第一章：Qt概述

(一)Qt简介:

1.Qt的定义：

Qt是一个跨平台的C++图形用户界面应用程序框架。它为应用程序开发者提供建立艺术级图形用户界面所需的所有功能。它是完全面向对象的，很容易扩展，并允许真正的组件编程。随着不断的扩充和发展，Qt集成了更多系统级的功能组件，现在的Qt不仅用于开发用户界面，也可以开发控制台工具和服务器程序。

2.Qt的特点：

（1）Qt是自由且开放源代码的软件，支持广泛的编译器，包括GCC的C++编译器和Visual Studio。

（2）Qt中开发应用程序既可以采用Qt Designer设计器方式，也可以采用编写代码的方式。

（3）Qt具有优良的跨平台特性。Qt在各个平台上都封装了同一套对外的编程接口API,开发者可以在一个平台上编写Qt程序, 同时可以将不经过任何修改的代码放到其它平台中编译, 可以生成对应平台的应用程序。简单来说就是一份代码到处编译。

4.Qt Creator介绍：

（1）像其他可视化软件开发工具一样，Qt4以后提供了Qt Creator集成开发环境，我们可以在集成开发环境中 设计程序界面、编辑代码、调试编译构建可执行程序等。

（2）Qt Creator是一个用于Qt开发的轻量级跨平台集成开发环境。Qt Creator可带来两大关键益处：提供首

个专为支持跨平台开发而设计的IDE，并确保首次接触Qt框架的开发人员能迅速上手和操作。即使不开发Qt应用程序，Qt Creator也是一个简单易用且功能强大的IDE。

6.Qt Creator的组成：

（1）在Qt Creator界面中，顶层是菜单栏，包括文件、编辑、建立、调试、工具集、窗口和帮助操作菜单。

（2）在Qt Creator界面中，左上角是模式选择区，包括欢迎界面模式、编辑模式、设计模式、调试模式、工程模式和帮助模式。

（3）在Qt Creator界面中，左下角是建立运行区，分别是目标系统选择、运行、调试和建立快捷选项。

（4）在Qt Creator界面中，最下层是定位于输出面板快捷选项。

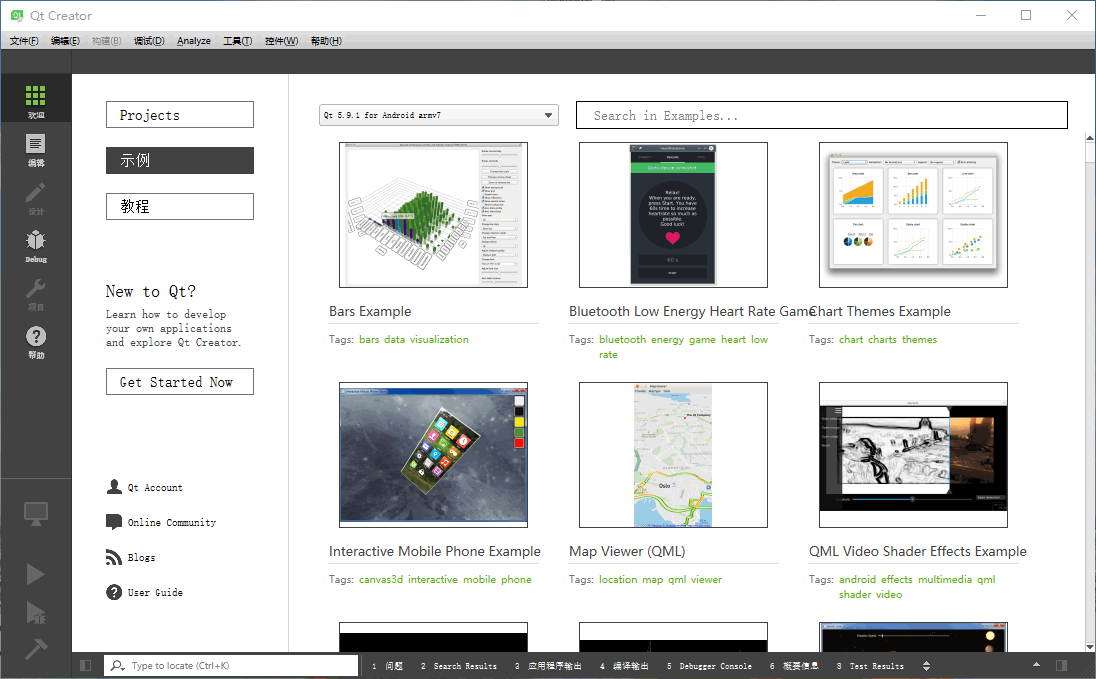


图1.1 Qt Creator主窗口

7.Qt Creator中模式选择区组成：

（1）欢迎界面模式：包括Qt的学习资料、参考例子、新建或打开Qt工程文件、最近的开发项目等，可以快速进入工作界面。即在此处可以选择自带的例子演示，在下一次打开欢迎界面时能够显示最近一次的一些项目，免除自己再去查找的麻烦。

（2）编辑模式：从其他模式或界面快速切换到编辑状态，在此处编写代码进行程序设计。

（3）设计模式：启动Qt Designer，在该模式下通过所见即所得的可视化显示，快速设计图形用户界面。在这里设计图形界面，进行部件属性设置、信号和槽设置及部件设置等操作。

（4）调试模式：在该界面下提供了调试界面。在此界面下可以调试程序，以便跟踪观察程序的运行情况。

（5）工程模式/项目模式：提供了快速设置建立（编译）、运行、调试和建立快捷选项。在此界面下可以完成开发环境的相关配置。

（6）帮助模式：快速检索和查看在线帮助系统。可在此处输入关键字，查找相关的帮助信息。

8.Qt开发环境：

在Qt程序开发过程中，除了可以通过手写代码实现软件开发功能以外，还可以通过Qt的GUI界面设计器Qt Designer进行界面的绘制和布局。该工具提供了Qt基本的可绘制窗口部件。在设计器中用鼠标直接拖拽这些窗口部件，能够高效、快速地实现GUI界面的设计，界面直观形象，所见即所得。

3. Qt按照不同的版本发行，分为商业版和开源版。Qt商业版为商业软件提供开发环境，它们提供传统商业软件发行版，并提供在协议期内有效的免费升级和技术支持服务。而Qt开源版是为了开发自由而设计的开放源码软件，它提供了和商业版本同样的功能，在GNU通用公共许可证下，它是免费的。

9.Qt/Qt SDK/Qt Creator的区别：

（1）Qt真正指的是Qt库，并不是Qt Creator，Qt Creator只是一个软件开发IDE环境，官网上面的Qt版本号也

是指的是Qt库的版本。即使开发者不使用Qt Creator集成开发环境，也可以使用Visual Studio集成开发环境来开发基于Qt的应用程序。

（2）QT SDK是一套完整的开发环境，包含了Qt库以及Qt的开发工具；Qt Library仅包含Qt库,是Qt SDk中的核心模块；Qt Creator是基于Qt构建的一个轻量级IDE。

14. Qt Designer设计界面的组成：

（1）顶层窗口部件：进入Qt设计器主界面后，看到中间主要区域就是将要设计的顶层窗口部件，顶层窗口部件是其他子窗口部件的载体。如果要将某个组件放置到待设计的顶层窗体上时，从组件面板上拖放一个组件到窗体上即可。

（2）对象查看器（Object Inspector）：窗口右上方是对象查看器（Object Inspector），用树状视图显示窗体上各组件之间的布局包含关系，视图有两列，显示了每个组件的对象名称（ObjectName）和从属的类名称（ClassName）。界面上的每个组件都需要一个唯一的对象名称，以便被引用。界面上的组件的命名应该遵循一定的法则，具体使用什么样的命名法则根据个人习惯而定，主要目的是便于区分和记忆，也要便于与普通变量相区分。

（3）组件面板：窗口左侧是界面设计组件面板，分为多个组，如Layouts、Spacers、Buttons、Container、Display Widgets等，界面设计的常见组件都可以在组件面板里找到。

（4）属性编辑器（Property Editor）：窗口右下方是属性编辑器，是界面设计时最常用的编辑器。属性编辑器显示某个选中的组件或窗体的各种属性及其取值，可以在属性编辑器里修改这些属性的值。即属性编辑器的内容分为两列，分别为属性的名称和属性的值。属性又分为多个组，实际上表示了类的继承关系。

（5）布局和界面设计工具栏：窗口上方是工具栏，工具栏上的按钮主要实现布局和界面设计。

（6）Signals和Slots编辑器与Action编辑器是位于待设计窗体下方的两个编辑器。Signals和Slots编辑器用于可视化地进行信号与槽的关联；Action编辑器用于可视化设计Action。

（7）信号和槽编辑器（Signals&Slots Editor）：列出了再Qt设计器中关联的信号和槽，通过双击列中的对象或信号/槽，可以进行对象的选择和信号/槽的

（8）动作编辑器（Action Editor）：列出了为窗口部件设计的QAction动作，通过添加或删除按钮可以新建一个可命名的QAction动作或删除指定的QAction动作。

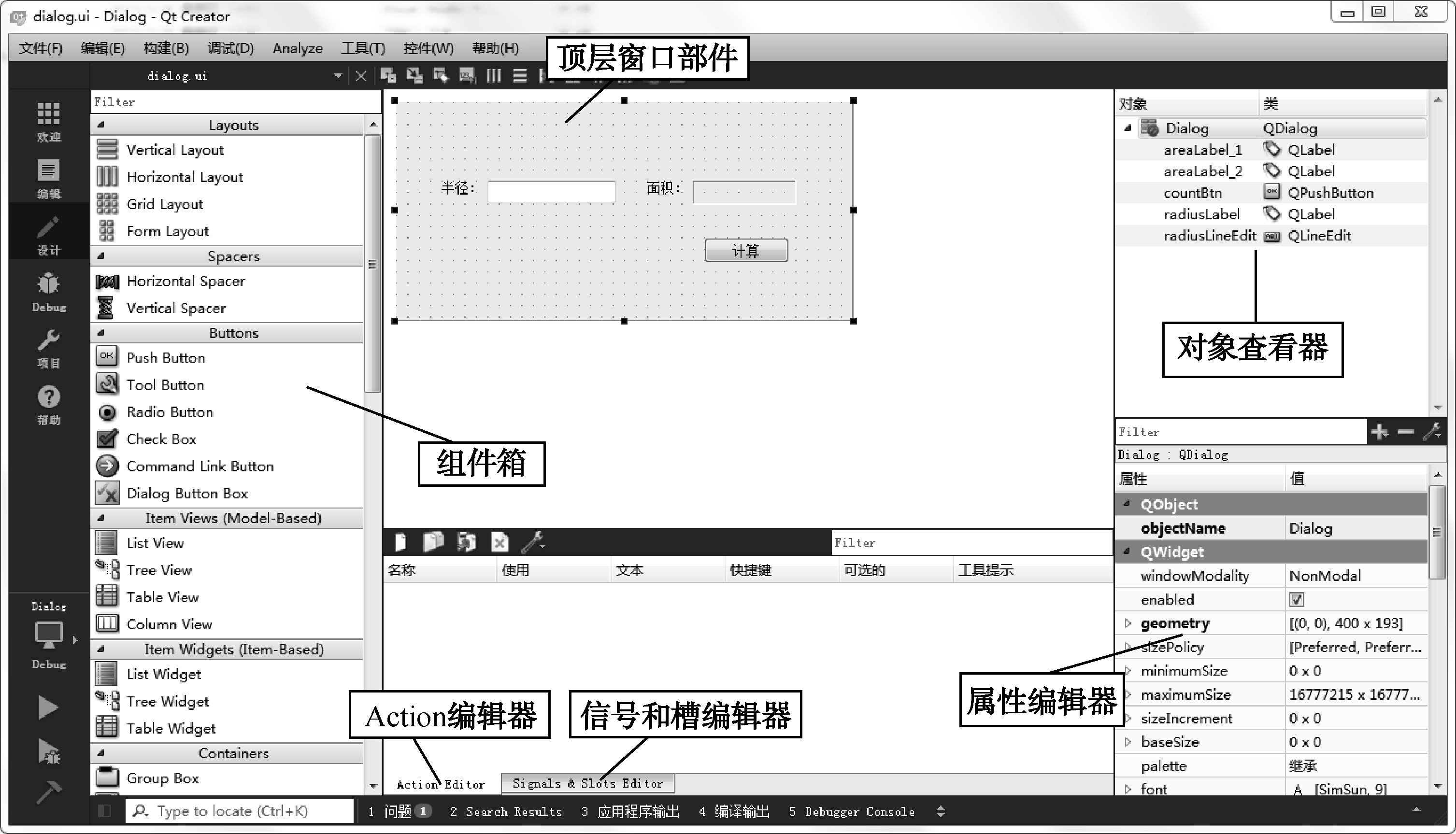


图1.2 Qt Designer界面

13.组件箱的作用：

在Qt设计器的左侧组件箱栏列出了经常使用的Qt标准窗口部件，可以直接拖拽相应的窗口部件图标到顶层窗口部件的界面上。同时也可以将设计的窗口部件组合或放置其他窗口部件的Qt容器类直接拖拽到组件箱栏中，Qt设计器会自动在组件箱栏中生成scratchpad组，并生成新的自定义的窗口部件。此后，可以像使用Qt提供的标准窗口部件一样使用新创建的窗口部件

15.Qt设计器的四种GUI窗口部件编辑模式：

（1）控件编辑模式（Edit Widgets）：可以在Qt设计器中添加GUI窗口部件并修改它们的属性和外观。

（2）信号/槽编辑模式（Edit Signals&Slots）：可以在Qt设计器中窗口部件上关联Qt已经定义好的信号和槽。

（3）伙伴编辑模式（Edit Buddlies）：可以在Qt设计器中的窗口部件上建立QLabel标签和其他窗口部件的伙伴关系。

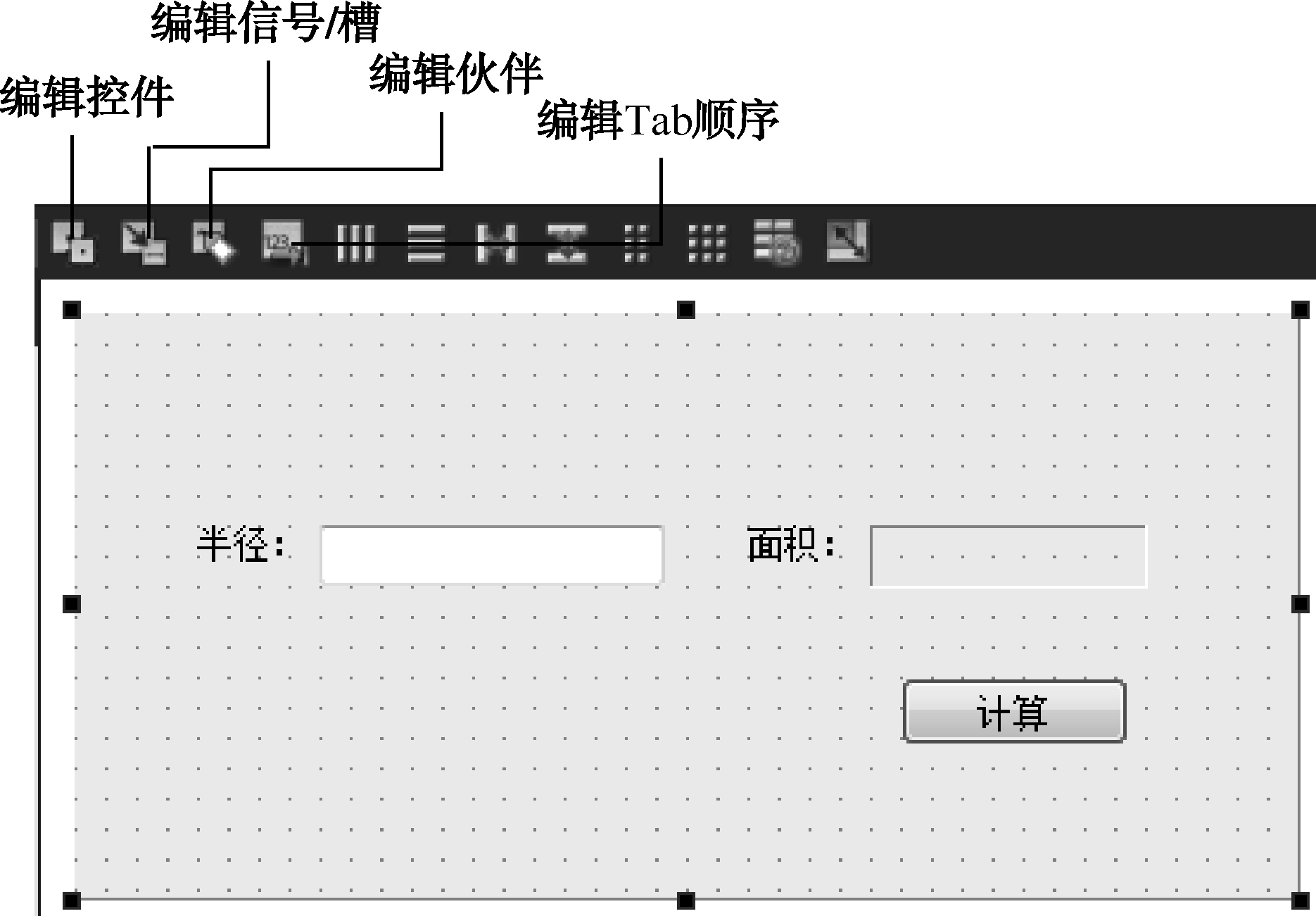
（4）Tab顺序编辑模式（Edit Tab Order）：可以在Qt设计器中的窗口部件上设置Tab键在窗口部件上的焦点顺序。

图1.3 窗口部件编辑模式

16.伙伴编辑模式（Edit Buddies）：

（1）QLabel标签和伙伴（buddy）窗口部件包括一个标签（QLabel）和一个窗口部件，它们具有伙伴关系，即指当用户激活标签的快捷键时，鼠标/键盘的焦点将会转移到它的伙伴窗口部件上。只有QLabel标签对象才可以有伙伴窗口部件，也只有该QLabel对象具有快捷键（在显示文本的某个字符前面添加一个前缀“&”，就可以定义快捷键）时，伙伴关系才有效。

（2）QLabel伙伴关系的作用：提供了使用助记符来设置鼠标/键盘焦点到对应窗口部件上的机制，这个窗口部件就叫QLabel的伙伴。助记符就是我们常说的加速键或快捷键。

（4）Qt提供了QLabel::setBuddy()函数来设置QLabel标签的伙伴窗口。此外Qt设计器还提供了伙伴关系编辑模式，可以通过鼠标拖拽操作快捷地建立标签QLabel和其他窗口部件的伙伴关系。

（3）QLabel伙伴关系的设置：

①使用英文标签时，在字符串的某个字母前面加上“&”符号，就可以指定'&'后面跟着的第一个字符为这个标签的快捷键。当用户按下“Alt+&后第一个字符”，就可以激活QLabel的快捷键将焦点跳转至QLabel的伙伴窗口上。

②对于中文标签，需要在小括号中指定加速键字母。

23.Qt提供的图形用户界面类：

（1）QMainWindow：是主窗口类，主窗口具有主菜单栏、工具栏和状态栏，类似于一般的应用程序的主窗口。

（2）QWidget：是所有具有可视界面类的基类，选择QWidget创建的界面对各种界面组件都可以支持。

（3）QDialog：是对话框类，可建立一个基于对话框的界面。

16.Qt中生成的工程文件：

（1）Qt生成的工程文件包括\*.pro工程文件、\*.ui样式文件、\*.h头文件和\*.cpp源文件。在Qt Creator集成开发环境编辑模式中可以分类看到，双击相应的文件，就会在编辑窗口中打开。

（2）\*.pro工程文件：工程文件定义了Qt用的类库、目标程序的文件名、采用模板、C++源文件、C++头文件和样式文件。

（3）\*.ui样式文件：样式文件定义了人机交互界面，在设计模式下是所见即所得的窗口和控件，在编辑模式下是XML定义文件。Qt编译器在编译样式文件时，自动将XML资源文件编译成C++头文件ui\_\*.h。该文件创建了窗口界面类Ui\_\*，封装了样式中的窗口和控件。了解该文件可以更好地了解Qt界面的实现过程。双击项目文件目录树中的文件\*.ui，会打开一个集成在Qt Creator中的Qt Designer对窗体进行可视化设计。

（4）\*.h头文件：头文件列出本工程包含的用户C++头文件。

（5）\*.cpp源文件：源文件列出本工程包含的用户C++源文件。

17.Qt中\*.pro工程文件的组成：

（1）像所有的集成开发工具一样,Qt集成开发环境需要建立工程文件，把生成软件所需的所有文件通过工程文

件统一组织管理。后缀为“.pro”的文件是项目的管理文件，文件名就是项目的名称

（2）“Qt += core gui”表示项目中加入core gui模块。core gui是Qt用于GUI设计的类库模块，如果创建的

是控制台（Console）应用程序，就不需要添加core gui。Qt类库以模块的形式组织各种功能的类，根

据项目涉及的功能需求，在项目中添加适当的类库模块支持。

（3）“TARGET = demo”表示生成的目标可执行文件的名称，即编译后生成的可执行文件是Demo.exe。

（4）SOURCES、HEADERS、FORMS记录了项目中包含的源程序文件、头文件和窗体文件（.ui文件）的名称。这

些文件列表是Qt Creator自动添加到项目管理文件里面的，用户不需要手动修改。当添加一个文件到项

目或从项目里删除一个文件时，项目管理文件里的条目会自动修改。

19.Qt的程序结构：

（1）Qt的程序结构主要包括main()主程序和主程序中实例化的一系列GUI组件。在Qt自动生成的程序文件中

主程序实例化应用程序和主窗口MainWindow，并显示主窗口及其包含的控件，然后在主窗口中完成主要

的操作和处理。主窗口中的所有的控制和操作是通过类的定义来实现的。

（2）主程序的程序结构如下：

#include "mainwindow.h"

#include <QApplication>

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

{

QApplication a(*argc*, argv); //创建QApplication应用程序实例。

MainWindow w; //创建MainWindow主窗口实例。

w.show(); //显示主窗口及其包含的控件。

return a.exec(); //进入应用程序事件循环。

}

①#include "mainwindow.h"：包含了程序中要完成功能的MainWindow类的定义，在MainWindow类中封装完

成所需要的功能。注意，使用哪个类就必须将包含该类的头文件引用过来。Qt中每个窗口都是QWidget对

象，或者其派生类的对象。

②#include <QApplication>：QApplication类的定义。在每一个使用Qt图形化应用程序中都必须使用一个

QApplication对象。QApplication管理了各种各样图形化应用程序的广泛资源、基本设置、控制流及事件

处理等。

③int main(int argc, char \*argv[])：应用程序的入口，几乎在所有使用Qt的情况下，main()函数只需要

将控制转交给Qt库之前执行初始化，然后Qt库通过事件向程序告知用户的行为。所有Qt程序都必须有且只

有一个main()函数。main()函数有两个参数，即argc和argv。argc是命令行变量的数量,argv是命令行变

量的数组。

④w.show()：当创建一个窗口部件的时候，默认它是不可见的，必须调用show()函数使它变为可见。

⑤return a.exec()：应用程序进入消息循环，等待可能的输入进行响应。这里就是main()函数将控制权转

交给Qt，Qt完成事件处理工作，当应用程序退出的时候，exec()函数的值就会返回。在exec()函数中，Qt

接收并处理用户和系统的事件并且将它们传递给适当的窗口部件。

⑥Q\_OBJECT宏的作用是启动Qt元对象系统的一些特性，如支持信号和槽操作等，它必须放置到类定

义的私有区中。

（3）主窗口类头文件mainwindow.h：

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

namespace **Ui** { //声明了一个名称为Ui的命名空间，包含一个类MainWindow。但

class **MainWindow**; //是这个类MainWindow并不是本文件里定义的类MainWindow而是

} // ui\_mainwindow.h文件里定义的类，用于描述界面组件的。这个 //声明相当于一个外部类型声明。

class **MainWindow** : public QMainWindow

{

Q\_OBJECT //使用Qt的信号与槽机制的类都必须加入的一个宏。

public:

explicit **MainWindow**(QWidget \*parent = nullptr);

~***MainWindow***();

private:

Ui::MainWindow \*ui; //这个指针是用前面声明的namespace Ui里的MainWindow类

//定义的，所以指针ui是指向可视化设计的界面，后面访问

}; //面上的组件，都需要通过这个指针ui。

#endif // MAINWINDOW\_H

（4）主窗口类源文件mainwindow.cpp：

#include "mainwindow.h"

#include "ui\_mainwindow.h"

MainWindow::**MainWindow**(QWidget \*parent) :

QMainWindow(parent), //执行父类QMainWindow的构造函数。

ui(new Ui::MainWindow) //创建一个Ui::MainWindow类的对象ui, 这个ui就是

{ //MainWindow的private部分定义的指针变量ui。

ui->setupUi(this); //执行Ui::MainWindow类的setupUi()函数，这个函数实现窗

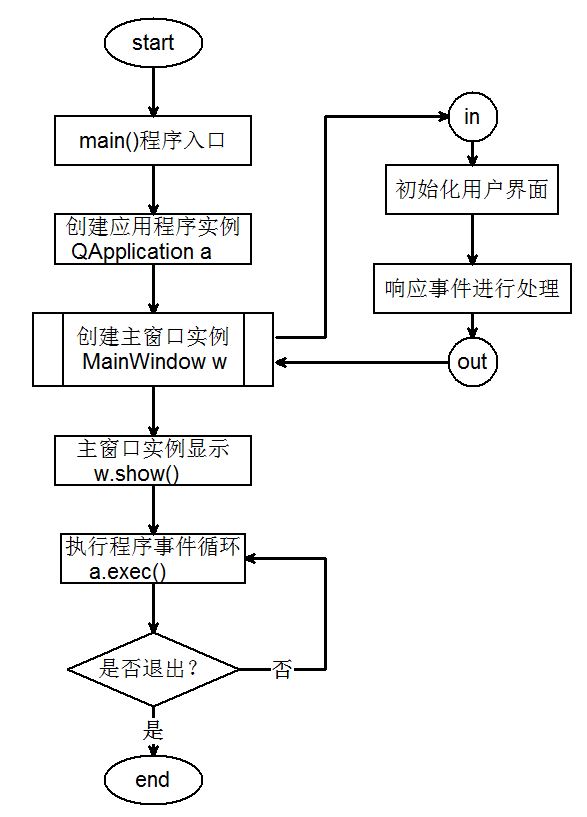
} //口的生成与各种属性的设置、信号与槽的关联。

MainWindow::~***MainWindow***()

{

delete ui; //析构函数只是简单地删除用new创建的指针ui。

}

(5)Qt程序结构流程图：

（6）在ui\_mainwindow.h文件里有一个namespace名称为Ui，里面有一个类MainWindow是用于描述可视化设计的窗体，且与mainwindow.h里定义的类同名。在MainWindow类里访问Ui::MainWindow类的成员变量或函数需要通过MainWindow类里的ui指针，如同构造函数里执行ui->setupUi(this)函数那样。

20. Qt的一些类，如QString、QList、QMap等，不需要QCoreApplication的支持也能够工作，但是在使用Qt编写应用程序时，如果是控制台应用程序，则建议初始化一个QCoreApplication对象，Qt5创建控制台项目时生成的main.cpp源文件中默认就创建了一个QcoreApplication对象；如果是GUI图形用户界面程序，则会初始化一个QApplication对象。QApplication用来管理界面应用的控制流以及主要的程序设置。在Qt中不管有多少的窗口，QApplication实例对象都只有一个。

22.Qt Creator提供的应用程序模板：

（1）Qt Widgets Application：支持桌面平台的有图形用户界面（Graphic User Interface，GUI）的应用程

序。GUI的设计完全基于C++语言，采用Qt提供的一套C++类库。

（2）Qt Console Application：控制台应用程序，无GUI界面，一般用于学习C/C++语言，只需要简单的输入

输出操作时可创建此类项目。

（3）Qt Quick Application：创建可部署的Qt Quick2应用程序。Qt Quick是 Qt支持的一套GUI开发架构，其

界面设计采用QML语言，程序架构采用C++语言。利用Qt Quick可以设计非常炫的用户界面，一般用于移

动设备或嵌入式设备上无边框的应用程序的设计。

（4）Qt Quick Controls2 Application：创建基于Qt Quick Controls2组件的可部署的Qt Quick2应用程序。

Qt Quick Controls2组件只有Qt5.7及以后版本才有。

（5）Qt Canvas 3D Application：创建Qt Canvas 3D QML项目，也是基于QML语言的界面设计，支持3D画布。

24.Qt Creator中项目的文件目录树的组成：

（1）Qt Creator中目录树显示了项目内文件的组织结构。项目的名称构成目录树的一个根节点，Qt Creator 可以打开多个项目，但是只有一个活动项目，活动项目的项目名称节点用粗体字体表示。

（2）\*.Pro文件：是项目管理文件，包括一些对项目的设置项。

（3）Headers：该节点下存放项目内的所有头文件（.h）。[C++](http://c.biancheng.net/cplus/)中任何窗体或界面组件都是用类封装的，一个 类一般有一个头文件（.h文件）和一个源程序文件（.cpp文件）。

（4）Sources：该节点下存放项目内的所有C++源文件（.cpp）。main.cpp是主函数文件，也是应用程的入口。

（5）Forms：该节点存放项目内的所有界面文件（.ui）。界面文件是文本文件，使用XML语言描述界面的组成。

25.Qt Creator中编译调试工具栏按钮的作用:

|  |  |  |
| --- | --- | --- |
| **图标** | **作用** | **快捷键** |
| http://c.biancheng.net/uploads/allimg/181228/2-1Q22Q52043426.gif | 弹出菜单选择编译工具和编译模式，如 Debug或 Release模式。 |  |
| http://c.biancheng.net/uploads/allimg/181228/2-1Q22Q52144152.gif | 直接运行程序，如果修改后未编译，会先进行编译。即使在程序中设置了断点，此方式运行的程序也无法调试。 | Ctrl+R |
| http://c.biancheng.net/uploads/allimg/181228/2-1Q22Q52211343.gif | 项目需要以Debug模式编译，点此按钮开始调试运行，可以在程序中设置断点。若是以 Release模式编译，点此按钮也无法进行调试。 | F5 |
| http://c.biancheng.net/uploads/allimg/181228/2-1Q22Q52230b6.gif | 编译当前项目。 | Ctrl+B |

（1）在项目编译设置界面中每个编译器又有Build和Run两个设置界面。在Build设置界面上，有一个“Shadow build”复选框。如果勾选此项，编译后将在项目的同级目录下建立一个编译后的文件目录，目录名称包 含编译器信息，这种方式一般用于使用不同编译器创建不同版本的可执行文件。如果不勾选此项，编译 后将在项目的目录下建立“Debug”和“Release”子目录用于存放编译后的文件。

（2）在Qt Creator中也可以对程序设置断点进行调试，但是必须以Debug模式编译，并以“Start Debugging”

快捷键F5方式运行程序。程序调试的方法与一般IDE工具类似。注意要在Qt Creator里调试MSVC2017编译

的程序，必须安装Windows软件开发工具包SDK。

26.在 [Qt](http://c.biancheng.net/qt/) Creator中可以使用MSVC编译工具对Qt项目进行编译。若有人比较习惯于使用Visual Studio或某些项目必须使用Visual Studio，也可以在Visual Studio里创建和管理Qt程序项目。要在Visual Studio中使用Qt，需要安装一个 [qt-vsaddin-msvc](http://download.qt.io/development_releases/vsaddin/qt-vsaddin-msvc2017-2.3.0.vsix)的Qt插件，这个插件程序由Qt公司提供。

27.qmake的作用:

（1）Qt Creator集成开发环境开发应用是通过qmake来组建项目，qmake与cmake类似，qmake语法更简单，cmake

功能更强大。qmake能够自动生成Makefile、Microsoft Visual Studio 专案文件。不管源代码是否是用

Qt写的，都能使用qmake，因此qmake能用于很多软件的构建过程，qmake文件的后缀一般为.pro。

（2）Qt的编译机制：先使用qmake.exe工具解析.pro文件，生成Makefile文件, 然后通过makefile-32bit.exe

执行Makefile文件来编译源文件，生成release版本的exe文件。如果使用makefile-32bit.exe工具来执

行Makefile.Debug文件编译源码文件，将生成Debug版本的可执行exe文件。

28.qmake文件的基本语法：

（1）​qmake中一行代表一条规则，#代表注释，如果想分行书写，可以使用转义字符“\”。

（2）qmake文件的常用变量:

①QT        #指定要用的哪些Qt支持库

②SOURCES #指定项目源文件。

③HEADERS #指定项目头文件。

④FORMS #指定项目中用到的ui文件（ui文件是用QT设计器自动生成的）。

⑤INCLUDEPATH #这个用来指定头文件路径, 例如：INCLUDEPATH += ./mitab。

⑥LIBS #加载动态库。例如：LIBS += ./mitab/libmitab.so。

⑦DEFINES #可以用来定义宏。

⑧CONFIG       #添加编译选项。例如: CONFIG+=c++11使用c++11特性，CONFIG+=console使用控制台。

⑨TRANSLATIONS #字典文件

⑩TEMPLATE #生成Makefile采用的模板。

（3）qmake文件中可供选择的TEMPLATE模板：

①app：建立一个应用程序Makefile，这个也是默认值，没有指定TEMPLATE模板时默认是app。

②lib：建立一个库的Makefile。

③vcapp：建立一个应用程序的visual studio项目文件。

④vclib：建立一个库的visual studio项目文件。

⑤subdirs：创建一个能够进入子目录并且为之生成Makefile。需要在后面用SUBDIRS=XX来指定那些子目录。

（5）HelloWorld.proc文件解析：

QT += core gui #在项目中添加core gu模块。

greaterThan(QT\_MAJOR\_VERSION, 4): QT += widgets #qmake中的条件执行语句，表示当Qt #版本大于4时，才加入widgets模块。

TARGET = HelloWorld #表示生成的目标可执行文件的名称是HelloWorld.exe

TEMPLATE = app #表示项目使用的模板是app，即一般的应用程序。

DEFINES += QT\_DEPRECATED\_WARNINGS #定义QT\_DEPRECATED\_WARNING宏，使用弃用了的Qt API

#编译器将产生警告。

CONFIG += c++11 #编译器支持C11特殊

SOURCES += \ #指定项目包含的源文件

main.cpp \

mainwindow.cpp

HEADERS += \ #指定项目包含的头文件

mainwindow.h

FORMS += \ #指定项目中用到的ui文件

mainwindow.ui

21.Qt中大部分的类都是继承至QObject，QObject是Qt中最基础的类。QWidget在Qt中是所有窗口类的基类。

通过帮助文档可查看类的继承关系。

22.Qt项目中的\*.ui文件是窗体界面定义文件，是一个XML文件，定义了窗口上的所有组件的属性设置、布局、及其信号和槽函数的关联等。用UI设计器可视化设计的界面都由Qt自动解析，并以XML文件的形式保存下来。在设计界面时，只需在UI设计器里进行可视化设计即可，而不用管\*.ui文件是怎么生成的。

23.Qt项目中的ui\_\*.h文件解析：

（1）ui\_\*.h文件是在对\*.ui编译后生成的一个头文件，它会出现在编译后的目录下，或与\*.ui同目录，这与 项目的shadow build编译设置有关。ui\_\*.h文件并不会出现在Qt Creator的项目文件目录树里，当然可 以手工将ui\_\*.h头文件添加到项目中。方法是在项目文件目录树上，右击项目名称节点，在弹出的快捷

菜单中选择“Add Existing Files…”，找到并添加ui\_\*.h文件即可。

（2）注意ui\_\*.h文件是对\*.ui样式文件编译后自动生成的，而\*.ui文件又是通过UI设计器可视化设计生成的。 所以对ui\_\*.h文件手工进行修改没有什么意义，所有涉及界面的修改都应该直接在UI设计器里进行。所以 ui\_\*.也没有必要添加到Qt Creater的项目文件目录树里。

（3）ui\_\*.h文件里实现界面功能的类是Ui\_\*。再定义一个Ui::\*类从Ui\_\*类继承而来，并定义在namespace Ui 里，这样Ui::\*与\*.h里的用户派生类同名，但是用namespace区分开来。所以界面的Ui::\*类与文件\*.h 里定义的用户派生类实际上是两个类，但是Qt的处理让用户感觉不到Ui::\*类的存在，只需要知道在用户 派生类里用ui指针可以访问可视化设计的界面组件就可以了。

（4）ui\_\*.h文件的作用：

①定义了一个类Ui\_\*，用于封装可视化设计的窗体界面。

②自动生成了窗体界面上各个组件的类成员变量。在Ui\_\*类中的public部分为界面上每个组件定义了一个 指针变量，变量的名称就是属性编辑器中设置的objectName。

③定义了setupUi()函数用于创建各个界面组件，并设置其位置、大小、文字内容、字体等属性，设置信号 与槽的关联。在窗口派生类的构造函数里调用ui->setupUI(this)，就实现了窗体上组件的创建、属性设 置、信号与槽的关联。

④定义namespace Ui，并定义一个从Ui\_\*继承的Ui::\*类。

（5）ui\_\*.h文件中setupUi()函数体的组成：

①根据可视化设计的界面内容，用C++代码创建界面上各组，并设置其属性。

②调用了函数retranslateUi()，用来设置界面各组件的文字内容属性，将界面上的文字设置的内容独立出

来作为一个函数retranslateUi()，在设计多语言界面时会用到这个函数。

③将在UI设计器里设置的信号与槽的关联转换为语句，并设置槽函数的关联方式。

28.认识Qt中的窗口：

（1）窗口大小：Qt中每一个窗口都可以使用resize()和size()函数设置其窗口大小，以像素为单位。

（2）窗口坐标系：Qt中每一个窗口都有一个坐标系统，默认窗口左上角为坐标原点，水平向右递增，水平向

左递减，垂直向下递增，垂直向上递减。原点即为（0，0）点，以像素为单位增减。

（3）窗口关系：普通窗口可以以顶级窗口方式显示，比如直接调用show()函数来显示，也可以嵌套在其它窗

口上来显示，这个时候被嵌套的窗口叫做父窗口，嵌套的窗口叫做子窗口。注意父子窗口是对象间的组

合关系，不是父子类继承关系。子窗口位置坐标参考的是父窗口的坐标系。顶级窗口其父窗口就是桌面，

桌面也可以认为是一个窗口。

（4）窗口移动的作用点是该窗口的坐标原点，也就是窗口的左上角。

(二)信号和槽机制:

1.信号和槽机制（Signal&Slot）：

（1）Qt提供了信号和槽机制用于完成界面操作的响应，是完成任意两个Qt对象之间的通信机制。其中信号

会在某个特定情况或动作下被触发，槽是等同于接收并处理信号的函数。若要将一个窗口部件的变化情

况通知给另外一个窗口部件，则一个窗口部件发送信号，另一个窗口部件的槽接收此消息并进行相应的

操作，即可实现两个窗口部件之间的通信。

（2）每个Qt对象都包含若干个预定义的信号和若干个预定义的槽，当某一个特定事件发生时，一个信号被发

出，与信号相关联的槽则会响应信号并完成相应的处理。当一个类被继承时，该类的信号和槽也同时被

继承，也可以根据需要自定义信号和槽。

（3）Qt中各部件已经预定义了很多信号和槽，用户也可以在类中增加新的信号和槽，以便实现特定的功能信

号和槽遵循C++的基本语法，在定义信号和槽是，只需要添加关键字signal和slot即可。要实现信号和槽

的关联，用connect操作实现。

2.信号和槽机制的意义：

（1）信号和槽用于两个QObject对象或者其派生类之间的通信，信号和槽机制是Qt的核心特征，也是Qt不

同于其它开发框架的最突出的特征。信号与槽（Signal & Slot）是 [Qt](http://c.biancheng.net/qt/) 编程的基础，也是Qt的一大创新。

因为有了信号与槽的编程机制，在Qt中处理界面各个组件的交互操作时变得更加直观和简单。

（2）Qt的信号槽机制并不仅仅是使用系统提供的那部分，还会允许我们自己设计自己的信号和槽。这也是Qt

框架的设计思路之一，用于我们设计解耦的程序。信号槽不是GUI模块提供的，而是Qt核心特性之一。因

此，我们可以在普通的控制台程序使用信号槽。

（3）信号和槽是松耦合的可以各自在类中定义和实现，彼此在连接时进行参数匹配之外，没有任何相关性。 一个信号可以触发多个槽，多个信号也可以连接至一个槽。需要注意的是，在槽中发射信号不要产生死 循环。

3.信号和槽机制的实现方法：

（5）信号和槽是QT自行定义的一种通信机制，它独立于标准的C++语言，是对标准C++的扩展。因此要正确 的处理信号和槽，必须借助一个称为Meta Object Compiler的QT元对象编辑器工具，该工具是一个C++ 预处理程序，它为高层次的事件处理自动生成所需要的附加代码。

（4）信号和槽机制实际就是观察者模式。当某个事件发生之后，它就会发出一个信号（signal）。这种发出

信号是没有目的的，类似广播。如果有对象对这个信号感兴趣，就可以调用连接（connect）函数来关联

此对象的一个槽函数（slot），来处理这个信号。也就是说当信号发出时，被连接的槽函数会自动被回

调。这就类似观察者模式，当发生了感兴趣的事件，某一个操作就会被自动触发。但Qt的信号槽使用了

额外的处理来实现，并不是GoF经典的观察者模式的实现方式。

4.自定义信号和槽的要求：

（1）只有继承至QObject的派生类才能使用信号槽。

（2）必需在头文件添加宏Q\_OBJECT，且Q\_OBJECT 这个宏只能在头文件中才能被展开，源文件中不能使用。

（3）关键字signals:指定声明信号区域。信号是一个函数声明，返回值为void，不需要实现函数体。

（4）关键字[public/protect/private] slot:指定申明槽函数区域。

（5）调用QObject::connect()函数连接信号和槽，并使用emit在恰当的位置发送信号。

5.信号和槽使用的注意事项

（1）信号和槽是Qt对象的重要特征，使用的信号和槽必须是QObject的子类，否则无法使用信号和槽来建立

关联。信号和槽是类型安全的，只要信号的参数在顺序上是大于槽的参数，即可建立匹配，在响应的时

候，系统自动匹配满足槽的参数，放弃多余的参数。

（2）只有继承了QObject的类，才具有信号槽的能力。所以为了使用信号槽，必须继承QObject。凡是QObject 类（不管是直接子类还是间接子类），都应该在第一行代码写上Q\_OBJECT。不管是否使用信号槽，都应 该添加这个宏。这个宏的展开将为我们的类提供信号槽机制、国际化机制以及Qt提供的不基C++ RTTI 的反射能力。因此如果你觉得你的类不需要使用信号槽，就不添加这个宏，就是错误的。其它很多操作 都会依赖于这个宏。

（3）Q\_OBJECT宏将由moc做特殊处理，不仅仅是宏展开这么简单。moc会读取标记了Q\_OBJECT的头文件，生成 以moc\_ 为前缀的文件。由于moc只处理头文件中的标记了Q\_OBJECT的类声明，不会处理cpp文件中的类似 声明。因此如果标记了Q\_OBJECT的类位于cpp文件中，是无法得到moc的处理的。解决方法是，我们手动 调用moc工具处理cpp文件，并且将cpp文件中include包含手动添加了moc\_前缀的头文件就可以了。

6.信号概念：

（1）信号（Signal）：就是在特定情况下被发射的事件。GUI程序设计的主要内容就是对界面上各组件的信号

的响应，只需要知道什么情况下发射哪些信号，合理地去响应和处理这些信号就可以了。

（4）信号可以看做是特殊的函数，需要带括号，有参数时还需要指明参数。当信号和槽函数带有参数时，在

connect()函数里，要写明参数的类型，但可以不写参数名称。

（2）信号的特点：

①信号的声明是在头文件中进行的，QT中的signals关键字指出进入了信号声明区，随后即可声明信号，信 号没有调用权限，不需要在申明的时候指定(public、protected、private)等权限。事实上如果信号是

private的，这个信号就不能在类的外面连接，也就没有任何意义。

②从形式上讲，信号的声明与普通的C++函数是一样的，但是信号却没有函数体定义，另外信号的返回类型 都是void，不要指望能从信号返回什么有用信息。即：信号只声明不实现。

③信号可以通过关键字emit来发射。即：emit signal()。

7.槽的概念：

（3）槽（Slot）：就是对信号响应的函数。槽就是一个函数，与一般的[C++](http://c.biancheng.net/cplus/)函数是一样的，可以定义在类的任

何部分（public、private或protected），可以具有任何参数，也可以被直接调用。槽函数与一般的函

数不同的是槽函数可以与一个信号关联，当信号被发射时，关联的槽函数被自动执行。

（4）Qt中任何成员函数、static函数、全局函数和Lambda表达式都可以作为槽函数。与信号函数不同槽函数

必须自己完成实现代码。槽函数就是普通的成员函数，因此作为成员函数，也会受到public、protected

和private访问控制符的影响。

（2）槽的特点：

①槽函数像普通的C++成员函数，可以被正常调用，与普通函数的不同之处是信号可以与槽函数相关联。当

与其关联的信号被发射时，这个槽就会被调用。

②槽函数与其它普通函数一样，它们也有存取权限。槽函数也分为三种类型，即public slots、private slots

和protected slots。

③protected slots：在这个区内声明的槽意味着当前类及其子类可以将信号与之相连接。

④private slots：在这个区内声明的槽意味着只有类自己可以将信号与之相连接。

⑤public slots:在这个区内声明的槽意味着任何对象都可将信号与之相连接。

8.连接信号和槽：

（1）信号和槽关联函数的一般格式：connect(sender,SIGNAL(signal()),receiver,SLOT(slot()))。其中第

一个参数sender是发出信号的对象，第二个参数signal()是发送对象发出的信号，第三个参数receiver

是接收信号的对象，第四个参数slot()是接收对象在接收到信号之后所需要调用的函数。

（2）​通过调用QObject对象的connect函数来将某个对象的信号与另外一个对象的槽函数相关联，这样当发射 者发射信号时，接收者的槽函数将被调用。信号作为connect的传入参数时候需要使用SIGNAL()这个宏函

数来包装一下，槽函数作为connect的传入参数时候需要使用SLOT()这个宏函数来包装一下。

（3）SIGNAL()和SLOT()是Qt定义的两个宏函数，它们返回其参数的C风格字符串（const char\*），即这两个

宏函数包装的内容可用C风格字符串替换，SIGNAL(signal())= “signal()”, SLOT(slot())=”slot()”。（4）connect()是QObject类的一个静态函数，而QObject是所有Qt类的基类，在实际调用时可以忽略前面的限

定符。emit是Qt对C++扩展的关键字，其实也是一个宏，它的含义是发出一个信号

（6）在Qt Designer中，右击鼠标，在弹出的快捷菜单中选择go to slot命令可以很快的在主窗口中建立对信

号的响应。

9.断开信号和槽：当信号与槽没有必要继续保持关联时，可以使用disconnect函数来断开连接。其一般形式为：disconnect(sender,SIGNAL(signal()),receiver,SLOT(slot()))。

10.信号和槽机制的连接形式：

（1）一个信号可以与另一个信号相连:

connect(Object,”signal()”,Object,” signal()”);

connect(Object,SIGNAL(signal()),Object,SIGNAL(signal()));

（2）一个信号和一个槽相连：

connect(Object,”signal()”,Object,”slot()”);

connect(sender,SIGNAL(signal()),receiver,SLOT(slot()));

（3）同一个信号可以与多个槽相连：

connect(Object1,SIGNAL(signal2),Object2,SIGNAL(slot2));

connect(Object1,SIGNAL(signal2),Object3,SIGNAL(slot1));

（4）同一个槽可以响应多个信号：

connect(Object1,SIGNAL(signal2),Object2,SIGNAL(slot2));

connect(Object3,SIGNAL(signal2),Object2,SIGNAL(slot2));

（5）一个信号跟多个槽函数链接时，当这个信号被发射，这些槽将会一个接一个地执行，但是它们执行的顺 序将会是随机的、不确定的，我们不能人为地指定哪个先执行、哪个后执行。多个信号和一个槽函数链

接，那么只要其中一个信号被发射，都能调用该槽函数。

（6）当一个信号被发射时，与其关联的槽函数通常被立即执行，就像正常调用一个函数一样。只有当信号关

联的所有槽函数执行完毕后，才会执行发射信号处后面的代码。

（7）在非多线程对象之间，使用是以直接链接方式，即只有当所有的槽返回以后发射函数才返回。多线程之

间对象的信号槽链接，是以队列的方式链接，即发射函数后不需要等待槽返回。

（8）信号槽要求信号和槽的参数一致，所谓一致是参数类型一致。如果不一致，允许的情况是，槽函数的参

数可以比信号的少，即便如此，槽函数存在的那些参数的顺序也必须和信号的前面几个一致起来。这是

因为，你可以在槽函数中选择忽略信号传来的数据（也就是槽函数的参数比信号的少），但是不能说信

号根本没有这个数据，你就要在槽函数中使用（就是槽函数的参数比信号的多，这是不允许的）。

4.信号与槽机制的优点:

（1）类型安全。需要关联的信号和槽的签名必须是等同的，即信号的参数类型和参数个数与接收该信号的槽

的参数类型和参数个数相同。不过一个槽的参数个数是可以少于信号的参数个数的，但缺少的参数必须

是信号参数的最后一个或几个参数。如果信号和槽的签名不符，编译器就会报错。

（2）松散耦合。信号和槽机制减弱了Qt对象的耦合度。激发信号的Qt对象无须知道是哪个对象的哪个槽需要

接收它发出的信号，它只需做的是在适当的时间发送适当的信号就可以了，而不需要知道也不关心它的

信号有没有被接收到，更不需要知道是哪个对象的哪个槽接收到了信号。同样对象的槽也不知道哪些信 号关联了自己，而一旦关联信号和槽，Qt就保证了适合的槽得到调用。即使关联的对象在运行时被删除， 应用程序也不会崩溃。

（3）一个类若要支持信号和槽，就必须从QObject或QObject的子类继承。注意Qt信号和槽机制不支持对模板

的使用。信号与槽机制是Qt GUI编程的基础，主要用于窗口部件间的通信。

5.信号与槽机制的效率：

（1）信号和槽机制增强了对象间通信的灵活性，然而这也损失了一些性能。同回调函数相比，信号和槽机制

运行速度有些慢。通常使用传递一个信号来调用槽函数将会比直接调用非虚函数运行速度慢10倍。

（2）信号和槽机制运行变慢的原因：

①需要定位接收信号的对象。

②安全地遍历所有的关联（如一个信号关联多个槽的情况）。

③编组（marshal）/解组（unmarshal）传递的参数。

④多线程的时候，信号可能需要排队等待。

（3）然而与创建堆对象的new操作及删除堆对象的delete操作相比，信号和槽的运行代价只是他们很小的一部

分。信号和槽机制导致的这一点性能损耗，对实时应用程序是可以忽略的；同信号和槽提供的灵活性简

便性相比，这点性能的损失也是值得的。

10.Qt中还有另外一种响应机制：事件。Qt中的事件主要来自底层的窗口系统，如鼠标、键盘、定时器、网络、触摸屏等。对事件的响应一般采用重载类的成员函数来实现，从实现上事件比信号和槽要更底层一些。

18. UI的可视化设计是对用户而言的，其实底层都是[C++](http://c.biancheng.net/cplus/)的代码实现，只是[Qt](http://c.biancheng.net/qt/)巧妙地进行了处理，让用户省去了很多繁琐的界面设计工作。由于界面设计的底层其实都是由C++语言实现的，底层实现的功能比可视化设计更加强大和灵活。某些界面效果是可视化设计无法完成的，或者某些人习惯了用纯代码的方式来设计界面，就可以采用纯代码的方式设计界面。Qt自带的实例基本都是用纯代码方式实现用户界面的。

(三)布局管理器:

1.界面组件布局：

（1）Qt的界面设计使用了布局（Layout）功能。所谓布局就是界面上组件的排列方式，使用布局可以使组件

有规则地分布，并且随着窗体大小变化自动地调整大小和相对位置。布局管理是GUI设计的必备技巧。

（2）GUI界面归根结底就是一堆组件的叠加。在拖放组件时，组件的位置尤其重要。我们必须要指定组放在哪 里，以便窗口能够按照我们需要的方式进行渲染。Qt提供了两种组件定位机制：绝对定位和布局定位。

（3）绝对定位：就是一种最原始的定位方法，给出这个组件的坐标和长宽值。这样Qt就知道该把组件放在哪 里以及如何设置组件的大小。但是这样做带来的一个问题是，如果用户改变了窗口大小，采用绝对定位 的组件是不会有任何响应的。这也很自然，因为你并没有告诉Qt，在窗口变化时组件是否要更新自己以 及如何更新。如果你需要让组件自动更新，就要自己编写相应的函数来响应这些变化。或者还有更简单 的方法，即禁止用户改变窗口大小。但这总不是长远之计。

（4）布局定位：针对窗口大小和位置经常变化的需求，Qt提供了另外的一种定位机制，即布局定位来解决这 个问题。你只要把组件放入某一种布局，布局由专门的布局管理器进行管理。当需要调整大小或者位置 的时候，Qt使用对应的布局管理器进行调整。

2.使界面上各组件之间具有层次关系的方法：

（1）为了将界面上的各个组件的分布设计得更加美观，经常使用一些容器类，如QGoupBox、QTabWidget、QFrame、 QWidget、QToolBox等。

（2）Qt为界面设计提供了丰富的布局管理功能，在UI设计器中，组件面板里有Layouts和Spacers两个组件面 板，在窗体上方的工具栏里有布局管理的按钮。

3.布局管理器的作用：

（1）在设计较复杂的GUI用户界面时，仅通过指定窗口部件的父子关系以期达到加载和排列窗口部件的方法是 行不通的，最好的办法是使用Qt提供的布局管理器。布局管理器可以让程序员摆脱繁琐的窗口位置大小设 置工作。并且布局管理器可以依据窗口大小变更布局控件的位置大小。

（2）每个窗口对象都可以设置一个布局管理器，来管理所有它的子窗口，由于布局管理器继承至QObject，那 么布局管理的内存释放也可以托管给窗口对象，而所有添加到布局管理器中的窗口对象，它们的内存释 放又都托管给这个布局管理器。

4.常用的布局组件：

（2）QHBoxLayout：是水平布局管理器，可以将所有托管给它的窗口对象按照水平方向自动从左向右进行布局。

（3）QVBoxLayout：是垂直布局管理器，可以将所有托管给它的窗口对象按照垂直方向自动从上到下进行布局。

（4）QGridLayout：是网格布局管理，类似于HTML的table，在一个网格中进行布局。网格布局大小改变时， 每个网格的大小都改变

（5）QFormLayout：是表格布局管理器，每一行前面是一段文本，文本后面跟随一个组件（通常是输入框）， 类似 HTML的form。与网格状布局类似，但是只有最右侧的一列网格会改变大小。

（6）QStackedLayout：是层叠布局管理器，允许我们将几个组件按照Z轴方向堆叠，可以形成向导那种一页一 页的效果。

（7）Horizontal Spacer：一个用于水平分隔的空格。

（8）Vertical Spacer：一个用于垂直分隔的空格

5.布局管理器基类为QLayout继承至QObject和QLayoutItem，它是一个抽象基类，需要被进一步继承才能使用。它的派生类有[QBoxLayout](qboxlayout.html)、[QFormLayout](qformlayout.html)、[QGridLayout](qgridlayout.html)和[QStackedLayout](qstackedlayout.html)。QHBoxLayout、QVBoxLayout它们又继承至QBoxLayout。

6.布局管理器的使用（QGridLayout）：

（1）QGridLayout \*layout = new QGridLayout(this):创建一个网格布局管理器对象layout，并用this指定 父窗口。还可以在创建布局管理器对象时不必指明父窗口。

（2）layout->addWidget():将控件对象放置到布局管理器中。

（3）QWidget::setLayout():将布局管理器添加到对应的窗口部件对象中。如果这里的主窗口就是父窗口， 直接调用setLayout(laytout)即可。如果主窗口中事先已经设置了一个布局管理器，那么会替换掉原来 的布局管理器。

7. 布局和界面设计工具栏的作用：

（1）在设计窗体的上方有一个布局和界面设计工具栏，用于调整设计器进入不同的状态，以及进行布局和界 面设计。使用工具栏上的布局控制按钮时，只需在窗体上选中需要设计布局的组件，然后点击某个布局 按钮即可。在窗体上选择组件时同时按住Ctrl键，可以实现组件多选，选择某个容器类组件，相当于选 择了其内部的所有组件。

（2）UI设计器工具栏各按钮的功能：

|  |  |
| --- | --- |
| **按钮及快捷键** | **功能** |
| Edit Widget (F3) | 界面设计进入编辑状态，就是正常的设计状态。 |
| Edit Signals/Slots(F4) | 进入信号与槽的可视化设计状态。 |
| Edit Buddies | 进入伙伴关系编辑状态，可以设置一个Label 与一个组件成为伙伴关系。 |
| Edit Tab Order | 进入Tab顺序编辑状态，Tab顺序是在键盘上按Tab键时，输入焦点在界面各组件之间跳动的顺序。 |
| Layout Horizontally (Ctrl+H) | 将窗体上所选组件水平布局。 |
| Layout Vertically (Ctrl+L) | 将窗体上所选组件垂直布局。 |
| Layout Horizontally in Splitter | 将窗体上所选组件用一个分割条进行水平分割布局。 |
| Layout Vertically in Splitter | 将窗体上所选组件用一个分割条进行垂直分割布局。 |
| Layout in a Form Layout | 将窗体上所选组件按窗体布局。 |
| Layout in a Grid | 将窗体上所选组件网格布局。 |
| Break Layout | 解除窗体上所选组件的布局，也就是打散现有的布局。 |
| Adjust Size(Ctrl+J) | 自动调整所选组件的大小。 |

8.在UI设计器里可视化设计布局时，要善于利用水平和垂直空格组件，善于设置组件的最大、最小宽度和高度来实现某些需要的布局效果。有些时候，我们需要实现复杂的布局效果，只使用一种布局管理器，无法完成，那么这个时候需要嵌套使用布局管理器。即布局管理器中添加布局管理器。

10.在UI设计器工具栏上单击“Edit Tab Order”按钮进入Tab顺序编辑状态。Tab顺序是指在程序运行时，按下键盘上的Tab键时输入焦点的移动顺序。一个好的用户界面，在按Tab键时，焦点应该以合理的顺序在界面上移动，而不是随意地移动。进入Tab顺序编辑状态后，在界面上会显示具有Tab顺序组件的编号，依次按希望的顺序单击组件，就可以重排Tab顺序了。没有输入焦点的组件是没有Tab顺序的。

11. 常用的布局管理器方法：

（1）void QBoxLayout::addWidget(QWidget \*widget)：将控件添加到布局管理器中。

（2）void QBoxLayout::setStretch(int index,int stretch)：设置伸缩系数。

（3）void QBoxLayout::addStretch(int stretch= 0)：添加类似弹簧的东西。

（4）void QBoxLayout::setSpacing(int spacing)：设置布局管理中控件的间隔。

（5）void QBoxLayout::addSpacing(int size)：布局管理器中添加间隙。

（6）void QLayout::setMargin(int margin)：设置布局管理器边界。

（7）bool QLayout::setAlignment(QWidget \*w, Qt::Alignment alignment)：设置对齐方式。

（8）void QBoxLayout::addLayout(QLayout \*layout, int stretch = 0)：QHBoxLayout与QVBoxLayout使 用这个函数来嵌套其它布局管理器。

（9）void QGridLayout::addLayout(QLayout \*layout,int row,int column)：QGridLayout使用这个函数来 嵌套其它布局管理器。

（10）void QGridLayout::setRowStretch(int row,int stretch)：网格布局管理器设置行伸缩系数。

（11）void QGridLayout::setColumnStretch(int column,int stretch)：网格布局管理器设置列伸缩系数。（12）void QGridLayout::setRowMinimumHeight(int row,int minSize)：网格布局管理器设置行最小高度。

（13）void QGridLayout::setColumnMinimumWidth(int column,int minSize)：网格布局管理器设置列最小 宽度。

（10）virtual QSize sizeHint() const：如果想获取布局管理器中组件的大小，这时就不能使用size 函数 来获取，需要通过sizeHint函数来获取。这是因为size获取的是窗口实际大小，一般通过resize来设 置，不设置的时候也有默认大小。然而如果设置了布局管理器，那么窗口在显示的时候还会按照特定的 布局策略重置窗口大小，而sizeHint获取到的是重置后的窗口大小。如果没有使用布局管理器，那么 sizeHint返回无效的值。

(四)Qt模块介绍:

1. Qt5与Qt4最大的一个区别之一是底层架构有了修改。Qt5引入了模块化的概念，将众多功能细分到几个模块之中。Qt4也有模块的概念但是是一种很粗的划分，而Qt5则更加细化。Qt5模块分为Essentials Modules基础模块和Add-on Modules扩展模块两部分。基础模块在所有平台上都可用，扩展模块建立在基础模块的基础之上，在能够运行Qt的平台之上可以酌情引入。由于Qt的扩展模块并不是Qt必须安装的部分，因此Qt在未来版本中可能会提供更多的扩展模块。

2. Qt基础模块：

（1）Qt Core：提供核心的非GUI功能，所有模块都需要这个模块。这个模块的类包括了动画框架、定时器、

各个容器类、时间日期类、事件、IO、JSON、插件机制、智能指针、图形（矩形、路径等）、线程、XML

等，所有这些类都可以通过<QtCore>头文件引入。

（2）Qt Gui：提供GUI程序的基本功能，包括与窗口系统的集成、事件处理、OpenGL和OpenGL ES集成、2D图 像、字体、拖放等。这些类一般由Qt用户界面类内部使用，当然也可以用于访问底层的OpenGL ES图像API。 Qt Gui模块提供的是所有图形用户界面程序都需要的通用功能。

（3）Qt Multimedia：提供视频、音频、收音机以及摄像头等功能。这些类可以通过<QtMultimedia>头文件引 入，而且需要在pro文件中添加 QT += multimedia。

（4）Qt Network：提供跨平台的网络功能。这些类可以通过<QtNetwork>头文件引入，而且需要在pro文件中 添加QT +=network。

（5）Qt Qml：提供QML（一种脚本语言，也提供JavaScript的交互机制）使用的C++ API。这些类可以通过<QtQml> 头文件引入，而且需要在pro文件中添加QT += qml。

（6）Qt Quick：允许在Qt/C++程序中嵌入Qt Quick（一种基于Qt的高度动画的用户界面，适合于移动平台开 发）。这些类可以通过<QtQuick>头文件引入，而且需要在pro文件中添加 QT += quick。

（7）Qt SQL：允许使用SQL语言访问数据库。这些类可以通过<QtSql>头文件引入，而且需要在pro文件中添加 QT += sql。

（8）Qt Test：提供Qt程序的单元测试功能。这些类可以通过<QtTest>头文件引入，而且需要在pro文件中添 加 QT += testlib。

（9）Qt Webkit：基于WebKit2的实现以及一套全新的QML API。

2. Qt常用的扩展模块：

（1）Qt 3D：提供声明式语法，在Qt程序中可以简单地嵌入3D图像。Qt 3D为Qt Quick添加了3D内容渲染。Qt 3D 提供了QML和C++两套API，用于开发3D程序。

（2）Qt Bluetooth：提供用于访问蓝牙无线设备的C++和QML API。

（3）Qt Concurrent：封装了底层线程技术的类库，方便开发多线程程序。

（4）Qt Location：提供定位机制、地图和导航技术、位置搜索等功能的QML和C++ API。

（5）Qt OpenGL：方便在Qt应用程序中使用OpenGL。该模块仅仅为了程序从Qt 4移植到Qt 5的方便才保留下来， 如果你需要在新的Qt5 程序中使用OpenGL相关技术，需要使用的是QtGui模块中的QOpenGL。

（6）Qt Sensors：提供访问各类传感器的QML和C++ API。

（7）Qt Service Framework：提供客户端发现其他设备的服务。Qt Service Framework为在不同平台上发现、 实现和访问服务定义了一套统一的机制。

（8）Qt System Info：提供一套API，用于发现系统相关的信息，比如电池使用量、锁屏、硬件特性等。

（9）Qt Tools：提供了Qt开发的方便工具，包括Qt CLucene、Qt Designer、Qt Help以及Qt UI Tools。

（10）Qt Widgets：使用C++扩展的Qt Gui模块，提供了一些界面组件，比如按钮、单选框等

（11）Qt XML：SAX和DOM的C++实现。该模块已经废除，请使用QXmlStreamReader/Writer。

(五)Qt Creator使用技巧:

1. [Qt](http://c.biancheng.net/qt/)  Creator在设计界面或编辑代码时，有一些快捷键和使用技巧，熟悉这些快捷键和使用技巧，可以提高工作效率。另外在使用Qt时，要善于使用Qt自带的帮助文件，对于一个编程语言或类库来说，其自带的帮助文件是最全面最权威的资料。当光标停留在一个类名或函数上时，按F1可以调出其帮助文件的内容。

2. 在Qt Creator主窗口左侧的主工具栏上有“Help”按钮，单击可以打开Qt的帮助文件系统，也可以使用“开始”菜单Qt程序组里的Assistant单独打开帮助系统。在Qt帮助系统里可以搜索查看每个类的详细资料，包括在这个中类定义的公共类型、属性、公共函数、信号、公共槽等。另外若要查看类的继承关系，可以访问 Qt官网的“Inheritance Hierarchy”页面。

3.Qt帮助文件系统的四种显示模式：

（1）Bookmarks模式：左边框里显示已存储的Bookmarks（书签），任何帮助页面下，点击窗口上方工具栏上 的“Add Bookmark”可以添加书签。

（2）Contents模式：左边框里以目录树形式显示Qt的所有模块，可以分类浏览想看的内容。

（3）Index模式：可以输入查找内容，左边框里会列出与输入内容前匹配的帮助主题列表。

（4）Search模式：可以输入关键字进行搜索

4.  Qt Creator中常用的快捷操作：

|  |  |  |
| --- | --- | --- |
| **功能** | **快捷键** | **解释** |
| Switch Header/Source | F4 | 在同名的头文件和源程序文件之间切换。 |
| Follow Symbol Under Cursor | F2 | 跟踪光标下的符号，若是变量，可跟踪到变量声明的地方；若是函数体或函数声明，可在两者之间切换。 |
| Switch Between Function Declaration and Definition | Shift+F2 | 在函数的声明（函数原型）和定义（函数实现）之间切换。 |
| Refactor\Rename Symbol Under Cursor | Ctrl+Shift+R | 对光标处的符号更改名称，这将替换到所有用到这个符号的地方。 |
| Refactor\Add Definition in .cpp |  | 为函数原型在cpp文件里生成函数体。 |
| Auto-indent Selection | Ctrl+I | 为选择的文字自动进行缩进。 |
| Toggle Comment Selection | Ctrl+/ | 为选择的文字进行注释符号的切换，即可以注释所选代码，或取消注释。 |
| Context Help | F1 | 为光标所在的符号显示帮助文件的内容。 |
| Save All | Ctrl+Shift+S | 文件全部保存。 |
| Find/Replace | Ctrl+F | 调出查找/替换对话框。 |
| Find Next | F3 | 查找下一个。 |
| Build | Ctrl+B | 编译当前项目。 |
| Start Debugging | F5 | 开始调试。 |
| Step Over | F10 | 调试状态下单步略过，即执行当前行程序语句。 |
| Step Into | F11 | 调试状态下跟踪进入，即如果当前行里有函数，就跟踪进入函数体。 |
| Toggle Breakpoint | F9 | 设置或取消当前行的断点设置。 |

(六) Qt元对象和属性系统详解：

1. [Qt](http://c.biancheng.net/qt/) 是一个用标准 [C++](http://c.biancheng.net/cplus/) 编写的跨平台开发类库，它对标准C++进行了扩展，引入了元对象系统、信号与槽、属性等特性，使应用程序的开发变得更高效。标准 [C++](http://c.biancheng.net/cplus/) 语言里并没有property关键字，property是Qt对标准C++的扩展，使得在Qt Designer里就可以可视化设置类的数据。

2. Qt元对象系统提供了对象间的信号和槽通信机制、运行时类型信息和动态属性系统的支持，是标准C++的一个扩展，它使Qt能够更好地实现GUI图形用户界面编程。Qt的元对象系统不支持C++模板尽管模板扩展了标准C++的功能，但是元对象系统提供了模板无法提供的一些特性。

3.Qt 5元对象系统的组成基础：

（1）基类QObject：任何需要使用元对象系统功能的类必须继承自QObject。

（2）Q\_OBJECT宏：Q\_OBJECT宏必须出现在类的私有声明区，用于启动元对象的特性。

（3）元对象编译器（Meta-Object Compiler或moc）：为QObject子类实现元对象特性提供必要的代码实现。 当构建Qt项目时，MOC工具读取C++源文件，当它发现类的定义里有Q\_OBJECT宏时，它就会为这个类生成 另外一个包含有元对象支持代码的C++源文件，这个生成的源文件连同类的实现文件一起被编译和连接。

4.Qt元对象提供的常用功能：

（1）QObject::connect()：用于实现信号与槽函数关联。

（2）QObject::metaObject()：返回类所关联的元对象。元对象类QMetaObject包含了访问元对象的一些接口 函数。Qt的反射是基于Qt元对象系统实现的。

（3）QMetaObject::newInstance()：创建类的一个新的实例。

（4）QObject::inherits(const char \*className)：判断一个对象实例是否是名称为className的类或QObject 的子类的实例。

（5）QObject::tr()/QObject::trUtf8()：可翻译字符串，用于多语言界面设计。

（6）QObject::setProperty()/QObject::property()：用于通过属性名称动态设置和获取属性值。对于QObject 及其子类，还可以使用qobject\_cast()函数进行动态投射。使用动态投射，可以让程序在运行时对不同 的对象做不同的处理。

5. Qt提供一个Q\_PROPERTY()宏可以定义属性，它也是基于元对象系统实现的。Qt的属性系统与C++编译器无关，可以用任何标准的C++编译器编译定义了属性的Qt C++程序。使用Q\_PROPERTY()宏为继承于QObject的类声明属性后，被声明的属性行为与类数据成员一样，但是他们可以通过元对象系统进行访问。

6. Qt属性系统：

（1）在QObject的子类中，用宏Q\_PROPERTY()定义属性的常用格式：

①Q\_PROPERTY(类型 名称 READ get函数 WRITE set函数 RESET reset函数)。

②Q\_PROPERTY(type name READ getFunction [WRITE setFunction] [RESET resetFunction])。

（2）Q\_PROPERTY宏定义一个返回值类型为type，名称为name的属性，用READ、WRITE关键字定义属性的读 取、写入函数，还有其他的一些关键字定义属性的一些操作特性。属性的类型可以是QVariant支持的任 何类型，也可以用户自定义类型。

（3）属性名称和类型以及READ是必须的，其中属性类型可以是QVariant支持的所有类型或者自定义。还有 几个属性是可选的，但是WRITE和RESET功能很常见。除了USER默认true其他均默认为false。

（4）Q\_PROPERTY()是一个宏，用来在一个类中声明一个属性property，由于该宏是Qt特有的，需要用moc 工具进行编译，故必须继承于QObject类。一个属性的行为就像类的数据成员，但是它还具有附加的READ、 WRITE、RESET、NOTIFY、DESIGNABLE、SCRIPTABLE、STORED、USER、CONSTANT和FINAL等特性，这些特性 可以被元数据对象系统操作。

7.Q\_PROPERTY宏定义属性的一些主要关键字含义：

（1）READ：指定一个读取属性值的函数，没有MEMBER关键字时必须设置READ。

（2）WRITE：是可选的，用于指定一个设定属性值的函数，只读属性没有WRITE设置。WRITE访问器函数必须 返回空并且至少具有一个属性类型值、指针或引用的参数。

（3）MEMBER：指定一个成员变量与属性关联，成为可读可写的属性，无需再设置 READ 和 WRITE。

（4）RESET：是可选的，用于指定一个设置属性默认值的函数。

（5）NOTIFY：是可选的，用于设置一个信号，当属性值变化时发射此信号。NOTIFY信号必须带有一个参数， 这个参数的类型必须与属性相同；参数保存的是属性的新值。

（6）DESIGNABLE：指定一个bool型变量，表示属性是否在Qt Designer的属性编辑器中可见，缺省值为true-。

（7）SCRIPTABLE：指定一个bool型变量，表明属性是否可以被一个脚本引擎操作，默认值是true。

（8）STORED：指定一个bool型的变量，表明属性是否被认为是独立存在还是依赖于其它的值而存在。它也表明是否在保存对象状态时保存此属性的值。大多数属性都是需要保存的

（9）

（8）CONSTANT：表示属性值是一个常数，对于一个对象实例，READ 指定的函数返回值是常数，但是每个实例 的返回值可以不一样。具有 CONSTANT 关键字的属性不能有 WRITE 和 NOTIFY 关键字。

（9）FINAL：表示所定义的属性不能被子类重载。

第二章：Qt模板库、工具类及控件

(一)字符串类:

1.QString类：

标准的C++提供了两种字符串：一种是C语言风格的以”\0”字符结尾的字符数组；另一种是字符串类QString。而Qt字符串类QString的功能更强大。QString类保存16位Unicode值，提供了丰富的操作、查询和转换等函数。该类还进行了使用隐式共享、高效的内存分配策略等多方面的优化。另外与C风格的字符串不同，QString中间是可以包含'\0'符号的，而length()函数则会返回整个字符串的长度，而不仅仅是从开始字符到以'\0'结尾的长度，使用QString让用户不用担心内存分配以及关于'\0'结尾的这些注意事项。

2.QString类的作用：

QString 类是Qt中用于表示字符串的类，其实现在QtCore共享库中。Qt把QString类做得非常方便，几乎涵盖了我们需要的所有字符串操作，QString类使用基于Unicode的字符串，支持多种语言字符集。如果要使用8位字符串，Qt提供了QByteArray类，使用方法和QString类似。

3.QString与QByteArray的区别：

（1）Qt的QByteArray类提供了很方便的对字节流操作的接口，QByteArray没有继承任何Qt基类。

（2）QString是专门用来处理字符串的，除了能处理ASCII编码字符，还包括各国语言的编码，默认情况下

QString会把所有数据当做utf-8编码来处理。QByteArray只是单纯用来处理数据的，除了能处理ASCII

编码字符，其它复杂的编码都不能处理，直接以字节流的方式来对待。

4.QString类的实现特征：

（1）Qt的QString类提供了很方便的对字符串操作的接口，采用Unicode内部编码，可以表示世界上大多数语 言的文字，QString没有继承任何Qt基类。

（2）字符串的存储采用引用计数，当一个QString对象被复制为另一个QString对象时，它们实际上指向相同

的存储空间，仅仅是增加一个引用计数。

（3）采用按需复制的技术，当指向相同存储空间的多个QString对象中的一个要被修改时，将真正复制

一个新的字符串并进行修改。

6.NULL字符串与空字符串的区别：

一个NULL字符串就是用QString的默认构造函数或者使用“(const char\*)0”作为参数的构造函数创建的QString字符串对象；而一个空字符串是一个大小为0的字符串。一个NULL字符串一定是一个空字符串，而一个空字符串未必是一个NULL字符串。

5. QString构造函数与析构函数：

（1）QString(); //构造空字符串,字符串的长度和数据指针都为0。

（2）QString(QChar ch); //构造一个长度为一个字符，内容为给定字符*ch*的字符串。

（3）QString(const QChar \*pch, int size); //构造一个[QChar](http://www.kuqin.com/qtdocument/qchar.html)数组中开始*size*个字符的[深度复制](http://www.kuqin.com/qtdocument/shclass.html#deep-copy)的字符串。

（4）QString(const QString &s); //拷贝构造函数，构造一个字符串*s*的[隐含共享](http://www.kuqin.com/qtdocument/shclass.html#implicitly-shared)的复制。

（5）QString(const char \*str); //构造一个被解释为经典C字符串的*str*的[深度复制](http://www.kuqin.com/qtdocument/shclass.html#deep-copy)的字符串。

（6）[~QString](http://www.kuqin.com/qtdocument/qstring.html#~QString) () //销毁这个字符串并且如果这是这个字符串的最后一个复制，就释放“真实的”字符串。

6.创建QString字符串方法：

（1）直接定义：使用构造函数或经过重载的赋值符号，创建QString字符串。

①QString str = “This is a string”;

②QString str(“This is a string”);

③QString str = QString(“This is a string”);

（2）从其他字符串类型转换：使用Qt提供的字符串类型转换函数，可以很方便的将其他格式的字符串或第三 方软件包支持的字符串转换为Qt的字符串。

①char \*otherStr = ”this is 8-bit character string”;

QString str = QString::fromAscii(otherStr);

②Qt提供的常用的几种其他类型字符格式转换为QString字符串的函数：

❶QString fromAscii(const char \*str, int size = -1);

❷QString fromLatin1(const char \*str, int size = -1);

❸QString fromRawDate(const QChar \*unicode, int size);

❹QString fromStdString(const Std::string &str);

❺QString fromUtf8(const char \*str, int size = -1);

（3）采用追加的方法：

QString str;

Str.append(“this is a append string”);

（4）格式化生成方法：

QString str = QString(”value is %1\n”).arg(100); //用占位符%1生成value is 100的字符串。

QString str.sprintf(”%s %s”,”Hello”,”world”);

8.字符串运算符：

（1）字符串赋值运算符（=）：

QString& [operator=](http://www.kuqin.com/qtdocument/qstring.html#operator-eq-3) (const char \*str)运算符传递给QString一个const char\*类型的ASCII字符串，它

被解释为一个典型的以“\0”结尾的C类型字符串。这将会导致调用QString(const char \*str)构造函数，

来初始化一个QString字符串。被传递的const char\*类型指针又被QString:: fromAscii()转换为Unicode

编码。默认情况下，函数QString::fromAscii()会将超过128的字符作为Latin-1进行处理。

（2）字符串组合运算符（+ 与 +=）：

①QString提供了一个二元的“+”操作符用于组合两个字符串，并提供了一个“+=”操作符用于将一个字

符串追加到另一个字符串的末尾。

②与Java的String类相似，QString也重载了+和+=运算符。这两个运算符可以很方便的把两个字符串连到

一起，QString可以自动的对占用内存空间进行扩充，这种连接操作是很迅速的。

③加法操作符可以将两个字符串或字符连接起来并以QString临时对象的方式返回。但必须注意加法操作 符的两个操作数中必须至少有一个是QString对象，否则无法重载操作符。

（5）字符串索引运算符（[]）：

QString类的取字符操作[]就类似于对一个字符数组的[]定位操作，根据下标得到某个位置上的字符,其 提供了两种重载的形式，即通过[]操作符得到的字符不能被修改和可以被修改，两种重载形式。

（4）QString类的字符串操作运算符都是通过重载C++中的标准运算符实现的。由于C++语言允许赋值操作符 和复合赋值操作符的返回值作为左值使用，因此它们的返回值也被设计为QString对象自己的引用，故 也可以连续操作。但由于复合赋值操作符的结合顺序是从右往左，要想先计算左边的操作数需要加括号。

（3）字符串比较运算符（< , <=, ==, >，>= 与 !=）：

①operator<(const QString&)：比较一个字符串是否小于另一个字符串。如果是，则返回true。

②operator<=(const QString&)：比较一个字符串是否小于等于另一个字符串。如果是，则返回true。

③operator==(const QString&)：比较两个字符串是否相等。如果相等，则返回true。

④operator>(const QString&)：比较一个字符串是否大于另一个字符串。如果是，则返回true。

⑤operator>= (const QString&)：比较一个字符串是否大于等于另一个字符串。如果是，则返回true

⑥operator!=(const QString&)：比较两个字符串是否不相等。如果不相等，则返回true。

⑦字符串比较运算符具有两种重载形式，一种是作为QString类的成员函数重载,它们可以使用this指针自 由的访问本类的数据成员，因此可以少一个QString类型参数，这时比较运算符既可以实现QString对象 与QString对象比较，也可以QString对象与普通的字符串比较,但它们的局限性是第一个操作数必须是 QString对象;另一种是作为QString类外的普通函数重载,这时比较运算符的第一个参数是普通字符串, 因此可以实现普通字符串与QString对象或普通字符串的比较。

7.判断是否为空字符串与NULL字符串：

（1）isEmpty()函数：如果字符串为空，也就是如果[length](http://www.kuqin.com/qtdocument/qstring.html#length)()==0返回真，即判断字符串是否为空。NULL字符 串总是空的。

（2）isNull()函数：如果字符串为NULL，也就是返回真，即判断字符串是否存在。NULL字符串也是空字符串。

9.字符串追加操作：

（1）append()函数：具有与+=操作符同样的功能，实现在一个字符串的末尾追加另一个字符串。

（2）prepend()函数：与append()函数功能类似, 也能够将另一个字符或字符串与QString对象连接起来,但是

接在原字符串的前面，即在原字符串的开头插入另一个字符串。

10.字符串插入与删除操作：

（1）insert()函数：在原字符串特定的位置插入另一个字符或字符串，其position参数指定了要插入的位置。

（2）remove()函数：在原字符串特定的位置删除另一个字符或字符串，其position参数指定了要删除的位置。

11.字符串替换操作：

replace()函数：用指定的字符串代替原字符串中的某些字符。replace()函数有两种重载形式，一种是给定起始位置，使用特定的字符或字符串从原字符串中替换指定长度的字符数；另一种是从原字符串中搜索指定的字符或字符串并开始替换，即定位替换和搜索替换两种形式。

11. 字符串大小写操作：

（1）toUpper()函数：将给定字符串转换为大写，toUpper()函数不会改变QString对象本身，而是将大写转换 后的结果作为返回值。

（2）toLower()函数：将给定字符串转换为小写，toLower()函数不会改变QString对象本身，而是将小写转换 后的结果作为返回值。

13.字符串填充与清空操作：

（1）fill()函数：使用指定个数的特定字符填充字符串。如果未给定填充字符的个数，默认为-1，则表示将

原字符串填充满，也就是说原来字符串里的所有字符都用等长度的给定字符来代替。

（2）clear()函数：可以清空一个QString对象的内容，使之成为空字符串。

12.字符串索引/定位操作：

at()函数：其功能与[]操作符相似，返回在索引index处的字符，如果index超过字符串的长度返回0。at()

函数返回的index位置处字符不能被修改。

14.空白字符操作：

（1）在实际编程过程中，很多时候去掉一个字符串两端的空白非常有用，如获取用户输入的账号时。但是值 得注意的是Qt中空白字符包括回车字符“\n”、换行字符“\r”、制表符“\t”和空格字符“ ”等，而 不仅仅是空格。

（2）trimmed()函数：移除字符串两端的空白字符。即：trimmed()函数可以截去QString对象头部和尾部的空 白字符

（3）simplified()函数：移除字符串两端的空白字符，并且使用单个空格字符“ ”代替字符串中出现的空白 字符。即：simplified()函数可以将字符串中的所有连续的空白字符替换成一个，并且把两端的空白字 符去除。

（4）trimmed()与simplified()函数的比较：simplified()函数功能比trimmed()更强大，它不仅能去掉 QString对象头尾的空白字符，还能将中间的连续多个空白字符全部替换成一个空格。

11.字符串截断操作：

（1）truncate()函数：在给定索引位置处截断字符串，也就是去掉指定位置后字符串中的所有内容。如果指

定的索引位置超出字符串的长度，则什么也不做。

（2）chop()函数：从字符串末尾删除n个字符。如果n大于或等于字符串长度，则结果为空字符串；如果n是负

的，它就说明一个字符也不截掉，什么也不发生。

（3）truncate()与chop()函数的区别：truncate()是从指定位置往后删除字符串中的内容，而chop()是从字

符串末尾往前删除指定个数的字符。

15.字符串统计操作：

（1）size()函数：返回QtString字符串中字符的个数，在QtString字符串中最后一个字符的位置为size()-1。

（2）length()函数：返回字符串的长度，零字符串和空字符串的长度都为0。

（3）size()与length()函数的作用完全相同，都可以得到QString对象中字符的个数，注意字符的个数并不一 定等于字节数。

（4）count()函数：统计QString字符串中包含某个特定字符或字符串的个数。

16.字符串截取操作：

（1）left()函数：截取字符串最左端*len*个字符组成的子字符串。如果*len*超过字符串的长度，则整个字符串 被返回。

（2）rigt()函数：与left()函数功能类似，都是接受一个起始位置参数，对字符串进行截取。不同之处在于， left()函数从左侧截取len个字符，而right()从右侧开始截取。

（3）mid()函数：截取字符串从起始位置index开始*len*个字符的子字符串。如果这个字符串为空或者*index*超 出范围则返回零。如果*index+len*超过这个字符串的长度，返回从*index*开始的整个字符串。即mid()函数 接受两个参数，第一个是起始位置，第二个是取串的长度。如果省略第二个参数，则会从起始位置截取 到末尾。

（4）left()函数从左边截取字符串；right()函数从右边截取字符串；mid()函数从中间截取字符串。注意三 个函数并不修改QString对象自身，而是返回一个临时对象以供使用。

18.字符串查找操作：

（1）startsWith()函数：判断一个字符串是否以某个特定的字符串开头。此函数具有两个参数，第一个参数 指定了一个字符串，第二个参数指定是否大小写敏感，默认情况下是大小写敏感的。

（2）endsWith()函数：判断一个字符串是否以某个特定的字符串结尾。startsWith()和endsWith()函数可以

检测字符串是不是以某个特定的子字符串开始或结尾

（3）contains()函数：判断一个子字符串是否在当前字符串中出现过，即：判断QString对象是否包含指定的

字符或字符串。

（4）indexOf()函数：返回QString对象中某个特定字符或字符串出现的位置，查找的方向是从前往后，如果

没有找到则返回 -1。其from参数表示查找的起点，它可以为负数，-i表示查找至倒数第i个字符。

（5）lastIndexOf()函数：与indexOf()函数功能类似，也能字符串中某个特定字符或字符串出现的位置，但 其查找方向是从后往前查找。

（6）indexOf()和lastIndexOf()这两个方法都是从一个字符串中搜索给定的字符串，然后返回子字符串的位

置，如果没有子字符串的位置，则返回-1。它们的区别在于，indexOf()方法从字符串的开头向后搜索字

符串，而lastIndexOf()方法是从字符串的末尾向前搜索子字符串。两种方法都可以接受可选的第二个参

数，表示从字符串中的哪个位置开始搜索。换句话说，indexOf()会从该参数指定的位置向后搜索，忽略

该位置之前的所有字符；而lastIndexOf()则会从指定的位置向前搜索，忽略该位置之后的所有字符。

19.字符串格式化组合操作：

（1）sprintf()函数：从一个格式化字符串*cformat*和一个任意的参数列表安全地构建一个格式化的字符串。

根据不同的格式化字符串*cformat*，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值， 替换format参数中指定的每个%标签。参数的个数应与%标签的个数相同。sprintf()函数支持的格式定义 符和C++库中的函数sprintf()定义的一样

（2）C语言中有sprintf()函数把格式化的数据写入某个字符串，QString则提供了一个sprintf()成员函数实现了 相同的功能。但Qt不建议使用sprintf()方法来格式化输出字符串，因为Qt使用Unicode字符串，sprintf() 会导致类型匹配错误。QString类提供了另外一个功能更强大、类型更安全、使用更方便的arg()方法以 代替sprintf()实现格式化输出。

（3）arg()函数：Qt还提供了另一种方便的字符串格式化组合方式，使用QString::arg()函数，此函数的重载 可以处理很多的数据类型。此外一些重载具有额外的参数对字段的宽度、数字基数或者浮点数精度进行 控制。通常相对于函数QString::sprintf()，函数QString::arg()是一个比较好的解决方案，因为它 类型安全，完全支持Unicode，并且允许改变"%n"参数的顺序

（3）arg( )函数中的参数在进行替换时，只会按照%占位符中数字从小到大的顺序进行替换，只有比当前数字 小的所有数字都替换完成了，才会替换当前数字，否则会将当前数字和%符号按字符串处理。%占位符中 的数字必须为自然数。

（4）arg()函数的结构：第一个参数是要输出的值，包括字符串和各种基本数据类型；第二个参数是位段宽度； 第三个参数一般是用2、8、10、16表示进制；最后一个参数是填充字符。因为arg()操作返回的是QString 对象，所以可以连续使用。

（5）格式化输出字符串的基本形式：

QString(const char\* formt).arg(…) … tr(const char\* formt).arg(…) …。其中format表示的是 用1%、2% … n%表示的占位符。

14.字符串比较操作：

（1）localeAwareCompare()：可以用于比较前后两个字符串。如果前面字符串小于后面字符串，则返回负整 数值；如果等于则返回0；如果大于则返回正整数值。该函数的比较是基于本地（locale）字符集的，而 且是平台相关的。

（2）compare()：该函数可以指定是否进行大小写的比较，而大小写的比较是完全基于字符的Unicode编码值 的，而且是非常快的，返回值类似于localeAwareCompare()函数。如果前面字符串小于后面字符串，则 返回负整数值；如果等于则返回0；如果大于则返回正整数值。

（3）字符串的比较可以直接使用比较运算符，也可以使用QString类的静态成员函数compare()或

localeAwareCompare()比较两个字符串。但一般使用QString类的比较操作符更为直观。

20.字符串转换操作：

（1）QString类提供了丰富的转换函数，可以将一个字符串转换为数值类型或者其他的字符编码集。

（2）数值转换为字符串：

①number()函数：可以把各种基本数据类型的数字转换成字符串。number()函数它会返回一个新的临时对

象以供使用，而不会改变QString对象自身。number()是静态成员函数，因此与某个具体的对象无关，可

以直接通过类名调用。

②setNum()函数：非static的setNum()成员函数和static的number()函数功能相同，都可以将各种基本数

值类型转换为QString字符串，但setNum()函数返回值是QString&类型，它会改变QString对象本身的值。

③当使用number()与setNum()函数将浮点数转换为字符串时，都需要指定format与precision参数。其中 format参数用于指定转化格式，precision参数用于指定转换结果的精度，既有效数组的个数。

（2）字符串转换为数值：

①toInt()函数：将字符串转换为整型数值。函数QString::toInt()有两个参数，第一个参数是一个bool

类型的指针，用于返回转换的状态，当转换成功是设置为true，否则设置为false；第二个参数指定了

转换的基数。当基数设置为0时，将会使用C语言的转换方式，即如果字符串以“0x”开头，则基数为16；

如果字符串以“0”开头，则基数为8；其他情况下一律是10。

②与toInt()类似的函数还有toDouble()、toFloat()、toLong()、toLongLong()等。QString提供了一系 列的to函数用于将字符串转换成其他基本类型，它们都接受一个bool指针作为参数，函数结束之后将根 据是否转换成功设置为true或者false。这些函数都不会改变QString对象自身。

（3）字符串转换为其他字符编码：

①QString提供的字符编码集的转换函数将会返回一个const char\*类型版本的QByteArray，即调用构造函 数QByteArray(const char\*)创建的QByteArray对象。QByteArray类具有一个字节数组，它既可以存储 原始字节（raw bytes），也可以存储传统的以“\0”结尾的8位的字符串。在Qt中使用QByteArray 比使用const char\*更方便，且QByteArray也支持隐式共享。

②QString类型无法直接转换成char\*, 需要先转换成QByteArray, 然后由QByteArray再转char\*。

③toAscii()函数：返回一个ASCII编码的8位字符串。

④toLatin1()函数：返回一个Latin-1（ISO8859-1）编码的8位字符串。

⑤toUtf8()函数： 返回一个UTF-8编码的8位字符串（UTF-8是ASCII码的超集，它支持整个Unicode字符集）。

⑥toLocal8Bit()函数**：**返回一个系统本地（locale）编码的8位字符串。

**(二)容器类:**

1.Qt容器类：Qt提供了一组通用的基于模板的容器类，可以用这些类存储指定类型的项。对比C++的STL标准模

板库中的容器类，Qt的这些容器更轻量、更安全并且更容易使用。此外Qt容器类是隐式共享的、可重入的、

并且在速度、内存消耗和内联代码等方面进行了优化，当它们作为只读的容器时是线程安全的，所有线程都

可以使用它们。

2. QT不仅支持C++的STL模板库，同时自己也定义了一套容器类和与之操作的算法类，使用QT定义的这一套库，

可以使在各个平台的表现相同。QT的这些容器被设计为更轻便，更安全和更容易使用，内联代码扩展少从而

可以产生更小的可执行文件。容器类是隐含共享的、可重入的和线程安全的。此外Qt还为容器遍历提供了

foreach关键字和Java/STL两种风格的迭代器，更加方便Qt容器中元素遍历。

3.Qt容器中存储的数据类型要求：

（1）存储在Qt容器中的数据必须是可赋值的数据类型，也会是说这些数据类型必须提供一个不需要带参数的 默认构造函数、一个拷贝构造函数和一个赋值操作运算符。这样的数据类型包含了通常使用的大多数数 据类型，包括基本数据类型（如int和double等）和Qt的一些数据类型（如QString、QDate和QTime等）。 不过Qt中的QObject及其子类（如QWidget和Qdialog等）是不能够存储在容器中的，因为它们没有复制构 造函数和赋值操作运算符，一个可代替的方案是存储QObject及其子类的指针。

（2）对于有些容器对它们能够存储的数据类型可能还有特殊的要求(如QMap<Key,T>的Key类型必须提供<<运 算符)，这些特定的要求在容器类的文档中都有详细说明，一般如果某个要求没被满足，编译器就会报错。

11.如果没有为类提供拷贝构函数或赋值运算符，C++会提供逐成员拷贝或赋值的默认实现。同样如果没有显式

的声明任何构造函数，C++将提供一个默认的构造函数，它会使用成员变量所对应数据类型的默认值对各个

成员进行初始化。

4.Qt的容器类是可以嵌套的。例如：QHash<QString, QList<double> >、其中QHash的键类型是QString，它的

值类型是QList<double>。注意，在最后两个“>”符号之间要保留一个空格，否则C++编译器会将两个“>”

解释为一个“>>”符号，导致无法通过编译器编译。

6.foreach语句：

（1）Qt提供了foreach关键字使我们方便地遍历容器中的项。在标准C++中并没有foreach关键字。但是在QT 中可以使用这一个关键字，其主要原因是QT自己增加了这一个关键字，就像slots和signals、emit等一 样。增加的foreach关键字在编译时会进行预处理。

（2）foreach语句格式：foreach（variable ，container），其中variable是容器container中的一个项，相 当于variable=container.item，遍历会从头遍历到尾。foreach语句用于循环访问集合以获取所需信息， 但不应用于更改集合内容以避免产生不可预知的副作用。

10. Qt容器还提供了<<和>>运算符，以使它们可以很方便的使用DataStream类来读取或写入数据，这也意味着

存储在容器中的数据类型必须支持这两种操作。QList类重载了“+=”和“<<”操作运算符，使用它们可 以很方便的将一个数据项存储到QList列表中。

12.算法复杂性的表示: O(1)—常数时间复杂度；O(logn)—对数时间复杂度； O(nlogn)—线性对数时间复杂

度；O(n)—线性时间复杂度；O(n²).—平方时间复杂度

7.Qt容器分类：

（1）Qt中的容器和STL中的类似，也分为顺序容器和关联容器。

（2）经常使用的Qt容器类有QList、QLinkList和QVector等，在开发一个较高性能需求的应用程序时，程序员 会比较关注这些容器类的运行效率。不同类型的容器对其中存储数据的增、删、改、查等操作速度各不 相同，可以根据业务需求和每一种容器的特点选择最佳的容器。

（3）顺序容器：QList、QLinkedList、QVector、QStack、QQueue。对大部分应用程序来说，QList都是一 个很好的选择。尽管它在底层是用一个数组array-list实现的，但它为我们提供了非常快速的首尾添加 操作。如果你确实需要一个链表linked-list，可以使用QLinkedList；如果你想确保你的元素占用连续 的内存空间可以使用QVector。而QStack栈和QQueue队列分别提供了后进先出（LIFO）和先进先出（FIFO） 的机制。

（4）关联容器：QMap，QMultiMap，QHash，QMultiHash，QSet。这些容器中存储的都是key-value对。其中， "Multi"容器又支持一个键key可以关联多个值value。"Hash"容器通过使用一个hash函数而不是二分搜索 提供了更快速的查找操作。

9. QList、QLinkedList和QVector容器的时间复杂度比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 容 器 类 | 查 找 | 插 入 | 头 部 添 加 | 尾 部 添 加 |
| QList | O(1) | O(*n*) | Amort.O(1) | Amort.O(1) |
| QLinkedList | O(*n*) | O(1) | O(1) | O(1) |
| QVector | O(1) | O(*n*) | O(*n*) | Amort.O(1) |

其中Amort.O(1)表示，如果仅完成一次操作，可能会有O(*n*)行为；但是如果完成多次操作（如*n*次），平均

结果将会是O(1)。

6.QList类：

（1）QList<T>是迄今为止最常用的容器类，它存储给定数据类型T的一列数值。虽然它是以数组列表的形式实 现的，但是在其前或后添加数据非常快，QList可以使用下标索引的方式对数据项进行访问。继承自QList 类的子类有QItemSelection、QQueue、QSignalSpy及QStringList和QTestEventList。

（2）QList不仅提供了可以在列表中进行追加的QList::append()和Qlist::prepend()函数，还提供了在列表

中间完成插入操作的函数QList::insert()。相对于任何其他的Qt容器类，为了使可执行代码尽可能少，

QList被高度优化。

（3）QList<T>维护了一个指针数组，该数组存储的指针指向QList<T>存储的列表项的内容。因此QList<T>提 供了基于下标的快速访问。对于不同的数据类型，QList<T>采取不同的存储策略。

（4）QList<T>的两种存储策略：

①如果T是一个指针类型或指针大小的基本类型（即该基本类型占有的字节数和指针类型占有的字节数相 同），QList<T>会将数值直接存储在它的数组中。

②如果QList<T>存储对象的指针，则该指针指向实际存储的对象。

（5）QList的常用函数：

| **函数** | **用途** | **函数** | **用途** | **函数** | **用途** |
| --- | --- | --- | --- | --- | --- |
| at()/value() | 定位列表项 | insert() | 插入数据项 | move() | 移动数据项 |
| indexof() | 检索列表项 | removeAt() | 删除数据项 | swap() | 交换数据项 |
| append() | 列表头追加 | removeFirst() | 删除列表中第一项 | replace() | 替换数据项 |
| prepend() | 列表尾追加 | removeLast() | 删除列表中最后项 | contains() | 判断是否包含指定值 |
| isEmpty() | 判断列表是否为空 | size() | 返回列表项个数 | count() | 统计列表中某值出现次数 |

8. QVector类：

（1）QVector<T>在相邻的内存中存储给定数据类型T的一组数值。在一个QVector的前部或者中间位置进行插 入或删除操作的速度是很慢的，这是因为这样的操作将导致内存中的大量数据被移动，由QVector存储数 据的方式所决定的。

（2）QVector<T>既可以使用下标访问数据项，也可以使用迭代器访问数据项。继承自QVector类的子类有 QPolygon、QPolygonF和QStack。

（3）QVector<T>提供了动态数组的功能，可以使用下标索引访问数据。QVector的函数接口与QList几乎完全 相同，QVector<T>的性能比QList<T>更高，因为QVector的数据项是连续存储的。

7.QLinkedList类：

（1）QLinkedList<T>是一个链式列表，它以非连续的内存块保存数据。由于QLinkedList是链式存储结构，因 此它比较适合随机插入或删除数据项。它提供了常量时间的插入和删除操作，却不能进行快速的随机存 取操作。不支持[]定位操作符，遍历它的元素是通过迭代器完成的。除了不提供基于下标索引的数据项 访问外，QLinkedList的其他接口函数与QList基本相同

（2）QLinkedList<T>不能使用下标，只能使用迭代器访问它的数据项。与QList相比，当对一个很大的列表进 行插入或删除操作时，QLinkedList具有更高的效率。QLinkedList的插入和删除数据项操作时间相等。

8.QMap类：

（1）QMap<Key,T>提供了一个从类型为Key的键到类型为T的值的映射。

（2）通常QMap存储的数据形式是一个键对应一个值，并且按照键Key的顺序存储数据。为了能够支持一键多值 的情况，QMap提供了QMap<Key,T>::insertMulti()和QMap<Key,T>::values()函数。存储一键多值的数据 时，也可以使用QMultiMap<Key,T>容器，它继承自Qmap。

9. QHash类：

（1）QHash<Key,T>具有与QMap几乎完全相同的API。QHash维护着一张哈希表（Hash Table），哈希表的大小 与QHash的数据项的数目相适应。

（2）QHash以任意的顺序组织它的数据。当存储数据的顺序无关紧要时，建议使用QHash作为存放数据的容器。

QHash也可以存储一键多值形式的数据，它的子类QMultiHash<Key,T>实现了一键多值的语义。

10. QMultiMap类：

（1）QMultiMap是QMap的子类，是用于处理多值映射的便利类。多值映射就是一个键可以对应多个值。QMap 正 常情况下不允许多值映射，除非使用QMap::insertMulti()添加键值对。

（2）QMultiMap是QMap的子类，所以QMap的大多数函数在QMultiMap都是可用的，但是有几个特殊的函数，

QMultiMap::insert()等效于QMap::insertMulti(),QMultiMap::replace()等效于QMap::insert()。

8. QMap类和QHash类的区别：

（1）QHash具有比QMap更快的查找速度。

（2）QHash以任意的顺序存储数据项，而QMap总是按照键Key的顺序存储数据。

（3）QHash的键类型Key必须提供operator==()和一个全局的qHash(Key)函数，而QMap的键类型Key必须提供 operator<()函数

（4）QMap类和QHash类的时间复杂度比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 容 器 类 | 键 查 找 | | 插 入 | |
| 平 均 | 最 坏 | 平 均 | 最 坏 |
| QMap | O(log *n*) | O(log *n*) | O(log *n*) | O(log *n*) |
| QHash | Amort.O(1) | O(*n*) | Amort.O(1) | O(*n*) |

其中“Amort.O(1)”表示，如果仅完成一次操作，则可能会有O(*n*)行为；如果完成多次操作（如*n*次），

则平均结果将是O(1)。

5.Qt的容器类为遍历其中的内容提供了两种方法:

（1）迭代器为访问容器中的元素提供了一套统一的方式。Qt容器类有两种类型的迭代器：Java风格的迭代 器和STL风格的迭代器。但是当容器中的数据被修改后或由于调用了non-const成员函数导致其脱离了隐 式共享，那么这两种迭代器都会失效。

（2）Java风格的迭代器（Java-style iterators）,Java风格的迭代器更易于使用，并且提供了更高级的功能。

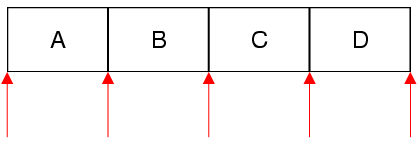
（3）STL风格的迭代器（STL-style iterators），STL风格的迭代器会更高效，并且可以和Qt和STL的通用算

法结合使用。

11.Java风格迭代器：

（1）Java风格的迭代器在Qt4中被引入，成为Qt应用程序的标准组件，它们的API以Java的迭代器类为模型， 其具体使用方法在每个迭代器类的文档中有详细说明。Java风格的迭代器同STL风格的迭代器相比，使用 起来更简单方便，不过这也是以轻微的性能损耗为代价的。对于每一个容器类，Qt都提供了两种类型的 Java风格迭代器数据类型，即只读访问和读写访问。

（2）Java风格迭代器的迭代点位于列表项的中间，而不是直接指向某个列表项。因此它的迭代点或者在第一 个列表项的前面，或者在两个列表项之间，或者在最后一个列表项之后。



|  |  |  |
| --- | --- | --- |
| 容 器 类 | 只读迭代器类 | 读写迭代器类 |
| QList<T>,QQueue<T> | QListIterator<T> | QMutableListIterator<T> |
| QLinkedList<T> | QLinkedListIterator<T> | QMutableLinkedListIterator<T> |
| QVector<T>,QStack<T> | QVectorIterator<T> | QMutableVectorIterator<T> |

图2.2 Java风格迭代器位置

（3）顺序容器常用的Java风格迭代器：

12. QListIterator<T>迭代器的用法：

（1）#include <QList>:引入QList的头文件。注意<QDebug>中也包含了QList的头文件

（2）QListIterator<T> iter(T)：以该list为参数初始化一个QListIterator对象iter。此时迭代点处在第一 个列表项List[0]的前面,注意并不是指向该列表项。

（3）while(iter.hasNext())：调用QListIterator<T>::hasNext()函数检查当前迭代点之后是否有列表项。 如果有则调用QListIterator<T>::next()函数进行遍历。next()函数将会跳过下一个列表项（即迭代点 将位于第一个列表项和第二个列表项之间），并返回它跳过的内容。

（5）QListIterator<T>迭代器对列表进行向后遍历的函数：

①QListIterator<T>::hasNext()：检查当前迭代点之后是否具有列表项。

②QListIterator<T>::next()：返回后一个列表项的内容并将迭代点移动到后一个列表项之后。

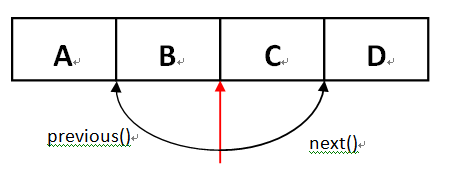


图2.3 previous()与next()函数的区别

（4）QListIterator<T>迭代器对列表进行向前遍历的函数：

①QListIterator<T>::toBack()：将迭代点移动到最后一个列表项的后面。

②QListIterator<T>::hasPrevious()：检查当前迭代点之前是否具有列表项。

③QListIterator<T>::previous()：返回前一个列表项的内容并将迭代点移动到前一个列表项之前。

④QListIterator<T>迭代器向后遍历和向前遍历的代码类似，除了我们在一开始调用了toBack）函数将迭 代器移到最后一个元素的后面。

（5）QListIterator<T>提供的其他接口：

①toFront()：移动迭代点到列表的前端（第一个列表项的前面）。

②peekNext()：返回下一个列表项，但不移动迭代点。

③peekPrevious()：返回前一个列表项，但不移动迭代点。

④findNext()：从当前迭代点开始向后查找指定的列表项，如果找到，则返回true，此时迭代点位于匹配 列表项的后面；如果没有找到，则返回false，此时迭代点位于列表的后端（最后一个列表项的后面）。

⑤findPrevious()：与findNext()类似，不同的是它的方向是向前的，查找操作完成后的迭代点在匹配项 的前面或整个列表的前端。

14.QListIterator类的常用API及其作用：

|  |  |
| --- | --- |
| toFront() | 移动迭代器到第一个元素之前 |
| toBack() | 移动迭代器到最后一个元素之后 |
| hasNext() | 如果迭代器还未遍历到列表的最后，返回true |
| next() | 返回下一个元素，并将迭代器向前移动一个位置。 |
| peekNext() | 返回下一个元素，不移动迭代器。 |
| hasPrevious() | 如果迭代器还未遍历到列表的前端，返回true。 |
| previous() | 返回前一个元素，并将迭代器向后移动一个位置。 |
| peekPrevious() | 返回前一个元素，不移动迭代器。 |

13.QListIterator<T>是只读迭代器，它不能完成该迭代器在遍历的过程中对列表项的插入和删除操作。读写

迭代器QMutableListIterator<T>除了提供基本的遍历操作（与QListIterator的操作相同）外，还提供了

insert()插入操作函数、remove()删除操作函数和setValue()修改数据函数等，它们不会使迭代器失效，

所以是安全的。

13. QMutableListIterator<T>迭代器的用法：

（3）while(iter.hasNext())：调用QMutableListIterator <T>::hasNext()函数检查当前迭代点之后是否有 列表项。

（1）for(iter.toFront();iter.hasNext();)：将迭代器的迭代点移动到列表的前端，完成对列表的正序遍历。

（2)for(iter.toBack();iter.hasPrevious();)：移动迭代器的迭代点到列表的后端，对列表进行倒序遍历。

（3）QMutableListIterator<T>::next()：返回后一个列表项的非const引用并将迭代点移动到后一个列表项

之后。可以直接对next()函数的返回值进行赋值以修改列表项中的内容，而一定使用setValue()函数。

（4）QMutableListIterator<T>::previous()：返回前一个列表项的非const引用并将迭代点移动到前一个列 表项之前。可以直接对previous()函数的返回值进行赋值以修改列表项中的内容。

（5）QMutableListIterator<T>::setValue()：修改遍历函数next()、previous()、findNext()和findPrevious() 跳过的列表项的值，但不会移动迭代点的位置。如果是正序遍历，跳过的列表项在当前迭代器之前；如果 是倒序遍历，跳过的列表项在当前迭代器之后。对于findNext()和findPrevious()有些特殊：当findNext() 或findPrevious()查找到列表项的时候，setValue()将会修改匹配的列表项；如果没有找到，则对 setValue()的调用将不会进行任何修改。

11.STL风格迭代器：

（1）自从Qt2发布就可以使用STL风格的迭代器了，它们适用于Qt和STL的泛型算法，并且对速度作了优化。对 于每一个容器类，Qt都提供了两种类型的STL风格迭代器数据类型：一种提供只读访问；另一种提供读写 访问。由于只读类型的迭代器的运行速度要比读写迭代器的运行速度快，所以应尽可能地使用只读类型的 迭代器。

（2）STL风格迭代器的API是建立在指针操作的基础上的。可以使用“++”操作运算符移动迭代器到下一个item 项，而“\*”操作运算符返回迭代器指向的项内容。事实上对于QVector<T>和QStack<T>，它们的数据项在 内存中存储在相邻的位置，迭代器类型正是T \*，const迭代器类型正是const T \*。

（3）不同于Java风格的迭代器，STL风格迭代器的迭代点直接指向列表项。begin()函数返回指向容器中第一 项的迭代器。end()函数返回指向容器中最后一项后面一个位置的迭代器，end()标记着一个无效的位置， 不可以被解引用，它经常被用来在一个循环中做为结束条件或break条件。如果list是空的，begin()等 于end()，所以我们永远不会执行循环。

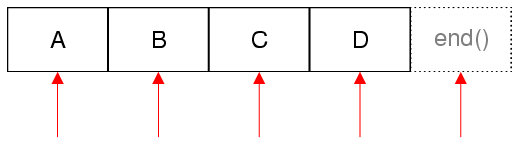


图2.4 STL风格迭代器的位置

（3）顺序容器常用的STL风格迭代器：

|  |  |  |
| --- | --- | --- |
| 容 器 类 | 只读迭代器类 | 读写迭代器类 |
| QList<T>,QQueue<T> | QList<T>::const\_iterator | QList<T>::iterator |
| QLinkedList<T> | QLinkedList<T>::const\_iterator | QLinkedList<T>::iterator |
| QVector<T>,QStack<T> | QVector<T>::const\_iterator | QVector<T>::iterator |

14. QList<T>::iterator迭代器的用法：

（1）for(iter=list.begin();iter!=list.end();++iter)：将迭代器的迭代点移动到列表的第一项，完成对 列表的正序遍历。QList<T>::begin()函数返回指向第一个列表项的迭代器；QList<T>::end()函数返回 一个容器最后列表项之后的虚拟列表项，为标记无效位置的迭代器，用于判断是否到达容器的底部。

（2）for(iter= list.end()-1;iter!= list.begin();--iter)：将迭代器的迭代点移动到列表的最后一项， 完成对列表的倒序遍历。在倒序遍历时，需要注意将迭代器移动到列表最后一项的操作是end()-1。

（3）list.insert(list.end(),value)：第一个参数是QList<T>::iterator类型，表示在该列表项之前插入一 个新的列表项（使用QList<T>::end()函数返回的迭代器，表示在列表的最后插入一个列表项）；第二个 参数指定了需要插入的值。

15. STL风格迭代器支持的运算符：

| **表达式** | **用途** | **表达式** | **用途** |
| --- | --- | --- | --- |
| \*iter | 返回当前列表项 | iter += n | 迭代器向前移动n项 |
| ++iter | 将迭代器指向下一项 | iter -= n | 将迭代器你向后移动n项 |
| --iter | 将迭代器指向上一项 | iteri - iterj | 返回迭代器i和j之间项的数目 |

（1）++和--运算符可以使用前缀(++i,--i)和后缀(i++,i--)的形式，前缀的版本修改迭代器并返回修改后迭 代器的引用，后缀版本在修改之前先复制迭代器，然后返回它的拷贝。在不需要考虑返回值的情况下，推 荐使用前缀运算符(++i,--i)，因为它们稍微快一点。

（2）对于非const的迭代器类型，一元运算符\*可以被用在赋值运算符的左边。

16.关联容器常用的Java风格迭代器：

|  |  |  |
| --- | --- | --- |
| 容 器 类 | 只读迭代器类 | 读写迭代器类 |
| QMap<Key,T>,QMultiMap<Key,T> | QMapIterator<Key,T> | QMutableMapIterator<Key,T> |
| QHash<Key,T>,QMultiHash<Key,T> | QHashIterator<Key,T> | QMutableHashIterator<Key,T> |

17. QMapIterator<Key,T>迭代器的用法：

（1）while(iter.hasNext())：完成对QMap的向后遍历。注意在获取QMap的键和值时，调用的函数是同的。 因为在获取键的时候，不需要使迭代点移动到下一个位置，所以调用QMapIterator<T,T>::key()就可以 了；而在获取值的时候调用需要先调用QMapIterator<T,T>::next()将迭代点移动到下一个位置，再调 用QMapIterator<T,T>::value()得到值。

（2）QMapIterator<T,T>::findNext()和findPrevious（）：根据值查找某个<键,值>对。Java风格的迭代器 并没有提供查找键的 函数，只提供了查找值的函数。

（3）类似于QListIterator，QMapIterator也提供了toFront()，toBack(），peekPrevious()，hasPrevious()， peekNext()，next()，hasNext()。至于具体的key和value，我们可以调用key()和value()函数，从

next()，peekNext()，previous()或者peekPrevious()返回的对象中提取，即QMapIterator提供了相 应的key()和value()函数，可以直接作用于迭代器本身，返回上次跳过的元素的键和值。

18. 关联容器常用的STL风格迭代器：

|  |  |  |
| --- | --- | --- |
| 容 器 类 | 只读迭代器类 | 读写迭代器类 |
| QMap<Key,T>,QMultiMap<Key,T> | QMap<Key,T>::const\_iterator | QMap<Key,T>::iterator |
| QHash<Key,T>,QMultiHash<Key,T> | QHash<Key,T>::const\_iterator | QHash<Key,T>::iterator |

19. QMap<Key,T>::iterator迭代器的用法：

（1）for(iter=map.begin();iter!=map.end();++iter)：将迭代器的迭代点移动到QMap的第一个<键,值>对， 完成对列表的正序遍历。

（2）for(iter= list.end()-1;iter!= list.begin();--iter)：将迭代器的迭代点移动到QMap的最后一个

<键,值>对，完成对列表的倒序遍历。在倒序遍历时，需要注意将迭代器移动到列表最后一项的操作是 end()-1。

（3）对于QMap和QHash，“\*”运算符返回<键,值>对中的值，如果你想要获得键，只需在迭代器上调用key() 函数。为了对称，迭代器还提供了value()函数来获得值。

20.QVariant类：

（1）QVariant类似于C++中的union联合体数据类型，它不仅能保存很多Qt类型的值，包括QString、QBrush、 QFont、QPen、QRect，QColor和QSize等,也能够存放Qt的容器类型的值。Qt的很多功能都是建立在 QVariant基础上的，如Qt的对象属性及数据库功能等。

（2）由于QVariant是QCore模块的类，所以它没有为QtGui模块中的数据类型提供转换函数，因此需要使用 QVariant::value或者QVariantValue()模块函数。

（3）QVariant::type()函数返回存储在QVariant变量中的值的数据类型。QVariant::StringList是Qt定义 的一个QVariant::type枚举类型的变量。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 变 量 | 对应的类型 | 变 量 | 对应的类型 | 变 量 | 对应的类型 |
| QVariant::Invalid | 无效类型 | QVariant::Time | QTime | QVariant::DateTime | QDateTime |
| QVariant::Region | QRegion | QVariant::Line | QLine | QVariant::Double | double |
| QVariant::Bitmap | QBitmap | QVariant::Palette | QPalette | QVariant::Font | QFont |
| QVariant::Bool | bool | QVariant::List | QList | QVariant::Icon | QIcon |
| QVariant::Brush | QBrush | QVariant::SizePolicy | QSizePolicy | QVariant::Pixmap | QPixmap |
| QVariant::Size | QSize | QVariant::String | QString | QVariant::Rect | QRect |
| QVariant::Char | QChar | QVariant::Map | QMap | QVariant::Image | QImage |
| QVariant::Color | QColor | QVariant::StringList | QStringList | QVariant::UserType | 用户自定义类型 |
| QVariant::Cursor | QCursor | QVariant::Point | QPoint |  |  |
| QVariant::Date | QDate | QVariant::Pen | QPen |  |  |

21.QVariant类中的枚举类型变量：

|  |  |
| --- | --- |
| **类** | **概述** |
| QList<T> | 这是目前使用最频繁的容器类，它存储了指定类型(T)的一串值，可以通过索引来获得。本质上QList是用数组实现的，从而保证基于索引的访问非常快。可以通过QList::append()和QList::prepend()在两端添加项，或者通过QList::insert()在中间插入项。QStringList是从QList<QString>得到的。 |
| QLinkedList<T> | 类似于QList，但它使用迭代器而不是整数索引来获得项。当在一个很大的list中间插入项时，它提供了更好的性能，并且它有更好的迭代器机制。指向QLinkedList中某个元素的迭代器，只要该元素存在就会一直保持有效，而指向QList中某元素的迭代器，在向QList进行任意插入或删除操作时都会导致该迭代器失效。 |
| QVector<T> | 在内存中相邻的位置存储一组值，在开头或中间插入会非常慢，因为它会导致内存中很多项移动一个位置。 |
| QStack<T> | QVector的一个子类，提供类似于堆栈的后进先出（LIFO）机制。在当前的QVector中增加了几个方法：push()、pos()、pop()。 |
| QQueue<T> | QList的一个子类，提供了类似于队列的先进先出（FIFO）机制，在当前的QList中增加了几个方法：head()、enqueue()、dequeue()。 |
| QSet<T> | QSet是单值的数学集合（即提供了不允许有重复值的集合），基于散列表的集合模板类，它存储数据的顺序是不定的，查找值的速度非常快。QSet<T>内部就是用QHash实现的。 |
| QMap<Key, T> | 提供了一个字典（关联数组）将类型Key的键对应类型T的值。通常一个键对应一个值，QMap以Key的顺序存储数据，如果顺序不重要QHash是一个更快的选择。 |
| QMultiMap<Key, T> | QMultiMap是QMap的子类，用于处理多值映射的便利类，一个键对应多个值。QMultiMap是 QMap的子类，所以QMap的大多数函数在QMultiMap都是可用的，但是有几个特殊的函数，QMultiMap::insert()等效于QMap::insertMulti()；QMultiMap::replace()等效于QMap::insert()。 |
| QHash<Key, T> | QHash是基于散列表来实现字典功能的模板类，QHash<Key，T>存储的键值对具有非常快的查找速度。与QMap几乎有着相同的接口，但查找起来更快。在QMap中遍历时，数据项是按照键排序的，而QHash的数据项是任意顺序的。QMap 的键必须提供“<”运算符，QHash的键必须提供“==”运算符和一个名称为qHash() 全局散列函数。 |
| QMultiHash<Key, T> | QMultiHash是QHash的子类，用于处理多值映射的便利类，其用法与QMultiMap类似。 |

22.Qt容器总结：

**(三)算法及正则表达式:**