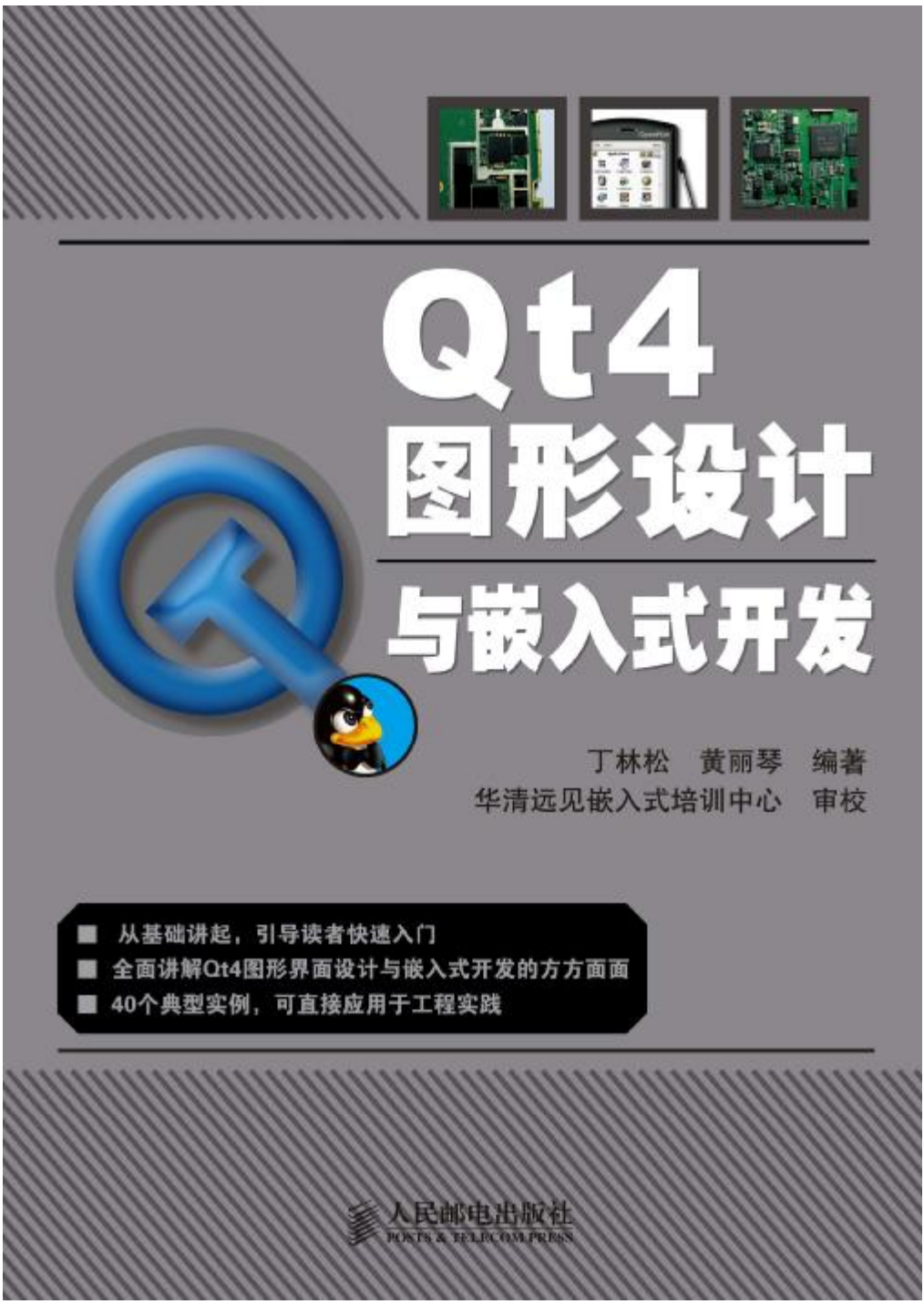

Qt4
图形设计
与嵌入式开发
丁林松　黄丽琴　编著
华清远见嵌入式培训中心　审校
从基础讲起，引导读者快速入门
全面讲解Qt4图形界面设计与嵌入式开发的方方面面
40个典型实例，可直接应用于工程实践
人民邮电出版社
POSTS & TELECOM PRESS

第 1 章　Qt 概述

Qt 是一个功能全面、可开发高性能的、多平台富客户端/服务器端的、C++图形用户界面应用程序框架。

Qt 包含一个丰富的可扩展的类库（Qt 类库），一个功能强大的 GUI 布局与窗体构造器（Qt 设计者），一套用来消除国际化工作流程中的障碍的工具（Qt 语言家）和一个完全可自定义、重新分配的帮助文件或文档浏览器（Qt 助手）。

在 1992 年，一批 Qt 的程序员就开始编写 Qt 程序，直到 1994 年 Trolltech 公司成立，该公司提供所有的有关 Qt 的服务；1996 年 Qt 进入商业领域，它已经成为全世界应用程序基础开发的重要角色。Qt 也是流行 Linux 桌面的 KDE 环境的基础，KDE 是所有主要 Linux 发行版的一个标准组件。

Trolltech 主要为诸如 eBay 公司的 Skype、Google Earth 和 Adobe Systems 的 Photoshop Elements 提供软件和应用程序平台，在 2008 年初 Trolltech 公司被移动电话巨头诺基亚公司接手。

诺基亚公司作为全球最大的移动电话制造商，在召开的关于对 Trolltech 的新闻发布会上表示将继续支持公司进行无线设备和电脑系统的研发，同时还包括网络设备。

1.1 Qt 程序设计简介

Qt 是 Trolltech 公司开发的给予标准框架的图形应用程序，它是一种高效与跨平台的应用程序的解决方案，Qt 支持的平台有微软操作系统、苹果机 OS 以及 Linux 操作系统，并支持了大部分商业的 UNIX 操作系统和 Linux 嵌入式操作系统。

在嵌入式操作系统当中所有的 API 都是在 Qtopia Core 中使用，在本书的最后一章将着重讲解 Qt 的嵌入式开发。

Qt 提供给应用程序开发者大部分的功能，来完成建立适合的、高效率的图形界面程序与后台执行的应用程序，它提供的是一种面向对象的可扩展性能和真正的基于组件的编程模式。建议读者在学习这本书之前最好首先熟悉官方所提供的白皮书。

Qt 的第一次商业版本发行是在 1996 年，Qt 发展到现在，已经提供了成千上万的应用程序，包括 Google 地图、Photoshop 的一些元素以及 Skype 软件。同时 Qt 还是 Linux 桌面系统 KDE 的开发环境，KDE 是 Linux 发行版最广泛的一种图形操作界面。

Qt 支持以下的平台：

Windows 98、Windows NT4.0、Windows ME、Windows 2000、Windows XP、UNIX/X11-Linux 、 SunSolaris 、 HP-UX 、 HPTru64UNIX 、 IBMAIX 、 SGIIRIX 、 MacOSXMacOSX10.3+，与其他的一些版本的 UNIX。在 EmbeddedLinux——Linux 嵌入式平台上需要使用 Framebuffer 帧缓冲的支持。

1.1.1 Qt 版本介绍

在 Qt 发行版本中将要涉及两个版本：Qt 商业版本和 Qt 开源版本。

1．Qt 商业版本

Qt 商业版本是设计商业软件的开发环境，这些商业软件使用了传统的商业软件来发布，它包含了一些更新的功能、技术上的支持和大量的解决方案，开发了使用于行业的一些特定的组件，有一些特殊的功能只在商业用户中使用。

2．Qt 开源版本

Qt 开源版本是用来开发开源的软件，它提供了一些免费的支持，并遵循 QPL 协议。

开放源代码是免费的软件，不牵涉用户的某些权益。任何人都有使用开源软件和参与它的修改的机会，这就意味着其他的人同样可获得你开发的代码。

开源软件应该较多地谈论软件自由，而不是金钱。在自由软件基金会的许可背景下，讲究软件自由，而不是不计成本地去使用软件。

原 Trolltech 公司提供了 Qt 的开源版本，让大部分的人可以使用开源的 Qt 来开发自由的开源软件，所有在 Qt 开源版本下开发的软件都需要遵守 GNU 的公共许可。

自从 Qt 开源以来已经成功地完成了很多优秀的项目，例如，KDE。全世界有相当部分的程序员在使用 Qt 的开源版本，它们没有被告知要缴纳使用费用。在 Qt4X

以上的版本，Qt 开源主要面向的是 UNIX/X11、MacOS 以及 Windows 用户。

开源版本的 Qt 可以在原 TrollTech 公司的网站上进行下载。当然程序员在开发程序的时候也可以使用 Qt 的商业版本，但这样就需要购买一个商业的许可，才可以将你的软件作为商业的软件进行发布。

这样的运作模式是很优秀的，开源软件不断地使用最小的成本增加了软件的功能，商业软件符合了商业软件的发行模式。许多大型的公司因需要保护软件的许可权益，所以愿意购买原 TrollTech 的商业许可权，使得原 TrollTech 公司保证了盈利的来源。

如果你选择了 Qt 的开源的版本，GNU 的权益也就强加给了你，也就是说你要遵守自由的理念，将你开发的开源代码分享给其他的朋友。总之，在 GNU 的协议下你需要将软件的源代码共享到底。

GNU 的协议：http://www.gnu.org。

开源协议 http://www.opensource.org/。

当遇到棘手的问题时，可以通过邮件发送给原 Trolltech 公司 info@trolltech.com。

1.1.2 创建可重用的软件模式

一个大型程序通常由很多模块构成，这些模块提供了程序当中给定的函数、过程和数据的结构。在一般情况下大部分的模块都是现成的，并且它们来自于库函数，而新的模块才需要重新设计。

理论上这些程序模块可以被重复使用，但事实并非如此，我们在设计程序的时候大部分代码要重新书写。

有编程经验的人都知道，在一个程序中大部分的操作都有一个共同的模式，只有那些涉及 I/O 操作以及具体的细节问题才使用到库函数，其他的都是使用组件或大批量的代码来完成工作。I/O API 架构如图 1.1 所示。

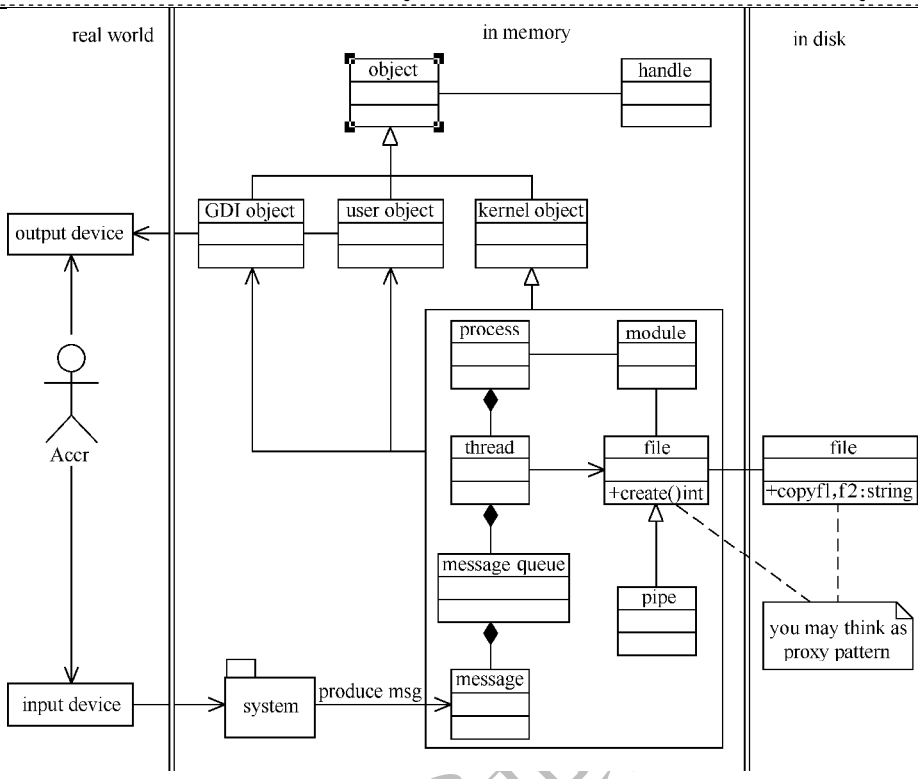


图 1.1　I/O API 架构

可重用的软件模式，一是为了实现软件的开发效率，二是为了提高程序的健壮性。现在可以被重用使用的模块很少，即使有，也缺少标准，即没有可鉴定的方法。C 语言在 1989 年被标准化，现在人们对它已不陌生，并把它作为一个很好的工具来使用。

Qt 程序在潜移默化地形成标准，如果把一个 QPushButton 部件看作一个 C++ 代码的集合，那么除了必须的维护，原 TrollTech 公司就会将它完全地保留在库文件中，它们彼此独立并被程序员调用，它们被看作功能的仓库，可以使用这些功能来完成程序所要做的工作。

有些模块不被重用还有一个原因，即它们往往包含在一个很大的库文件当中。程序员如果学习它的时间成本超过了学习一门新语言的成本，那么程序员不会学习它，企业也不会在这个上面花费更多的成本。

程序库的设计是艰难的，程序员必须小心翼翼地去设计，以保证它的通用性和简单的操作。但程序库的设计又不能太简单或太复杂，太简单了模块完成的功能太少，太复杂了程序员在库文件中又不好查找到合适的功能。

在 Qt 中，它们被集中起来，有元对象编译器对它们进行管理。程序员所做的是构建属性。

大多数的程序构造或者说程序到底怎样按照流程去工作，在大学的学习中已经涉及了，那些刚刚从大学毕业的学生可以很容易地进行一个软件的设计。不但这样，国内的软件企业需要寻求更简单的编写软件的方法，它们往往不重视模块的开发。

下面将深入说明 API 到底是什么。

1．理解接口

要想了解 API，首先从接口下手。接口只是负责表示客户端的标识，它和实现没有任何关系，可以使用不同的结构和算法实现它。

接口是对于实现的调用，它的表示很简单，主要是表现实现的功能。从小的操作上说，保存一个文件或以什么样式打开一个文件，或者通过传递一个数据采取怎样的加密算法并且以什么样的样式返回数值等。

在 C 语言中，这些接口必须在头文件中显示出来，这个文件也就被命名为*.h 文件。通常一些使用到的宏、类型、数据结构、变量以及例程，程序员在写程序的时候使用#include 导入接口。

下面我们来看一个实例：

```
arith_max(int x，int y);
arith_min(int x，int y);
arith_add(int x，int y);
arith_mul(int x，int y);
arith_div(int x，int y);  // 算术接口的实现方法
```

它们都是接口，用来实现不同的标识。

例如，当你使用函数来判断两个整数的大小的时候：10 和 20 哪一个大呢？用 arith_max(int x，int y)函数，它可以返回任何的数值类型，如 int 类型；它可以是 20，主要是比较后两个数的大的那一个被返回。

2．什么是实现

一个实现导出一个接口，它定义了必要的标量和函数，以提供接口所规定的功能。一个实现解释了表示的细节和接口给出的特定行为的算法。但事实上，客户调用程序时根本不需要看见这些细节，他们通常是通过从程序库中调用接口来完成所有目标代码的实现。

一个接口可能有多个实现，只要实现符合该接口，它就可以改变那些不会对客户调用程序产生影响的不同的事项，以提供良好的性能。

例如，程序设计良好的接口可以避免对机器的依赖性，但是也可能使得实现必须依赖机器，因此，对不同的接口可能需要不同的或部分不同的实现。

Qt 中的功能函数完全采取封装的形式放到库文件当中，当程序需要动态地调用或者直接加入到程序去运行时，对这些接口的掌握就变得很重要了。它虽然可以实现功能，但是用户应当想到很简单的办法来完成这些功能。

Qt Designer 工具在属性编辑框中提供了对于属性的直接访问，从而减少了代码的编写数量。程序员可以根据客户的需要来编写程序。当然那些需要动态修改的属性，必须在程序编写过程中做出代码声明。

3．构建自己的 API

我在读高中时曾经写过 API，使用汉语拼音来完成全部的功能。这个 API 后来被数学老师拿去做了教学的案例，这一直是我引以为豪的事情。用汉语拼音开发程序是很有意思的事情，里面封装了有关数学的坐标，有关字符串的良好操作，当然这些最初的动机是为了玩儿。

真正的企业软件开发，不但要保证代码的安全，而且在发生意外的错误时，程序有自动修复的能力，那些可被用来重用的代码对于编写软件很重要。

Java 开源也看到了这一点，从构建企业的模块到 Java 的部分底层代码公开，这对于程序员来说是好消息。

程序员可以使用那些已经存在的而且经过多次实验的模块来构建自己的程序，而把这些程序发布出去，并注明模块的来源和大量的编程文档，这样程序就变得很明了，后继的程序员也就可以很容易地开发相关软件的功能模块了。

以上就是开源的运作手段，它将是软件发展的必然，这也表明了软件开发的方向。

你也可以尝试在你所使用的工具上开发 API，并且试图发表它们，它们对于整个软件的发展是有帮助的。尤其是开源，开源所做出的贡献就是把在软件的许可权上所得到的利润，转嫁到软件的后期服务上。

你可以简单地在 Java 中写一个功能的实现，或者在 C 语言中写一个接口，完成后会发现自己已经不是那个只会写代码的小孩子了。

4．理解 API 的健壮与安全

谁都会写功能代码，只要代码可以表示功能，那就可以算作是 API。但问题的关键在于怎样编写健壮的 API。

健壮的意思是程序在多次调用时不会出现错误，而安全是在调用后程序运行期间不会出现意外的问题，即使出现意外的问题也会有办法恢复它。

程序运行得越快，程序的代码就会越多，出现不可避免的代码错误的机率也会增加，因此不能避免一个错误的螺旋出现，这时候应该专注于程序代码模块的修复，它会大幅度地提升软件的质量。

下面是两个不同层次的人编写的程序。

```
    char *strcpy(char dst[], const char src[])
    {
 /*
使用数组的方法进行操作
    */
            int i;
            for(i=0;src[i]!='\0';i++)
            dst[i]=src[i];
            dst[i]='\0';
```

```
            return dst;
        }
        char *strcpy(char *dst，const char *src)
        {
    /*
    使用指针的方法进行操作
    */
            char *s=dst;
            while(*dst++=*srC++);
            return s;
        }
```

你能看得出来哪一个是出自比较成熟的手笔？从上面两个小例子中可以看出，在 C 语言中最重要的资源是指针，使用指针可以编写出很出色的程序，而且运算的速度和占用的内存空间都比使用数组小得多。

1.1.3 怎样学习 Qt

1．从 C++到 Qt 的跨越

C++程序语言是一个成功的典范，虽然它有很多不足的地方（比如说快速开发，对程序员质量的要求很高）。但在 Qt 当中不需要计较这么多，Qt 的出色表现让 C++语言蓬荜生辉。

在 Qt 中我们不需要关心底层的类怎样工作，而着重于图形的开发、美工的修饰、功能的扩展以及接口的形成。当链接数据库的时候，C++语言会提供接口，SQL 的提供商也会开发这方面的包，以满足各种程序设计条件下的需要。

当试图链接一个服务器的时候，程序员需要对于网络协议和套接字的构造有充分的经验，这些在 Qt 中都被涉及，并且很好地提供了丰富的类。

从 C 到 C++是机制的升华，而从 C++到 Qt，则是在内容和方便性上的变革。

如果选择 C++开发一个大型的图形程序，就需要对 XWindow 和显卡硬件有所了解，编写大量的代码，还需要掌握很多知识，如各种显卡参数。而 Qt 从上到下都为用户做了这些，用户只需要负责对它调用就能完成工作，用户只要活动鼠标，就可以构造出美观的窗口。

2．C 语言和其他程序设计的关系

C 语言是一种通用的程序设计语言，它包含了紧凑的表达式、丰富的运算符、数据类型以及用户自定义的数据结构。C 语言功能丰富，可扩展性很强。

本书的重点是讲解 Qt 的编程，但有必要涉及部分 C 语言的内容。这是因为 Linux 内核是由 C 写成的，C++也是由 C 语言继承和扩展而来，Qt 则是一组 C++的工具包。

在 C 语言中使用的一种特殊的数据结构叫结构体，例如：

```
    struct stu_info
```

```
{
        unsigned long number ;
        char *name;
        char *sex;
};
```

C++对 C 的扩展：

```
struct stu_info
{
     unsigned long number;
     char *name;
     char *sex;
   public :
        void set_name();
        void get_number();
 };
```

定义类的模式：

```
 class  stu_info
{
        unsigned long number;
        char *name;
        char *sex;
        public :
           void set_name();
           void get_number();
}
```

类与结构体十分相似，上面的例子介绍了从结构体到类的演变过程。在面向对象设计中，可以将结构体看做是类的一种特殊变体。本书的主要内容是 Qt 类的操作，这就要求读者要有一定的 C++的基础。

Qt 是建立在 C++基础之上的，要学好它就必须掌握 C++。这样用户可以很顺利地去研究 Qt 的例子，分析各种类的功能，顺利地使用它们。

使用 Qt 编写程序时，主要面向的对象就是类的操作，即使大部分类的属性可以在属性编辑框中操作，但是有些信号与槽是必须要个人自定义设计的，程序往往要完成不同的操作，而 Qt 这个工具在适用更大范围的操作的时候会显得力不从心。

大量的槽和信号的编写可以帮助用户完成很多事情。

最好的学习方式还是阅读源代码，在国外有很多的软件都是使用 Qt 编写的，它们都有源码包，可以帮助用户熟悉 Qt 工具。另外本书也是由大量的实例构成的，也是为了让大家能够在学习中侧重于应用，分析并了解源代码的作用。

1.2 Qt 对象类模型

标准的 C++对象模型为对象范例提供了有效的运行期支持，但是这种 C++对象模型的静态性质在一定的问题领域不够灵活。

图形用户界面编程运行需要高效和灵活性，由于 Qt 是基于 C++的架构，提高了程序执行的速度，所编写的大部分类能有很好的灵活性。

Qt 程序开发框架如图 1.2 所示。

Qt 把下面这些特性添加到了 C++当中。

（1）一种关于无缝对象通信被称为信号/槽的非常强大的机制，可以直接设计属性。

（2）强大的事件和事件过滤器。

（3）根据上下文进行国际化的字符串翻译。

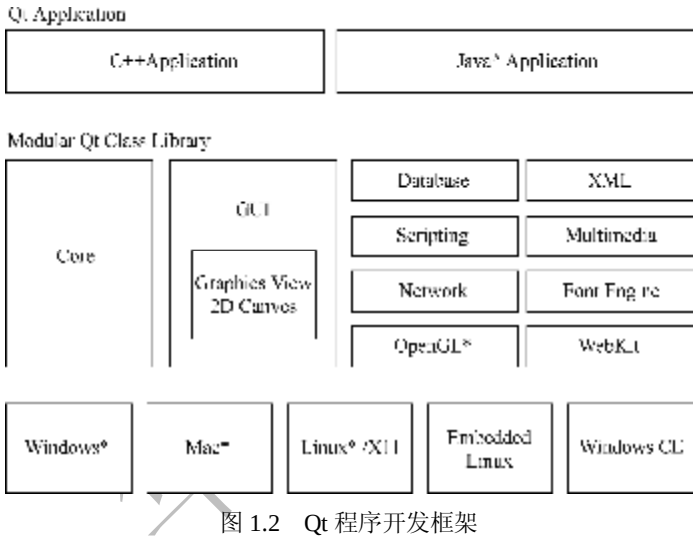


图 1.2 Qt 程序开发框架

（4）完善的时间间隔驱动的计时器，使得在一个事件驱动的图形界面程序中很好地集成许多任务成为可能。

（5）以一种自然的方式组织对象所有权的分层次和可查询的对象树。

当参考对象被破坏时，被守护的指针 QGuardedPtr 可以自动地设置为无效。而 C++指针在它的对象被破坏时变成了“摇摆指针”，这就是 Qt 区别于 C++的地方。许多 Qt 的特性是基于 QObject 的继承，通过标准 C++技术实现的。

其他的，例如，对象通信机制和虚拟属性系统，都需要 Qt 自己的元对象编译器提供的元对象系统。元对象系统是一个可使语言更加适用于真正的组件图形用户界面程序的 C++扩展，尽管模板也可以用来扩展 C++。

1.2.1 信号与槽

在图形用户界面编程中，我们经常希望一个窗口部件的变化能传递给另一个窗口部件。一般地，我们希望任何一类的对象都可以和其他对象进行通信。

例如，如果正在解析一个 XML 文件，当遇到一个新的标签时，也许希望通

知列表视图所要表达的 XML 文件结构。较旧的工具包使用一种被称作回调的通信方式来实现这一目的。

1．回调

回调是指一个函数的指针，所以如果希望一个处理函数通知你一些事件，可以把另一个函数（回调）的指针传递给处理函数。处理函数在适当的时候调用回调。回调有两个主要缺点。

（1）它们不是安全的类型，用户从来都不能确定处理函数是否使用了正确的参数来调用回调。

（2）回调和处理函数是非常强有力地联系在一起的，因为处理函数必须明确要调用哪个回调，这就需要有正确的选择。

图 1.3 所示是信号与槽的结构图。

2．信号与槽

信号与槽用于对象间的通信。信号与槽机制是 Qt 的一个中心特征，也是 Qt 与其他工具包的最不相同的部分。

在 Qt 中信号与槽是一种可以替代回调的技术。当一个特定事件发生的时候，一个信号就被发射。Qt 的窗口部件有很多预定义的信号，但是用户总是可以通过继承来加入自己的信号。槽就是一个可以被调用来处理特定信号的函数。

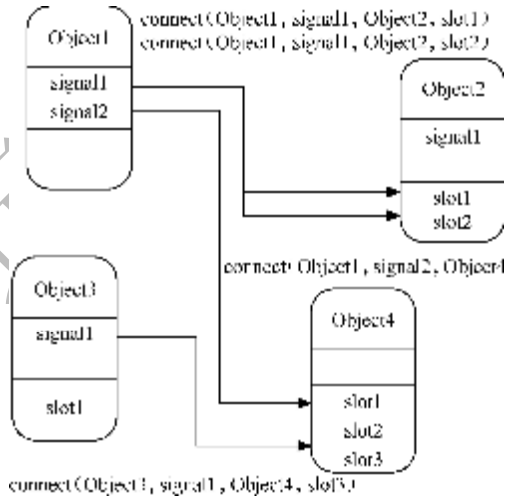


图 1.3　信号与槽的结构图

Qt 的窗口部件有很多可以预定义的槽，用户可以加入自定义的槽来处理感兴趣的信号。

信号与槽有以下特点。

（1）信号与槽的机制是安全的类型。

一个信号的签名必须与它的接收槽的签名相匹配（实际上一个槽的签名可以比它接收的信号的签名少，因为它可以忽略额外的签名）。因为签名是一致的，编译器就可以检测类型是否匹配。

（2）信号与槽的联系很宽松。

一个发射信号的类不必明确哪个槽要接收该信号。Qt 的信号与槽的机制可以保证如果用户把一个信号和一个槽连接起来，槽会在正确的时间使用信号的参数而被调用。

信号与槽可以使用任何数量、任何类型的参数。它们是完全安全的类型，不会再有回调核心转储（Coredump）。

从 QObject 类或者它的一个子类（例如，QWidget 类）继承的所有类可以包含信号与槽。

当对象改变它们的状态时，信号被发送，从某种意义上讲，它们也许对外面的世界感兴趣。这就是所有的对象进行通信的模式。这就是真正的信息封装，并且确保对象可以用作一个软件组件。

一个信号与槽连接的例子，如图 1.4 所示。

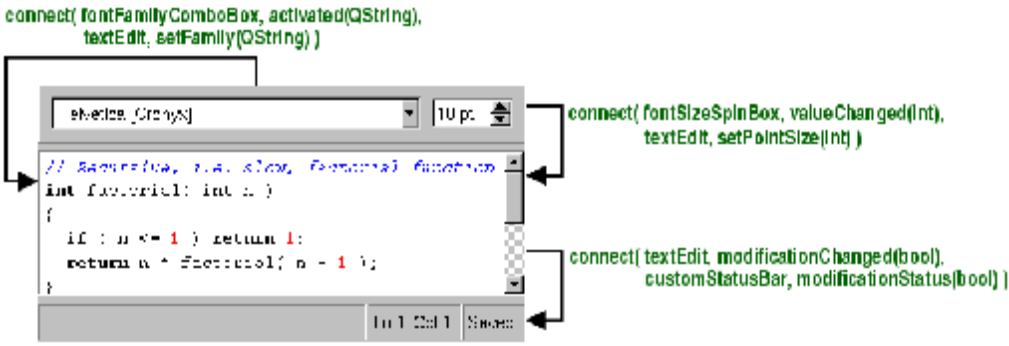


图 1.4　一个信号与槽连接的例子

信号与槽的封装机制使得槽可以用来接收正常的成员函数的信号，而不必明确是否被任意信号所连接。此外，对象对这种通信机制和能够被用作一个真正的软件组件是不知情的。

信号与槽的连接方式有 3 种：多信号对单一槽、单信号对多槽和信号对信号之间（这时，只要第一个信号被发射时，第二个信号立刻就被发射）。总体来看，信号与槽构成了一个强有力的组件编程机制。

一个小的 Qt 类如下：

```
class Foo : public QObject
    {
        Q_OBJECT
    public:
        Foo();
        int value() const { return val; }
    public slots:
        void setValue(int );
    signals:
        void valueChanged(int );
    private:
        int val;
    };
```

这个类有同样的内部状态，并且用共有方法来访问状态信息，另外它也支持使用信号与槽的组件编程。这个类通过发射一个信号 valueChanged()，来通知外界它的状态发生了变化，其对象可以发送信号给这个槽。

所有包含信号与槽的类必须在它们的程序中声明 Q_OBJECT。槽可以由应用程序的编写者来实现。这里是 Foo::setValue()的一个可能的代码实现。

```
void Foo::setValue(int v )
    {
       if (v != val ) {
          val = v;
          emit valueChanged(v);
       }
    }
```

emit valueChanged(v)这一行从对象中发射 valueChanged 信号，通过使用 emitsignal(ar guments)来发射信号。

下面是把两个对象连接在一起的一种方法。

```
Foo a, b;
connect(&a,SIGNAL(valueChanged(int)),&b,SLOT(setValue(int)));
b.setValue(11);
a.setValue(79);
b.value();
```

调用 a.setValue(79)会使 a 发射一个 valueChanged()信号，b 将会在它的 setValue()槽中接收这个信号，也就是 b.setValue(11)被调用。接下来 b 会发射同样的 valueChanged()信号，但是因没有槽被连接到 b 的 valueChanged()信号，信号消失了。

注意只有当 v!=val 的时候 setValue()函数才会设置这个值并且发射信号。这样就避免了在循环连接时（例如，b.valueChanged()和 a.setValue()连接在一起）出现无休止的循环的情况。

这个例子说明了对象之间只要最初时在中间建立信号连接就可以相互协同工作。

使用标准的 C++编译器的条件是预处理程序要改变或移除了信号、槽和发射这些关键字。在一个定义有信号/槽的类上运行元对象编译器，就可以生成能和其他对象文件编译和连接成引用程序的 C++源文件了。

1. 什么是信号

相对于 C++而言，Qt 的优越性就是使用了信号/槽机制。当对象的内部状态发生改变时，信号就被发射。只有定义了一个信号的类及其子类时，才能发射这个信号。

例如，一个列表框同时发射 highlighted()和 activated()这两个信号。绝大多数对象也许只对 activated()这个信号感兴趣，如果两个不同的对象都对这个信号感兴趣，

可以把这个信号和这两个对象连接起来。

当一个信号被发射，它所连接的槽会被立即执行，就像调用一个普通函数一样。信号/槽机制完全不依赖于任何一种图形用户界面的事件回路。当所有的槽都返回后，发射也将返回。

如果几个槽被连接到一个信号，当信号被发射时，这些槽就会按任意顺序一个接一个地执行。信号会由元对象编译器自动生成，并且一定不在.cpp 文件中实现。它们也不能有任何的返回类型（例如，使用 void）。

关于参数需要注意的是，如果信号/槽不使用特殊的类型，它们都可以多次使用。如果 QScrollBar::valueChanged() 使用了一个特殊的类型，例如，hypotheticalQRangeControl::Range，它就只能被连接到被设计成可以处理 QRangeControl 的槽。

2．什么是槽

当一个和槽连接的信号被发射的时候，这个槽被调用。槽也是普通的 C++函数，因此可以被调用；槽惟一的特点就是可以被信号连接。槽的参数和信号一样不能含有默认值。

为了槽的参数而使用自己特定的类型是很不明智的，因为槽就是普通成员函数，但它们也和普通成员函数一样有访问权限。一个槽的访问权限决定了谁可以和它相连，如：一个公共槽区包含了任何信号都可以相连的槽。

槽访问权限对于组件编程来说非常有用：用户生成了许多对象，它们互相并不知道，把它们的信号/槽连接起来，这样信息就可以正确地传递，就像一个火车模型，把它驱动，它就跑起来了。

在程序中一个受保护槽区包含了之后，这个类和它的子类的信号才能连接槽。这就是说，这些槽只是类实现的一部分，而不是它和外界的接口。还包含了这个类本身的信号可以连接的槽。这就是说它和这个类是非常紧密的，甚至它的子类都没有获得连接的权利。

也可以把槽定义为虚的，这在实践中非常有用。信号/槽的机制是非常有效的，但是它不像“真正的”回调那样快。信号/槽稍微有些慢，这是由它们的工作机制决定的，但在实际应用中这些缺点可以被忽略。

通常，发射一个和槽相连的信号，大约比直接调用那些非虚函数调用的接收器慢得多。这是定位连接对象可以安全地重复所有的连接所必需的开销（例如，在发射期间检查并发接收器是否被破坏），并且可以按一般的方式安排任何参数。

看起来调用 10 个非虚函数很多，时间开销比任何一个“新建”或者“删除”操作还要少些。当执行一个字符串、矢量或者列表操作时，需要“新建”或者“删除”，信号/槽仅仅对一个完整函数调用的时间开销中的一个非常小的部分负责。

无论何时用户在一个槽中使用一个系统调用或间接地调用超过 10 个函数的时间都是相同的。

在一台 i585-500 机器上，每秒钟可以发射 2 000 000 个左右的信号连接到一个接收器上，或者发射 1 200 000 个左右的信号连接到两个接收器上。信号/槽机制的

简单性和灵活性对于时间的开销来说是非常值得的，用户甚至察觉不出来。

1.2.2 对象与对象树

对象树都是通过 QObject 组织起来的，当一个对象创建时它会自动地被添加到父类的 children()队列中，之后通过调用 children()函数来获得这个对象链表。

QShortcut 是一个键盘的快捷键，被作为一个窗口的子部件来使用，当一个用户关闭一个窗口时，快捷键就会被自动地删除掉。

QWidget 类是指在屏幕上显示的所有部件，它是父类到子类关系的一种扩展。

一个子类就是一个子部件。举个例子，当用户删除已经运行的消息对话框的同时需要消除掉在对话框上的按钮与标签部件，可见按钮与标签被作为消息框的子部件来使用。这些部件就是这样联系的。

同时一个子部件也可以被单独地删除，即从父类部件进行自删除。例如，当使用工具的时候，QMainWindow 作为一个父类部件，QToolBar 可以将自己删除。

当一个应用程序看起来或者运行起来有些奇怪时，可用调试函数来进行调试。函数为：QObject::dumpObjectTree()与 QObject::dumpObjectInfo()。

构造与析构对象的规则如下。

当对象在堆当中建立，一个对象树就会按照它自己的方法来构造，之后采取同样的方法来删除。在删除一个对象之前需要侦测它是否有父类，如果有，它将会自动地在对象中删除，如果这个对象有子类，那么析构器也会自动删除子类。

```
int main()
{
QWidget window;
QPushButton quit("Quit",&window);
...
}
```

一个父类 window 以及作为子类的 quit 都是从 QWidget 继承来的，QWidget 是从 QObject 继承而来，在这里 quit 只能被调用一次，因为它是符合 C++标准语言，使用析构函数后自动销毁对象，销毁对象有先后顺序，先销毁子对象，后销毁父对象。

1.2.3 对象属性

Qt 的属性系统很强大，适用于编译器以及独立于平台的库，支持大部分的标准 C++编译器和系统平台，它是基于元对象系统发展起来的，能完成对象之间的访问。

在一个类中声明一个属性，它们需要符合两个条件：Q_PROPERTY()必须在声明属性前添加；另外，类必须是由 QObject 继承而来。在外界看来类的属性表现出了相似的数据成员。

1.2.4 事件和事件过滤器

在 Qt 里，一个事件是继承自 QEvent 的对象。事件通过调用 QObject::event()，被发送到继承自 QObject 的对象。事件发送即一个事件已经产生，由 QEvent 去表达，QObject 进行回应。多数事件针对 QWidget 和它的子类，此外还有一些与图形不相关的重要事件，例如，套接字激活，某种被用于 QSocketNotifier 运作的事件。

某些事件来自窗口系统，如 QMouseEvent；某些来自其他源头，如 QTimerEvent；

而某些来自应用程序，Qt 均一视同仁。因此用户可以准确地发送事件，这和 Qt 本身的事件循环所做的方式是一样的。

每个类派生自 QEvent 且添加事件特定的函数。例如，在 QResizeEvent 中，就被加入了 QResizeEvent::size()和 QResizeEvent::oldSize()。

某些类支持多种事件类型。例如：QMouseEvent 支持鼠标移动、按压、粘滞按压、拖曳、点击等。

Qt 的事件派发的灵活性有利于程序在多变且复杂的环境下起作用。QApplication::notify()的文档扼要地叙述事件起作用的整个过程，它展示在这里的内容可以满足 99%的应用。

派发事件的方法就是调用一个虚拟函数。例如，QPaintEvent 通过调用 QWidget::paint Event()而被使用。这个虚拟函数负责正确的响应，重画窗口部件。

有时，并不存在一个特定事件函数，或特定的某个函数不能充分地满足需求。最常用的事件如按下 Tab 键。正常情况下，被 QWidget 看成是去移动键盘焦点，但少数窗口部件需要自行解释。

这些对象能重新实现 QObject::event()，按常规，在通常的事件处理前后对它们的事件进行处理，或者完全重写。一个与众不同的的窗口部件，它既解释了 Tab 又包含有一个特定的自定义事件。其代码为：

```
bool MyClass:event(QEvent *e){
if(e->type()==QEvent::KeyPress){
QKeyEvent *ke=(QKeyEvent*)e;
if(ke->key()==Key_Tab){
//这里是特定的 Tab 处理
k->accept();
return TRUE;
}
}else if(e->type()>=QEvent::User){
QCustomEvent *c=(QCustomEvent*)e;
//这里是自定义事件处理
return TRUE;
}
QWidget::event(e);
}
```

更普遍的是，一个对象需要去考虑其他的事件。Qt 用 QObject::installEventFilter()支持这个目标对象（相应的有移除）。对话框通常要为某些窗口部件去过滤按键，例如，去修改回车键的处理。

一个事件过滤器在处理目标对象之前要先去处理事件。如果过滤器的 QObject::event Filter()实现被调用，它可以接受或放弃过滤，也可容许或拒绝进一步去处理事件。如果所有的事件过滤器允许进一步的处理事件，事件就被送达目标对象。

如果两者之一停止处理，目标对象和任何后面的事件过滤器就不能看到任何事件。

在 QApplication 上安装一个事件过滤器就可以过滤应用程序中的所有的事件。QToolTip 就是用这个方式过滤鼠标和键盘的全部动作。这个功能相当强大，但其在整个应用中也拖慢了单个事件的派送，因此最好避免使用这种方式。

全局事件过滤器在特定对象的过滤器调用之前被调用。

许多应用程序都要创建和发送它们自己的事件。

创建一种内置类型的事件很简单，其方法是：创建一个相应的类型的对象，然后调用 QApplication::sendEvent()或者 QApplication::postEvent()。

sendEvent()立即处理事件——当 sendEvent()返回时，事件过滤器和对象已经处理完事件了。对于很多事件类，调用 isAccepted()函数，它将告知该事件能否被当前调用的处理者所接受或者拒绝。

postEvent()投送事件于一个队列以备派发。在下次 Qt 的主事件循环运行时，它派发全部事件，并附带一些优化动作。举例，若有数个调整大小事件，它们就会被压缩成一个。

对于画图事件同样如此：QWidget::update()调用 postEvent()，最小化闪屏和避免多次重画，以加快速度。

在对象初始化期间常常使用 postEvent()函数，因为在对象完成初始化后，投送的消息会很快被派发。要创建一个自定义类型的事件，需要先定义一个事件号，其必须大于 QEvent::User，子类 QCustomEvent 可以传递有关自定义事件的特性。

1．Java 当中的事件处理过程

在 Java 程序设计中，事件的处理是非常重要的，尤其是在需要自定义事件和设计 JavaBean 时，对事件的处理过程能有一个完整的认识，对于编程来说是很有帮助的。下面用一个演示性的例子来说明事件及其处理过程。

2．事件的构成

如果想要自定义一个事件，则必须提供一个事件的监听接口以及一个事件类。在 Java 中监听接口继承自 Java.util.EventListener，事件类继承自 Java.util.EventObject。

很多基本的事件在编程环境中都已经定义，并可以很方便地使用，但在自定义事件时必须要了解这些相关的内容。

事件类可以向用户处理程序提供被监听类的信息。下面是一个事件类的代码。

```
import java.util.*;
public class PropertyEvent extends EventObject
{
   public PropertyEvent(){}
}
```

下面是监听接口的代码：

```
import java.util.*;
public interface PropertyListener extends EventListener {
   public void propertyChanged(PropertyEvent propertyEvent);
}
```

3. 事件的处理

下面是一段简要的被监听类代码。

```
import java.util.*;
public class Exam
{
  private int property;
  /*listeners 用来存放已注册的监听对象*/
  private Set listeners= new HashSet();
  .....
  public void addListener(PropertyListener propertyListener){
  /*listeners 必须保证只能被一个线程访问*/
    synchronized(listeners){
      listeners.add(propertyListener);
    }
  }
  public void firePropertyChange(){
    Iterator iterator;
    synchronized(listeners){
   /*将 listeners 中的类名放到 iterator*/
      iterator = new HashSet(listeners).iterator();
    }
   /*创建事件类*/
    PropertyEvent propertyEvent = new PropertyEvent();
    while(iterator.hasNext()){
      PropertyListener   propertyListener   =   (propertyListener)
iterator.next();
   /*调用用户的事件处理程序*/
      propertyListener.propertyChanged(propertyEvent);
    }
  }
}
```

当属性值发生变化时，首先进行内部处理，调用 firePropertyChange 产生一个事件对象，然后把事件对象作为参数来调用用户的事件处理程序。

4. 事件处理的使用

（1）下面是事件的基本用法：

```
public Exam exam;
exam.addListener(this);
```

```
public void propertyChange(PropertyEventevent){...}
```

注：exam 是被监听对象，this 为监听对象，这是已经实现了接口方法的当前类，addListener 将当前类注册到 listeners。

（2）一个被监听对象可以有多个监听对象：

```
exam.addListener(listener1);
exam.addListener(listener2);
```

这样当 exam 的 property 发生变化时，actionListener1 和 actionListener2 的处理程序都会被调用。当然 listener1 和 listener2 必须都是已实现接口方法的类。

（3）被监听的对象也可以是实现了方法的接口：

```
exam.addListener(new PropertyListener()
{
/*用户定义事件处理过程*/
public void propertyChange(PropertyEvent event)
{
...
}
```

1.2.5 元对象编译系统

Qt 的元对象系统是用来处理对象间通信的信号和槽，它运行信息类型和动态属性。Qt 的元对象系统包括以下 3 个部分的内容。

（1）QObject 类。

（2）类声明私有段中的 Q_OBJECT 宏。

（3）元对象编译器。

元对象编译器读取 C++源文件时，如果发现 C++源文件一个或多个类的声明中又含有 Q_OBJECT 宏，元对象编译器就会给含有 Q_OBJECT 宏的类生成另一个含有元对象代码的 C++源文件，这个生成的源文件可以被该类的源文件所包含，或与这个类一起编译和链接。

QObject 中的元对象代码除了提供对象间通信的信号和槽以外，还可实现其他特征。

（1）className()函数在运行时以字符串返回类的名称，不需要 C++编译器中的本地运行类型信息（RTTI）的支持。

（2）inherits()函数返回的对象是一个继承于 QObject 继承树中一个特定类的实例。

（3）tr()和 trUtf8()两个函数是用于国际化中的字符串翻译。

（4）setProperty()和 property()两个函数是用来通过名称动态设置而获得对象属性。metaObject()函数返回这个类所关联的元对象。

使用 QObject 作为一个基类而不使用 Q_OBJECT 宏和元对象代码是可以的，但是如果 Q_OBJECT 宏没有被使用，那么这个类声明的信号/槽以及其他特征描述都不会被调用。

根据元对象系统的观点，一个没有元代码的 QObject 的子类与它含有元对象代

码的最近的父类相同。举例来说，就是 className()将不会返回类的实际名称，它返回的是的它的父类的名称。

强烈建议 QObject 的所有子类使用 Q_OBJECT 宏，而不管它们是否实际使用了信号、槽和属性。

1．元对象编译器

元对象编译器解析一个 C++文件中的类声明，并生成初始化元对象的 C++代码。元对象包括所有信号/槽函数的名称和指针。元对象还包括一些额外的信息，例如，对象的类名称。用户也可以检查一个对象是否继承了一个特定的类，例如：

```
if(widget->inherits("QButton")){
//是的，它是一个 Push Button、Radio Button 或者其他按钮
}
```

实例：这是一个注释过的简单的例子（代码片断选自 qlcdnumber.h）。

```
#include"qframe.h"
#include"qbitarray.h"
class QLCDNumber:public QFrame
```

QLCDNumber 通过 QFrame、QWidget 和#include 的相关声明继承了含有绝大多数信号/槽知识的 QObject。

```
{
Q_OBJECT
```

Q_OBJECT 是由预处理器展开声明几个由元对象编译器来实现的成员函数，如果你得到了几行“virtualfunctionQButton::classNamenotdefined”这样的编译器错误信息，其原因可能是忘记运行元对象编译器或忘记在连接命令中包含元对象编译器的输出。

```
public:
        QLCDNumber(QWidget *parent=0, const char *name=0 );
        QLCDNumber(uint numDigits, QWidget *parent=0, const char
*name=0 );
```

它并不和元对象编译器直接相关，但是如果继承了 QWidget，当然想在构造器中获得 parent 和 name 这两个参数，而且把它们传递到父类的构造器中。一些解析器和成员函数在这里被省略掉了，元对象编译器忽略了这些成员函数。

```
signals:
void overflow();
```

当 QLCDNumber 被请求显示一个不可能的值时，它会发射一个信号。如果没有留意溢出，或者认为溢出不会发生，可以忽略 overflow()信号。也就是说，可以不把它连接到任何一个槽上。

另外当数字溢出发生，想调用两个不同的错误函数来解决这个问题时，可以把

这个信号和两个不同的槽连接起来。Qt 将会两个槽都调用。

```
public slots:
        void    display(int num );
        void    display(double num );
        void    display(const char *str );
        void    setHexMode();
        void    setDecMode();
        void    setOctMode();
        void    setBinMode();
        void    smallDecimalPoint(bool );
```

一个槽就是一个接收函数，它用来获得其他窗口部件状态变化的信息。QLCDNumber 使用这个槽（就像上面的代码一样）来设置显示的数字。因为 display()是这个类和程序其他部分的一个接口，所以这个槽是公有的。

几个例程把 QScrollBar 的 newValue 信号连接到 display()槽，所以 LCD 数字可以继续显示滚动条的值。请注意，这时的 display()已被重载了，当把一个信号和这个槽相连的时候，Qt 将会选择适当的版本。

2．QMetaClassInfo 类描述

QMetaClassInfo 提供了关于常用类的附加信息。元对象系统提供了信号/槽与对象之间通信的一种机制、运行期间的类型信息以及 Qt 属性系统。

在编写程序时，这个类不是必须的，但是它在编写一个源程序时将会有很大的用处。例如，当为一个 GUI 的用户来编写脚本的引擎时，一些经常使用到的函数是必须要使用的。

（1）className()：返回一个类的名字。

（2）superClass()：返回超级类的元对象。

（3）method()和 methodCount()。

QMetaClassInfo 类提供了类成员信息访问的方法，比如信号/槽和其他的成员函数。

（1）enumerator()和 enumeratorCount()提供了类的枚举。

（2）propertyCount()和 property()提供类的属性。

下面是一个相关的例子：

```
class MyClass
        {
            Q_OBJECT
            Q_CLASSINFO("author", "Sabrina Schweinsteiger")
            Q_CLASSINFO("url", "http://doc.moosesoft.co.uk/1.0/")
        public:
            ...
        };
```

1.3 Qt 全局函数

Qt 提供了一些底层的全局函数来优化应用程序。它们大部分被使用在指定的任务当中或者应用到指定平台下。这些函数从 QtCore 和 QtGui 中输出，但是它们并不包含在一个头文件中，如果在应用程序当中使用它们，就需要在调用之前声明。下面是一个例子：

```
#ifdef Q_WS_X11
void qt_x11_wait_for_window_manager(QWidget *widget);
#endif
int main(int argc, char *argv[])
{
QApplication app(argc, argv);
...
window.show();
#ifdef Q_WS_X11
qt_x11_wait_for_window_manager(&window);
#endif
...
return app.exec();
}
```

其他的全局函数做一些着重的声明：

```
void qt_set_library_config_file(const QString &fileName)
```

用来指定 Qt 的一个配置文件的位置。在调用这个功能之前，需要先构造 QApplication 或 QCoreApplication 对象。如果没有指定一个路径，就将自动地寻找一个最接近的路径。

```
void qt_x11_wait_for_window_manager(QWidget *widget)
```

阻塞，当一个 X11 窗口管理显示一个部件。

```
void qt_mac_secure_keyboard(bool enable)
```

提供了一个苹果机上的键盘特色的开关。

```
void qt_mac_set_dock_menu(QMenu *menu)
```

设置苹果机上应用程序的菜单的显示。

```
void qt_mac_set_menubar_merge(bool enable)
```

指定了一些标准的菜单项，比如关于 Qt。

```
void qt_mac_set_native_menubar(bool enable)
```

指定了单栏的样式特色。

1.4 Qt 的命名技巧

虽然那些类已经被完整地放在了库文件当中任由我们调用它，但随着程序功能的扩展，尤其到 Qt4.x 中添加了许多新的函数，为了便于查找这些函数，就需要对它们进行命名，并把它们存储在 Assistant 附件中，然后通过字母索引方式进行列举以便查找。

在程序设计中 C++语言比较出名的是微软公司的“匈牙利”命名规则，它用着重词的首字母的大写来表示。下面是两种命名方法的对比。

（1）普通的命名方法。

```
int i;
float b;
double c;
```

（2）匈牙利的命名规则。

```
int Ii;
float Fb;
double Dc;
```

其他的程序员看到变量的时候就知道它是什么类型的。当使用一个函数的时候，GetGurrentDirectory 直观的翻译函数功能是：“得到当前的工作目录”。每个大写的字母用来分辨每个单词。通过隐式的链接形成一个完整的意思。在 Qt 中命名的方法很简单，几乎和 Java 程序设计的命名方法是一样的。

```
bool QFont::bold()
bool QFont::setBold()
```

以上两个都是对于字体的操作。第一个是判断是否设置的字体是一个粗体的类型，而第二个则是通过设置得到一个粗体的类型。

显而易见，当只有两个单词的时候，第一单词的首字母是小写的，第二个则是大写的，因为两个单词隔开的话只需要一个大写字母。

1.5 Qt 开发工具的优点

1. 面向对象

良好的封装机制使得 Qt 的模块化程度非常高，可重用性较好，非常有利于用户的开发。Qt 提供了一种称为信号/槽的安全机制来替代回调，这使得各个元件之间的协同工作变得十分简单，另外 Qt 还提供了信号/槽的编译器支持可视化的编辑功能。

2. 丰富的 API

Qt 有多达 250 个以上的 C++类，还有基于模板的集合、串行、文件、I/O 设备、目录管理和日期/时间类，甚至还包括正则表达式的处理功能。

3．大量的开发文档

Qt 提供了丰富的开发文档，有利于方便地查找有用的信息，Qt 在许多类当中提供了相似的接口支持以便于使用 Qt 的官方文档编写程序。

4．XML 支持

XML 是可扩展标识语言（TheExtensibleMarkupLanguage）的简写。

目前推荐遵循的是 W3C 组织于 2000 年 10 月 6 日发布的 XML 1.0 版本，具体请参考（www.w3.org/TR/2000/REC-XML-20001006）。Qt 还包含了大量的关于 XML 的接口。

1.6 各种平台安装的方法 X11/Window

设计应用程序一般都在 X11/Window 这两个平台上进行。但最终还是取决于使用的环境和使用的范围，国内现在大多是用 Windows 操作系统。Linux 只是占有很小的一部分，所以大部分的程序要在 Window 操作系统上开发出来。

Qt 对不同平台（UNIX、Windows 和 Mac）的专门 API 进行了封装，如文件处理、网络（操作，协议）、进程处理、线程和数据库访问等。

在所有主要平台上，Qt 应用程序本地化运行，即类似于本地化应用程序，只需从单一源代码中汇编而成。使用 Qt 精确的平台非相关运行，编程后可在任何地方配置。确立新的平台，仅需重新汇编一个单一源代码库而已。

Qt 的编程可以无需考虑编写系统的平台，代码只要在一个平台下能够运行，那么就可以顺利地移植到其他的平台下。

Qt 已有成千上万个商业与开放源应用程序开发员，在多个操作系统与编译器上进行了战术测试，奠定了高性能与资源性应用程序的基础。

Qt 无需“虚拟器”，模拟层或大容量的运行时间环境。它如本地化的应用程序一样，直接写入底层的图形函数，因而 Qt 程序能以源代码的速度执行。

通过使用原 Trolltech 的双重授权模式，Qt 在商业支持并行之有效的框架下，呈现出开放源的所有优势：开放源优势包括一个活动的开放源开发员社团。

由于 Qt 的不间断开发，以及完整的代码透明性，已允许 Qt 开发员进行“彻底深入地查看”、自定义并扩展 Qt 来满足一些特殊的需求。商业产品的担保包括客户认可的产品支持，专门的 Qt 开发小组，以及一个第三方工具、组件与服务的成长生态体系。

1.6.1 tar 包安装方式

Linux 上的大部分软件需要用编译源代码的方式来进行安装，除了少量的程序不用安装直接解压外，还提供二进制的程序安装。

tar 包的一个好处是能够根据硬件和软件的结构体系来定制安装，做到最优化的安装。在安装 Qt 的时候需要具备 root 用户的安装权限，并选择一个合适的路径。

现在最新的包是 4.3，用户可以到以下的 Qt 的官方网站去下载最新的包。

http://trolltech.com/developer/downloads/Qt/index

```
    [root@localhost ] #cd /usr/local
    [root@localhost              ]#              tar              zxvf
Qt-x11-opensource-desktop-4.3.0rc1.tar.gz
```

解压包，然后归档管理包。归档管理就是把一些文件按照目录的层次打成一个包，在归档的时候还是保留原来的文件和目录层次。

```
    [root@localhost]./configure
```

./configure 文件找到机器的类型以方便在编译的时候获得很好的支持，并且根据后面跟上的选项来建立一个目录。如果不是根目录的话，Qt 被默认的安装在/usr/local/Trolltech/ Qt-4.3.0rc1 目录下，可以使用--prefix 来定制目录。如果对 configure 还有不知道的地方，可以查看帮助。

```
    [root@localhost]./configure
    [root@localhost]make
```

以上命令是建立有关的库文件、编译文件、帮助文档、所需要的数据及指南等。

```
    [root@localhost]make install
```

执行完命令后一个 Qt 就安装在 Linux 系统上了，源代码的编译需要很长的时间。Qt 的强大功能让用户觉得等待是有价值的，到目前为止还不能使用 Qt，和 Java 一样，当安装完程序之后还需要对它进行配置。

PATH 环境变量配置是为了使用本地的程序，例如，qmake、元对象编译器和其他的 Qt 工具，如果使用的 shell 是 bash、ksh 或者 zsh，那么使用以下方法，在.profile 中加入以下语句：

```
    PATH=/usr/local/Trolltech/Qt-4.3.0rc1/bin:$PATH
    export PATH
```

如果使用的是 csh 或 tcsh，那么需要加入以下语句：

```
    setenv PATH/usr/local/Trolltech/Qt-4.0.1/bin:$PATH
```

注销系统或者重新启动计算机输入以下的命令。在 Linux 上关联库是不必要的，而主要集中在工具的使用上。

```
    [root@localhost~]#qmake-v
    qmake version2.01a
    Using Qt version4.3.0in/usr/local/qt4/lib
    [root@localhost~]#export $PATH
    -bash:
    export:'/usr/local/qt4/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/us
r/local/
sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/ro
ot/bin': notavalididentifier
```

这里需要注意，当使用不同的 shell 的时候，需要配置不一样的变量。这样一个 Qt 就可以在 Linux 操作系统上运行了，可以做一个超级的链接到桌面，另外，可在那个菜单中找到新编译的 Qt。

有时候 Qt 可能编译不成功，需要查找是不是以前装过旧的版，它们可能会在系统当中产生冲突。如有冲突，可在环境变量当中明确地分开两个不同版本的 Qt，或者干脆将旧的版本先删除掉，然后再编译新的版本。

1.6.2 Window 平台安装

Qt 是在 Xwindow 上开发出来的，它经过编译可以在 UNIX 操作系统或者类 UNIX 操作系统（如 Linux）上很好地运行。

如果把它移植到 Window 平台之上，需要安装一个平台编译器，它的作用是构建一个 make 的环境，并将平台的差异性隐藏起来，所以在 Windows 上安装 Qt 必须首先安装 MinGW。

MinGW 程序主界面，如图 1.5 所示。

此向导是让我们在最近的镜像网站上下载所需要的包，至于选择什么样的

包，请参看接下来的向导页。

如果安装这个软件必须同意相关协议，单击“I Agree”按钮继续。

选择一个镜像的网站下载这些包，选择下载并安装选项，然后选择要安装的包。

图 1.6 是提示选择安装的包，包括 g++complier、g77 complier、Object c Complier 以及 MinGW Make。

图 1.5　MinGW 程序主界面

图 1.6　需要下载的包

G77 可以不安装。

选择一个安装的目录，目录的安装没有要求，选择一个比较大的磁盘分区就可以，在 Linux 上没有分区的概念，一般是在./configure 中选择要编译 Qt 的目录。

正在安装，这个时候要求网络已经链接。将下载的包安装到本地磁盘上。最后单击“Finish”按钮结束安装。

1.6.3　在 X11 平台上安装

本书所使用的版本是 4.3 版本，Qt 是向下兼容的软件，在笔者写书的时候 4.3 版本只是作为一个测试版本，而写完书后，Qt 官方网站已经将 4.3 版本正式发布出来。

安装向导可以帮助用户安装 Qt，用户只需要输入部分信息，例如，MinGW 安装目录以及要把 Qt 安装在哪个目录当中。按照安装向导即可安装完成。

我们安装的开源版本已经实现了很多的功能。这个工具可以作为一个平

台，企业用户如果希望发布商业软件的话，需要购买许可以得到很好的帮助。

Qt4.3 版本的界面相当优秀，支持动态的界面显示，给人感觉很人性化，与 4.2 版本当中的精美界面比起来，它显得很个性化，原 Trolltech 公司为我们在几个大模块上提供了不同的例子，我们这本书就是通过具有代表性的例子，对于所使用到的重要的类进行讲解。

Qt 这个工具运行在 Window 上可以和 Visual Studio 相媲美，它较传统的可视化的编写程序的方法有了巨大的改变，当用户链接一个信号，象征在这图形下面已经完成了代码层次的编写。

有人曾经预言人类将可以在 21 世纪脱离人的干预而让程序自动编写程序，Qt 在思想方面给出了可以借鉴的经验，在程序员看来，程序的任何一个动作都是程序代码的功劳。

Qt 有这样的一个机制，编写少量的代码，负责那些特殊的操作，而让大量的工作由图形界面来完成。

1.7 X Window 桌面系统

以前用 Linux 的时候都是使用 shell，使用 shell 操作系统，并让其有效率地工作，Linux 处理接收到的任务，并把完成任务的结果发送到屏幕上。

通常使用 shell 的人都被称为高手，不过在计算机日益普及的今天，图形模式以其平易近人的特点正被广泛地使用。在 Windows 下通常使用图形界面，而在 Linux 下为了管理的方便也创建了 X Window。

1．X Window 打造桌面环境

在介绍 KDE 和 Gnome 之前，我们有必要先来介绍 UNIX/Linux 图形环境的概念。对一个习惯 Windows 的用户来说，要正确理解 UNIX/Linux 的图形环境可能颇为困难，因为它与纯图形化 Windows 并没有多少共同点。

Linux 实际上是以 UNIX 为模板的，它继承了 UNIX 内核设计精简、高度健壮的特点，无论系统结构还是操作方式也都与 UNIX 无异。简单地说，可以将 Linux 看成是 UNIX 类系统中的一个特殊版本。

微软公司的 Windows 在早期只是一个基于 DOS 的应用程序，用户必须首先进入 DOS 后再启动 Windows 进程，而从 Windows 95 开始，微软公司将图形界面作为默认，命令行界面只有在需要的情况下才开启，后来的 Windows 98/Me 实际上也都隶属于该体系。

在 Windows 2000 之后，DOS 就被彻底清除了，Windows 成为一个完全图形化的操作系统。但 UNIX/Linux 与之不同，强大的命令行界面始终是它们的基础。

在 20 世纪八十年代中期，图形界面风潮席卷操作系统业界，麻省理工学院（MIT）也在 1984 年与当时的 DEC 公司合作，致力于在 UNIX 系统上开发一个分散式的视窗环境，这便是大名鼎鼎的“X Window System”项目。

不过，X Window（请注意不是 X Windows）并不是一个直接的图形操作环境，而是作为图形环境与 UNIX 系统内核沟通的中间桥梁，任何厂商都可以在 X Window 基础上开发出不同的 GUI 图形环境。

MIT 和 DEC 的目的只在于为 UNIX 系统设计一套简单的图形框架，以使 UNIX 工作站的屏幕上可显示更多的命令，对于 GUI 的精美程度和易用程度并不讲究，毕竟那时候能够熟练操作 UNIX 的都是一些习惯命令行的高手，根本不在乎 GUI 存在与否。

1986 年，MIT 正式发行 X Window，此后它便成为 UNIX 的标准视窗环境。紧接着，全力负责发展该项目的 X 协会成立，X Window 进入了新阶段。

与此同步，许多 UNIX 厂商也在 X Window 原型上开发适合自己的 UNIX GUI 视窗环境，其中比较著名的有 SUN 与 AT&T 联手开发的“Open Look”、IBM 主导

下的 OSF（Open Software Foundation，开放软件基金会）开发出的“Motif”。

一些爱好者成立了非营利的 XFree86 组织，致力于在 X86 系统上开发 X Window，这套免费且功能完整的 X Window 很快就进入了商用 UNIX 系统中，且被移植到多种硬件平台上，后来的 Linux 也直接从该项目中获益。

当然，这些早期的 X Window 环境都设计得很简单，许多 GUI 元素模仿于微软的 Windows，但 X Window 拥有一个小小的创新：当鼠标指针移动到某个窗口时，该窗口会被自动激活，用户无需点击便能够直接输入，简化了用户操作。这个特性在后来的 KDE 和 Gnome 中也都得到完整的继承。

由于必须以 UNIX 系统作为基础，X Window 注定只能成为 UNIX 上的一个应用，而不可能与操作系统内核高度整合，这就使得基于 X Window 的图形环境不可能有很高的运行效率，但它的优点在于拥有很强的设计灵活性和可移植性。

X Window 从逻辑上分为 3 层。

（1）最底层的 X Server（X 服务器）主要处理输入/输出信息并维护相关资源，它接受来自键盘、鼠标的操作并将它交给 X Client（X 客户端）作出反馈，而由 X Client 传来的输出信息也由它来负责输出。

（2）最外层的 X Client 则提供一个完整的 GUI 界面，负责与用户的直接交互（KDE、Gnome 都是一个 X Client）。

（3）中间层就是“X Protocol（X 通信协议）”，它负责衔接 X Server 与 X Client，它的任务是充当这两者的沟通管道。

尽管 UNIX 厂商采用相同的 X Window，但由于终端的 X Client 并不相同，这就导致不同 UNIX 产品搭配的 GUI 界面看起来非常不一样。

2．X Window 的简史

X 于 1984 年在麻省理工学院电脑科学研究所开始发展。当时 Bob Scheifler 正在发展分布式系统，同一时间 DEC 公司的 Jim Gettys 也在麻省理工学院做 Athena 计划的一部分，两个计划都需要一个相同的东西：一套在 UNIX 机器上优良的视窗系统。因此他们开始合作。

从斯坦福大学得到了一套叫做 W 的实验性视窗系统，由于是基于 W 视窗系统的基础上开始发展的，当发展到了足以和原先系统有明显区别时，它们把这个新系统叫做 X。X 被称为名称，没有任何的意义，然而人们现在经常使用的是 11 版本，所以经常也叫做 X11。

3．X 的基本部件

X 系统不像早期的视窗系统那样把一堆同类软件集中在一起，而是由 3 个相关的部分组合起来的。一个字体服务器端与客户端体现了这 3 个相关部分的关系，如图 1.7 所示。

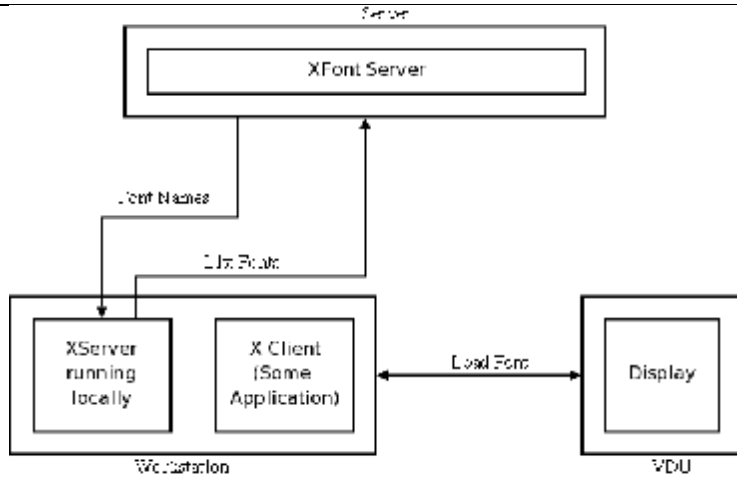


图 1.7　一个字体服务器端与客户端

（1）Server（服务器）。

控制实际显示器和输入设备的程序。服务器的责任是等待用户的任务，然后执行它并把结果返回给用户。Server 是控制显示器和输入设备（键盘和鼠标）的软件。

Server 可以建立视窗，在视窗中画图形和文字，响应 Client 程序的“需求”（requests），但它不会自己动作，只有在 Client 程序提出需求后才完成动作。

每一套显示设备只对应一个惟一的 Server，而且 Server 一般由系统的供应商提供，通常无法被用户修改。Server 只是一个普通的用户程序而已，因此很容易换个新的版本，甚至是第三方提供的原始程序。

（2）Client（客户端）。

Client 是使用系统视窗功能的一些应用程序。在 X 下的应用程序称作 Client。它作为 Server 的客户，申请 Server 端完成特定的动作。

Client 无法直接影响视窗或显示，它们只能发送一个请求（request）给 Server，由 Server 来完成它们的请求。比如在窗口中显示一个字符串或从 A 到 B 画一条直线，除了向服务器发送请求，它还具有处理一些用户传递的信息的功能，例如输入文字信息、作图、计算等。

通常，Client 程序的这一部分是和 X 独立的，它对于 X 几乎不需要知道什么。

应用程序（特别是大型的标准绘图软件、统计软件等）对许多的输出设备具有输出的能力，而在 X 视窗中的显示只是 Client 程序许多输出格式中的一种，所以，Client 程序中和 X 相关的部分在整个程序中只占非常小的份额。

用户可以通过不同的途径使用 Client 程序：通过系统提供的程序来使用；使用来自于第三方的软件；或者是用户自己为了某种特殊应用而编写的自己的 Client 程序。

（3）通信通道。

有了 Server 和 Client，它们之间就要传输一些信息，这种传输信息的媒介就是我们所要介绍的 X 的第三个组成部件：通信通道。凭借这个通道，Client 传送“需求”给 Server，而 Server 回传状态（status）及其他一些信息给 Client。

Client 使用函数库来使用通信通道。在系统或网络上支持通信形态需求的是内建于系统的基本的 X 视窗函数库（library）。只要 Client 程序利用了函数库，自然就有能力使用所有可用的通信方法。这时通道本身就变得不再重要了，而只是一个概念而已。

（4）Server 和 Client 之间的通信。

Server 和 Client 通信的方法大致有两类，都是对应于 X 系统的两种基本操作模式。

第一种，Server 和 Client 在同一台机器上执行，它们可以共同使用机器上任何可用的通信方法做交互式信息处理。

在这种模式下，X 可以同其他传统的视窗系统一样，高效工作。其中我们使用的所有有关的系统的操作比如显示窗口、下拉的菜单、鼠标的点击等，都是通过本地计算机上 Server 进行的。

第二种，Client 在一部机器上运行，而显示器和 Server 则在另一部机器上运行。因此两者的信息交换就必须通过彼此都遵守的网络协议进行，最常用的协议为 TCP/IP 协议，在此模式下我们可以使用一个常用的图形管理工具 xmanager，根据这种原理来显示出屏幕上的操作。

4．X 的用户界面

X 的设计目标之一就是能创建许多不同形式的用户界面。而 X 只提供一般的架构，让系统建立者建造所需的交互风格。这种特性使得开发者可以在 X 的基础上建造全新的界面，并且可以在任何时刻根据自己的需要选用适当的界面。

一般来说，用户界面可以分为管理界面和应用界面两部分。管理界面也就是视窗管理器，是命令的最高层，它负责在屏幕上建构或重建视窗，改变视窗的大小、位置，或者将视窗改变成图标等。

应用界面确定了用户和应用程序之间的交互风格，即用户如何利用视窗系统的设备程序来控制应用程序及输入资料。例如，如何用鼠标来选定一个选项。

5．X 的独立性

X 并不是内置于操作系统，它只是比用户层次稍高一些。在系统中它也是一个相对独立的组件。这样做有如下优点。

（1）易于安装和改版，甚至去除。这种工作不需要重启系统，也不会对其他应用程序造成干扰，网络上可以很容易地进行升级，官方网站会即时把更新的版本发放出来。

（2）第三方很容易支持并加强它的功能。例如你的制造厂商提供的系统不够好，你可以向别人买更好或更快的版本。例如 KDE 和 GNOME 就是根据 X window 为基础建立起来的两种常用的桌面系统。

（3）X 不会定制操作系统，因此成为一种标准，这也是第三方发展软件的原动力。

（4）为了发展者的利益，在 Server 上进行工作时，如果程序异常中断，只会影

响到视窗系统，不会造成机器的损坏或操作系统核心的破坏。

6．GNOME 桌面

GNOME（GNU Network Object Model Environment）是构建一个桌面系统的环境，同时也是简单创建应用程序集成到桌面系统中的一个框架。

GNOME 是一个自由软件，它是 GNU 工程的一部分。GNOME 还有许多第三方软件来支持它，在各种版本的 Linux 操作系统上它们也是第三方软件。

操作系统使用 GNOME 的版本就是这样的，另外可以随时在 GNOME 的官方网站下载最新的源码包来更新用户的 GNOME，使 GNOME 变得更稳定和美观。

如果用户使用的是中文的 GNOME 桌面，可以到中国的镜像网站（http://www.gnome- cn.org/getstart/garnomecn）上去下载最新的源码包，并且上面有安装的方法和具体的操作规程。

7．KDE 桌面

KDE 是一个用于 UNIX 操作的现代化桌面环境，它提供给 UNIX 操作系统简单而富有效率的桌面环境和不断更新的优化服务。

UNIX 操作系统应该说是最好的操作系统，但它却经历了将近 40 年的历史。

Linux 的出现已经打破了这个事实：KDE 是一个规模宏大的项目，我们很难用数字去量化它，但在很短的时间内，KDE SVN 代码仓库已经储存了超过 400 万行的代码（作为比较，Linux 内核 2.5.17 版的代码量是 370 万行左右）。Linux 优秀的原因如下。

（1）超过 800 名贡献者在协助进行 KDE 的开发。

（2）独立的翻译小组大约有 300 人。

（3）仅在 2002 年 5 月间，距今就有 11014 次 CVS 代码提交。

（4）KDE 在 12 个国家有 17 个以上的官方 WWW 镜像。

（5）KDE 在 39 个国家有 106 个以上的官方 FTP 镜像。

KDE 现在在 Linux 上非常好地运行，任何人都可以免费获得源代码并对它进行修改或优化。KDE 认识到了在一个计算平台上，平台和对于这个特定平台用户可用的一流应用程序的集合是同等重要的。

因此，KDE 项目已经开发了一流的复合文档应用程序框架，实现了最先进的框架技术，并且把它自己直接置身于和诸如微软的 MFC/COM/ActiveX 技术等流行开发框架相竞争的位置。

KDE 的 KParts 复合文档技术使得开发人员可以快速创建一流的应用程序以实现最尖端的技术。因为 KDE 应用程序开发框架的优势，已经有大量的应用程序存在于 KDE 桌面环境了，如图 1.8 所示。

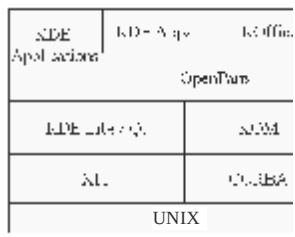


图 1.8　KDE 结构图

KDE 的发行版中包含了桌面应用程序的选择。现在 KDE 也拥有了一个基于 KDE 的 KParts 技术，由电子表格、幻灯片制作程序、组织者、新闻客户端和更多应用组成的办公应用套件。KDE 的幻灯片

制作程序 KPresenter 已被成功用于很多演示。

1.8 QApplication 类

QApplication 类管理图形用户界面应用程序的控制流和主要设置。它包含主事件循环，处理和调度来自窗口系统和其他资源的所有事件。例如，处理应用程序的初始化和结束，并提供对话管理；处理绝大多数系统范围和应用程序范围的设置。

对于任何一个使用 Qt 的图形用户界面应用程序，都存在一个 QApplication 对象，这个应用程序在同一时间内可以有多个窗口。

QApplication 对象可以通过全局变量 qApp 进行访问。QApplication 类负责的主要范围如下。

（1）它使用用户的桌面设置，例如 palette()、font()和 doubleClickInterval()来初始化应用程序。如果用户改变全局桌面（例如，用户通过控制面板改变全局桌面），它会对这些属性保持跟踪。

（2）执行事件处理，即从底层的窗口系统接收事件并把它们分派给相关的窗口部件。通过调用 sendEvent()和 postEvent()，就可以发送事件到窗口部件。

（3）分析命令行参数并根据它们设置内部状态。关于这点的详细情况请参考下面的构造函数文档。

（4）定义了由 QStyle 对象封装的应用程序的观感。在运行状态下，可以通过 setStyle()来进行设置。

（5）指定了应用程序如何分配颜色。详细情况请参考 setColorSpec()。

（6）定义了默认文本编码（请参考 setDefaultCodec()）并提供了通过 translate()用户可见的本地化字符串。

（7）提供了一些像 desktop()和 clipboard()这样的对象。

QApplication 类了解应用程序的窗口。用户可以使用 widgetAt()来询问在一个确定点上存在哪个窗口部件，以获得一个 topLevelWidgets()（顶级窗口部件）的列表和通过 closeAll Windows()来关闭所有窗口等。

如果程序员要知道应用程序的鼠标光标的处理，请参考 setOverrideCursor()和 etGlobal MouseTracking()。在 X 窗口系统上，它提供刷新和同步通信流的函数，请参考 flushX()和 syncX()。

QApplicaiton 还提供复杂的对话管理支持。这使得当用户注销时，它可以让应用程序很好地结束，如果无法终止，撤消关闭进程并且甚至为未来的对话保留整个应用程序的状态。详细情况请参考 isSessionRestored()、sessionId()、commitData()和 saveState()。

应用程序排演实例包含了一个 QApplication 通常用法的典型完整的 main()。类似于 C 语言下的初始化函数，QApplication 对象必须要做更多的初始化，它必须在所有与用户界面相关的其他类被创建之前被创建。

在现 main 函数传递参数的时候提供了可处理的命令行参数，在程序运行的前

期一些必要的参数必须传递到内部当中。

QApplication 对象所使用到的函数如表 1.1 所示。

表 1.1　　QApplication 相关函数

函 数 分 组	函　数　名
系统设置	desktopSettingsAware()、setDesktopSettingsAware()、cursorFlashTime()、setCursorFlash、Time()、doubleClickInterval()、setDoubleClickInterval()、wheelScrollLines()、setWheel、ScrollLines()、palette()、setPalette()、font()、setFont()、fontMetrics()
事件处理	exec()、processEvents()、enter_loop()、exit_loop()、exit()、quit()、sendEvent()、postEvent()、sendPostedEvents()、removePostedEvents()、hasPendingEvents()、notify()、macEvent、Filter()、qwsEventFilter()、x11EventFilter()、x11ProcessEvent()、winEventFilter()
图形用户界面风格	style()、setStyle()、polish()
颜色使用	colorSpec()、setColorSpec()、qwsSetCustomColors()
文本处理	setDefaultCodec()、installTranslator()、removeTranslator()、translate()

续表

函 数 分 组	函　数　名
窗口部件	mainWidget()、setMainWidget()、allWidgets()、topLevelWidgets()、desktop()、activePopup、Widget()、activeModalWidget()、clipboard()、focusWidget()、winFocus()、active、Window()、widgetAt()
高级光标处理	hasGlobalMouseTracking()、setGlobalMouseTracking()、overrideCursor()、setOverride、Cursor()、restoreOverrideCursor()
X 窗口系统同步	flushX()、syncX()
对话管理	isSessionRestored()、sessionId()、commitData()、saveState()
线程	lock()、unlock()、locked()、tryLock()、wakeUpGuiThread()
杂项	closeAllWindows()、startingUp()、closingDown()、type()

1.9　实例：Hello the World

第一个程序往往是很珍贵的，因为它告诉你，已经配置好了平台环境，并且各种库已经安装成功，接下来可以畅通无阻地学习程序的编写了。Hello the World 程序是每个程序设计都会有的一个小小的测验。

```
#include <QApplication>
#include <QLabel>
int main(int argc,char **argv)
{
   QApplication a(argc,argv);
   QLabel  mylabel("Hello the World");
```

```
    mylabel.show();
    return a.exec();
}
```

#include<QApplication>这一行包含了 QApplication 类的定义。在每一个使用 Qt 的应用程序中都必须有一个 QApplication 对象。QApplication 管理了各种各样的应用程序的广泛资源，例如，默认的字体和光标等。

#include<QLabel>包含了标签类 QLabel 用于显示文字或图像的无用户交互功能的服务，在外观上的标签可以设定为不同的方式。

intmain(intargc,char**argv)函数是程序的入口。在使用 Qt 的所有情况下，main()只需要在把控制转交给 Qt 库之前执行一些初始化，然后 Qt 库即可通过事件来向程序告知用户的行为。

argc 是命令行变量的数量，argv 是命令行变量的数组。这是 C/C++的特征。它不是 Qt 专有的，无论如何 Qt 需要处理这些变量（请看下面）。

```
QApplicationa(argc,argv);
```

A 就是这个程序的 QApplication。它在这里被创建，并处理这些命令行变量（例如，在 X 窗口下的-display）。请注意，所有被 Qt 识别的命令行参数都会从 argv 中被移除。

```
C:\hello>qmake -project
C:\hello>qmake
C:\hello>make
mingw32-make -f Makefile.Release
mingw32-make[1]: Entering directory 'C:/hello'
g++ -c -O2 -O2 -frtti -fexceptions -Wall -DUNICODE -DQT_LARGEFILE_SUPPORT
-DQT_D
LL   -DQT_NO_DEBUG   -DQT_GUI_LIB   -DQT_CORE_LIB   -DQT_THREAD_SUPPORT
-DQT_NEEDS_QMAIN
 -I"D:/QT/include/QtCore"                    -I"D:/QT/include/QtCore"
-I"D:/QT/include/QtGui" -I"D:
/QT/include/QtGui" -I"D:/QT/include" -I"." -I"D:/QT/include/ActiveQt"
-I"release
" -I"." -I"d:\QT\mkspecs\win32-g++" -o release\hello.o hello.cpp
hello.cpp:9:2: warning: no newline at end of file
g++   -mthreads   -Wl,-enable-stdcall-fixup   -Wl,-enable-auto-import
-Wl,-enable-runt
ime-pseudo-reloc    -Wl,-s    -Wl,-s    -Wl,-subsystem,windows    -o
"release\hello.exe" rel
ease\hello.o  -L"d:\QT\lib" -lmingw32 -lqtmain -lQtGui4 -lQtCore4
mingw32-make[1]: Leaving directory 'C:/hello'
```

经过编译生成的文件如图 1.9 所示。

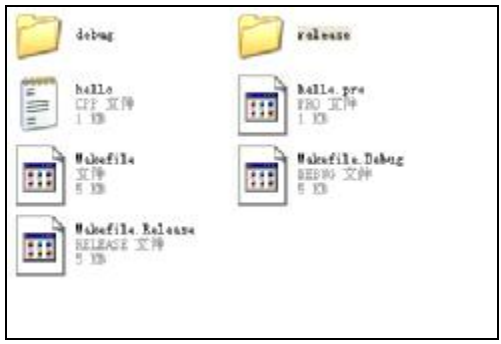


图 1.9　编译生成的文件

经过编译后，生成以上的文件和文件夹。要编译一个 C++应用程序，需要创建一个 makefile。创建一个 Qt 的 makefile 的最容易的方法是使用 Qt 提供的工具 qmake。

qmake-project 形成一个项目文件。第二个命令根据系统平台来生成一个和系统相关的 makefile 文件。

```
C:\hello>cd release
C:\hello\release>hello.exe
```

以上的命令是在 Windows 上运行 Qt 程序的方法，在 Linux 操作系统当中通常使用“./”并在其后跟上程序的名称即可。

下面编译如图 1.10 所示的第一个程序“Hello the World”。

```
#include <QApplication>
 #include <QPushButton>
 int main(int argc, char *argv[])
 {
     QApplication app(argc, argv);
     QPushButton hello("Hello world!");
     hello.resize(100, 30);
     /*
     重新设置了组件的尺寸，宽为 100 像素，高为 30 像素
     */
     hello.show();
     return app.exec();
 }
```

这个程序是 Qt 给提供的，一个程序经过编译会在按钮上显示出“Hello the Word”字符串，大家可以编写出来，然后按照上面介绍的方式进行重新编译。程序在 Linux 与 Windows 下的编译方法是一样的，而从 Windows 上移植到 X11 上基本上不用修改代码。

如果你编译的程序出现了如图 1.11 所示的图标，恭喜你，你设计成功并完整地编写出了第一个 Qt 程序。

图 1.10　编译的第一个程序“Hello the World”

图 1.11　已成功编译的第一个程序“Hello World”

1.10　窗口的基础类 QWidget

QWidget 类从 QObjet 类和 QPaintDevice 类继承，QObject 类管理对象树、信号、槽机制以及元对象。

QPaintDevice 类是绘画图片浏览对象的基类，它是绘画设备的抽象，能使用 QPainter 并在其中绘制二维空间图象，可被 QWidget、QPixmap、QPicture 和 QPrinter 类继承。

一个绘制设备的坐标系统的原点 O 坐标在屏幕上左上角的位置，X 轴是向右增长，Y 轴是向下增长，增长的单位是像素。

QPaintDevice 类使用 PDevCmd 枚举类型，列举了常用的绘制命令，其成员函数可以设置或得到绘制设备的一些通用的属性，如颜色映射和可视性等。

对于 Qt/X11 来说，调用 Xlib 库函数；对于 Qt/Embbedded 来说，使用 Qt/Embedded 服务器接口函数*qwsDisplay()等，全局的位图复制函数 bitBlt 提供了从源绘画设备对象向目的绘制设备对象进行像素复制。

函数分组如表 1.2 所示。

表 1.2　　窗口相关函数

上　下　文	函　　数
窗口函数	show()、hide()、raise()、lower()、close()
顶级窗口	caption()、setCaption()、icon()、setIcon()、iconText()、setIconText()、isActiveWindow()、setActiveWindow()、showMinimized()、showMaximized()、showFullScreen()、showNormal()

续表

上　下　文	函　　数
窗口内容	update()、repaint()、erase()、scroll()、updateMask()
几何形状	pos()、size()、rect()、x()、y()、width()、height()、sizePolicy()、setSizePolicy()、sizeHint()、updateGeometry()、layout()、move()、resize()、setGeometry()、frameGeometry()、geometry()、childrenRect()、adjustSize()、mapFromGlobal()、mapFromParent()、mapToGlobal()、mapToParent()、maximumSize()、minimumSize()、sizeIncrement()、setMaximumSize()、setMinimumSize()、setSizeIncrement()、setBaseSize()、setFixedSize()
模式	isVisible()、isVisibleTo()、visibleRect()、isMinimized()、isDesktop()、isEnabled()、isEnabledTo()、isModal()、isPopup()、isTopLevel()、setEnabled()、hasMouseTracking()、setMouseTracking()、isUpdatesEnabled()、setUpdatesEnabled()

观感	style()、setStyle()、cursor()、setCursor()、font()、setFont()、palette()、setPalette()、backgroundMode()、setBackgroundMode()、colorGroup()、fontMetrics()、fontInfo()
键盘焦点函数	isFocusEnabled()、setFocusPolicy()、focusPolicy()、hasFocus()、setFocus()、clearFocus()、setTabOrder()、setFocusProxy()
鼠标和键盘捕获	grabMouse()、releaseMouse()、grabKeyboard()、releaseKeyboard()、mouseGrabber()、keyboardGrabber()
事件处理器	event()、mousePressEvent()、mouseReleaseEvent()、mouseDoubleClickEvent()、mouse-MoveEvent()、keyPressEvent()、keyReleaseEvent()、focusInEvent()、focusOutEvent()、wheelEvent()、enterEvent()、leaveEvent()、paintEvent()、moveEvent()、resizeEvent()、closeEvent()、dragEnterEvent()、dragMoveEvent()、dragLeaveEvent()、dropEvent()、childEvent()、showEvent()、hideEvent()、customEvent()
变化处理器	enabledChange()、fontChange()、paletteChange()、styleChange()、windowActivationChange()
系统函数	parentWidget()、topLevelWidget()、reparent()、polish()、winId()、find()、metric()
帮助文件	customWhatsThis()
内部核心函数	focusNextPrevChild()、wmapper()、clearWFlags()、getWFlags()、setWFlags()、testWFlags()

每一个窗口部件构造函数接受两个或三个标准参数。

QWidget *parent=0 是新窗口部件的父窗口部件。如果为 0（默认），新的窗口部件将是一个顶级窗口部件。如果不是，它将会使 parent 的一个子部件，并且被 parent 的部件的几何形状所规范。

Constchar *name=0 是新窗口部件的窗口部件名称，可以使用 name()来访问它。窗口部件名称很少被程序员用到，但是对于图形用户界面构造程序，比如 Qt 设计器，是相当重要的（可以在 Qt 设计器中命名一个窗口部件，并且在代码中使用这个名字来连接它）。dumpObjectTree()调试函数也使用它。

WFlags f=0（在可用的情况下）设置窗口部件标记，默认设置对于几乎所有窗口部件都是适用的，但是，一个没有窗口系统框架的顶级窗口部件，必须使用特定的标记。

用户将需要为自己的窗口部件提供内容，这里需要一些简要的运行事件，只要窗口部件需要被重绘就被调用，就会涉及 paintEvent()事件。每个要显示输出的窗口部件必须实现它，并且不在 paintEvent()之外而在屏幕上绘制，这样做是明智的。

当窗口部件被重新定义大小时 resizeEvent()被调用。mousePressEvent()在鼠标键被按下时被调用。有 6 个鼠标相关事件，但是鼠标按下和鼠标释放事件是到目前为止最重要的。当鼠标在窗口部件内或者当它使用 grabMouse()来捕获鼠标时，它接收鼠标按下事件。

mouseReleaseEvent()：当鼠标键被释放时被调用。当窗口部件已经接收相应的鼠标按下事件时，它接收鼠标释放事件。这也就是说如果用户在窗口部件内按下鼠标，然后拖着鼠标到其他某个地方，然后释放，窗口部件接收这个释放事件。这里有一个例外：如果出现在弹出菜单中，当鼠标键被按下时，这个弹出菜单立即会偷掉这个鼠标事件。

mouseDoubleClickEvent()：和它看起来也许不太一样。如果用户双击，窗口部件接

收一个鼠标按下事件（如果它们没有拿牢鼠标，也许会出现一个或两个鼠标移动事件）、一个鼠释放事件并且最终是这个事件。直到看到第二次点击是否到来之前，不能从一个双击中辨别一个点击。这是绝大多数图形用户界面图书建议双击是单击的一个扩展，而不是一个不同行为的触发的一个原因。

如果你的窗口部件仅仅包含子窗口部件，也许不需要实现任何一个事件处理器。如果你想检测在子窗口部件中的鼠标点击，请在父窗口部件的 mousePressEvent()中调用子窗口部件的 hasMouse()函数。

接收键盘的窗口部件需要重新实现一些更多的事件处理器。keyPressEvent()：只要键被按下和当键已经被按下足够长的时间可以自动重复了就被调用。

注意：如果 Tab 和 Shift+Tab 键被用在焦点变换机制中，它们仅仅被传递给窗口部件。为了强迫那些键被窗口部件处理，用户必须重新实现 QWidget::event()。

focusInEvent()：当窗口部件获得键盘焦点（假设已经调用 setFocusPolicy()）时被调用。写得好的窗口部件意味着它们能按照一种清晰且谨慎的方式来获得键盘焦点。

focusOutEvent()：当窗口部件失去键盘焦点时被调用。一些窗口部件也许需要实现一些不太普通的事件处理器：mouseMoveEvent()——只要当鼠标键被按下时鼠标移动就会被调用。

举例来说，对于拖动，这个很有用。如果调用 setMouseTracking(TRUE)，尽管没有鼠标键被按下，也会获得鼠标移动事件。

keyReleaseEvent()：只要键被释放和当这个键是自动重复的并且被按下一段时间时就被调用。在这种情况下，窗口部件接收一个键释放事件并且对于每一个重复立即有一个键按下事件。

注意：如果 Tab 和 Shift+Tab 键被用在焦点变换机制中，它们就仅仅被传递给窗口部件。为了强迫那些键被窗口部件处理，必须重新实现 QWidget::event()。

wheelEvent()：当窗口部件拥有焦点时，只要用户转动鼠标滚轮就被调用。

enterEvent()：当鼠标进入这个窗口部件屏幕空间时被调用。这不包括被这个窗口部件的子窗口部件所拥有的屏幕空间。

leaveEvent()：当鼠标离开这个窗口部件的屏幕空间时被调用。

moveEvent()：当窗口部件相对于它的父窗口部件已经被移动时被调用。

closeEvent()：当用户关闭窗口部件时（或这当 close()被调用时）被调用。

这里还有一些不太明显的事件。它们在 qevent.h 中被列出并且需要重新实现 event()来处理它们。event()的默认实现处理 Tab 和 Shift+Tab（移动键盘焦点）并且其他绝大多数事件给上面提到的一个或更多的特定处理器。

当实现一个窗口部件时，还有更多的事情要考虑。

在构造函数中，在可能收到一个事件的任何机会之前，请确认尽早地设置你的成员变量。重新实现 sizeHint()在绝大多数情况下都是很有用的，并且使用 setSizePolicy()来设置正确的大小策略，这样你的同事可以更容易地设置布局管理器。

一个大小策略可以为布局管理器提供好的默认情况，这样其他窗口部件可以很容易地包含和管理窗口部件。

sizeHint()为这个窗口部件说明一个“好的”大小。如果你的窗口部件是一个顶级窗口部件，setCaption()和 setIcon()分别设置标题栏和图标。

1.11 入门级实例：设计一个用户界面

本节我们设计了一个使用图形用户界面来开发一个地址簿应用程序。

我们使用一个图形的方法建立一个用户的界面。在下面的介绍内容需要完成一个为应用程序输入的区域，如图 1.12 所示是一个缩略图。

在这个实例中我们设置了两个 QLabel 对象：nameLabel 和 addressLabel；还定义了两个输入的区域：一个 QLineEdit（它的实例化对象是 nameLine）与一个文本编辑器 QTextEdit（其对象是 addressText）。它提供了用户输入姓名与联系地址的方法，图 1.13 列举了所使用到的部件位置。

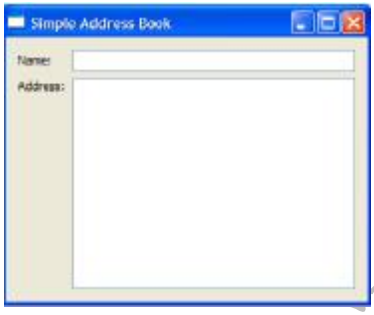


图 1.12 用户界面缩略图

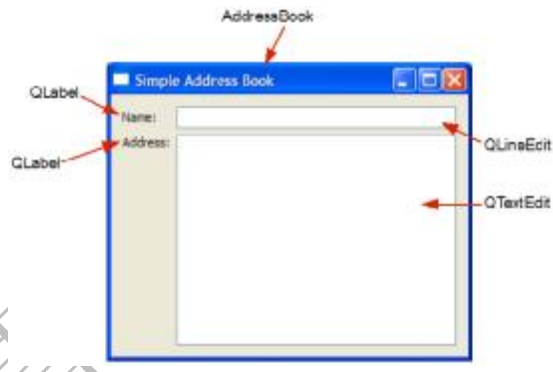


图 1.13 部件位置示意图

在这里通过 3 个文件来执行一个地址簿的程序。

（1）addressbook.h 提供了 AddressBook 类的定义。

（2）addressbook.cpp 提供了 AddressBook 类的实现文件。

（3）main.cpp 这个文件当中包含了一个 main()函数，它是一个 AddressBook 的实例。

1. Qt 编程-子类

在编写 Qt 程序的时候，我们通常使用一个子类来扩展 Qt 对象的功能，它是一个功能的扩展，用于建立自定义的部件，并包含了一些标准部件的使用方法，这些部件提供了一些部件作用的动作。

当需要建立一个用户自定义的界面时，在实现功能的同时还可以定义一些虚拟函数， 这些函数使得其他的应用程序在使用时不必再定义。

在同一个应用程序或一个库文件中，一个子类可以同时建立多个自定义的部件。并且这些代码还可以在其他的工程中做到重用。在 Qt 中并没有给我们提供一个地址簿的部件，因此我们将通过子类来建立具有地址簿特色的部件。

2．定义 AddressBook 类

在 addressbook.h 文件当中定义一个 AddressBook 类作为 QWidget 部件之后，还要声明一个构造函数，并且在类中使用 Q_OBJECT 以适用于元对象编译器与国际化的需要，同时在其中还体现了信号与槽的特色，这些特色将在下面的代码当中展示出来。

```
    class AddressBook : public QWidget
{
    Q_OBJECT
public:
    AddressBook(QWidget *parent = 0);
private:
    QLineEdit *nameLine;
    QTextEdit *addressText;
};
```

在这个类当中声明了 nameLine 与 addressText 两个类，它提供 QLineEdit 与 QTextEdit 实例化。

Q_OBJECT 宏提供了很多的 Qt 的特色，比如它有两个函数 tr()与 connect()可用。下面将在 addressbook.cpp 文件中实现 AddressBook 类。

3．实现 AddressBook 类

在构造函数当中 AddressBook 接收了 QWidget 作为参数。

```
AddressBook::AddressBook(QWidget *parent)
    : QWidget(parent)
{
    QLabel *nameLabel = new QLabel(tr("Name:"));
    nameLine = new QLineEdit;
    QLabel *addressLabel = new QLabel(tr("Address:"));
    addressText = new QTextEdit;
```

在构造函数当中声明并初始化了两个本地的 QLabel 对象：nameLabel 和 addressLabel，同时也初始化 nameLine 与 addressText。tr()提供了一个字符串的翻译功能，在以后的应用程序编写的过程中，会发现很多例子都会使用到 tr()函数来进行字符串的翻译。

当编写 Qt 程序的时候，需要使用一定的布局来协同工作，Qt 提供了 3 个主要的布局模式，分别是 QHBoxLayout、QGridLayout 与 QVBoxLayout，用于控制部件的位置。

在主程序当中使用的是网络布局，根据每个部件的位置在添加部件到布局的时候指定一个位置，下面的代码做到了与图 1.14 的网络布局样式相对应。

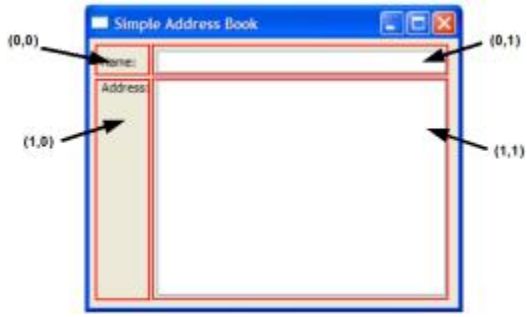


图 1.14　网络布局样式

```
    QGridLayout *mainLayout = new QGridLayout;
    mainLayout->addWidget(nameLabel, 0, 0);
    mainLayout->addWidget(nameLine, 0, 1);
    mainLayout->addWidget(addressLabel, 1, 0, Qt::AlignTop);
    mainLayout->addWidget(addressText, 1, 1);
```

在这里需要注意的是 addressLabel 的位置使用了 Qt::AlighTop 作为一个附加的参数，它设置为当前布局的上部，为了将一个布局的对象设置在一个部件上，将使用一个部件布局的设置函数 setLayout()。

```
    setLayout(mainLayout);
        setWindowTitle(tr("Simple Address Book"));
    }
```

最后设置窗口部件的主题为“Simple Address Book”。

4．运行应用程序

在 main.cpp 文件当中使用到了 main()函数，在 main()函数中初始化一个 QApplication 对象。在这个对象当中提供了应用程序扩展的资源（例如，使用默认字体与默认的光标）以及应用程序执行事件的循环，下面是实现 main()函数的方法：

```
    int main(int argc, char *argv[])
    {
        QApplication app(argc, argv);
        AddressBook *addressBook = new AddressBook;
        addressBook->show();
        return app.exec();
    }
```

在构造函数当中使用到了一个 show()函数来显示一个部件，当一个应用程序的事件循环开始的时候，这个部件才显示。

1.12 入门级程序：添加地址

接下来我们将在应用程序中添加一些功能，如图 1.15 所示为添加地址薄界面，允许用户输入。

在这里我们添加了标准按钮。

1. 定义 AddressBook 类

下面的代码中主要声明几个功能。

```
public slots:
    void addContact();
    void submitContact();
    void cancel();
```

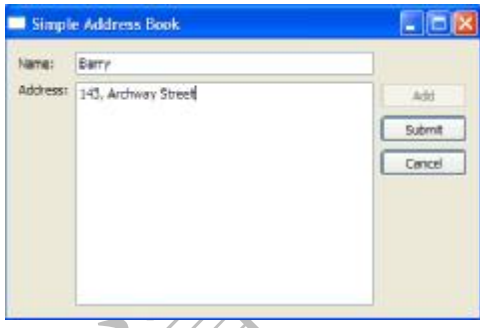


图 1.15 添加地址簿界面

一个槽是相应标准信号的功能单位，3 个 QPushButton 的对象：addButton、submitButton 与 cancelButton，它们被声明为私有的变量。

```
private:
    QPushButton *addButton;
    QPushButton *submitButton;
    QPushButton *cancelButton;
    QLineEdit *nameLine;
    QTextEdit *addressText;
```

接下来需要使用一个容器来存储一个地址簿的联系方式，下面是一个容器类的声明方式，采用了 QMap：

```
    QMap<QString, QString> contacts;
    QString oldName;
    QString oldAddress;
};
```

在这里同时还提供了两个私有的字符串对象：oldName 与 oldAddress，这两个对象是用来承载最后在地址簿改变的变量。

2．执行 AddressBook 类

在 AddressBook 类的构造函数当中，我们设置 nameLine 和 addressText 为只读的属性，因此在使用编辑器时，只能作为显示来使用。

```
    ...
    nameLine->setReadOnly(true);
    ...
    addressText->setReadOnly(true);
```

下面，我们添加了 3 个标准按钮：addButton、submitButton 与 cancelButton。

```
    addButton = new QPushButton(tr("&Add"));
    addButton->show();
    submitButton = new QPushButton(tr("&Submit"));
    submitButton->hide();
    cancelButton = new QPushButton(tr("&Cancel"));
    cancelButton->hide();
```

当按钮需要被显示出来时，调用 show()函数。submitButton 与 cancelButton 在程序运行的时候是隐藏的，因此需要调用 hide()函数。

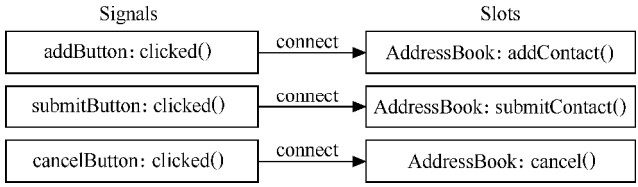
```
    connect(addButton, SIGNAL(clicked()), this,
SLOT(addContact()));
    connect(submitButton, SIGNAL(clicked()), this,
SLOT(submitContact()));
    connect(cancelButton, SIGNAL(clicked()), this, SLOT(cancel()));
```

如图 1.16 所示显示了一个信号与槽相对应的情况。

在下面的代码中，我们使用了一个纵向的布局将几个按钮添加到一个 QVBoxLayout 中：

```
    QVBoxLayout *buttonLayout1 = new QVBoxLayout;
    buttonLayout1->addWidget(addButton, Qt::AlignTop);
    buttonLayout1->addWidget(submitButton);
    buttonLayout1->addWidget(cancelButton);
    buttonLayout1->addStretch();
```

在此，最后使用了 addStretch()函数，下面的图 1.17 显示了使用与不使用这两个函数之间的不同。



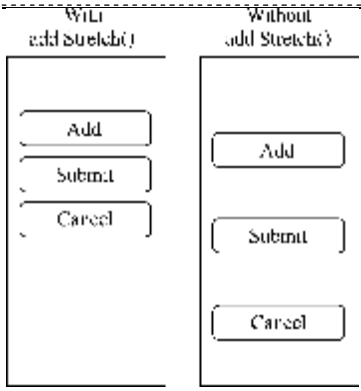


图 1.16　程序槽与信号图　　图 1.17　addStretch()函数的不同使用

之后调用函数 addLayout()将按钮的布局添加到主要布局中：

```
    QGridLayout *mainLayout = new QGridLayout;
    mainLayout->addWidget(nameLabel, 0, 0);
    mainLayout->addWidget(nameLine, 0, 1);
    mainLayout->addWidget(addressLabel, 1, 0, Qt::AlignTop);
    mainLayout->addWidget(addressText, 1, 1);
    mainLayout->addLayout(buttonLayout1, 1, 2);
```

最终布局如下面的图 1.18 所示。

在 addContact()函数中我们提供了存储最后一个联系方式的细节显示 oldName 与 oldAddress，之后清除所有的输入区域，并关闭只读模式。

```
void AddressBook::addContact()
 {
     oldName = nameLine->text();
     oldAddress = addressText->toPlainText();
     nameLine->clear();
     addressText->clear();
     nameLine->setReadOnly(false);
     nameLine->setFocus(Qt::OtherFocusReason);
     addressText->setReadOnly(false);
     addButton->setEnabled(false);
     submitButton->show();
     cancelButton->show();
 }
```

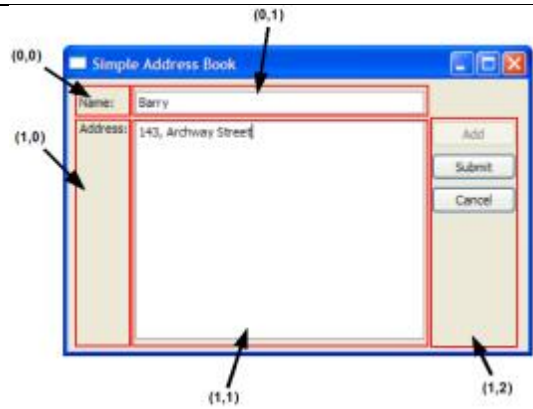


图 1.18　网络布局示意图

submitContact()函数将分成 3 个部分。

（1）首先声明一个 name 字符串变量来获得当前所输入的姓名，之后保证在提交内容之前两个输入区域不为空，最后通过使用 QMessageBox 来显示涉及的信息。

```
    void AddressBook::submitContact()
{
    QString name = nameLine->text();
    QString address = addressText->toPlainText();
    if (name == "" || address == "") {
        QMessageBox::information(this, tr("Empty Field"),
            tr("Please enter a name and address."));
        return;
    }
```

（2）检测当前输入的联系人是否存在，如果存在，就将通过 QMessageBox 来显示信息。

```
      if (!contacts.contains(name)) {
        contacts.insert(name, address);
        QMessageBox::information(this, tr("Add Successful"),
            tr("\"%1\" has been added to your address
book.").arg(name));
    } else {
        QMessageBox::information(this, tr("Add Unsuccessful"),
            tr("Sorry, \"%1\" is already in your address
book.").arg(name));
        return;
    }
```

如果联系方式已经存在，就通过使用 QMessageBox 类来显示关于联系人的信息，这里保证每个姓名都是惟一的。

（3）当用户完毕操作后，需要进行以下的处理。

```
        if (contacts.isEmpty()) {
          nameLine->clear();
          addressText->clear();
        }
        nameLine->setReadOnly(true);
        addressText->setReadOnly(true);
        addButton->setEnabled(true);
        submitButton->hide();
        cancelButton->hide();
    }
```

下面是取消一个出错操作的具体实现办法。

```
    void AddressBook::cancel()
    {
        nameLine->setText(oldName);
        nameLine->setReadOnly(true);
        addressText->setText(oldAddress);
        addressText->setReadOnly(true);
        addButton->setEnabled(true);
        submitButton->hide();
        cancelButton->hide();
    }
```

在最后我们将通过一个图形的模式来做整个程序的流程图片。

图 1.19 用来显示一个联系信息被添加的提示信息。

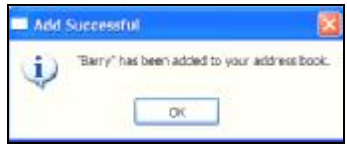


图 1.19　添加成功信息框

1.13　入门实例：地址簿浏览程序

现在这个通信录应用程序已经完成了一半，这里需要使用添加一些通信程序浏览的功能。在添加之前，我们需要声明一个可排列的数据结构来管理联系人内容。在第二个入门实例当中我们提供了一个 QMap 类，当使用字符的时候这是最好的一种选择。

使用 QMap 类保证了每个数据结构都采用统一的结构，并提供了一个可循环访问

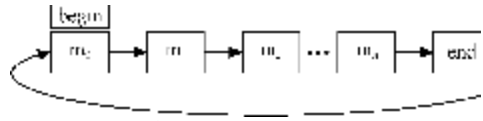


图 1.20　链表逻辑图

的列表，下面的图 1.20 是所使用的列表的示例。

1．定义 Address 类

在一个地址簿应用程序中，要实现一个浏览的功能需要添加更多的槽，next()与 previous()，把它们添加到 addressbook.h 文件中。

```
        void next();
        void previous();
```

在这里需要定义另外的两个 QPushButton 对象：nextButton 与 previousButton，它们是私有变量。

```
        QPushButton *nextButton;
        QPushButton *previousButton;
```

2．执行 AddressBook 类

在构造函数当中需要为 nextButton 与 previousButton 提供一个 disable 的树形结构，这是因为在浏览时需要使用到多个联系人的内容。

```
        nextButton = new QPushButton(tr("&Next"));
        nextButton->setEnabled(false);
        previousButton = new QPushButton(tr("&Previous"));
        previousButton->setEnabled(false);
```

通过标准按钮与相应的槽进行连接。

```
        connect(nextButton, SIGNAL(clicked()), this, SLOT(next()));
        connect(previousButton,        SIGNAL(clicked()),        this,
SLOT(previous()));
```

应用程序的缩略图如图 1.21 所示。

下面的代码定义了基本函数 next()和 previous()，以及一个从左到右显示部件的方法。

```
     QHBoxLayout *buttonLayout2 = new QHBoxLayout;
        buttonLayout2->addWidget(previousButton);
        buttonLayout2->addWidget(nextButton);
```

最后将按钮的部件添加到主部件当中。

```
    mainLayout->addLayout(buttonLayout2, 3, 1);
```

下面提供了一个包含坐标的主布局，如图 1.22 所示：

首选通过函数将两个按钮设置成为不可用。

```
        nextButton->setEnabled(false);
        previousButton->setEnabled(false);
```

之后通过 sumbitContact()函数来使得导航按钮变为可用。nextButton 与

previousButton 以当前联系人的数据个数作为依据，通过判断当前数据的大小来获知按钮是可用或不可用。其实现的代码如下：

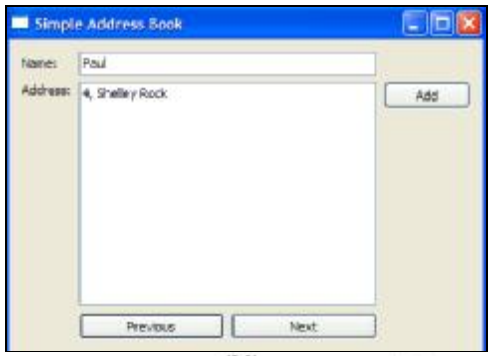


图 1.21　应用程序的缩略图

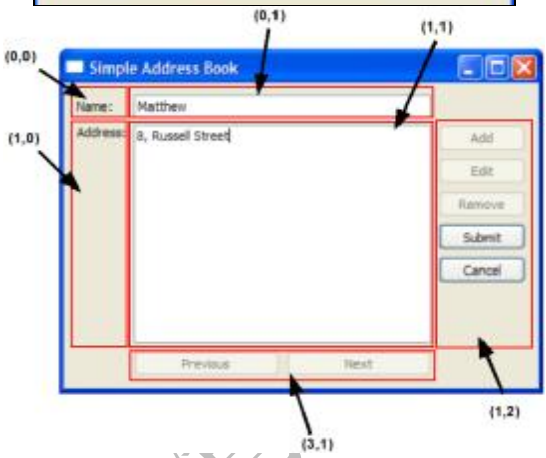


图 1.22　网络布局图

```
    int number = contacts.size();
    nextButton->setEnabled(number > 1);
    previousButton->setEnabled(number > 1);
```

在 cancel()函数当中也需要包含上面的几行代码。

```
void AddressBook::next()
 {
     QString name = nameLine->text();
     QMap<QString, QString>::iterator i = contacts.find(name);
     if (i != contacts.end())
         i++;
     if (i == contacts.end())
         i = contacts.begin();
     nameLine->setText(i.key());
     addressText->setText(i.value());
 }
```

当一个迭代器找到了相应的用户名，内容将通过 nameLine 与 addressText 来显示。同时 previous()函数也使用了一个迭代器的方法。

```
void AddressBook::previous()
{
    QString name = nameLine->text();
    QMap<QString, QString>::iterator i = contacts.find(name);
    if (i == contacts.end()){
        nameLine->clear();
        addressText->clear();
        return;
    }
    if (i == contacts.begin())
        i = contacts.end();
    i--;
    nameLine->setText(i.key());
    addressText->setText(i.value());
}
```

然后将内容全部显示在相应的位置。

1.14 入门级实例：编辑与删除地址

在这个小节中，我们将需要调整应用程序当中的地址内容。下面是应用程序的缩略图，如图 1.23 所示。

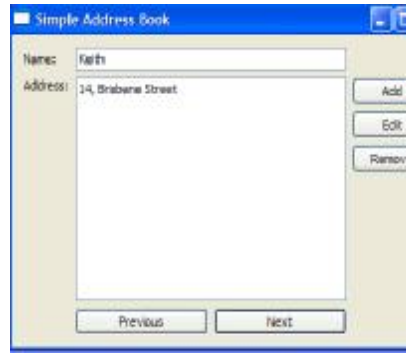


图 1.23 程序主界面

一个地址簿的应用程序不但可以实现浏览，而且还需要有方便的功能来实现编辑与删除，用来调整和改变应用程序中的一些细节，在这里只需要一些小小的改进就可以做到这一点。在这里需要使用两种模式：一种是添加模式，一种是浏览模式。

1. 定义 AddressBook 类

在 addressbook.h 中需要声明几种不同的模式。

```
enum Mode{NavigationMode,AddingMode,EditingMode};
```

同时还需要添加两个新的槽：editContact()和 removeContact()，用作两个共有的槽。

```
void editContact();
void removeContact();
```

为了可以在两种模式之间进行切换，需要使用 updateInterface()函数来控制 QPushButton 独享可用与不可用。

```
        void updateInterface(Mode mode);
        ...
        QPushButton *editButton;
        QPushButton *removeButton;
        ...
        Mode currentMode;
```

2．执行 AddressBook 类

在这里需要执行两种模式转化特色的地址簿应用程序，所以 editButton 与 removeButton 需要使用默认的方式进行初始化，它们被使用为不可用。

```
        editButton = new QPushButton(tr("&Edit"));
        editButton->setEnabled(false);
        removeButton = new QPushButton(tr("&Remove"));
        removeButton->setEnabled(false);
```

之后按钮被链接到合适的槽中，它们分别为 editContact()与 removeContact()，并将它们添加到一个布局当中。

```
         connect(editButton,          SIGNAL(clicked()),          this,
SLOT(editContact()));
        connect(removeButton,         SIGNAL(clicked()),          this,
SLOT(removeContact()));
        ...
        buttonLayout1->addWidget(editButton);
        buttonLayout1->addWidget(removeButton);
```

editContact()函数是用来存放联系人旧的一些细节保存在 oldName 与 oldAddress 中，在转换成为编辑模式之前，submitButton 与 cancelButton 都是可用的。

```
    void AddressBook::editContact()
    {
        oldName = nameLine->text();
        oldAddress = addressText->toPlainText();
        updateInterface(EditingMode);
    }
```

submitContact()函数分为两部分，使用 if-else 语句完成。

```
    void AddressBook::submitContact()
    {
        ...
        if (currentMode == AddingMode) {
```

```
            if (!contacts.contains(name)) {
                contacts.insert(name, address);
                QMessageBox::information(this, tr("Add Successful"),
                    tr("\"%1\" has been added to your address
book.").arg(name));
            } else {
                QMessageBox::information(this, tr("Add Unsuccessful"),
                    tr("Sorry, \"%1\" is already in your address
book.").arg(name));
                return;
            }
```

另外，如果当前的模式被检测为编辑模式，将使用当前的 name 与 oldName 进行比较，当 name 已经被改变，删除旧的并插入新的。

```
             } else if (currentMode == EditingMode) {
            if (oldName != name) {
                if (!contacts.contains(name)) {
                    QMessageBox::information(this, tr("Edit Successful"),
                        tr("\"%1\" has been edited in your address
book.").arg(oldName));
                    contacts.remove(oldName);
                    contacts.insert(name, address);
                } else {
                    QMessageBox::information(this,              tr("Edit
Unsuccessful"),
                        tr("Sorry, \"%1\" is already in your address
book.").arg(name));
                    return;
                }
            } else if (oldAddress != address) {
                QMessageBox::information(this, tr("Edit Successful"),
                    tr("\"%1\" has been edited in your address
book.").arg(name));
                contacts[name] = address;
            }
        }
        updateInterface(NavigationMode);
    }
```

如果只是地址改变，那么更新联系的地址，最后将当前的模式改变为浏览的模式。从一个地址簿当中删除一个联系人，通过 removeContact()执行，在删除之前需

要检测当前的联系人是否存在。

```
    void AddressBook::removeContact()
    {
        QString name = nameLine->text();
        QString address = addressText->toPlainText();
        if (contacts.contains(name)) {
            int button = QMessageBox::question(this,
                tr("Confirm Remove"),
                tr("Are you sure you want to remove \"%1\"?").arg(name),
                QMessageBox::Yes | QMessageBox::No);
            if (button == QMessageBox::Yes) {
                previous();
                contacts.remove(name);
                QMessageBox::information(this,              tr("Remove
Successful"),
                tr("\"%1\"  has  been  removed  from  your  address
book.").arg(name));
            }
        }
        updateInterface(NavigationMode);
    }
```

在这里使用了两个 QMessageBox：一个是提供给用户确认删除掉当前的联系人；另一个是提供了显示信息，确定联系人是否已经删除。

3. 更新用户界面

在 updateInteface()函数中定义了按钮的可用与不可用的功能，它依赖于不同的模式。

```
       void AddressBook::updateInterface(Mode mode)
    {
        currentMode = mode;
        switch (currentMode) {
        case AddingMode:
        case EditingMode:
            nameLine->setReadOnly(false);
            nameLine->setFocus(Qt::OtherFocusReason);
            addressText->setReadOnly(false);
            addButton->setEnabled(false);
            editButton->setEnabled(false);
            removeButton->setEnabled(false);
            nextButton->setEnabled(false);
            previousButton->setEnabled(false);
```

```
        submitButton->show();
        cancelButton->show();
        break;
```

在浏览模式下，必须保证 editButton 与 removeButton 是可用的。

```
    case NavigationMode:
        if (contacts.isEmpty()) {
            nameLine->clear();
            addressText->clear();
        }
        nameLine->setReadOnly(true);
        addressText->setReadOnly(true);
        addButton->setEnabled(true);
        int number = contacts.size();
        editButton->setEnabled(number >= 1);
        removeButton->setEnabled(number >= 1);
        nextButton->setEnabled(number > 1);
        previousButton->setEnabled(number >1 );
        submitButton->hide();
        cancelButton->hide();
        break; } }
```

1.15 入门级实例：地址簿查找功能

在本节当中不但要实现前面的功能，同时还添加了一个查找的功能，程序的缩略图如图 1.24 所示。

如果地址簿程序当中包含了很多的联系人地址，通过使用前面的实例，逐个浏览将会造成很大的不便，所以需要在这个实例当中提供一个查找的功能，使得用户在很短的时间当中找到需要的联系人，上面的缩略图中使用到了一个“Find”按钮。

当用户点击了“Find”按钮，将会弹出一个对话框，用户可通过使用对话框来输入联系人的名字进行查找。下面详细讲解这个查找对话框的制作 过程。

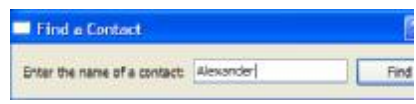


图 1.24 程序缩略图

1. 定义 FindDialog 类

为了可以构建一个对话框的子类，需要在一个头文件当中包含<QDialog>头文件，另外还需要声明 QLineEdit 和 QPushButton，这两个部件都是在对话框部件中需要使用到的。

在 AddressBook 当中还需要定义 Q_OBJECT 宏，同时可以接受父窗口，并从一个独立的窗口中打开一个对话框。

```
    #include <QDialog>
```

```
class QLineEdit;
class QPushButton;
class FindDialog : public QDialog
{
    Q_OBJECT
public:
    FindDialog(QWidget *parent = 0);
    QString getFindText();
public slots:
    void findClicked();
private:
    QPushButton *findButton;
    QLineEdit *lineEdit;
    QString findText;
};
```

在这里使用到了一个共有的函数 getFindText()，同时还包含了一个共有的槽 findClicked()。

2．执行 FindDialog 类

在 FindDialog 构造函数当中，设置了私有变量：lineEdit、findButton 与 findText，我们将使用 QHBoxLayout 来设置每个部件的位置。

```
FindDialog::FindDialog(QWidget *parent): QDialog(parent)
{
    QLabel *findLabel = new QLabel(tr("Enter the name of a
contact:"));
    lineEdit = new QLineEdit;
    findButton = new QPushButton(tr("&Find"));
    findText = "";
    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(findLabel);
    layout->addWidget(lineEdit);
    layout->addWidget(findButton);
    setLayout(layout);
    setWindowTitle(tr("Find a Contact"));
    connect(findButton,          SIGNAL(clicked()),          this,
SLOT(findClicked()));
    connect(findButton, SIGNAL(clicked()), this, SLOT(accept()));
}
```

设置了窗口的标题，提供槽与相应的信号的连接。当 findButton 按钮发送 clicked()信号时，就自动执行 findClicked()槽。下面介绍了主类与 FindDialog 对话框的关系，如图 1.25 所示。

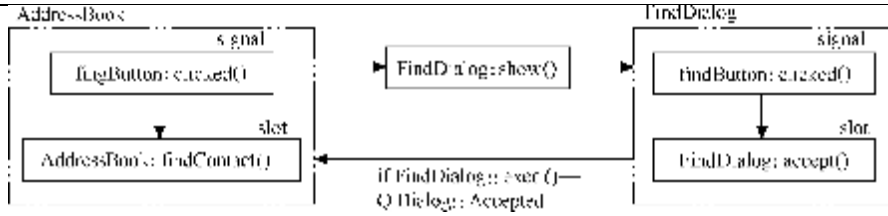


图 1.25　AddressBook 与 FindDialog 类关系

当点击了一个 findClicked()，将通过 findText()来查找字符串。

```
  void FindDialog::findClicked()
{
    QString text = lineEdit->text();
    if (text.isEmpty()) {
        QMessageBox::information(this, tr("Empty Field"),
            tr("Please enter a name."));
        return;
    } else {
        findText = text;
        lineEdit->clear();
        hide();
    }
}
```

findText 是一个全局共有的变量，被使用在 getFindText()函数中，它被直接返回，因为当前找到的 text 文本复制给 findText 变量了。

```
QString FindDialog::getFindText()
{
    return findText;
}
```

3．定义 AddressBook 类

在声明 AddressBook 类之前，需要在 addressbook.h 中包含头文件。

```
   #include "finddialog.h"
```

在这里需要定义一个私有的槽 fingContact。

```
       void findContact();
    ...
    QPushButton *findButton;
```

最后在主类当中实例化 FindDialog。

```
       FindDialog *dialog;
```

一旦初始化 dialog，它在应用程序中可多次使用，所以定义为私有的变量。

4．执行 AddressBook 类

在构造函数当中需要初始化两个私有对象：findButton 与 findDialog。

```
        findButton = new QPushButton(tr("&Find"));
        findButton->setEnabled(false);
        ...
        dialog = new FindDialog;
```

这里将 findButton 的 clicked()信号与 findContact()槽连接。

```
        connect(findButton,          SIGNAL(clicked()),          this,
SLOT(findContact()));
```

下面的代码是实现 findContact()函数。

```
    void AddressBook::findContact()
    {
        dialog->show();
        if (dialog->exec() == QDialog::Accepted) {
            QString contactName = dialog->getFindText();
            if                        (contacts.contains(contactName))
{nameLine->setText(contactName);
               addressText->setText(contacts.value(contactName));
            } else {
                QMessageBox::information(this,    tr("Contact    Not
Found"),
                     tr("Sorry,   \"%1\"   is   not   in   your   address
book.").arg(contact Name));
                return;
            }
        }
        updateInterface(NavigationMode);
    }
```

在 findDialog 中继承了 QDialog::Accepted，它是一个表示状态的语法，当查找到或者没有查找到时，将通过 QMessageBox 来报告一个信息。

1.16 入门实例：从文件中加载与保存到文件

在这个程序当中涉及对文件的特色的使用，从文件中加载与保存到文件示意图如图 1.26 所示。

虽然在上一节的程序设计中提供了浏览与查找的功能，但是当程序一旦关闭，

应用程序的数据将丢失，因此需要使用文件模式来保存当前联系人的信息，方便以后应用程序在需要时加载。

Qt 提供一些类用于输入输出的功能，在这里将通过使用 QFile 与 QDataStream 类完成具体的功能。

QFile 是一个向一个磁盘文件写入和读取的功能类，它是 QIODevice 的一个子类，说明文件是设备的一种。QDataStream 类使用串行的二进制数据，它向 QIODevice 中写入数据，并在以后使用时找回数据，打开一个设备读取并写入数据时即打开了一个设备流，此时将设备作为一个参数来使用。

图 1.26　从文件中加载与保存到文件示意

1．定义 AddressBook 类

在构造函数当中声明了两个槽 saveToFile()与 loadFromFile()，同时声明了两个对象 loadButton 与 saveButton。

```
    void saveToFile();
    void loadFromFile();
    ...
    QPushButton *loadButton;
    QPushButton *saveButton;
```

2．执行 AddressBook 类

在构造函数当中初始化两个按钮 loadButton 与 saveButton，理想的更加人性化的设计需要为每个标签进行说明，使用 setToolTip()函数来完成，说明了当前的按钮是怎样的功能。

```
    loadButton->setToolTip(tr("Load contacts from a file"));
    ...
    saveButton->setToolTip(tr("Save contacts to a file"));
```

之后将两个按钮添加到按钮的布局中，使用 clicked()信号与对应的槽进行连接。

为了能够保存一个文件，需要通过 QFileDialog::getSaveFileName()获得一个文件名称，它是使用 QFileDialog 类来提供的一个方便的功能，作为一个对话框弹出，用户需要选择以.abk 文件结尾的文件名称，或者用户自定义的建立以.abk 为结尾的文件。下面讲解 saveToFile()函数的实现方法。

```
    void AddressBook::saveToFile()
{
    QString fileName = QFileDialog::getSaveFileName(this,
        tr("Save Address Book"), "",
```

```
            tr("Address Book (*.abk);;All Files (*)"));
```

如果 fileName 不为空，将建立 QFile 对象，使用的文件名称通过 fileName 来说明，QFile 将通过 QDataStream 来读取。之后通过只读的模式打开所选定文件，如果不成功，将提供一个消息对话框显示给用户，如图 1.27 所示。

```
        if (fileName.isEmpty())
        return;
    else {
        QFile file(fileName);
        if (!file.open(QIODevice::WriteOnly)) {
            QMessageBox::information(this, tr("Unable to open file"),
                file.errorString());
            return;
        }
```

图 1.27　弹出对话框选择文件

当初始化 QDataStream 对象，从文件中读取一个数据，QDataStream 将请求一个相同的版本的流，同用于流的读取与写入。

```
        QDataStream out(&file);
        out.setVersion(QDataStream::Qt_4_3);
        out << contacts;
    }
}
```

在加载的功能中，同时也需要为 fileName 获得一个文件名称，使用 QFileDialog::getOpen FileName()。

```
void AddressBook::loadFromFile()
{
    QString fileName = QFileDialog::getOpenFileName(this,
        tr("Open Address Book"), "",
```

```
            tr("Address Book (*.abk);;All Files (*)"));
```

如果 fileName 不为空，将建立 QFile 对象，同时赋予文件为 ReadOnly 只读模式，在这里通过 saveToFile()功能来实现，如果操作不成功，将使用 QMessageBox 显示出错误信息。

```
          if (fileName.isEmpty())
             return;
        else {
             QFile file(fileName);
             if (!file.open(QIODevice::ReadOnly)) {
                 QMessageBox::information(this, tr("Unable to open file"),
                     file.errorString());
                 return;
             }
             QDataStream in(&file);
             in.setVersion(QDataStream::Qt_4_3);
             contacts.empty();
             in >> contacts;
```

这里在构造 QDataStream 对象需要保证读写的版本一致。

```
                if (contacts.isEmpty()) {
                QMessageBox::information(this,   tr("No   contacts   in
file"),
                    tr("The file you are attempting to open contains no
contacts."));
            } else {
                QMap<QString, QString>::iterator i = contacts.begin();
                nameLine->setText(i.key());
                addressText->setText(i.value());
            }
        }
        updateInterface(NavigationMode);
    }
```