":/images/momor_head.jpg"), "momor"));
listWidget->insertItem(2, new QListWidgetItem(
                QIcon(":/images/bush_head.jpg"), "bush"));
listWidget->insertItem(3, new QListWidgetItem(
                QIcon(":/images/bee_head.jpg"), "bee"));
listWidget->insertItem(4, new QListWidgetItem(
                QIcon(":/images/cat_head.jpg"), "cat"));
```

您也可以为资源文件的路径设置别名（Alias），例如：

```
<file alias="caterpillar_head.jpg">images/caterpillar_head.jpg</file>
```

之后在程序中指定路径时，就可以直接使用别名，例如：

```
listWidget->insertItem(0, new QListWidgetItem(
                QIcon(":/caterpillar_head.jpg"), "caterpillar"));
```

您也可以为别名设置前置（Prefix），例如：

```
<qresource prefix="/resources">
    <file alias="caterpillar_head.jpg">images/caterpillar_head.jpg</file>
</qresource>
```

之后每个别名都会自动加上前置，使用时如下：

```
listWidget->insertItem(0, new QListWidgetItem(
                QIcon(":/resources/caterpillar_head.jpg"), "caterpillar"));
```

您也可以搭配语系来使用嵌入的资源档，例如若这么设定：

```
<qresource>
    <file>caterpillar_head.jpg</file>
</qresource>
<qresource lang="zh_TW">
    <file alias="caterpillar_head.jpg">caterpillar_head_zh_TW.jpg</file>
</qresource>
```

当路径指定为:/caterpillar_head.jpg，如果使用者是使用 zh_TW 语系，则会自动对应使用 caterpillar_head_zh_TW.jpg，此一方法也可以用来加载.qm 档案，以实现多国语系支持，可参考 翻译应用程序 与 多国语系选择与切换。

6.3 数据库

6.31Qt 的 MySQL 驱动程序

在 Qt 在线文件 SQL Database Drivers 中有提及如何建构 Qt 的数据库驱动程序，在这边简介一下，如何在 Windows 下使用 Qt OpenSource 4.3.3 自行编译 MySQL 驱动程序的 plugin。

(1)安装 MySQL 时必须有 Include Files / Lib Files 选项

首先是当您在安装 MySQL 的时候，必须选择自订安装，并选择安装 Include Files / Lib Files，安装完成时在 MySQL 安装目录下，会有 include 目录与 lib 目录，如果您先前没有选择安装 Include Files / Lib Files，则只要再执行一次 MySQL 安装程序，选择「Modify」项目，即可再增加 Include Files / Lib Files 的安装。

(2)复制 MySQL 的 include 与 lib 目录

若 MySQL 预设的安装目录名称中预设有空白（例如 MySQL Server 5.0 这样的名称），在编译驱动程序时会有问题,您可以建立一个 mysql 目录,例如 c:\mysql,然后将 MySQL 的 include 与 lib 目录复制至 c:\mysql 之中。

(3)下载 mingw-utils

下载 mingw-utils-0.3.tar.gz，将之解压缩，将其中的 reimp 复制至 MinGW 的 bin 目录中。

(4)使用 reimp 与 dlltool

开启文字模式主控台，执行以下指令：

```
cd c:\mysql\lib\opt
reimp -d libmysql.lib
dlltool -k -d libmysql.def -l libmysql.a
```

(5)编译驱动程序

在文字模式主控台中执行以下指令，%QTDIR%为您的 Qt 安装路径环境变量：

```
cd %QTDIR%\src\plugins\sqldrivers\mysql

qmake -o Makefile "INCLUDEPATH+=C:\mysql\include" "LIBS+=C:\mysql\lib\opt\libmysql.a" mysql.pro

make
```

完成以上的步骤并编译完成之后，可以在 Qt 安装目录中的 plugins\sqldrivers 目录中，找到编译好的 MySQL 驱动程序 plugin。

接下来可以编写程序测试数据库连结，Qt 的数据库支持是放置在 QtSql 模块之中，您可以使用 QSqlDatabase 的静态 addDatabase() 方法指定"QMYSQL"，这会加载驱动程序并传回 QSqlDatabase 实作对象，之后可以使用 setHostName()、 setDatabaseName()、setUserName()、setPassword()等方法，设定数据库的 URL 地址、数据库名称、使用者与密码，然后使用 open()方法开启联机，使用 close()方法关闭联机。

下面这个程序是个简单示范：

```
#include <QApplication>
#include <QtSql>
#include <QLabel>

bool createConnection() {
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");

    db.setHostName("localhost");
    db.setDatabaseName("demo");
    db.setUserName("root");
    db.setPassword("123456");

    if (!db.open()) {
        return false;
    }

    db.close();

    return true;
```

```
}

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QLabel *label = new QLabel;
    label->setWindowTitle("Qt Database");

    if(createConnection()) {
        label->setText("<h1>Connected to database!</h1>");
    }
    else {
        label->setText("<h1>Connection fail!</h1>");
    }

    label->show();

    return app.exec();
}
```

接着执行 qmake -project 产生.pro 档案，为了要连结 QtSql 模块，记得编辑.pro 档案，在当中加上一行：

```
QT += sql
```

接着就可以再次执行 qmake 产生 Makefile，执行 make 进行程序编译了。下图为程序执行时的参考画面：

6.32QSqlQuery

要在 Qt 中执行与处理 SQL，可以使用 QSqlQuery，假设您建立的表格如下：

```
create table T_USER (
     id bigint not null auto_increment,
     name varchar(255),
     age bigint,
     primary key (id)
);
```

您可以如下使用 QSqlQuery 执行 SQL 语句：

```
QSqlQuery query;
```

```
query.exec("INSERT INTO T_USER (name, age) "
                    "VALUES ('bush', 9)");
```

这会在指定的数据库表格中插入一笔数据，也可以直接在建构函式中指定 SQL 语句，这会在建构对象之后，直接执行 SQL 语法：

```
QSqlQuery query("INSERT INTO T_USER (name, age) "
                    "VALUES ('bush', 9)");
```

您可以使用 numRowsAffected()传回影响的笔数，例如：

```
QSqlQuery query;
query.exec("DELETE FROM T_USER WHERE name = 'bee'");
cout << query.numRowsAffected() << endl;
```

如果要查询数据，可以用 next()方法移至所查得的下一笔数据，如果可以找到下一笔数据则传回 true，否则传回 false，例如：

```
QSqlQuery query;
query.exec("SELECT * FROM T_USER");
while (query.next()) {
    int id = query.value(0).toInt();
    QString name = query.value(1).toString();
    int age = query.value(2).toInt();
    cout << id << " " << qPrintable(name) << " " << age << endl;
}
```

value()方法中的自变量，是您所指定的表格字段顺序，因为数据库表格中所储存的型态是未知的，value()方法传回的是 QVariants 型态的对象，QVariants 在 Qt 中用来持有各种不同型态的数据，从基本的 int 到对象型态都可以，使用 value()方法传回 QVariants 对象之后，再使用相关方法，如 toString()、toInt()等尝试转为指定的型态。

除了 next()方法之外，QSqlQuery 还有 previous()、seek()、first()、last()等方法，方便您进行数据检索，但是效能上负担较重，如果您只是要循序取出资料，则可以使用 QSqlQuery()的 setForwardOnly()设定为 true，再用 next()方法取出数据，以获得较好的效能。

如果数据库表格的字段很多，直接使用字符串的方式撰写 SQL 时，在安插数据时就会有点麻烦，尤其是在需要正确 escape 某些字符时，而且对于大量数据安插时，将一些型态转换为字符串也有损效能，您可以用 prepare()方法搭配占位字符 '?' 的方式来指定数据安插处，

再使用 bindValue()来指定真正要安插的数据，例如：

```
QSqlQuery query;
query.prepare("INSERT INTO T_USER (name, age) VALUES (?, ?)");
query.addBindValue("bee");
query.addBindValue(1);
query.exec();
```

这样的写法是 ODBC 风格的写法，另一种方式则是直接指定 Oracle 风格的语法，可以直接指定占位名称，再搭配 bindValue()方法指定实际值，例如：

```
QSqlQuery query;
query.prepare("INSERT INTO T_USER (name, age) VALUES (:name, :age)");
query.bindValue(":name", "justin");
query.bindValue(":age", 33);
query.exec();
```

如果数据库支持交易（Transaction），则您可以使用 Qt 的交易支持，例如：

```
QSqlDriver *driver = QSqlDatabase::database().driver();
if (driver->hasFeature(QSqlDriver::Transactions)) {

    QSqlDatabase::database().transaction();
    QSqlQuery query;
    query.exec("INSERT .....");
    ....
    QSqlDatabase::database().commit();
}
```

QSqlDatabase 的静态 database()方法，传回目前数据库连结的 QSqlDatabase 实例，您使用其 driver()传回驱动程序实例，并使用 hasFeature()测试是否支持交易，如果支持，则使用 transaction()开启交易，使用 commit()提交执行，或使用 rollback()方法撤消。

如果需要对多个数据库的连结，则可以使用 QSqlDatabase 的 addDatabase()方法时指定别名，例如：

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", "DEMO2");
db.setHostName("localhost");
db.setDatabaseName("demo2");
db.setUserName("admin");
db.setPassword("123456");
```

如果要传回该数据库连结的 QSqlDatabase 实例，则在使用 database()方法时指定别名，并在使用 QSqlQuery 时，指定传回的 QSqlDatabase 实例，例如：

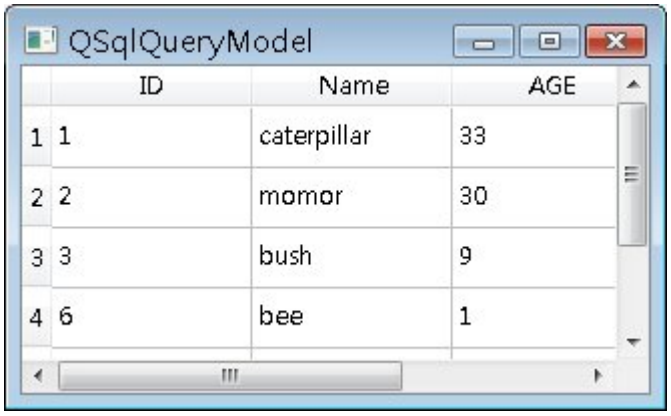
```
QSqlDatabase db = QSqlDatabase::database("DEMO2");
QSqlQuery query(db);
query.exec("....");
```

6.33QSqlQueryModel 与 QSqlTableModel

QSqlQueryModel 提供可编辑的数据模型，协助您从单一数据表格中读取或写入数据，可搭配 View 类别 简单的达到以图形组件显示表格字段记录，下面的程序是个简单的示范：

```
#include <QApplication>
#include <QtSql>
#include <QTableView>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("localhost");
    db.setDatabaseName("demo");
    db.setUserName("root");
    db.setPassword("123456");
    db.open();

    QSqlQueryModel *model = new QSqlQueryModel;
    model->setQuery("SELECT * FROM T_USER");
    model->setHeaderData(0, Qt::Horizontal, "ID");
    model->setHeaderData(1, Qt::Horizontal, "Name");
    model->setHeaderData(2, Qt::Horizontal, "AGE");

    QTableView *view = new QTableView;
    view->setWindowTitle("QSqlQueryModel");
    view->setModel(model);
    view->show();

    db.close();

    return app.exec();
}
```

QTableView 会自动提取 QSqlQueryModel 的数据，并显示在表格之中，一个执行的参考画面如下所示：



如果想要直接使用 QSqlQueryModel 进行数据查询，则可以如下：

```
QSqlQueryModel *model = new QSqlQueryModel;
model->setQuery("SELECT * FROM T_USER");

for (int i = 0; i < model->rowCount(); ++i) {
    QSqlRecord record = model->record(i);
    int id = record.value("id").toInt();
    QString name = record.value("name").toString();
    int age = record.value("age").toInt();
    cout << id << " " << qPrintable(name) << " " << age << endl;
}
```

QSqlQueryModel 的 rowCount()方法传回所查询得到的数据笔数，record()方法指定索引表示要查询第几笔数据，这会传回一个 QSqlRecord 对象，您可以使用 value()方法指定字段名称以取得数据的 QVariants 对象，并使用 toInt ()、toString()等方法转换为指定的数据型态，value()方法也可以指定数字索引，索引值从 0 开始，建议采用数字索引，以降低与数据表格的耦合度，例如：

```
QSqlQueryModel *model = new QSqlQueryModel;
model->setQuery("SELECT * FROM T_USER");

for (int i = 0; i < model->rowCount(); ++i) {
    QSqlRecord record = model->record(i);
    int id = record.value(0).toInt();
    QString name = record.value(1).toString();
    int age = record.value(2).toInt();
    cout << id << " " << qPrintable(name) << " " << age << endl;
}
```

QSqlQueryModel 的 setQuery()方法基本上也可以设定 INSERT、UPDATE、DELETE 等 SQL 语句，例如以下的程序片段会删除名称为"justin"的数据：

```
model->setQuery("DELETE FROM T_USER WHERE name = 'justin'");
```

QSqlTableModel 是 QSqlQueryModel 的子类别，提供对象导向的方式来对数据库表格进行存取，透过 QSqlTableModel，您可以不用撰写 SQL 语句，就可以进行查询、更新、新增、删除等动作，例如：

```
QSqlTableModel *model = new QSqlTableModel;
model->setTable("T_USER");
model->setFilter("age >= 30");
model->select();

for (int i = 0; i < model->rowCount(); ++i) {
    QSqlRecord record = model->record(i);
    int id = record.value(0).toInt();
    QString name = record.value(1).toString();
    int age = record.value(2).toInt();
    cout << id << " " << qPrintable(name) << " " << age << endl;
}
```

您可以使用 setTable()方法指定要查询的表格，使用 setFilter()设定查询条件，使用 select()方法进行 SELECT 查询，这相当于使用以下的 SQL 语句：

```
SELECT * FROM T_USER WHERE age >= 30
```

如果要更新数据，则可以使用如下，以下取得第一笔资料，并进行更新：

```
QSqlTableModel *model = new QSqlTableModel;
model->setTable("T_USER");
// 先 SELECT 资料到 Model
model->select();
// 取得第一笔记录
QSqlRecord record = model->record(0);
// 更新数据记录
record.setValue("name", "justin");
// 设定 Model 的第一笔记录
model->setRecord(0, record);
// 记得要 submitAll()才会更新至数据库
model->submitAll();
```

也可以直接在 QSqlTableModel 对象上直接使用 setValue()进行更新，例如：

```
QSqlTableModel *model = new QSqlTableModel;
model->setTable("T_USER");
// 先 SELECT 资料到 Model
model->select();
// 指定 Model 中储存格索引更新数据
model->setData(model->index(0, 1), "caterpillar");
model->setData(model->index(0, 2), 34);
// 记得要 submitAll()才会更新至数据库
model->submitAll();
```

如果要新增数据，则如下进行：

```
QSqlTableModel *model = new QSqlTableModel;
model->setTable("T_USER");
// 先 SELECT 资料到 Model
model->select();
// 只要指定为 0 即可
int row = 0;
// 在 Model 中新增一列
model->insertRows(row, 1);
model->setData(model->index(row, 1), "ww");
model->setData(model->index(row, 2), 10);
model->submitAll();
```

虽然 insertRows()是用来在指定的列(Row)后插入指定的列数,但实际上在 QSqlTableModel 中，insertRows()被重新定义为只要指定列数为 0 即可，数据表格的新增数据就一律新增至最后一列。

如果要删除数据，则使用 removeRows()方法，例如：

```
QSqlTableModel *model = new QSqlTableModel;
model->setTable("T_USER");
model->setFilter("name = 'duke'");

// 先 SELECT 资料到 Model
model->select();

// 从 Model 中移除资料
if (model->rowCount() > 0) {
    model->removeRows(0, model->rowCount());
```

```
    // 记得要 submitAll()才会更新至数据库
    model->submitAll();
}
```

6.4 绘图

6.41QPainter

QPainter、QPaintEngine、QPaintDevice 组成了 Qt 的 绘图系统（The Paint System），QPainter 提供低阶的绘图 API，在内部使用 QPaintEngine 作为接口，在 QPaintDevice 进行绘图，只要是 QPaintDevice 的子类别，就可以建立 QPainter 在其上进行图形绘制，像是 QWidget、QImage、QPicture、QPrinter 等都是 QPaintDevice 的子类别。

建立 QPainter 的方式如下，其中 qPainterDevice 是个指向 QPaintDevice 子类别的名称：

```
QPainter painter(qPainterDevice);
```

若是图形组件，通常会重新定义 QWidget 的 paintEvent()，当绘图装置（Paint Device）需要重绘时，就会发出 QPaintEvent 并分派给这个方法来处理事件，例如组件出现、被覆盖又重现时，您也可以呼叫 repaint() 或 update()，这也会执行 paintEvent()。

QPainter 提供各种绘制图形的 API，从基本的线绘制、方块、矩形、圆形、渐层到复杂的图片等，QPainter 都有提供相对应的 API，使用的方式，在 QPainter 的说明文件 中，基本上都有提供，在这边基本上要先了解的是，QPainter 的三个基本设定：笔触（Pen）、笔刷（Brush）与字型（Font）。

笔触在 Qt 中是以 QPen 作代表，用来于绘制线条或轮廓时决定样式，像是颜色、笔宽、转折、线条样式（实线、曲线、点状线之类的样式）等。

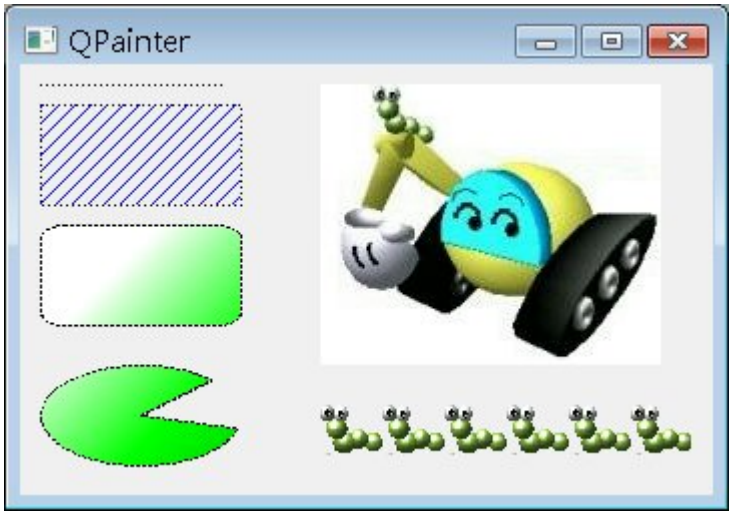
笔刷在 Qt 中是以 QBrush 作代表，用来于绘制矩形、圆形、扇形等几何图形时决定样式，像是颜色、填满样式、渐层等。

以下先看一个简单的程序，了解一下 QPainter 的几个 API，以及 QPen、QBrush 的使用：

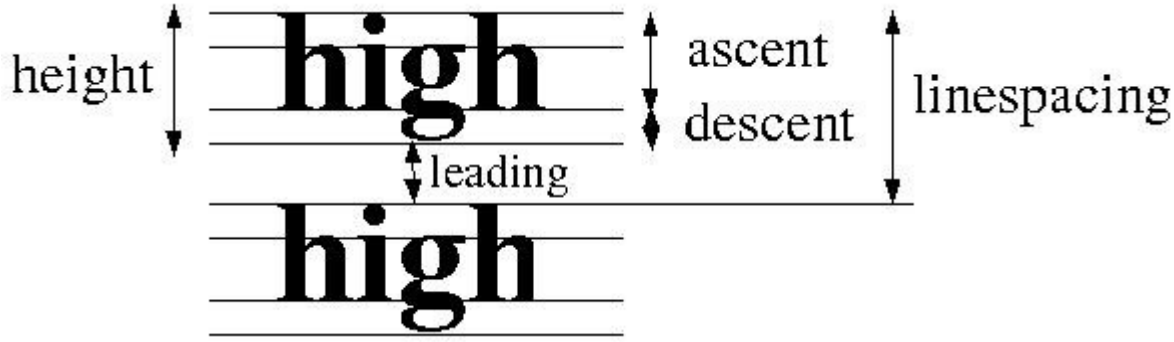
```
#include <QApplication>
#include <QWidget>
#include <QPainter>

class PainterWidget : public QWidget {
protected:
    // 重新定义 paintEvent()事件处理
    void paintEvent(QPaintEvent*);
};

// 实作事件处理
```

```
void PainterWidget::paintEvent(QPaintEvent *event) {
    // 建立 QPainter
    QPainter painter(this);

    // 设定笔触为点状线
    painter.setPen(Qt::DotLine);

    // 指定 x、y、width、height 绘制线条
    painter.drawLine(10, 10, 100, 10);

    // 设定笔刷为蓝色、对角斜线样式
    painter.setBrush(QBrush(Qt::blue, Qt::BDiagPattern));

    // 指定 x、y、width、height 绘制矩形
    painter.drawRect(10, 20, 100, 50);

    // 设定线形渐层，x1,y1 为起点，x2,y2 为终点
    QLinearGradient gradient(50, 100, 300, 350);

    // 设定渐层颜色过渡
    gradient.setColorAt(0.0, Qt::white);
    gradient.setColorAt(0.2, Qt::green);
    gradient.setColorAt(1.0, Qt::black);

    // 以渐层对象建立笔刷
    painter.setBrush(QBrush(gradient));

    // 绘制圆角矩形
    painter.drawRoundRect(10, 80, 100, 50);

    // 绘制扇形，单位为 1/16 角度，下例为 45 度 到 300 度
    painter.drawPie(10, 150, 100, 50, 45 * 16, 300 * 16);

    // 绘制图片
    painter.drawPixmap(150, 10, QPixmap("caterpillar.jpg"));

    // 绘制填满图形
    painter.drawTiledPixmap(150, 170, 185, 25, QPixmap("caterpillar_smaill.gif"));
}

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    PainterWidget pWidget;
```

```
    pWidget.setWindowTitle("QPainter");
    pWidget.resize(350, 200);
    pWidget.show();

    return app.exec();
}
```



QPainter 类别中的 drawText()方法可以在绘图装置上绘制文字，也可以设定绘制时所用的字型，字型在 Qt 中是使用 QFont 代表，在设定好字型之后，可以用 fontMetrics()方法取得字型的几何信息，例如 ascent（字符最高点至字符底线 baseline 距离）、descent（字符最低点到字符底线距离）、leading（两行之间的空间值）height（字体印字时的高度，相当于 ascent+descent +1，1pixel 是字符底线的高度）与 linespacing（height+leading）等。



下面这个程序是文字绘制的简单示范，利用循环展示三种字型、五种大小不同组合下的文字绘制效果：

```
#include <QApplication>
#include <QWidget>
#include <QPainter>

class PainterWidget : public QWidget {
protected:
    void paintEvent(QPaintEvent*);
};
```

```
void PainterWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    static const char *fonts[] = {"Helvetica", "Courier", "Times", 0};
    static int    sizes[] = {10, 12, 18, 24, 36, 0};
    int f = 0;
    int y = 0;
    while (fonts[f]) {
        int s = 0;
        while (sizes[s]) {
            QFont font(fonts[f], sizes[s]);
            painter.setFont(font);
            QFontMetrics fm = painter.fontMetrics();

            y += fm.ascent();
            painter.drawText(10, y, "Hello! caterpillar!");
            y += fm.descent();
            s++;
        }
        f++;
    }
}

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    PainterWidget pWidget;
    pWidget.setWindowTitle("QPainter");
    pWidget.resize(600, 450);
    pWidget.show();

    return app.exec();
}
```

下图为执行时的结果画面：

6.42QMatrix

QPainter 预设的坐标系统是绘图装置的坐标系统，也就是左上角为原点，向右为正 X，向下为正 Y 的坐标系统，坐标系统的转换常使用矩阵的方式来表现及进行运算，QMatrix 的作用，正是让您可以利用其内建的矩阵，设定好相关的参数，然后让 QPainter 根据 QMatrix 的设定，来进行一些二维坐标系统的转换动作。

QMatrix 的内部使用一个 3x3 的矩阵：

m11	m12	0
m21	m22	0
dx	dy	1

dx 与 dy 定义了水平与垂直移动，m11 与 m22 定义了水平与垂直缩放（scaling），m12 与 m21 定义了垂直与水平扭曲（shearing），想象您是坐在宇宙飞船中，在宇宙飞船从左上原点开到某个点之后，（x, y）是以您为中心所看到的坐标，但实际上宇宙飞船相对于左上角为原点的坐标为（x', y'），QMatrix 的矩阵可以如以下的公式，将（x, y）转换为（x', y'）：

x' = m11*x + m21*y + dx
y' = m22*y + m12*x + dy

当您使用 QPainter 要进行绘图时，可以您为中心所看到的坐标系统(x, y)，使用 QPainter 的相关 API 来进行相关图形的绘制，这就像您在宇宙飞船中画图一样的方便，若有设定 QMatrix，则会自动转换为计算机绘图时所看到的坐标系统（x', y'），如此就不用亲自进行一些复杂的转换动作，进行绘图时也较为直觉。

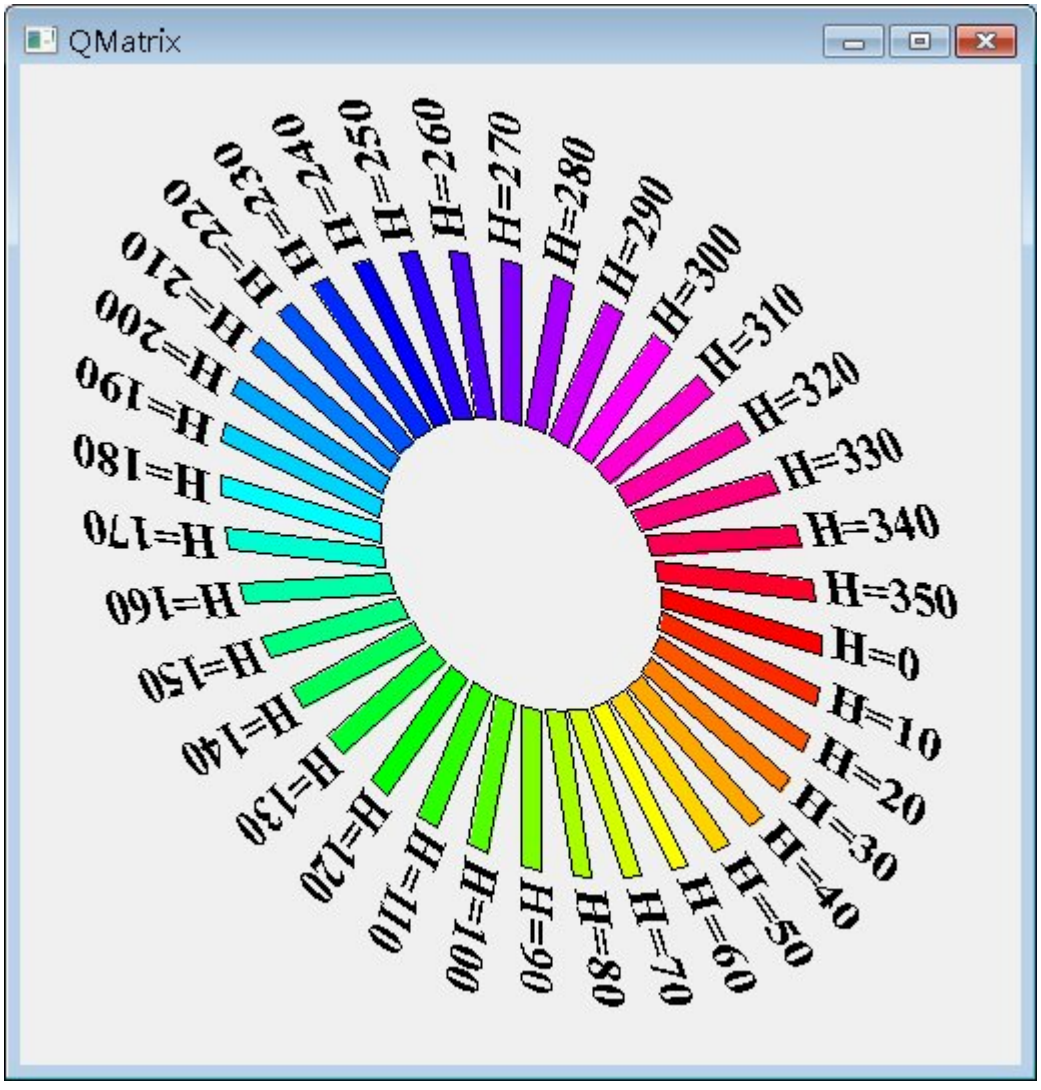
您可以藉由 QMatrix 的 setMatrix()方法设定 m11、m12、m21、m22、dx、dy，或者是直接使用 translate()、rotate()、scale()与 shear()等方法来直接进行移动、旋转、缩放、扭曲等坐标转换。

以下的范例为色彩轮的绘制，藉由设定 HSV（Brightness,Hue, Saturation）中的「色相」来完成彩虹般的效果， HSV 中的「色相」(Hue）是錂镜分光，主要有红、橙、黄、绿、蓝、紫...等八个主要色相。「亮度」(Brightness）是明暗表现，由白至黑的表现，在 P.C.C.S（Practical Color Coordinate System）配色系统中，将之分为白、浅灰（浅，深)、浅中灰、中灰、暗中灰、暗灰（浅，深)、黑等。「彩度」(Saturation)也就是色彩的饱和程度，彩度最高的称之为「纯色」，最低为「无颜色」。

```
#include <QApplication>
#include <QWidget>
#include <QPainter>
#include <QMatrix>

class PainterWidget : public QWidget {
protected:
    void paintEvent(QPaintEvent*);
};

void PainterWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    QFont font("times", 18, QFont::Bold);
    painter.setFont(font);
    painter.setPen(Qt::black);

    for (int i = 0; i < 36; i++) {                  // 进行 36 次旋转
        QMatrix matrix;
        matrix.translate(250, 250);                 // 移动中心至 (250, 250)
        matrix.shear(0.0, 0.3);                     // 扭曲
        matrix.rotate((float) i * 10);              // 每次旋转 10 度
        painter.setWorldMatrix(matrix);             // 使用这个 QMatrix

        QColor color;
        color.setHsv(i * 10, 255, 255);             // 设定彩虹效果
        painter.setBrush(color);                    // 设定笔刷颜色
        painter.drawRect(70, -10, 80, 10);          // 画矩形
```

```
            QString str;
            str.sprintf("H=%d", i*10);
            painter.drawText(80 + 70 + 5, 0, str);       // 绘制角度文字
        }
}

int main(int argc, char *argv[]) {
        QApplication app(argc, argv);

        PainterWidget pWidget;
        pWidget.setWindowTitle("QMatrix");
        pWidget.resize(500, 500);
        pWidget.show();

        return app.exec();
}
```

下图为执行时的参考画面：



6.43QPixmap、 QBitmap、QImage 与 QPicture

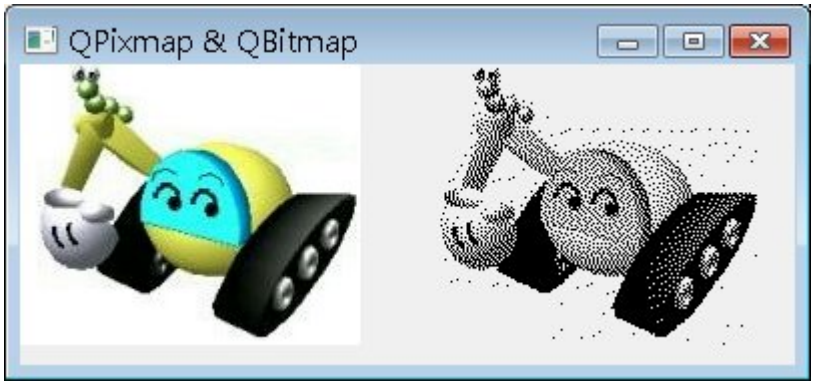
在处理影像数据上，Qt 提供了 QPixmap、QBitmap、QImage 与 QPicture 等类别。

QPixmap 继承了 QPaintDevice，您可用以建立 QPainter 并于上进行绘图，您也可以直接指定图案加载 Qt 所支持的图档，像是 BMP、GIF、JPG、JPEG、PNG 等，并使用 QPainter 的 drawPixmap()绘制在其它的绘图装置上。您可以在 QLabel、QPushButton 上设定 QPixmap 以显示图像。QPixmap 是针对屏幕显示图像而设计并最佳化，依赖于所在平台的原生绘图引擎，所以一些效果的展现（像是反锯齿），在不同的平台上可能会有不一致的结果。

QBitmap 是 QPixmap 的子类别，提供单色图像，可用于制作光标（QCursor）或笔刷（QBrush）物件。下面的程序加载相同的图文件，以观看 QPixmap 与 QBitmap 的呈现效果：

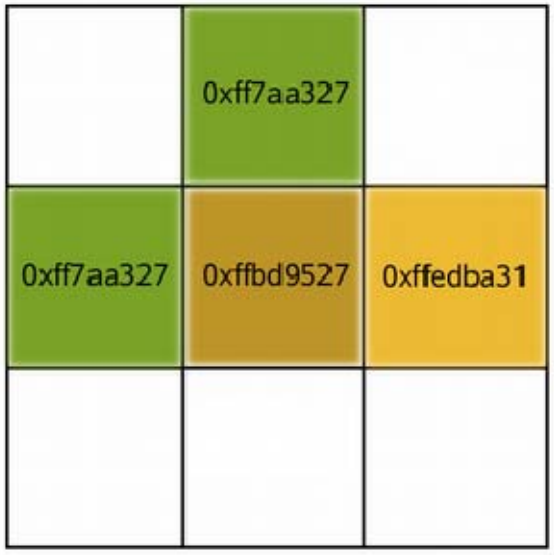
```
#include <QApplication>
#include <QWidget>
#include <QPainter>
#include <QBitmap>

class PainterWidget : public QWidget {
protected:
    void paintEvent(QPaintEvent*);
};

void PainterWidget::paintEvent(QPaintEvent *event) {
    QPixmap pixImg("caterpillar.jpg");
    QBitmap bitImg("caterpillar.jpg");
    QPainter painter(this);
    painter.drawPixmap(0, 0, pixImg);
    painter.drawPixmap(200, 0, bitImg);
}

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    PainterWidget pWidget;
    pWidget.setWindowTitle("QPixmap & QBitmap");
    pWidget.resize(400, 150);
    pWidget.show();

    return app.exec();
}
```

执行后的结果如下图所示：



QPixmap 使用平台的绘图引擎，在不同的平台所呈现的效果不一，无法提供个别像素的存取，QImage 使用 Qt 自身的绘图引擎，可提供在不同平台上相同的图像呈现效果，并可透过 setPixpel()、pixel()等方法，直接存取指定的像素，例如在 Qt 的 QImage 文件中，就有提供以下的范例：

```
QImage image(3, 3, QImage::Format_RGB32);
  QRgb value;

  value = qRgb(189, 149, 39); // 0xffbd9527
  image.setPixel(1, 1, value);

  value = qRgb(122, 163, 39); // 0xff7aa327
  image.setPixel(0, 1, value);
  image.setPixel(1, 0, value);

  value = qRgb(237, 187, 51); // 0xffedba31
  image.setPixel(2, 1, value);
```



QPicture 则是个绘图装置，可以记录并回放 QPainter 的绘图指令，您可以使用 QPainter 的 begin()方法，指定在 QPicture 上进行绘图，使用 end()方法结束绘图，使用 QPicture 的 save()方法将 QPainter 所使用过的绘图指令存至档案，例如：

```
QPicture picture;
QPainter painter;
painter.begin(&picture);
painter.drawRect(10, 20, 100, 50);
painter.end();
picture.save("draw_record.pic");
```

要回放绘图指令的话，建立一个 QPicture，使用 load()方法加载绘图指令的档案，然后在指定的绘图装置上绘制 QPicture：

```
QPicture picture;
picture.load("draw_record.pic");
QPainter painter;
painter.begin(this);
painter.drawPicture(0, 0, picture);
painter.end();
```

6.44QPrinter

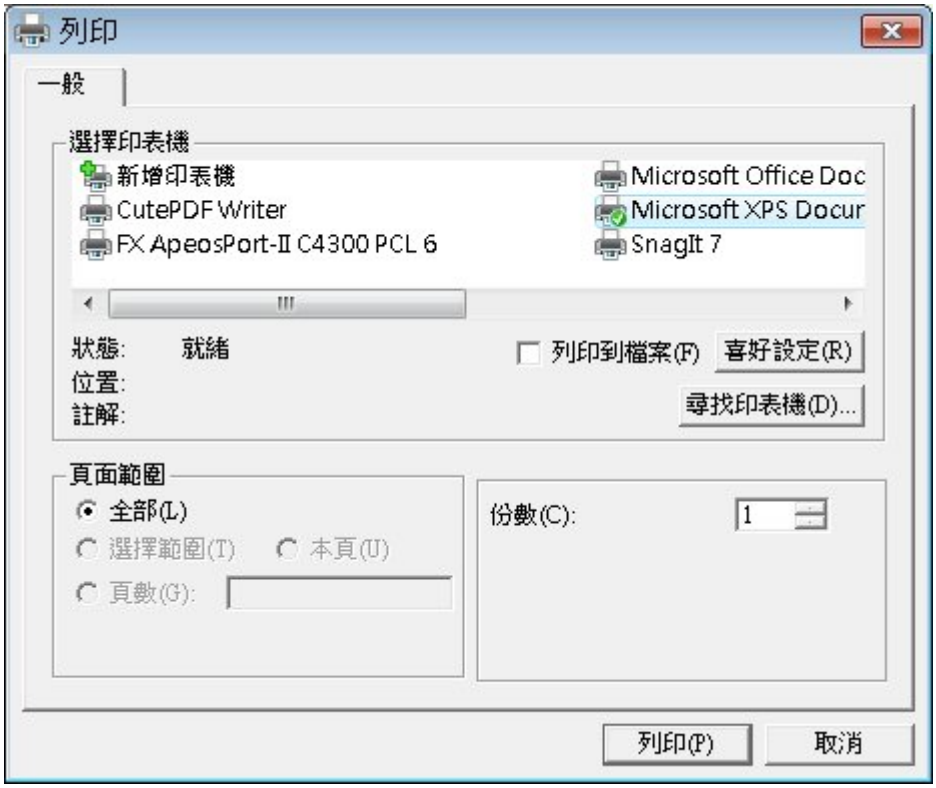
打印机打印，基本上就是透过打印机在纸上进行绘图的动作，打印机为一种绘图装置，在 Qt 中使用 QPrinter 作为打印机绘图装置的表现，您可以基于 QPrinter 建立 QPainter，然后使用 QPainter 进行图形绘制，至于打印机的选择、相关打印参数的设定，若是在图形环境中，可以直接使用 QPrintDialog 来显示打印对话框，让使用者可以选择打印机及相关参数。

下面这个简单的程序，示范如何使用 QPrinter 及 QPrintDialog，您可以使用 QFileDialog 加载一个图片档案，并使用 QPrintDialog 设定打印机，然后将选择的图文件名称及图片本身打印出来：

```
include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QPixmap>
#include <QPrinter>
#include <QPainter>
#include <QFileDialog>
#include <QPrintDialog>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QLabel *label = new QLabel("<center>Image</center>");
    label->setWindowTitle("QPrinter");
    label->resize(500, 100);
```

```
    label->setFont(QFont( "Times", 18, QFont::Bold ));
    label->show();

    QString fileName = QFileDialog::getOpenFileName(label, "Open Image",
                        "C:\\", "Image Files (*.png *.xpm *.jpg *.gif)");

    if(fileName != NULL) {
        label->setWindowTitle(fileName);

        QPixmap pixmap(fileName);
        label->setPixmap(pixmap);
        label->resize(pixmap.width() , pixmap.height());

        QPrinter printer;
        QPrintDialog printDialog(&printer, label);
        if (printDialog.exec()) {
            QPainter painter(&printer);
            painter.drawText(50, 50, fileName);
            painter.drawPixmap(50, 100, pixmap);
        }
    }
    return app.exec();
}
```

下图为执行时出现 QPrintDialog 的参考画面：



若要打印时进行换页，可以使用 QPrinter 的 newPage()方法，若要中断打印机打印，则可

以呼叫 abort()方法，您也可以透过 QPrinter 来产生 pdf 档案，只要执行 QPrinter 的 setOutputFormat(QPrinter::PdfFormat)，并使用 setOutputFileName()设定输出的文件名称，例如：

```
printer.setOutputFormat(QPrinter::PdfFormat);
printer.setOutputFileName("QPrinterOutput.pdf");
```

若不想透过 QPrintDialog 的方式设定 QPrinter 的相关参数，则可以使用 QPrinter 上的几个方法来设定，像是 setOrientation()设定纸张方向，setPageSize()设定纸张大小，setResolution()设定打印的 DPI（Dots per inch）分辨率，setFullPage()设定是否整张纸作为打印，setNumCopies()设定打印份数等。

6.5 拖放（Drag & Drop）与剪贴

6.51 拖放事件

想要在图形组件上启用拖放功能，可以使用 QWidget 上所继承下来的 setAcceptDrops()方法，设定组件接受拖放动作，在拖放动作发生时，会有相对应的 QDragEnterEvent、QDragMoveEvent、QDragLeaveEvent 与 QDropEvent 等事件发生，您可以重新定义 dragEnterEvent()、dragMoveEvent()、dragLeaveEvent()与 dropEvent()等事件处理者，以处理相对应的拖放事件，通常会使用的是 dragEnterEvent()与 dropEvent()。

以下是一个简单的拖放事件处理程序，您可以将图档拖放至 QLabel 上，QLabel 会自动加载图片并显示出来：

ImageLabel.h

```
#ifndef IMAGE_LABEL_H
#define IMAGE_LABEL_H

#include <QLabel>

class QDragEnterEvent;
class QDropEvent;

class ImageLabel : public QLabel {
    Q_OBJECT

public:
    ImageLabel(QWidget *parent = 0);

protected:
    void dragEnterEvent(QDragEnterEvent *event);
    void dropEvent(QDropEvent *event);

private:
```

```
    void readImage(const QString &fileName);
};

#endif
```

ImageLabel 继承了 QLabel，并且将重新定义 dragEnterEvent()及 dropEvent()，以处理对应的拖放事件，readImage()则会在拖放图档至 QLabel 上时，将图片档案加载并设定给 QLabel 来显示，ImageLabel 的实作如下：

ImageLabel.cpp

```
#include <QDragEnterEvent>
#include <QDropEvent>
#include <QUrl>
#include <QFile>
#include <QTextStream>
#include "ImageLabel.h"

ImageLabel::ImageLabel(QWidget *parent) : QLabel(parent) {
    this->setAcceptDrops(true);
}

void ImageLabel::dragEnterEvent(QDragEnterEvent *event) {
    if(event->mimeData()->hasFormat("text/uri-list")) {
        event->acceptProposedAction();
    }
}

void ImageLabel::dropEvent(QDropEvent *event) {
    QList<QUrl> urls = event->mimeData()->urls();
    if (urls.isEmpty()) {
        return;
    }

    QString fileName = urls.first().toLocalFile();
    if (fileName.isEmpty()) {
        return;
    }

    this->setWindowTitle(fileName);
    readImage(fileName);
}

void ImageLabel::readImage(const QString &fileName) {
    QPixmap pixmap(fileName);
    this->setPixmap(pixmap);
```

```
    this->resize(pixmap.width(), pixmap.height());
}
```

setAcceptDrops(true)设定接受拖放，当图片档案被拖入组件时，发生 QDragEnterEvent 并分派给 dragEnterEvent()处理。QDragEnterEvent 的 mimeData()方法传回 QMimeData，当中包括了 MIME （Multipurpose Internet Mail Extensions）类型的相关信息，hasFormat("text/uri-list")测试是否含有文字信息的 URI （Universal Resource Identifier），也可以使用 hasUrls()来进行同样的测试。

当拖放行为开始执行时，执行拖放行为的一方会需要知道接受放置的另一方接受何种操作，例如拖放档案时，当拖放完成时，原档案是被复制或移动。执行拖放行为的一方会设定可被接受的动作，而接受放置的一方可选择接受何种动作，并传回相关信息给执行方，acceptProposedAction()的作用为接受执行拖放行为的一方所设定的预设动作。

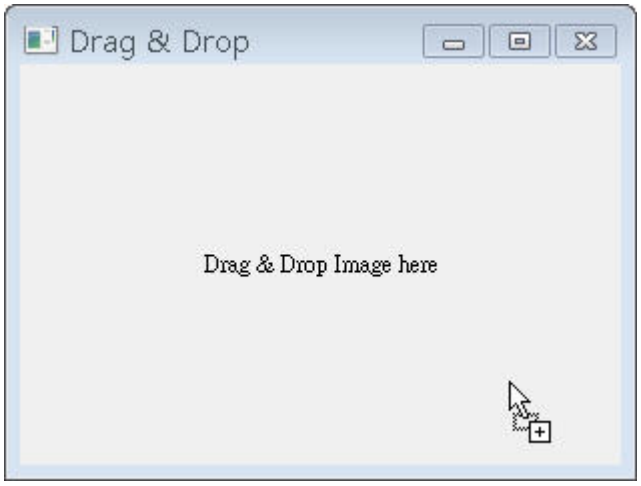
当放置时会发生 QDropEvent 并分派给 dropEvent()来处理，QMimeData 的 urls()取得所拖放档案的 QUrl，由于您所拖放的档案可能不只一个，所以 urls()传回的是内含 QUrl 的 QList 对象，您取得第一个 QUrl 并取得文件名信息，再进行图文件的读取并设定至 QLabel 上。

可以撰写以下的程序来执行：

main.cpp

```
#include <QApplication>
#include <QMainWindow>
#include "ImageLabel.h"
#include <QTextEdit>

int main(int argc, char **argv) {
    QApplication app(argc, argv);

    ImageLabel *imgLabel = new ImageLabel;
    imgLabel->setWindowTitle("Drag & Drop");
    imgLabel->setText("<center>Drag & Drop Image here</center>");
    imgLabel->resize(300, 200);
    imgLabel->show();

    return app.exec();
}
```

执行时，组件若可以进行拖放，则会出现相关的光标以提醒使用者，在 Windows 下是个「+」的光标：



有些组件预设即接受拖放行为，例如 QLineEdit，您可以直接将文字拖曳至 QLineEdit，这会将拖曳的文字置放至您置放的位置，若您想改变这个行为，可以设定 QLineEdit 的 setAcceptDrops()为 false，然后设定其父组件的 setAcceptDrops()为 true，让拖放事件传播至父组件，由父组件来处理拖放相关事件，以改变 QLineEdit 的预设拖放行为，例如让纯文本文件拖至 QLineEdit 时，可以自动加载文字至 QLineEdit 中。

6.52 拖放的执行与接受

您可以决定何时执行拖放，并设定拖放时要于组件或窗口之间沟通的信息，要执行拖放的几个基本动作为：

执行 setAcceptDrops()设定为 true
建立 QMimeData，设定拖放时要携带的信息，例如文字、影像等
建立 QDrag，将 QMimeData 设定给 QDrag，并设定拖放时所要显示的图标等信息
执行 QDrag 的 exec()方法，设定使用者可选择的放置动作
使用 exec()的传回值判断使用者所接受的放置动作，以进行后续处理

当拖放行为开始执行时，执行拖放行为的一方会需要知道接受放置的另一方接受何种操作，例如拖放档案时，当拖放完成时，原档案是被复制或移动。执行拖放行为的一方会设定可被接受的动作，而接受放置的一方可选择接受何种动作，并传回相关信息给执行方，exec()方法完成后所传回的信息作用即是如此。

以下制作一个简单的清单程序示范拖放的执行与接受,您可以将清单中的项目拖放至另一个清单之中。

通常会重新定义 mousePressEvent()，记录鼠标按下时的位置，而后再重新定义 mouseMoveEvent()，判断鼠标按下后移动的距离，是否达到所建议的拖放执行距离：

```
void ListWidget::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        startPoint = event->pos();
    }
    QListWidget::mousePressEvent(event);
```

```
}

void ListWidget::mouseMoveEvent(QMouseEvent *event) {
    if (event->buttons() & Qt::LeftButton) {
        if ((event->pos() - startPoint).manhattanLength()
                >= QApplication::startDragDistance()) {
            execDrag();
        }
    }
    QListWidget::mouseMoveEvent(event);
}
```

execDrag()中示范了，如何建立 QMimeData 及 QDrag，并使用 exec()执行拖放：

```
void ListWidget::execDrag() {
    QListWidgetItem *item = currentItem();
    if (item) {
        QMimeData *mimeData = new QMimeData;
        // 设定所要携带的文字信息
        mimeData->setText(item->text());
        // 设定所要携带的影像数据
        mimeData->setImageData(item->icon());
        QDrag *drag = new QDrag(this);
        drag->setMimeData(mimeData);
        // 设定拖放时所显示的图标
        drag->setPixmap(item->icon().pixmap(QSize(22, 22)));
        if (drag->exec(Qt::MoveAction) == Qt::MoveAction) {
            delete item;
        }
    }
}
```

exec()方法执行时，设定接受拖放的一端可以采用的动作，接受拖放的一方所打算采用的动作，可使用事件的 setDropAction()来设定，并使用 accept()方法接受事件。上面的程序片段中，若接受拖放的一方接受 Qt::MoveAction，则原清单中的项目会被删除。

若 QDragEnterEvent 的 setDropAction()所设定的动作，不在 exec()方法所设定的允许动作中，则会出现禁止符号，表示拖放无法完成：

```
void ListWidget::dragEnterEvent(QDragEnterEvent *event) {
    ListWidget *source =
            qobject_cast<ListWidget *>(event->source());
    if (source && source != this) {
```

```
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

QDropEvent 的 setDropAction()设定放置时所采取的动作，将成为 exec()的传回值，以下例而言，在接受拖放的清单从 QMimeData 中取得文字与图标并新增项目，QDropEvent 使用 setDropAction()设定为 Qt::MoveAction 并 accept()之后，exec()将传回 Qt::MoveAction：

```
void ListWidget::dropEvent(QDropEvent *event) {
    ListWidget *source =
            qobject_cast<ListWidget *>(event->source());
    if (source && source != this) {
        QIcon icon = event->mimeData()->imageData().value<QIcon>();
        QString text = event->mimeData()->text();
        addItem(new QListWidgetItem(icon, text));

        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

dragMoveEvent()在这个例子中，直接覆写保持空白，这是为了要覆写父类别的 dragMoveEvent()定义

```
void ListWidget::dragMoveEvent(QDragMoveEvent *event) {}
```

以下是完整的程序范例参考：

ListWidget.h

```
#ifndef LISTWIDGET_H
#define LISTWIDGET_H

#include <QListWidget>

class QMouseEvent;
class QDragEnterEvent;
class QDragMoveEvent;
class QDropEvent;
class QPoint;
```

```
class ListWidget : public QListWidget {
    Q_OBJECT

public:
    ListWidget(QWidget *parent = 0);

protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void dragEnterEvent(QDragEnterEvent *event);
    void dragMoveEvent(QDragMoveEvent *event);
    void dropEvent(QDropEvent *event);

private:
    void execDrag();
    QPoint startPoint;
};

#endif
```

ListWidget.cpp

```
#include "ListWidget.h"

#include <QApplication>
#include <QPoint>
#include <QMouseEvent>
#include <QMimeData>
#include <QDrag>
#include <QListWidgetItem>
#include <QIcon>

ListWidget::ListWidget(QWidget *parent) : QListWidget(parent) {
    setAcceptDrops(true);
}

void ListWidget::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        startPoint = event->pos();
    }
    QListWidget::mousePressEvent(event);
}

void ListWidget::mouseMoveEvent(QMouseEvent *event) {
    if (event->buttons() & Qt::LeftButton) {
        if ((event->pos() - startPoint).manhattanLength()
```

```
                    >= QApplication::startDragDistance()) {
                execDrag();
            }
        }
        QListWidget::mouseMoveEvent(event);
}

void ListWidget::execDrag() {
        QListWidgetItem *item = currentItem();
        if (item) {
            QMimeData *mimeData = new QMimeData;
            mimeData->setText(item->text());
            mimeData->setImageData(item->icon());
            QDrag *drag = new QDrag(this);
            drag->setMimeData(mimeData);
            drag->setPixmap(item->icon().pixmap(QSize(22, 22)));
            if (drag->exec(Qt::MoveAction) == Qt::MoveAction) {
                delete item;
            }
        }
}

void ListWidget::dragEnterEvent(QDragEnterEvent *event) {
        ListWidget *source =
                qobject_cast<ListWidget *>(event->source());
        if (source && source != this) {
            event->setDropAction(Qt::MoveAction);
            event->accept();
        }
}

void ListWidget::dragMoveEvent(QDragMoveEvent *event) {}

void ListWidget::dropEvent(QDropEvent *event) {
        ListWidget *source =
                qobject_cast<ListWidget *>(event->source());
        if (source && source != this) {
            QIcon icon = event->mimeData()->imageData().value<QIcon>();
            QString text = event->mimeData()->text();
            addItem(new QListWidgetItem(icon, text));

            event->setDropAction(Qt::MoveAction);
            event->accept();
        }
```

```
}
```

main.cpp

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

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    ListWidget *listWidget1 = new ListWidget;
    listWidget1->setWindowTitle("Head store");
    listWidget1->insertItem(0, new QListWidgetItem(
                    QIcon("caterpillar_head.jpg"), "caterpillar"));
    listWidget1->insertItem(1, new QListWidgetItem(
                    QIcon("momor_head.jpg"), "momor"));
    listWidget1->insertItem(2, new QListWidgetItem(
                    QIcon("bush_head.jpg"), "bush"));
    listWidget1->insertItem(3, new QListWidgetItem(
                    QIcon("bee_head.jpg"), "bee"));
    listWidget1->insertItem(4, new QListWidgetItem(
                    QIcon("cat_head.jpg"), "cat"));

    ListWidget *listWidget2 = new ListWidget;
    listWidget2->setWindowTitle("Buy head");


    listWidget1->show();
    listWidget2->show();

    return app.exec();
}
```

程序的执行画面如下所示：

dragMoveEvent()可以用于将拖放的区域限制在某个范围中，在 Qt 的 Drag and Drop 文件

中有个例子：

```
void Window::dragMoveEvent(QDragMoveEvent *event) {
     if (event->mimeData()->hasFormat("text/plain")
         && event->answerRect().intersects(dropFrame->geometry())) {
         event->acceptProposedAction();
    }
}
```

接受拖放的一方可以直接执行拖放事件的 acceptProposedAction()接受所建议的动作，或者是使用 proposedAction()来判断不同的动作该采取的行为：

```
if (event->proposedAction() == Qt::MoveAction) {
    event->acceptProposedAction();
     // 处理事件 ....
} else if (event->proposedAction() == Qt::CopyAction) {
    event->acceptProposedAction();
    // 处理事件 ....
} else {
    // 处理事件 ....
    return;
}
```

您不仅可以在拖放时携带文字或图片，符合 MIME 类型 的数据都可以于拖放时携带，或者您也可以将资料转换为 QByteArray，使用 QMimeData 的 setData()方法设定以进行拖放时数据的携带。

6.53 剪贴簿（QClipboard）

您可以将数据放在剪贴簿中，让应用程序透过剪贴簿共享一些信息，Qt 中剪贴簿的代表对象是 QClipboard，您可以透过 QApplication 的 clipboard()方法来取得：

```
QClipboard *clipboard = QApplication::clipboard();
```

取得 QClipboard 之后，您可以使用 setImage()、setPixmap()、setText()等方法，将图片或文字讯息设定至剪贴簿，在使用这些方法时，在 X11 环境下，也可以使用常数 QClipboard::Selection 指定设定至鼠标选择区，预设是设定为 QClipboard::Clipboard：

```
clipboard->setText(lineEdit->text(), QClipboard::Clipboard);
```

您可以使用 QClipboard 的 supportsSelection()测试是否支持鼠标选择区。要从剪贴簿中取得资料，可以使用 image()、pixmap()、text()等方法，同样的，您可以指定

QClipboard::Selection 从鼠标选择区取得数据。

您也可以使用 QMimeData 设定好相关数据，再使用 QClipboard 的 setMimeData()方法将 QMimeData 设定至剪贴簿，这让剪贴簿可以携带各种类型，而不仅受限于文字或图片。

QClipboard 拥有一些预设的 Signal，像是 changed(QClipboard::Mode mode)、dataChanged()、findBufferChanged()、selectionChanged()，可以让您得知剪贴簿的状态变化。

6.6 网络

6.61QHttp

QHttp 是 Qt 所提供有关网络的高阶 API，可以协助您进行 HTTP 协议的进行，QHttp 发出请求时是异步的，请求的过程中会发出相关的 Signal，您可以用 Slot 来接收这些 Signal，并进行相关的处理。

以下先示范一个最基本的 QHttp 使用，程序将设计一个 HttpGet 类别：

HttpGet.h

```
#ifndef HTTPGET_H
#define HTTPGET_H

#include <QObject>

class QUrl;
class QHttp;
class QFile;

class HttpGet : public QObject {
    Q_OBJECT

public:
    HttpGet(QObject *parent = 0);
    void downloadFile(const QUrl &url);

signals:
    void finished();

private slots:
    void done(bool error);

private:
    QHttp *http;
    QFile *file;
```

```
};

#endif
```

这个 HttpGet 可以让您指定档案的 URL 地址，以 HTTP 方式取得档案并储存在本地端，URL 在 Qt 中以 QUrl 代表，当档案下载完成时，会发出 finished()的 Signal，当 QHttp 所排定的全部请求完成时，会发出 done()的 Signal，HttpGet 类别中自定的 Slot，就是用来接收 QHttp 的 done() Signal 以进行相关处理，这可以在 HttpGet 的实作看到：

HttpGet.cpp

```
#include <QtNetwork>
#include <QFile>
#include <iostream>
#include "HttpGet.h"
using namespace std;

HttpGet::HttpGet(QObject *parent) : QObject(parent) {
    http = new QHttp(this);
    connect(http, SIGNAL(done(bool)), this, SLOT(done(bool)));
}

void HttpGet::downloadFile(const QUrl &url) {
    QFileInfo fileInfo(url.path());
    QString fileName = fileInfo.fileName();
    if (fileName.isEmpty()) {
        fileName = "index.html";
    }

    file = new QFile(fileName);
    if (!file->open(QIODevice::WriteOnly)) {
        cerr << "Unable to save the file" << endl;
        delete file;
        file = 0;
        return;
    }

    http->setHost(url.host(), url.port(80));
    http->get(url.path(), file);
    http->close();
}

void HttpGet::done(bool error) {
    if (error) {
        cerr << "Error: " << qPrintable(http->errorString()) << endl;
    } else {
```

```
            cerr << "File downloaded as " << qPrintable(file->fileName())
                 << endl;
    }
    file->close();
    delete file;
    file = 0;

    emit finished();
}
```

要使用 Qt 的网络相关类别，必须引进 QtNetwork，并且必须在..pro 档案中，加入以下这行以在建构过程中使用 Qt 网络模块：

```
QT += network
```

当呼叫 HttpGet 类别的 downloadFile()方法时，程序中使用 QUrl 的 path()来取得路径讯息，如果路径讯息中没有包括文件名，就使用预设的"index.html"作为请求的对象及下载后存盘时的档名，要使用 QHttp 来请求档案时，必须使用 setHost ()来设定主机及连接端口信息，接着使用 get()方法发出请求，并告知下载的档案要到用哪个 QFile 来存盘。

当 QHttp 所有请求处理完毕后，会发出 done()的 Signal，程序中将之连接至 HttpGet 的 done()来处理，处理完成之后，再发出 finished()的 Signal。

以下写个简单的程序来测试 HttpGet：

main.cpp

```
#include <QCoreApplication>
#include <QUrl>
#include "HttpGet.h"
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    QCoreApplication app(argc, argv);

    HttpGet getter;
    getter.downloadFile(QUrl("http://caterpillar.onlyfun.net/index.html"));

    QObject::connect(&getter, SIGNAL(finished()), &app, SLOT(quit()));

    return app.exec();
}
```

程序中将 HttpGet 的 finished()的 Signal 连接至 QCoreApplication 的 quit()，如此当下载档案完成后，可以直接关闭应用程序。

Qt 的 QHttp 与 QFtp 在使用上有许多类似的地方，可以在以上的范例看到一些特性，以下再整理出相关特性：

非阻断行为，请求是异步的。
您可以排定一连串的请求，每个请求都有一个 Command ID，QHttp 的 requestStarted()与 requestFinished()等 Signal 会带有请求的 Command ID，您可以用以追踪请求的执行。
在数据传输的过程中，有相关的 Signal 可以追踪进度，像是 QHttp 的 dataReadProgress()、dataSendProgress()等 Signal。
支持 QIODevice 的写入（下载）与读取（上传），还有以 QByteArray 为基础的 API。

QHttp 还可以针对请求标头、HTTPS 等加以处理，在 Qt 的在线文件中，有个 QHttp 的范例 Http Example，对 QHttp 的使用有更完整的示范。

6.62QFtp

QFtp 在使用上与 QHttp 非常类似，只不过 FTP 机制比 HTTP 来得更为复杂一些，所以 QFtp 上可以进行的相关操作更为丰富，以下先制作一个简单的程序，使用 QFtp 进行 FTP 档案下载，首先是 FtpGet 的定义：

FtpGet.h

```
#ifndef FTPGET_H
#define FTPGET_H

#include <QObject>

class QUrl;
class QFtp;
class QFile;

class FtpGet : public QObject {
    Q_OBJECT

public:
    FtpGet(QObject *parent = 0);
    void downloadFile(const QUrl &url);

signals:
    void finished();

private slots:
    void done(bool error);
```

```
private:
    QFtp *ftp;
    QFile *file;

};

#endif
```

定义上与上一个 QHttp 故意设计的类似，基本上实作也非常接近：

FtpGet.cpp

```
#include <QtNetwork>
#include <QFile>
#include <iostream>
#include "FtpGet.h"
using namespace std;

FtpGet::FtpGet(QObject *parent) : QObject(parent) {
    ftp = new QFtp(this);
    connect(ftp, SIGNAL(done(bool)), this, SLOT(done(bool)));
}

void FtpGet::downloadFile(const QUrl &url) {
    QFileInfo fileInfo(url.path());
    QString fileName = fileInfo.fileName();

    file = new QFile(fileName);
    if (!file->open(QIODevice::WriteOnly)) {
        cerr << "Unable to save the file" << endl;
        delete file;
        file = 0;
        return;
    }

    ftp->setTransferMode(QFtp::Passive);
    ftp->connectToHost(url.host(), url.port(21));
    ftp->login("user", "passwd");
    ftp->get(url.path(), file);
    ftp->close();
}

void FtpGet::done(bool error) {
    if (error) {
        cerr << "Error: " << qPrintable(ftp->errorString()) << endl;
    } else {
```

```
            cerr << "File downloaded as " << qPrintable(file->fileName())
                 << endl;
        }
        file->close();
        delete file;
        file = 0;

        emit finished();
}
```

您可以使用 QFtp 的 setTransferMode()来设定传送模式，例如上面的程序设定为被动（Passive）模式，要连接至服务器，使用 connectToHost()方法，要登入则使用 login()方法，下载档案使用 get()（上传则使用 put()）等，接下来写个简单的测试程序：

main.cpp

```
#include <QCoreApplication>
#include <QUrl>
#include "FtpGet.h"
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    QCoreApplication app(argc, argv);

    FtpGet getter;
    getter.downloadFile(
        QUrl("ftp://caterpillar.onlyfun.net/public_html/index.html"));

    QObject::connect(&getter, SIGNAL(finished()),
                     &app, SLOT(quit()));

    return app.exec();
}
```

Qt 的说明文件中，有个 FTP Example 范例，您可以参考当中的程序，了解更多有关 QFtp 的使用方式。

6.63QTcpSocket

QTcpSocket 提供了针对 TCP（Transmission Control Protocol）的接口，可以让您进行以 TCP 为基础的通讯协议、数据传输，像是 SMTP、POP3 等，QTcpSocket 本身设计为异步的（Asynchronized）操作，各个操作阶段会发出相关的 Signal，像是 connected()表示联机建立、bytesWritten()表示传输多少数据、error()带有网络操作过程中的相关错误讯息，QTcpSocket 继承自 QIODevice，所以您可以搭配 QTextStream 或 QDataStream 来使用。

这篇文件与 QTcpServer 将制作一个简单的范例，可以让您进行档案选取并透过网络传送，

这边将制作的是客户端的程序，首先看到定义的部份：

ClientDialog.h

```
#ifndef CLIENTDIALOG_H
#define CLIENTDIALOG_H

#include <QDialog>
#include <QHostAddress>
#include <QTcpSocket>

class QProgressBar;
class QPushButton;
class QFile;

class ClientDialog : public QDialog {
    Q_OBJECT

public:
    ClientDialog(QWidget *parent = 0);
    void setHostAddressAndPort(QString hostAddress, quint16 port);
    void closeConnection();

public slots:
    void start();
    void startTransfer();
    void updateProgress(qint64 numBytes);
    void displayError(QAbstractSocket::SocketError socketError);

private:
    QProgressBar *progressBar;
    QPushButton *startBtn;

    QHostAddress hostAddress;
    quint16 hostPort;
    QTcpSocket client;

    QFile *file;
};

#endif
```

程序中将会有一个按钮与进度列，按下按钮后将可以选择档案并进行网络联机，这是在 start()中会定义，startTransfer()则实际进行数据传输，updateProgress()负责更新进度列状态，若有相关错误讯息，则会由 displayError()来显示。

实作部份如下：

ClientDialog.cpp

```
#include <QApplication>
#include <QProgressBar>
#include <QPushButton>
#include <QVBoxLayout>
#include <QFileDialog>
#include <QMessageBox>
#include <iostream>
using namespace std;

#include "ClientDialog.h"

ClientDialog::ClientDialog(QWidget *parent) : QDialog(parent) {
    progressBar = new QProgressBar;
    progressBar->setValue(0);

    startBtn = new QPushButton("Start");

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(progressBar);
    layout->addWidget(startBtn);
    this->setLayout(layout);

    connect(startBtn, SIGNAL(clicked()), this, SLOT(start()));
    connect(&client, SIGNAL(connected()), this, SLOT(startTransfer()));
    connect(&client, SIGNAL(bytesWritten(qint64)),
            this, SLOT(updateProgress(qint64)));
    connect(&client, SIGNAL(error(QAbstractSocket::SocketError)),
            this, SLOT(displayError(QAbstractSocket::SocketError)));
}

void ClientDialog::setHostAddressAndPort(QString address, quint16 port) {
    hostAddress.setAddress(address);
    this->hostPort = port;
}

void ClientDialog::start() {
    QString fileName = QFileDialog::getOpenFileName(
                                    this, "Open File", "F:\\", "All Files (*.*)");
    if(fileName == NULL) {
        return;
    }
```

```
    file = new QFile(fileName);

    startBtn->setEnabled(false);

    client.connectToHost(hostAddress, hostPort);

    QApplication::setOverrideCursor(Qt::WaitCursor);
}

void ClientDialog::startTransfer() {
     if (!file->open(QIODevice::ReadOnly)) {
          cerr << "Unable to read the file" << endl;
          delete file;
          file = 0;
          return;
     }

     client.write(file->readAll());
}

void ClientDialog::updateProgress(qint64 numBytes) {
    int written = progressBar->value() + (int)numBytes;

    progressBar->setMaximum(file->size());
    progressBar->setValue(written);

    if(progressBar->value() == progressBar->maximum()) {
        closeConnection();
    }
}

void ClientDialog::closeConnection() {
    client.close();

    file->close();
    delete file;
    file = 0;

    progressBar->reset();
    startBtn->setEnabled(true);

    QApplication::restoreOverrideCursor();
}
```

```
void ClientDialog::displayError(QAbstractSocket::SocketError socketError) {
    QMessageBox::information(this, "Network error",
                             "The following error occurred: " + client.errorString());
    closeConnection();
}
```

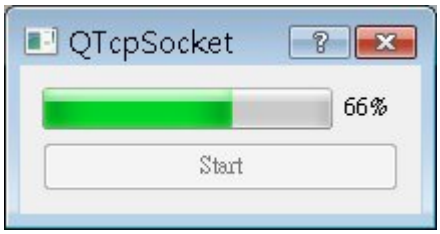
QTcpSocket 的 connected()是在联机成功时发出，此时连接 startTransfer()进行档案读取与传送，每送出一笔数据就会发出 bytesWritten()，并带有送出的数据长度讯息，这个讯息可用以更新进度列，当进度达百分之百之后，关闭连结。

联机过程中若有错误发生，会发出 error()，其中错误若为 QTcpSocket::RemoteHostClosedError，则表示远程中断联机，由于在这个简单范例中，档案传输完成后，客户端会主动中断联机，所以 QTcpSocket::RemoteHostClosedError 在 QTcpServer 中的范例，就作为判断档案传送是否完成之用。

您可以如下使用这个 ClientDialog：

```
ClientDialog *client = new ClientDialog;
client->setHostAddressAndPort("127.0.0.1", 9393);
client->setWindowTitle("QTcpSocket");
client->show();
```

下图为执行时的参考画面：



6.64QTcpServer

QTcpSocket 用来与远程服务器联机，如果您要接受客户端联机，则使用 QTcpServer，QTcpServer 使用 listen()方法开始倾开所指定的连接埠，您可以使用 isListening()方法测试是否正在倾听联机，当联机发生时，QTcpServer 会发出 newConnection()的 Signal，您可以使用 QTcpServer 的 nextPendingConnection()取得代表客户端联机的 QTcpSocket 对象，接下来就可以使用它来与客户端进行数据传输。

配合 QTcpSocket 中的范例，以下制作一个 ServerDialog 来接受客户端的档案传送，首先是 ServerDialog 的定义：

ServerDialog.h

```
#ifndef SERVERDIALOG_H
#define SERVERDIALOG_H

#include <QDialog>
```

```
#include <QTcpServer>

class QLabel;
class QFile;
class QTcpSocket;

class ServerDialog : public QDialog {
    Q_OBJECT

public:
    ServerDialog(QWidget *parent = 0);
    void setReceivedFileName(QString fileName);
    void listen(quint16 port);

public slots:
    void acceptConnection();
    void updateProgress();
    void displayError(QAbstractSocket::SocketError socketError);

private:
    QLabel *label;
    QTcpServer server;
    QTcpSocket *clientConnection;
    int bytesReceived;
    QFile *file;
};

#endif
```

当联机发生时，QTcpServer 会发出 newConnection()的 Signal，程序中将之连接至 acceptConnection()，而每当有数据可以准备读取时，代表客户端联机的 QTcpSocket 会发出 readyRead()信号，这将之连接至 updateProgress()，当中将进行档案储存与目前接受容量显示，以下为实作内容：

ServerDialog.cpp

```
#include <QApplication>
#include <QProgressBar>
#include <QVBoxLayout>
#include <QHostAddress>
#include <QLabel>
#include <QAbstractSocket>
#include <QTcpSocket>
#include <QMessageBox>
#include <QFile>
```

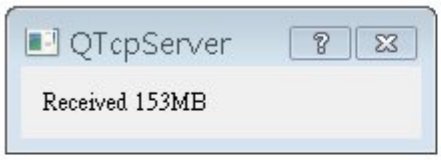
```
#include <iostream>
using namespace std;

#include "ServerDialog.h"

ServerDialog::ServerDialog(QWidget *parent) : QDialog(parent) {
	label = new QLabel("Received:");
	QVBoxLayout *layout = new QVBoxLayout;
	layout->addWidget(label);
	this->setLayout(layout);

	connect(&server, SIGNAL(newConnection()),
			this, SLOT(acceptConnection()));
}

void ServerDialog::setReceivedFileName(QString fileName) {
	file = new QFile(fileName);
}

void ServerDialog::listen(quint16 port) {
	server.listen(QHostAddress::Any, port);
}

void ServerDialog::acceptConnection() {
	if (!file->open(QIODevice::WriteOnly)) {
		cerr << "Unable to write the file" << endl;
		delete file;
		file = 0;
		return;
	}

	clientConnection = server.nextPendingConnection();

	connect(clientConnection, SIGNAL(readyRead()),
			this, SLOT(updateProgress()));
	connect(clientConnection, SIGNAL(error(QAbstractSocket::SocketError)),
			this, SLOT(displayError(QAbstractSocket::SocketError)));

	server.close();

	QApplication::setOverrideCursor(Qt::WaitCursor);
}

void ServerDialog::updateProgress() {
```

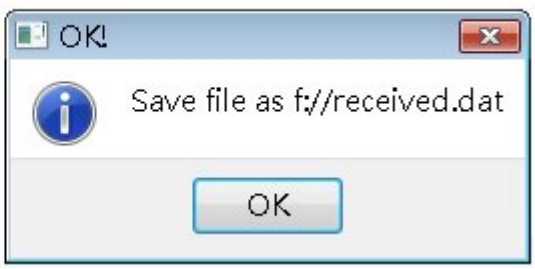
```
    bytesReceived += (int) clientConnection->bytesAvailable();
    file->write(clientConnection->readAll());

    QString txt = "Received %1MB";

    label->setText(txt.arg(bytesReceived / (1024 * 1024)));
}

void ServerDialog::displayError(QAbstractSocket::SocketError socketError) {
    file->close();

    if (socketError == QTcpSocket::RemoteHostClosedError) {
        QMessageBox::information(this,
                "OK!", "Save file as " + file->fileName());
    }
    else {
        QMessageBox::information(this, "Network error",
            "The following error occurred: " + clientConnection->errorString());
    }
    delete file;
    file = 0;

    QApplication::restoreOverrideCursor();
}
```

您可以用以下方式来使用 ServerDialog：

```
ServerDialog *server = new ServerDialog;
server->setReceivedFileName("f://received.dat");
server->listen(9393);
server->setWindowTitle("QTcpServer");
server->resize(200, 10);
server->show();
```

下图为执行时的参考画面：





第七章 进阶议题

7.1 多执行绪（Multithreading）

7.11QThread

一个进程（Process）是一个包括有自身执行地址的程序，在一个多任务的操作系统中，可以分配 CPU 时间给每一个进程，CPU 在片段时间中执行某个进程，然后下一个时间片段跳至另一个进程去执行，由于转换速度很快，这使得每个程序像是在同时进行处理一般。

一个执行绪是进程中的一个执行流程，一个进程中可以同时包括多个执行绪，也就是说一个程序中同时可能进行多个不同的子流程，这使得一个程序可以像是同时间处理多个事务，例如一方面接受网络上的数据，另一方面同时计算数据并显示结果，一个多执行绪程序可以同时间处理多个子流程。

在 Qt 中，有许多类别，其本身在一些操作上即设计为异步，透过 Signal 与 Slot，可以让您不用了解多执行绪，也可以实现非阻断的操作，但某些时候，您仍必须亲自实作多执行绪功能。

在 Qt 中要实现执行绪功能，可以继承 QThread 类别，并重新定义 run()方法，之后要启动一个执行绪，则建构这个自订的对象，并执行 start()方法。

下面这个程序是个简单的程序，您可以看到如何继承 QThread、重新定义 run()方法及如何启动执行绪，程序中将以两个执行绪「同时」对一个 QPixmap 画圆，显示两个「同时」进行的流程：

CircieThread.h

```
#ifndef CIRCLETHREAD_H
#define CIRCLETHREAD_H

#include <QThread>

class QLabel;
class QPixmap;

class CircleThread : public QThread {
    Q_OBJECT

public:
    CircleThread(QLabel *label, QPixmap *pixmap, int y);

protected:
    void run();

private:
    QLabel *label;
```

```
    QPixmap *pixmap;
    int y;
};

#endif
```

CircleThread 建构函式中，QPixmap 是 QLabel 将显示的图片，而 y 值是画圆时的位置，CircleThread 实作如下：

CircleThread.cpp

```
#include "CircleThread.h"
#include <QPainter>
#include <QLabel>
#include <QPixmap>

CircleThread::CircleThread(QLabel *label, QPixmap *pixmap, int y) {
    this->label = label;
    this->pixmap = pixmap;
    this->y = y;
}

void CircleThread::run() {
    QPainter painter(pixmap);

    for(int i = 10; i < 300; i += 10) {
        painter.drawEllipse(i, y, 30, 30);
        label->setPixmap(*pixmap);
        QThread::msleep(500);
    }
}
```

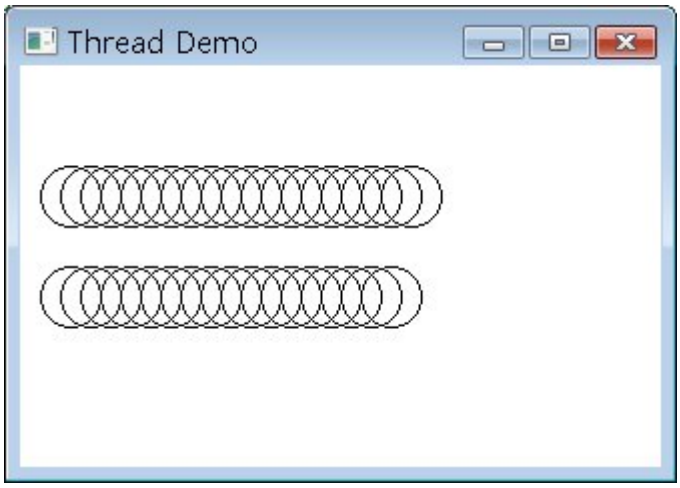
在 run()方法中,将在 QPixmap 上建构 QPainter,然后依序画 10 个圆,接着将画好的 QPixmap 再次设置给 QLabel，以重新在 QLabel 上显示新的绘制画面。QThread::msleep()可以令目前的执行绪暂停所设置的毫秒数。您可以撰写以下的程序来使用 CircleThread：

main.cpp

```
#include <QApplication>
#include <QLabel>
#include "CircleThread.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QLabel *label = new QLabel;
    label->resize(320, 200);
```

```
    QPixmap pixmap(320, 200);
    pixmap.fill(Qt::white);

    CircleThread *thread1 = new CircleThread(label, &pixmap, 50);
    CircleThread *thread2 = new CircleThread(label, &pixmap, 100);

    label->setWindowTitle("Thread Demo");
    label->show();

    thread1->start();
    thread2->start();

    return app.exec();
}
```

启动执行绪是使用 start()方法，下图为执行时的参考画面，您可以看到一个窗口，两个圆在「同时」绘制，虽说是同时，其实也只是错觉而已，其实是 CPU 往来两个流程之间不断的进行绘制圆的动作而已。：



7.12 执行绪的停止

如果想要停止执行绪，QThread 有个 terminate()方法，但是这个方法并不建议使用，因为执行绪会直接停止正在进行的程序流程，无论现在是在流程的哪个位置，这会使得一些资源的善后工作无法完成，或因程序流程嘎然中止而导致不可预期的程序错误。

一个执行绪要停止，基本上就是执行完 run()方法，让它进入完成（Finished），简单的说，如果您想要停止一个执行绪的执行，就要提供一个方式让执行绪可以执行完 run()，而这也是您自行实作执行绪停止的基本概念。

如果执行绪的 run()方法中执行的是一个重复执行的循环，您可以提供一个 flag 来控制循环是否执行，藉此让循环有可能终止、执行绪可以离开 run()方法以终止执行绪，以下面的实例来说，您提供一个 bool 的 stopped 变量：

MessageThread.h

```
#ifndef MESSAGETHREAD_H
#define MESSAGETHREAD_H

#include <QThread>

class MessageThread : public QThread {
    Q_OBJECT

public:
    MessageThread();
    void setMessage(const QString &message);
    void stop();

protected:
    void run();

private:
    QString msg;
    bool stopped;
};

#endif
```

在这个类别中，您可以藉由 setMessage()设定要显示的消息正文，在 run()方法中，while 循环由 stopped 变量判断是否继续循环：

MessageThread.cpp

```
#include "MessageThread.h"
#include <iostream>
using namespace std;

MessageThread::MessageThread() {
    this->stopped = false;
}

void MessageThread::setMessage(const QString &msg) {
    this->msg = msg;
}

void MessageThread::stop() {
    stopped = true;
}

void MessageThread::run() {
```

```
    while (!stopped) {
        cerr << qPrintable(msg) << endl;
        QThread::sleep(1);
    }
}
```

若要停止执行绪，可以呼叫 stop()方法，这会使得 stopped 变量设为 true，而使得 while 循环可以结束，从而可以让执行绪执行完 run()而进入完成。

要判断执行绪是否正在执行，可以使用 QThread 的 isRunning()方法，要判断执行绪是否完成，可以使用 QThread 的 isFinished()，以下可以写个简单的程序来使用以上的 MessageThread：

DemoDialog.h

```
#ifndef DEMODIALOG_H
#define DEMODIALOG_H

#include "MessageThread.h"
#include <QDialog>

class QWidget;
class QPushButton;
class QCloseEvent;

class DemoDialog : public QDialog {
    Q_OBJECT

public:
    DemoDialog(QWidget *parent = 0);

public slots:
    void startOrStop();

protected:
    void closeEvent(QCloseEvent *event);

private:
    QPushButton *btn;
    MessageThread thread;
};

#endif
```

程序中会有个按钮，按下后可以启动另一个执行绪，并设定按钮文字为「Stop」，若再按下，则会呼叫 MessageThread 的 stop()停止执行绪，此时设定按钮文字为「Finished」，并设定按钮为不可按下，执行绪完成后，再呼叫其 start()方法是没有作用的：

DemoDialog.cpp

```
#include "DemoDialog.h"
#include "MessageThread.h"
#include <QPushButton>
#include <QVBoxLayout>
#include <QCloseEvent>


DemoDialog::DemoDialog(QWidget *parent) : QDialog(parent) {
	btn = new QPushButton("Start", this);
	QVBoxLayout *layout = new QVBoxLayout;
	layout->addWidget(btn);
	this->setLayout(layout);

	thread.setMessage("message....");

	connect(btn, SIGNAL(clicked()), this, SLOT(startOrStop()));
}

void DemoDialog::startOrStop() {
	if(thread.isRunning()) {
		thread.stop();
		btn->setText("Finished");
		btn->setEnabled(false);
	}
	else {
		thread.start();
		btn->setText("Stop");
	}
}

void DemoDialog::closeEvent(QCloseEvent *event) {
	thread.stop();
	thread.wait();
	event->accept();
}
```

QThread 的 wait()方法，可以确实的等待执行绪完成，再进行接下来的动作，您也可以指定 wait()的时间，在时间到时，无论如何就进行接下来的动作。

可以撰写以下的简单程序来使用 DemoDialog：

main.cpp

```
#include <QApplication>
```

```
#include "DemoDialog.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    DemoDialog *demoDialog = new DemoDialog;
    demoDialog->setWindowTitle("Thread Demo");
    demoDialog->resize(200, 50);
    demoDialog->show();

    return app.exec();
}
```

有关于执行绪的终止，还可以参考 Two-phase Termination 模式。

7.13QMutex 与 QMutexLocker

如果您的程序只是一个单执行绪，单一流程的程序，那么通常您只要注意到程序逻辑的正确，您的程序通常就可以正确的执行您想要的功能，但当您的程序是多执行绪程序，多流程同时执行时，那么您就要注意到更多的细节，例如在多执行绪共享同一对象的数据时。

如果一个对象所持有的数据可以被多执行绪同时共享存取时，您必须考虑到「数据同步」的问题，所谓数据同步指的是两份数据的整体性一致，例如对象 A 有 name 与 id 两个属性，而有一份 A1 数据有 name 与 id 的数据要更新对象 A 的属性，如果 A1 的 name 与 id 设定给 A 对象完成，则称 A1 与 A 同步，如果 A1 数据在更新了对象的 name 属性时，突然插入了一份 A2 数据更新了 A 对象的 id 属性，则显然的 A1 数据与 A 就不同步，A2 数据与 A 也不同步。

数据在多执行绪下共享时，就容易因为同时多个执行绪可能更新同一个对象的信息，而造成对象数据的不同步，因为数据的不同步而可能引发的错误通常不易察觉，而且可能是在您程序执行了几千几万次之后，才会发生错误，而这通常会发生在您的产品已经上线之后，甚至是程序已经执行了几年之后。

这边举个简单的例子，考虑您设计这么一个类别：

UserInfo.h

```
#ifndef USERINFO_H
#define USERINFO_H

#include <QString>

class UserInfo {
public:
    UserInfo();
    void setNameAndID(const QString &name, const QString &id);
```

```
private:
    bool checkNameAndID();

    QString name;
    QString id;
    long count;
};

#endif
```

UserInfo.cpp

```
#include "UserInfo.h"
#include <QString>
#include <iostream>
using namespace std;

UserInfo::UserInfo() {
    name = "nobody";
    id = "N/A";
}

void UserInfo::setNameAndID(const QString &name, const QString &id) {
    this->name = name;
    this->id = id;
    if(!checkNameAndID()) {
        cout << count
            << ": illegal name or ID....."
            << endl;
    }
    count++;
}

bool UserInfo::checkNameAndID() {
    return (name.at(0) == id.at(0)) ? true : false;
}
```

在这个类别中，您可以设定使用者的名称与缩写 id，并简单检查一下名称与 id 的第一个字是否相同，单就这个类别本身而言，它并没有任何的错误，但如果它被用于多执行绪的程序中，而且同一个对象被多个执行存取时，就会"有可能"发生错误，来写个简单的测试程序：

CheckerThread.h

```
ifndef CHECKERTHREAD_H
#define CHECKERTHREAD_H
#include <QThread>
#include <QString>
```

```
class UserInfo;

class CheckerThread : public QThread {
public:
    CheckerThread(UserInfo *userInfo,
                  const QString &name, const QString &id);

protected:
    void run();

private:
    UserInfo *userInfo;
    QString name;
    QString id;
};

#endif
```

CheckerThread.cpp

```
#include "CheckerThread.h"
#include "UserInfo.h"

CheckerThread::CheckerThread(UserInfo *userInfo,
                             const QString &name, const QString &id) {
    this->userInfo = userInfo;
    this->name = name;
    this->id = id;
}

void CheckerThread::run() {
    while(true) {
        userInfo->setNameAndID(name, id);
    }
}
```

main.cpp

```
#include <QCoreApplication>
#include "UserInfo.h"
#include "CheckerThread.h"

int main(int argc, char *argv[]) {
    QCoreApplication app(argc, argv);

    UserInfo *userInfo = new UserInfo;

    CheckerThread *thread1 =
```

```
            new CheckerThread(userInfo, "Justin Lin", "J.L.");
    CheckerThread *thread2 =
            new CheckerThread(userInfo, "Shang Hwang", "S.H.");

    thread1->start();
    thread2->start();
    thread1->wait();
    thread2->wait();

    return 0;
}
```

来看一下执行时的一个例子（为简化范例，并无设置停止条件，请直接使用工作管理员结束程序）：

```
2522482: illegal name or ID.....
2522498: illegal name or ID.....
2522514: illegal name or ID.....
2522530: illegal name or ID.....
2522542: illegal name or ID.....
2522560: illegal name or ID.....
2522815: illegal name or ID.....
2522832: illegal name or ID.....
2522858: illegal name or ID.....
```

看到了吗？如果以单执行绪的观点来看，上面的讯息在测试中根本不可能出现，然而在这个程序中却出现了错误，而且重点是，第一次错误是发生在第 2522482 次的设定（您的计算机上可能是不同的数字），如果您在程序完成并开始应用之后，这个时间点可能是几个月甚至几年之后。

问题出现哪？在于这边：

```
void UserInfo::setNameAndID(const QString &name, const QString &id) {
    this->name = name;
    this->id = id;
    if(!checkNameAndID()) {
        cout << count
             << ": illegal name or ID....."
             << endl;
    }
    count++;
}
```

虽然您设定给它的参数并没有问题，在某个时间点时，thread1 设定了"Justin Lin", "J.L." 给 name 与 id，在进行测试的前一刻，thread2 可能此时刚好呼叫 setNameAndID("Shang Hwang", "S.H.")，在 name 被设定为"Shang Hwang"时，checkNameAndID()开始执行，此时

name 等于"Shang Hwang"，而 id 还是"J.L."，所以 checkNameAndID()就会传回 false，结果就显示了错误讯息。

您必须同步数据对对象的更新，也就是在有一个执行绪正在设定 userInfo 对象的数据时，不可以又被另一个执行绪同时进行设定，您可以使用 QMutex 来进行这个动作，例如在 UserInfo 中宣告 QMutex：

```
class UserInfo {
...
private:
    ...
    QMutex mutex;
    ....
};
```

然后改写一下 setNameAndID()，您使用 QMutex 的 lock()与 unlock()方法来锁定同步区域：

```
void UserInfo::setNameAndID(const QString &name, const QString &id) {
    mutex.lock();
    this->name = name;
    this->id = id;
    if(!checkNameAndID()) {
        cout << count
             << ": illegal name or ID....."
             << endl;
    }
    count++;
    mutex.unlock();
}
```

当执行绪执行 QMutex 的 lock()时，它会锁定接下来的程序流程，其它尝试再执行 lock()的执行绪必须等待目前执行绪先执行了 QMutex 的 unlock()，才可以取得锁定，QMutex 还有个 tryLock()，如果 QMutex 已经锁定，则 tryLock()立即返回。

您也可以使用 QMutexLocker，这是个方便的类别，建构时以 QMutex 对象作为自变量并进行锁定，而解构时自动解除锁定，例如可以改写一下 setNameAndID()如下，效果相同：

```
void UserInfo::setNameAndID(const QString &name, const QString &id) {
    QMutexLocker locker(&mutex);
    this->name = name;
    this->id = id;
    if(!checkNameAndID()) {
```

```
        cout << count
             << ": illegal name or ID....."
             << endl;
    }
    count++;
}
```

7.14QWaitCondition

在执行绪的同步化时，有些条件下执行绪必须等待，有些条件下则不用，这可以使用QWaitCondition来达到。

例如在生产者（Producer）与消费者（Consumer）的例子中，如果生产者会将产品交给店员，而消费者从店员处取走产品，店员一次只能持有固定数量产品，如果生产者生产了过多的产品，店员叫生产者等一下（wait），如果店中有空位放产品了再唤醒（wake）生产者继续生产,如果店中没有产品了,店员会告诉消费者等一下(wait),如果店中有产品了再唤醒(wake)消费者来取走产品。

以下举一个最简单的：生产者每次生产一个int整数交给在店员上，而消费者从店员处取走整数，店员一次只能持有一个整数，以程序实例来看，首先是生产者：

Producer.h

```
#ifndef PRODUCER_H
#define PRODUCER_H

#include <QThread>

class Clerk;

class Producer : public QThread {

public:
    Producer(Clerk *clerk);

protected:
    void run();

private:
    Clerk *clerk;
};

#endif
```

Producer.cpp

```
#include "Producer.h"
```

```
#include "Clerk.h"

Producer::Producer(Clerk *clerk) {
    this->clerk = clerk;
}

void Producer::run() {
    // 生产 1 到 10 的整数
    for(int product = 1; product <= 10; product++) {
        // 暂停随机时间
        QThread::msleep(qrand() / 100);
        // 将产品交给店员
        clerk->setProduct(product);
    }
}
```

再来是消费者：

Consumer.h

```
#ifndef CONSUMER_H
#define CONSUMER_H

#include <QThread>

class Clerk;

class Consumer : public QThread {

public:
    Consumer(Clerk *clerk);

protected:
    void run();

private:
    Clerk *clerk;
};

#endif
```

Consumer.cpp

```
#include "Consumer.h"
#include "Clerk.h"

Consumer::Consumer(Clerk *clerk) {
    this->clerk = clerk;
```

```
}

void Consumer::run() {
    // 消耗 10 个整数
    for(int i = 1; i <= 10; i++) {
        // 暂停随机时间
        QThread::msleep(qrand() / 10);
        // 从店员处取走整数
        clerk->getProduct();
    }
}
```

生产者将产品放至店员，而消费者从店员处取走产品，所以店员来决定谁必须等待并等候唤醒：

Clerk.h

```
#ifndef CLERK_H
#define CLERK_H

#include <QMutex>
#include <QWaitCondition>

class Clerk {
public:
    Clerk();
    void setProduct(int product);
    int getProduct();

private:
    int product;
    QMutex mutex;
    QWaitCondition waitCondition;
};

#endif
```

Clerk.cpp

```
#include "Clerk.h"
#include <iostream>
using namespace std;

Clerk::Clerk() {
    product = -1;
}

void Clerk::setProduct(int product) {
```

```
    mutex.lock();

    if(this->product != -1) {
        // 目前店员没有空间收产品，请稍候!
        waitCondition.wait(&mutex);
    }

    this->product = product;
    cout << "生产者设定 " << this->product << endl;

    // 唤醒一个消费者可以继续工作了
    waitCondition.wakeOne();

    mutex.unlock();
}

int Clerk::getProduct() {
    mutex.lock();

    if(this->product == -1) {
        // 缺货了，请稍候!
        waitCondition.wait(&mutex);
    }

    int p = this->product;
    cout << "消费者取走 " << this->product << endl;

    this->product = -1;

    // 唤醒一个生产者可以继续工作了
    waitCondition.wakeOne();

    mutex.unlock();

    return p;
}
```

使用这么一个程序来测试：

main.cpp

```
#include <QCoreApplication>
#include "Clerk.h"
#include "Producer.h"
#include "Consumer.h"
```

```
int main(int argc, char *argv[]) {
    QCoreApplication app(argc, argv);

    Clerk *clerk = new Clerk;

    Producer *producer = new Producer(clerk);
    Consumer *consumer = new Consumer(clerk);

    producer->start();
    consumer->start();
    producer->wait();
    consumer->wait();

    return 0;
}
```

执行结果：

```
生产者设定 (1)
消费者取走 (1)
生产者设定 (2)
消费者取走 (2)
生产者设定 (3)
消费者取走 (3)
生产者设定 (4)
消费者取走 (4)
生产者设定 (5)
消费者取走 (5)
生产者设定 (6)
消费者取走 (6)
生产者设定 (7)
消费者取走 (7)
生产者设定 (8)
消费者取走 (8)
生产者设定 (9)
消费者取走 (9)
生产者设定 (10)
消费者取走 (10)
```

生产者会生产 10 个整数，而消费者会消耗 10 个整数，由于店员处只能放置一个整数，所以每生产一个就消耗一个，其结果如上所示是无误的。

7.15QReadWriteLock 与 QSemaphore

使用 QMutex 与 QMutexLocker 时，被锁定的区域一次只允许一个执行绪，其它执行绪必须等待解除锁定，方可尝试取得锁定并执行程序，在大量执行绪存取共享资源的情况下，执行

绪的等待必然造成效能上的瓶颈。

在某些时候，共享资源是可以被多个执行绪以只读方式进行读取，而不会影响资源共享的安全性，在这种情况下，不必让执行绪等待，您可以使用 QReadWriteLock 来区分只读或写入的共享锁定，例如：

```
QReadWriteLock lock;

void ReaderThread::run() {
    ...
    lock.lockForRead();
    // 读取共享数据
    lock.unlock();
    ...
}

void WriterThread::run() {
    ...
    lock.lockForWrite();
    // 设定、写入共享资源
    lock.unlock();
    ...
}
```

QReadWriteLock 的 lockForRead()方法，在共享资源正在进行设定或写入，也就是另一个区域已被 lockForWrite()时，才进行执行绪的阻断，如果没有设定或写入的动作，则 lockForRead()并不会让其它执行绪等待。

QReadLocker 与 QWriteLocker 为一个方便的类别，以 QReadWriteLock 对象为自变量来建构，建构时进行读取锁定或写入锁定，解构时解除鑌定。

有些共享资源拥有一定的可存取次数，在多执行绪存取的情况下，可以同时允许一定数量的执行绪来存取共享资源，您可以自行计数执行绪进入与离开的次数，例如在 QWaitCondition 中生产者与消费者的例子，若店员可以持有不只一个产品，则生产者或消费者可以存取店员共享区的次数，则必须自行实作计算。

您也可以直接使用 QSemaphore，它为您提供计数信号，在建构 QSemaphore，可以指定资源可获取的量（次数），不设定则预设为 0，您可以使用 acqire()来表示将存取多少资源，使用 release()表示将释放多少资源，使用 available()来得知有多少资源可以存取，例如：

```
QSemaphore sem(5);      // sem.available() == 5

sem.acquire(3);         // sem.available() == 2
```

```
sem.acquire(2);          // sem.available() == 0
sem.release(5);          // sem.available() == 5
sem.release(5);          // sem.available() == 10
```

在 Qt 的文件中有个 Semaphores Example，使用 QSemaphore 来实作生产者与消费者，您可以观看其 原始程序代码，该程序有一个 8192 空间的 char 型态 buffer 数组，可以被生产者与消费者存取，其中：

```
void Producer::run() {
     qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));
     for (int i = 0; i < DataSize; ++i) {
         freeBytes.acquire();
         buffer[i % BufferSize] = "ACGT"[(int)qrand() % 4];
         usedBytes.release();
     }
}
```

freeBytes 为 QSemaphore 的实例，初始为 8192 次的资源，每 acqire()一次，表示将用掉 buffer 数组中的一个索引位置，用以限制生产者可以放入 buffer 数组的字符数量，而 usedBytes 为 QSemaphore 的实例，是用来告诉消费者，buffer 中已使用的索引量，每次存入 buffer 一个字符，就 release 一次 usedBytes，表示消费者可以多消费一次字符，在消费者这边：

```
void Consumer::run() {
     for (int i = 0; i < DataSize; ++i) {
         usedBytes.acquire();
         fprintf(stderr, "%c", buffer[i % BufferSize]);
         freeBytes.release();
     }
     fprintf(stderr, "\n");
}
```

每当消费者要取出 buffer 中的一个字符时，就 acqiire 一次 usedBytes，表示用掉一个索引位置，最后再 release 一次 freeBytes，这用以告知生产者，多出一个可以存放 buffer 的次数。

7.16QThreadStorage

无论如何，要编写一个多执行绪安全（thread-safe）的程序总是困难的，为了使用的共享资源，您必须小心的对共享资源进行同步，同步带来一定的效能延迟，而另一方面，在处理同步的时候，又要注意对象的锁定与释放，避免产生死结，种种因素都使得编写多执行绪程序变得困难。

Thread-Specific Storage 模式尝试从另一个角度来解释多执行绪共享资源的问题，其思考点很简单，即然共享资源这么困难，那么就干脆不要共享，何不为每个执行绪创造一个资源的复本，将每一个执行绪存取数据的行为加以隔离，其实现的方法，就是给予每一个执行绪一个特定空间来保管该执行绪所独享的资源，也因此而称之为 Thread- Specific Storage 模式。

实作 Thread-Specific Storage 模式，最基本的方式，就是使用一个关联容器对象，例如 关联容器（QMap、QHash...），在执行绪获得个别资源时，使用 QThread::currentThread()取得执行绪 ID，将 ID 为键（Key）、资源为值（Value）存入关联容器之中，要取得执行绪个别资源时，则以执行绪 ID 为键来取得相对应的资源。

下面这个简单的 MessageThreadLocal 简单实作了 Thread-Specific Storage 的概念：

```
class MessageThreadLocal {
public:
    QString get();
    void set(const QString &message);

private:
    QMap<QThread*, QString> map;
};

QString MessageThreadLocal::get() {
    QThread *thread = QThread::currentThread();
    QString message = map.value(thread, "N.A.");
    if(message == "N.A." && !map.contains(thread)) {
        map.insert(thread, "N.A.");
    }
    return message;
}

void MessageThreadLocal::set(const QString &message) {
    map.insert(QThread::currentThread(), message);
}
```

在 Qt 中，您不用亲自实作这样的 ThreadLocal 类别，它提供有 QThreadStorage 类别，可以让您直接用来实现 Thread-Specific Storage 模式，例如 API 文件中 QThreadStorage 的说明中，提供一个简单的范例片段，示范如何为每个执行绪储存一个快取对象：

```
QThreadStorage<QCache<QString, SomeClass> *> caches;

void cacheObject(const QString &key, SomeClass *object) {
```

```
    if (!caches.hasLocalData())
        caches.setLocalData(new QCache<QString, SomeClass>);

    caches.localData()->insert(key, object);
}

void removeFromCache(const QString &key) {
    if (!caches.hasLocalData())
        return;

    caches.localData()->remove(key);
}
```

使用 QThreadStorage 时要注意的是，由于某些编译器的限制，QThreadStorage 只能储存指标。

7.2 国际化（Internationalization）

7.21 使用 Unicode

在 简单的显示中文（使用 Unicode 转换） 中，介绍过简单的中文显示，在 Qt 中，使用 QString 来储存字符串，QString 中的每个字符则是 QChar 的实例，QChar 使用 Unicode 来储存，Unicode 包括了 ASCII 及 ISO 8859-1（Latin-1），您可以直接指定 Unicode 编码来指定要储存的字符，例如：

main.cpp

```
#include <QApplication>
#include <QLabel>
#include <QTextCodec>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QString text;
    text[0] = QChar(0x826F); //  良
    text[1] = QChar(0x845B); //  葛
    text[2] = QChar(0x683C); //  格

    QLabel *label = new QLabel;
    label->setText(text);
    label->setWindowTitle("Unicode");
    label->resize(200, 50);
    label->show();
```

```
    return app.exec();
}
```

程序中使用 Unicode 编码来指定 QChar 的字符，并设定为 QString 的元素，程序的执行结果将如下：

若想要得知 QChar 的 Unicode 字符之编码，可以使用 QChar 的 unicode()方法，如果 QChar 实际是在 ASCII 子集中，则可以使用 isSpace()、isDigital()、isSymbol()、isUpper()、isLower()等方法来判断字符是否为空白、数字、符号、大写、小写等。

在 简单的显示中文（使用 Unicode 转换） 中，使用了 QTextCodec 来进行 C/C++ 的字符编码转换为 Unicode 的动作，最基本的作法即是以 codecForName()取得 QTextCode 实例，再使用该实例的 toUnicode()进行转换：

```
QTextCodec *codec = QTextCodec::codecForName("Big5-ETen");
...
label->setWindowTitle(codec->toUnicode("良葛格"));
```

另一种方式，则是使用 tr()方法，tr()是 QObject 上所定义的静态方法，并在事先设置 QTextCodec::setCodecForTr()为想要的 QTextCodec 编码实例，例如：

main.cpp

```
#include <QApplication>
#include <QLabel>
#include <QTextCodec>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QTextCodec::setCodecForTr(
            QTextCodec::codecForName("Big5-ETen"));

    QLabel *label = new QLabel;
    label->setText(
            QObject::tr("<center><h1>Qt4 学习笔记</h1></center>"));
    label->setWindowTitle(
            QObject::tr("良葛格"));
    label->resize(200, 50);
    label->show();
```

```
    return app.exec();
}
```

执行结果与 简单的显示中文（使用 Unicode 转换） 是一样的，您也可以用更简单的方法，使用 QTextCodec::setCodecForCStrings()，直接设定 C/C++的字符与 QChar 的转换，这在程序撰写上会更简洁一些，例如：

main.cpp

```
#include <QApplication>
#include <QLabel>
#include <QTextCodec>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QTextCodec::setCodecForCStrings(
            QTextCodec::codecForName("Big5-ETen"));

    QLabel *label = new QLabel;
    label->setText("<center><h1>Qt4 学习笔记</h1></center>");
    label->setWindowTitle("良葛格");
    label->resize(200, 50);
    label->show();

    return app.exec();
}
```

7.22 翻译应用程序

您可以在应用程序中的一些文字位置撰写预设语系文字，例如英语系文字，若要改变整个应用程序中的文字为另一个语系的文字，再提供 qm 文件来进行翻译，要拥有这个功能，首要条件是在您的应用程序中：

所有要翻译的文字，都必须撰写在 QObject::tr()函式之中
实例化 QTranslator，加载.qm 档案
为 QApplication 安装 QTranslator

以 简单的显示中文（使用 Unicode 转换） 为例，示范如何让它拥有置换.qm 文件即可翻译应用程序：

main.cpp

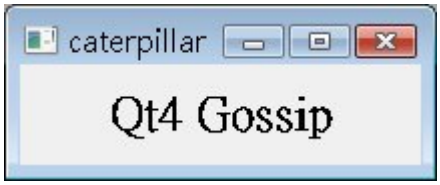
```
#include <QApplication>
#include <QLabel>
#include <QTranslator>
```

```
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QTranslator translator;
    translator.load("hello");
    app.installTranslator(&translator);

    QLabel *label = new QLabel;
    label->setText(
            QObject::tr("<center><h1>Qt4 Gossip</h1></center>"));
    label->setWindowTitle(
            QObject::tr("caterpillar"));

    label->resize(200, 50);
    label->show();

    return app.exec();
}
```

要翻译的文字，一定要直接撰写在 tr()之中，tr()之中不得使用变量，因为稍后使用 lupdate 工具程序处理时，会无法找出需要翻译的文字，例如以下的写法就不行：

```
QString name = "caterpillar";
label->setWindowTitle(name);
```

QTranslator 的 load()方法设置为 hello，这预设会去寻找 hello.qm 文件，档案中包括要进行翻译的文字，上面的应用程序已经可以执行，在不提供.qm 档案时，预设就是显示原始程序代码中的文字：



接着，确定在您的.pro 档案中，设置源文件名称与即将产生的.ts 文件名称：

```
SOURCES += main.cpp
TRANSLATIONS += hello.ts
```

如此就可以直接使用 Qt 附的 lupdate 工具程序来自动产生.ts 档案，指令为：

```
lupdate -verbose yourApp.pro
```

.ts 档案的格式内容为 XML 档案，您可以直接编辑它：

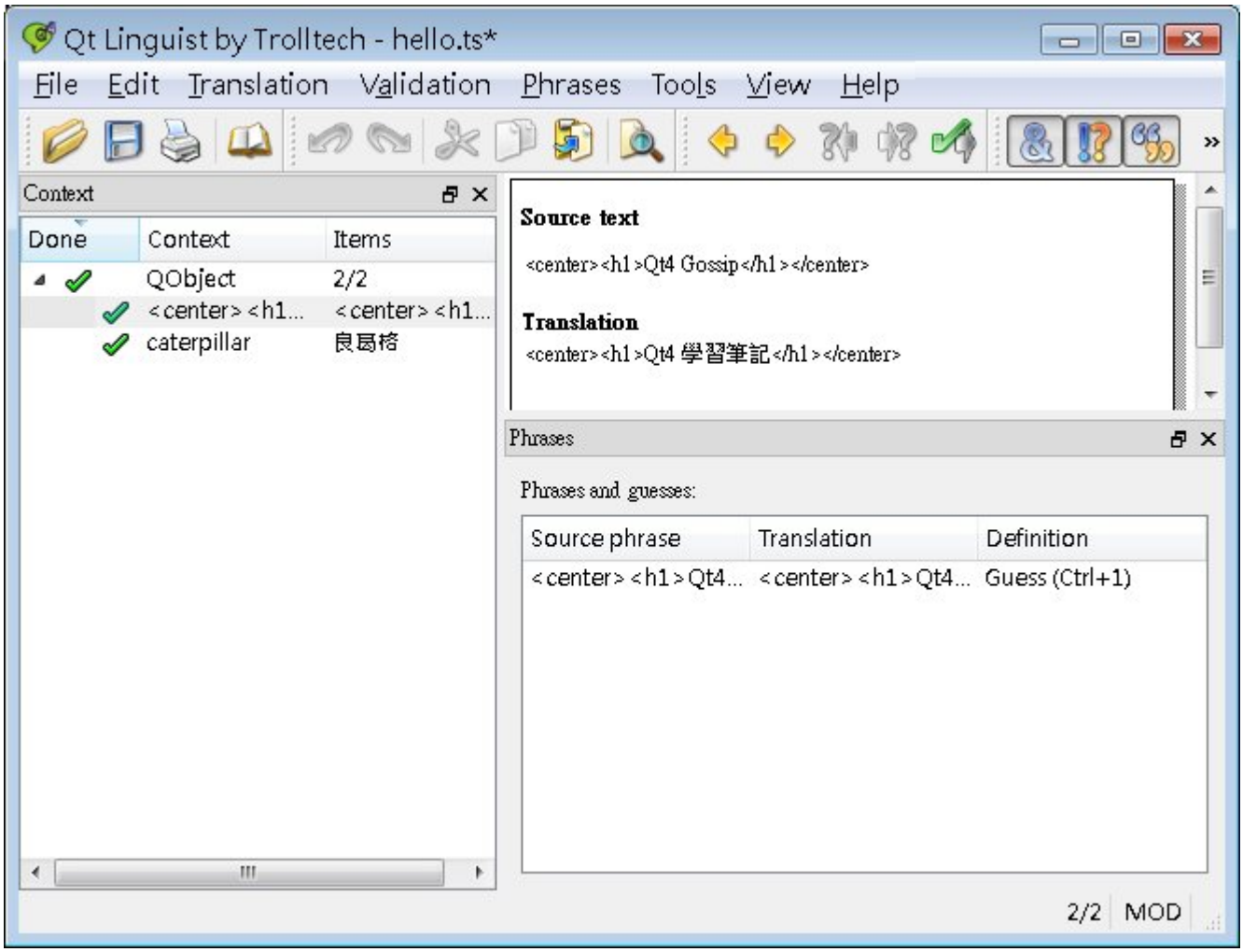
```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS><TS version="1.1">
<context>
    <name>QObject</name>
    <message>
        <location filename="main.cpp" line="15"/>
        <source>&lt;center&gt;&lt;h1&gt;Qt4
Gossip&lt;/h1&gt;&lt;/center&gt;</source>
        <translation type="unfinished"></translation>
    </message>
    <message>
        <location filename="main.cpp" line="17"/>
        <source>caterpillar</source>
        <translation type="unfinished"></translation>
    </message>
</context>
</TS>
```

但更简单的方法是使用 Qt 附的 linguist：

```
linguist hello.ts
```

这会出现 linguist 窗口程序，若要中文，可以执行「Edit/Translation File Settings」，设定为中文语系编辑：

接着可以在右上窗格进行翻译的文字编辑，例如：



修改所有要翻译的文字之后进行储存，原本的.ts 档将修改为如下：

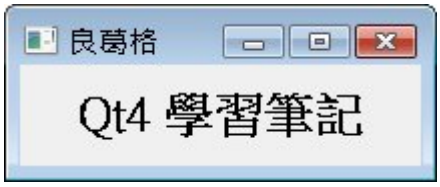
```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS><TS version="1.1" language="zh_TW">
<defaultcodec></defaultcodec>
<context>
    <name>QObject</name>
    <message>
        <location filename="main.cpp" line="15"/>
        <source>&lt;center&gt;&lt;h1&gt;Qt4
Gossip&lt;/h1&gt;&lt;/center&gt;</source>
        <translation>&lt;center&gt;&lt;h1&gt;Qt4　　学　　习　　笔　　记
&lt;/h1&gt;&lt;/center&gt;</translation>
    </message>
    <message>
        <location filename="main.cpp" line="17"/>
        <source>caterpillar</source>
        <translation>良葛格</translation>
    </message>
</context>
```

</TS>

接着使用 Qt 的 lrelease 工具程序，将.ts 档转换为.qm 檔：

```
lrelease hello.ts
```

接着启动应用程序，QTranslator 可以加载 hello.qm 档的内容，并将所有 tr()中的文字作置换，画面如下：



7.23 多国语系选择与切换

接续 翻译应用程序 的内容，在 tr()中基本上不使用变量，这会使得 lupdate 工具程序无法辨识要处理的翻译字符串，不过您可以使用 QT_tr_NOOP()宏函式，在将字符串指定给变量之前，先经过宏函式处理，例如：

```
QString FriendlyConversation::greeting(int type) {
    static const char *greeting_strings[] = {
        QT_TR_NOOP("Hello"),
        QT_TR_NOOP("Goodbye")
    };
    return tr(greeting_strings[type]);
}
```

若觉得麻烦而不想为每个字符串加上 tr()函式，或为了避免撰写时的疏忽而忘了为字符串加上 tr()函式，则可以在.pro 档案中撰写：

```
DEFINES += QT_NO_CAST_FROM_ASCII
```

这可以让前置处理器为每个字符串加上 tr()函式，若前置处理器无法处理的字符串，则会产生编译错误，此时您必须亲自为那些字符串加上 tr()函式。

若要应用程序依系统预设语系而自动选择正确的.qm 档案，以显示正确的语系文字，基本上可以使用 QLocale::System()传回代表预设语系的 QLocale 对象，例如：

```
QApplication app(argc, argv);
QTranslator translator;
translator.load("hello_" + QLocale::system().name()));
app.installTranslator(&translator);
...
```

以上例而言，若是在 zh_TW 之下，则会载入 hello_zh_TW.qm 档案，您也可以搭配 Qt 资源系统，例如撰写一个 qrresource.qrc：

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>hello.qm</file>
</qresource>
<qresource lang="zh_TW">
    <file alias="hello.qm">hello_zh_TW.qm</file>
</qresource>
</RCC>
```

再修改一下应用程序：

```
....
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QTranslator translator;
    translator.load(":/hello");
    app.installTranslator(&translator);
    ....
    return app.exec();
}
```

此时若语系设定为 zh_TW，则会自动去寻找 hello_zh_TW.qm，若无符合的语系对应，则使用预设的 hello.qm。

若要在应用程序中，可以让使用者透过功能选单来自由选择语系切换，则要让 QTranslator 重新加载.qm 档案，然后将重跑一遍设定字符串的程序，例如提供一个函式让功能选单来呼叫：

```
void changeLanguage(const QString &locale) {
    translator.load("hello_" + locale);
    label1->setText(QObject::tr("caterpillar"));
    label2->setText(QObject::tr("Gossip"));
    ....
}
```