Qt5 学习备忘录

Kreogist Team

2013-5-31

Contents

Chapter 1

关于本文

2013年夏，我们尝试使用Qt5去开发一个IDE，一个简单、好用，针对初学者/OIer/ACMer等等人群的IDE。当然，我们也是我们项目主要的目标用户群中的一员。这个人群常写的代码都有这样一个特点：单文件，命令行（或文件）输入输出，代码行数一般不超过1000行。对于他们而言，建立工程，写makefile等等都是不需要的，因为他们的代码只为学习算法或数据结构。

我们希望我们的作品可以提高我们写这类学习性代码的效率，同时，我们也希望在实现项目的过程中积累经验，学习那些我们所感兴趣的技术。学习，是这个项目的核心：源于学习，同时应用于学习。而本文则记录了我们按照怎样的步骤去学，又学到了什么，是我们学习过程的一个体现。我们希望当我们把知识又还给书本和网络的时候，可以从这里找到，我们都学到过什么。同时，我们也希望，这些文字对其他正在学习Qt5的人能有所帮助。

本文包含了原创内容和文档翻译，以及一些对别人文章的引用、整理，这些内容往往是混在一起的。这是我们的学习Qt5的过程的备忘录（说是学习笔记也可以），所以，并未严格区分这些内容。当然，必要的部分会按原作者的要求声明原文地址。

- 本文的目标读者

  所有希望了解Qt或者希望开发跨平台程序的学生、程序员等等；

- 阅读本文需要什么技能？

  我们假定你掌握了足够的关于C++知识和计算机基础知识。

Chapter 2

Qt学习备忘录1：Qt介绍

随着时代的发展，你或许越来越多地发现，很多软件的身影都不只出现在装有Windows XP/Vista/7的机器上。它们开始出现在Mac上面，出现在Linux上面，甚至出现在朋友的手机里。软件的跨平台已成为时代的大趋势。你有没有过让你的程序也同时部署在多个平台上的愿望呢？如果有，Qt可以帮你轻松实现它。

什么是Qt?

Qt是一个基于C++或QML的功能强大的跨平台应用程序开发工具和UI设计框架，且包含了很多用于开发应用程序的工具，以便于开发桌面、嵌入式及移动应用程序。

Qt包含了Qt Freamwork和Qt Creator IDE。Qt Framework提供了C++和CSS/Javascript的API，这些API基本都是跨平台的，你可以在不改变代码的情况下在多平台编译。你可以轻松地创建跨平台应用程序。Qt Creator IDE是一个功能强大的集成开发环境，集成了界面设计器、多种版本控制工具插件等工具，支持代码高亮、自动补齐等等的功能，使开发Qt应用程序变成一种享受。

为什么使用Qt?

最初的跨平台GUI程序都是把核心的平台无关的代码抽象出来，然后为每个平台单独开发对应的GUI。为每个平台分别维护代码图形化界面是个很烦人的工作。请设想这样一个场景：你刚刚在windows上实现一个复杂而繁琐的功能，然后为了保证各平台的一致，又得在Linux上再实现一遍，最后在Mac上又来一遍。万一你不行地发现这个功能的设计有些问题，又得分别修改三个平台上对应的代码。而且，这三个平台上开发应用程序的感觉有很大不同，你不但需要同时学会并适应在这三个平台上开发应用程序，还得及时切换。是不是感觉很累很麻烦呢？能不能不把宝贵的时间浪费在处理各个不同平台的差异上呢？

Qt开发的初衷正是希望为开发者开发跨平台程序提供便利。它提供了大量功能强大的类和API，为开发跨平台应用程序提供了极大的便利，就如它的标语所说的“Code less, Create more, Deploy everywhere”。你可以在一个你喜欢的平台上开发，然后轻松的部署到其他的许多平台上。

Qt5包含了如下模块，这些模块将极大地减轻你的工作量。

基本模块

- Qt Core 核心的被其他模块依赖的的图形无关的类；
- Qt GUI GUI组件的基础类，包括OpenGL；
- Qt Multimedia 提供音频、视频、广播和相机功能的方式；
- Qt Network 用于进行更简单且可移动的网络开发的方式；
- Qt QML 提供QML和Javascript语言的支持；
- Qt Quick 一个用于构建含自定义用户界面的高动态应用程序的声明式框架；
- Qt SQL 提供SQL数据库整合的类；
- Qt Test 用于对Qt程序和库进行单元测试的类；
- Qt WebKit 提供对WebKit2和新的QML API的支持；
- Qt WebKit Widgets WebKit1和源自Qt4的基于QWidget的类；
- Qt Widgets C++的widgets类。

附加模块

- Active Qt(Windows) 提供对ActiveX和COM的支持的类；
- Qt Concurrent(Windows) 提供用于编写多线程程序的高层API的支持；
- Qt D-Bus(Unix) 提供使用D-Bus协议进行进程内通讯的支持的类；
- Qt图形特效 Qt Quick2图形特效支持；
- Qt Image Formats 提供额外的图形格式支持：包括 TIFF，MNG，TGA和WBMP。
- Qt OpenGL OpenGL类支持；

  注意：此模块仅提供于Qt 4.x版本的移植。请在新的代码中使用QtGui类来代替QOpenGL。

- Qt Print Support 提供更加简易和可移动的打印方式。
- Qt Declarative Qt Declarative 是一个提供 Qt 4 兼容的类。提供到Qt 4.8 Qt Quick documentation的文档支持；
- Qt Script 让QT应用支持脚本化，此功能用于Qt 4.x的兼容支持。请在新的代码中使用 QtQml 模块中的 QJS* 类来替代Qt Script。
- Qt Script Tools 为使用Qt Script的应用提供额外功能的支持。
- Qt SVG 提供用于显示SVG文件的功能类；
- Qt XML 提供C++对SAX和DOM的支持；

  注意：不建议使用此类，请使用 `QXmlStreamReader` 和 `QXmlStreamWriter` 来启用新的功能；

- Qt XML Patterns 提供对 XPath，XQuery，XSLT 和 XML schema 语言的支持。

这些功能涵盖了开发应用程序时的所需的绝大部分功能。这就意味着，我们可以专注于开发我们项目中的特色部分，而不必为很多常规的细枝末节的功能而耗费宝贵的时间和精力，更不必每次都重造那些别人已经造过无数遍的轮子。同时，你也不必担心Qt的稳定性、易用性等等。Qt有大量的成功案例，例如著名的桌面环境KDE（K Desktop Environment）,VirtualBox虚拟机软件，国产办公软件的金山WPS等等，它绝不会让你失望。

如果上述文字仍无法打动你的话，我们再来随便看几个Qt提供的类，看看能否给你点震撼呢？ + QRegularExpression

这个类提供了对Perl兼容的正则表达式的支持，且完全支持Unicode。

- QJsonArray、QJsonDocument、QJsonObject、QJsonParseError、QJsonValue

  这些类提供了对解析和生成JSON的支持（JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式，广泛用于网络中）。

- QSystemTrayIcon

  这个类提供对系统托盘(System Tray)的支持，可以在托盘上显示图标，显示消息，支持托盘菜单等等。且这是跨平台的。支持所有Qt支持的windows版本、所有符合freedesktop.org关于系统托盘的规定的X11窗口管理器（包括最近的GNOME和KDE版本），以及所有Qt支持的Mac OS X版本。

- QCompleter

  QCompleter类为任意Qt Widget提供自动补齐功能，例如QLineEdit和QComboBox。

当然这些仅仅是Qt提供的大量功能中的少数几个，你是否动心了呢？动心了就请继续阅读本文，踏上Qt的学习和使用之旅吧。

关于Qt5的多平台支持

在开始之前，我们先来看看Qt5的多平台支持能力。

Qt5使用QPA（Qt Platform Abstraction，Qt平台抽象）将各个不同的平台的具体处理抽象化，以提供平台独立性，且可以通过插件的方式进行扩展。Qt原本在很多类的实现中都针对不同平台分别进行了特殊处理，提高了支持更多平台的难度。QPA把这些都抽象了出来，并以插件的形式实现对各平台的支持。只要实现一个新的插件就可以支持新的平台。这使得支持新的平台变成了一件轻而易举的事。

Qt5支持如下平台：

Qt5支持的桌面平台有Windows、Linux/X11和Mac OS X，嵌入式系统支持嵌入式Linux(DirectFB, EGLFS, KMS, and Wayland)、嵌入式Windows(精简版和标准版)以及实时操作系统，例如：QNX, VxWorks和INTEGRITY。 Qt5在以后的版本中将会提供对主流移动操作系统的支持（包括Android、iOS、Windows 8 (WinRT)、BlackBerry 10），这些支持正在开发中，将在未来的版本中提供（据最新消息，Qt 5.1将提供Android和iOS）。

Qt项目对如下配置进行过测试：

- Ubuntu Linux 11.10, X11 (64-bit) Ubuntu提供的编译器版本
- Ubuntu Linux 12.04, X11 (64-bit) Ubuntu提供的编译器版本
- Microsoft Windows 7 (32-bit) MSVC 2010 SP1
- Microsoft Windows 7 (32-bit) MinGW-builds gcc 4.7.2 (32-bit)
- Microsoft Windows 8 (64-bit) MSVC 2012
- Apple Mac OS X 10.7 “Lion”, Cocoa (64-bit) 苹果提供的Clang版本
- Apple Mac OS X 10.8 “Mountain Lion”, Cocoa (64-bit) 苹果提供的Clang版本

Qt提供的安装包

Qt项目为所有上述平台提供二进制安装包。如果想在其他Qt支持的平台上开发，你需要自己从源代码编译。如果想知道更多关于Qt在不同平台和编译器上的信息，请看平台和信息注意事项页面

Qt工具

工具是QT套件正式的一部分。正式的工具在所有的开发平台上都可运行。

下面是QT的工具列表：

- Qt Designer

  拓展Qt Designer功能的类；

- Qt Help

  在线帮助的类；

- Qt UI Tools

  UI工具的类；

Qt许可协议

看了这么多，你或许会问：这么强大的开发工具，得要多少钱啊？看看自己的钱包，不免有些紧张。其实不必那么紧张，Qt可以使用三种不同的协议以适应不同的用户需求：

- Qt商业许可协议（付费）

  如果你准备开发一个专有或商业软件且不希望向第三方开放任何源代码，或者你无法遵守GNU LGPL version 2.1 or GNU GPL version 3.0协议，你可以选择这个协议

- GNU LGPL version 2.1（免费）

  这个协议很著名，大家应该都很了解了。只要你采用动态链接的形式使用Qt，你就可以随意发布你的应用程序。

- GNU GPL version 3.0（免费）

  GPL开源协议大家更了解，不必多说了。如果你想使用GPLv3发布你的程序，或者代码中需要使用其他的一些使用GPLv3协议授权的代码，那么你可以使用GPLv3版本的Qt。

这三个协议的Qt库的源代码是完全一致的，也就是说功能上没有任何差异。只是收费版本提供很多支持、售后服务而已。如果想更多地了解LGPL的内容，可以参考FinderCheng的这篇博客

Chapter 3

Qt学习备忘录2：初试Qt

在对Qt做完了基本的介绍之后（其实，你认为这是布道我也没太大意见），我们开始尝试一下Qt。先配置一下Qt的环境，然后用Qt写个简单的程序（对，你猜对了，就是hello，world）。感受一下用Qt写程序的感觉。

如何安装Qt开发环境

本节将会说明如何安装Qt的开发环境，包括编译安装和二进制包安装。

二进制包安装

Ubuntu　　Ubuntu环境下有两种不同的选择。

- 选择发行版打包好的Qt

  Ubuntu发行版的软件仓库中有一个打包好的Qt。你可以直接打开Ubuntu软件中心，然后搜索Qt Creator，点击安装。也可以使用命令行：

  ```
  sudo apt-get install qtcreator qt5-default
  ```

  安装Qt Creator及开发Qt5必要的环境

- 从官方网站下载最新的二进制包

  发行版中包含的Qt版本很可能不是最新的Qt版本，或者版本无法使你满意。这种情况下，你可以选择从官网下载Qt。官网下载地址为：http://qt-project.org/downloads

  根据你的具体情况下载Qt (版本号) for Linux 32-bit(或64-bit)。然后打开命令行，到你下载下的安装包所在的目录下，执行：

```
chmod +x ./qt-linux-opensource-(你所下载的Qt的版本号)-x86_64-offline.run
```

这条命令使我们下载的.run文件具有可执行权限。然后执行：

```
sudo ./qt-linux-opensource-(你所下载的Qt的版本号)-x86_64-offline.run
```

使用root权限安装Qt。之后会打开一个和Microsoft Windows下的安装程序相同的界面，一直点下一步即可，相信各位都没问题。

（注意：笔者不使用sudo安装时会发生错误，但使用sudo后Qt Creator的配置文件的读写均需root权限，需手工修改权限为普通用户可读写，否则调Qt Creator设置的时候会报错。大牛们如果有更好的方法，还请赐教）

windows FIXME：待补齐

编译安装Qt5

Ubuntu 首先，下载源代码，编译，安装。 FIXME：没实际经验，先不写了。

windows FIXME：没实际经验，暂不写。

至此，Qt的开发环境就配置完毕了。接下来，我们开始完成一个简单的Hello World程序。

Hello World ！

敲出第一段Qt代码

步骤：

1. 「文件」->「新建文件或项目」
2. 选择「其他项目」中的「空Qt项目」
3. 跟着Qt Creator的引导创建项目，我们的项目名就叫Hello World吧。

创建好项目后，首先在HelloWorld.pro中输入：

```
    QT += core gui widgets

    SOURCES += \
        main.cpp
```

这段是用来描述我们的项目都需要包含什么内容的。qmake用它生成makefile（具体内容以后讨论）。我们需要使用Qt的core、gui和widgets三个模块，所以我们把它加入到项目中(`QT += core gui widgets`)。source指我们项目所包含的文件。这行不用自己敲，直接在Qt Creator->「项目」->「Hello World」上右键，然后添加main.cpp即可自动产生这行代码。

在main.cpp中输入以下代码：

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

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

    QLabel label("Hello World!");
    label.show();

    return a.exec();
}
```

编译，运行。然后一个Hello world程序就完成了，很简单，不是吗？

接下来，我们来具体分析一下这个Hello World程序。头两行引用头文件，这个自然不用多说。然后先实例化一个QApplication并将argc和argv参数传入。显然，我们大致猜到，QApplication类是用于处理Qt应用程序中最基本的功能的。比如处理外部传入的参数等等，是Qt应用程序中最基本的类。QLabel顾名思义，是一个标签控件，用于显示文字。那么，向构造函数传入的字符串，自然就是该标签的内容了。下一行`label.show()`的意思应该是显示该控件。最后，返回程序运行结果。

通过刚刚的目测，我们已经大致猜测了整段程序的含义，接下来，我们来细细地查阅Qt5的文档，来验证一下我们的想法。

查阅梳理Qt文档

QApplication类　　QApplication继承了QGuiApplication类，而QGuiApplication继承了QCoreApplication类，这是Qt程序的核心。

QApplication类管理着GUI程序的控制流和主要设置。QApplication继承了QGuiApplication类，并实现了一些基于QWidget的应用程序需要的功能。它负责处理关于widget的一些特定的初始化和终止化操作。注意！无论你的程序有几个窗口，你只能有 一个 QAppliaction 对象。对于那些不基于QWidget的Qt程序，你应当使用QGuiApplication代替QApplication，因为它不必依赖于QtWidgets库。

一些GUI程序会提供批处理模式（从命令行输入一些参数，然后自动执行任务而不需要人工操作）。在这类非GUI的模式下，初始化一些和图形用户界面相关的资源就显得有些浪费了。为了避免这个问题，Qt提供了QCoreApplication。下面这个例子（源自Qt5Doc）展示了如何动态选择创建QCoreApplication还是QApplication。

```
QCoreApplication* createApplication(int &argc, char *argv[])
{
    for (int i = 1; i < argc; ++i)
        if (!qstrcmp(argv[i], "-no-gui"))
            return new QCoreApplication(argc, argv);
    return new QApplication(argc, argv);
}

int main(int argc, char* argv[])
{
    QScopedPointer<QCoreApplication> app(createApplication(argc, argv));

    if (qobject_cast<QApplication *>(app.data())) {
       // 启动GUI版本...
    } else {
       // 启动非GUI版本...
    }

    return app->exec();
}
```

这段代码运行时，首先检查是否传入-no-gui。如果传入了，则实例化QCoreApplication类，否则就创建QApplication类。QScopedPointer是Qt提供的智能指针类之一，可以自动进行回收内存等等工作，目前我们先不理它，把它当作普通指针就可以。qobject_cast可以将父类指针转化为子类指针，如果强制转型成功则返回子类指针。失败则返回NULL。通过判断其返回值，我们就可以知道我们创建的是否是QApplication类，从而启动对应的处理。这样就实现了动态选择创建QApplication或QCoreApplication的功能。

QApplication对象可以通过instance()函数访问。这是一个静态程序函数，返回QApplication对象的指针，这个指针和全局的qApp指针是相同的。函数原型及功能如下：

```
QCoreApplication * QCoreApplication::instance() [static]
//返回指向程序的QCoreApplication（或QApplication）对象的指针
//如果没有QCoreApplication（或QApplication）被创建，则会返回NULL
```

QApplication主要负责这样一些工作：

- 它使用用户的桌面设置初始化应用程序，例如palette(), font()和doubleClickInterval()。它会对这些属性进行跟踪，以应对用户修改桌面的全局设置（changes the desktop globally）。例如用户通过某种控制面板一类的东西修改某些设置。
- 它负责处理事件，也就是说，他从窗口系统接受事件消息并把他们分发到相应的部件。你可以通过调用sendEvent()和postEvent()发送你自己的事件给一些窗口部件(widgets)。
- 它解析命令行参数并且根据它们设置内部状态。所有的Qt应用程序都自动支持如下命令行选项：

  -style= style，设置程序的图形用户界面风格。可行的值取决于系统设定。如果你使用其它风格编译Qt或者有其他的风格插件，那么它们都可以被用作-style命令行选项。你可以通过设置QT_STYLE_OVERRIDE环境变量来为所有的Qt程序设置风格。

  -style style 和上面的一项功能相同。

  -stylesheet= stylesheet，设置应用程序的样式表。这个值必须是一个包含样式表的文件的路径。

  注意：样式表文件中的相对路径是相对于当前项目表文件的路径的。

  -stylesheet stylesheet，和上面的一项相同。

  -widgetcount，程序最后会输出调试信息，包括未销毁的部件数和同时存在的部件数的最大值。

  -reverse，将应用程序的层（layout）的方向设为Qt::RightToLeft

  -qmljsdebugger=，有具体端口的活动的QML/JS调试器。这个值必须是这样的格式端口:1234[,block]， 其中block是可选的。这个参数会使程序等待直到一个调试器连接它。

- 它定义由QStyle对象封装的程序的外观。这个可以在运行时使用setStyle()修改。
- 它确定程序如何分配颜色。可以使用setColorSpec()函数设置。

  void QApplication::setColorSpec(int spec) [static]

  设置颜色规格（color specification）为spec。

  颜色规格控制当运行在有限的颜色数之下（例如：8位/256色）时，程序如何分配颜色。这个参数必须在你创建QApplication对象前设定好。

  更具体的内容可以参考Qt5的文档。

- 它提供通过translate()函数可见的本地化字符串。
- 它提供一些魔法般的对象，如desktop()和clipboard()。
- 它了解应用程序的所有窗口。你可以使用widgetAt()知道哪个窗口部件在当前位置上，通过topLevelWidgets()获得顶级窗口部件（或窗口）列表，以及调用closeAllWindows()关闭所有窗口等等。
- 它管理应用程序的鼠标光标处理，具体请参考setOverrideCursor()

关于qApp宏：qApp是一个全局指针，指向唯一的应用程序中的对象。它等于QCoreApplication::instance()函数返回的指针。在GUI程序里，它指向一个QApplication实例。只有一个应用程序对象可以被创建。

QCoreApplication 按照官方文档的说法，QCoreApplication为非GUI的Qt应用程序提供事件循环。GUI程序需要使用QApplication。但由于QCoreApplication是QApplication的基类，且它们都提供事件循环功能，所以，我们可以猜测大部分的事件循环都是在QCoreApplication中实现的。

QCoreApplication包含了主事件循环。源自操作系统的事件(例如：定时器和网络事件等等)和其他的一些资源的处理和分派都是在这里实现的。它同时处理应用程序的初始化和终止化，以及系统层面和程序层面上的设置。

- 关于事件循环

  如果你没有写过GUI程序，你可能会困惑事件循环是个什么东西，为什么我们要如此关注它。甚至不惜笔墨和精力，来大段地翻译QCoreApplication的类以搞清Qt中和事件循环相关的一些内容。

  事件循环是程序，特别是GUI程序在现代操作系统上运行所不可或缺的一部分。如你所见，现代操作系统都是多任务的。多个程序在同时运行。以windows为例，你在操作电脑的时候很可能会同时打开多个GUI程序。它们都有图形界面，且有些部分甚至相互重叠、相互覆盖。那么它们怎么知道用户是在向它输入消息，还是在向它附近的某个窗口输入呢？这个工作就由操作系统来完成。操作系统进行一系列的判断，确定用户到底向哪个程序输入，然后通知对应的程序。

  那么windows怎样通知具体的应用程序呢？首先，windows为每一个应用程序维护一个队列，每发生一个事件（例如用户按了哪个键，点了程序窗口的哪个位置，窗口尺寸改变了等等），windows就会把这个消息插入到对应程序的队列中。例如你点击了Firefox窗口中的某个按钮，windows就会向Firefox的队列中插入一个消息，通知它用户点击了它的某个位置。由于只向Firefox队列中插入消息，所以旁边的记事本就不必为Firefox所关心的事件头疼，更不会莫名其妙地响应用户对Firefox的操作了。这个队列就叫消息队列。

那么，作为应用程序，如果想要知道用户输入了什么，只需要不断读取自己的消息队列就可以了。读取，处理，读取，处理，遵循这样的步骤，应用程序就可以和用户不断交互，直到接到要求它关闭的消息。这个不断读取消息，处理事件，然后再读取，再处理的循环就叫事件循环。

程序的事件循环的伪代码如下：

```
int main()
{
    while(getEvent() && event!=QUIT) //不断获取事件列表中的事件，直到收到用户要求退出的消息。
    {
        switch(event)
        {
            case MOUSE_EVENT:
                //处理鼠标输入
                break;

            case KEYBOARD_EVENT:
                //处理键盘输入
                break;

            case PAINT_EVENT:
               //需要重绘（比如之前被别的窗口覆盖了一部分，这部分的内容需要重新绘制）
                //绘制Hello World什么的，是吧:）
                break;

            default:
                //不关心的事件，直接无视掉。
        }
    }
}
```

可以说，事件循环是运行在多任务系统上的GUI程序的一个主线。所有的交互和响应基本都是在事件循环中完成的。但直接用面向过程的方式使用事件循环十分不方便，不便于思考程序的逻辑。所以，为了便于开发，多数的图形用户界面框架都对事件循环进行了封装。用各种不同的方法在框架内部分发处理消息。但无论怎样封装，只要把握主事件循环这条主线，很多图形用户界面框架就都很好上手了。例如：MFC的消息映射表、VB中的什么Click啊mouse move什么的等等。毕竟，只要知道怎样处理事件，我们就知道怎样与用户交互了。知道怎样与用户交互，其他的问题其实都好解决。

在Qt中，事件循环通过调用exec()来启动。如果你要执行一个事件较长的操作，你可以调用processEvent()来保持程序对新事件及时地响应（执行耗时较长的任务时，如果新事件一直被操作系统存放在队列中，而没有程序被及时地读取、处理。程序就会看上去像在埋头干自己的活而不理用户一样。显然这会严重影响用户体验）。

通常，Qt推荐你在main()函数中创建QCoreApplication或QApplication对象，而且越早越好（毕竟，这是Qt程序的核心嘛）。然后通过调用exec()函数进入事件循环。这个函数直到事件循环结束时才会返回（比如quit()函数被调用的时候）。

QCoreApplication中还提供了一些方便的静态成员函数。事件可以使用sendEvent(),postEvent()和sendPoste
发送。它还提供了quit()信号和aboutToQuit()槽（关于信号和槽的内容我们会在以后探究）。

QCoreApplication还提供这样几个功能：

- 应用程序和库的路径

  可以通过applicationDirPath()获取程序所在目录的路径，通过applicationFilePath()获取程序本身的路径。库的路径可以通过libraryPaths()获取，并通过setLibraryPaths(), addLibraryPath数进行操作。

- 国际化和翻译

  可以通过installTranslator()和removeTranslator()来添加或删除翻译文件。更多关于国际化的内容我们或者以后进行更深入的学习。

- 访问命令行参数

  可以通过arguments()函数访问被QCoreApplication处理过的命令行参数。

- 地域设置(Locale Settings)

  在Unix/Linux上Qt默认使用系统默认的本地化设置。当然，这可能会导致一些冲突.例如：当将浮点数和字符串相互转换时，可能遇到各地的记法不同的问题。你可以通过在初始化QApplication或QCoreApplication后调用POSIX函数setlocale(LC_NUMERIC,“C”)来重置地域的方法来解决这个问题。

深入解读Hello World

之前我们只对我们的Hello World做了目测式的简单理解，在详细查过Qt的文档后，我们重新解读一下这段代码。

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

```
int main(int argc, char *argv[])
{
    //这里可以添加一些初始化代码
    //用于进行所有应该在QApplication创建之前进行的初始化
    //比如：使用`setColorSpec()`分配颜色之类的

    QApplication a(argc, argv);

    //在QApplication初始化后，所有基于QWidget的类才能正常使用
    //例如下面的QLabel什么的。

    QLabel label("Hello World!");
    label.show();

    return a.exec();
}
```

程序头两行引用了QApplication类和QLabel类的头文件。

接着，在main函数中，我们首先创建了一个QApplication实例，并将参数传入。按照Qt文档的建议，QApplication越早创建越好，所以习惯上，在main函数中进行完必要的初始化之后，就应该立刻创建QApplication(或QCoreApplication，其区别我们在上面已经解读过了)。

接下来，我们就该创建Hello World程序的核心——一个标有Hello World字样的标签了。通过将Hello World字符串传入QLabel的构造函数，我们很轻松地实现了这个功能。当然，光这样是不行的，因为一般情况下基于QWidget的类（QLabel什么的）不会自动显示，需要我们手工调用show()函数让它显示出来（其实也有不少情况会自动显示，比如插入到QTabWidget中的QWidget，这个我们后面再慢慢学）。

最后，也是最重要的一步，进入事件循环。这样，我们的使用Qt的GUI程序Hello World就算正式跑起来了。麻雀虽小，五脏俱全。可以庆祝一下胜利了。

层层深入——跟着Hello World看看Qt内部

在高呼完Hello World!之后，我们或许又会产生新的问题，Qt的内部又是什么样子呢？众所周知，开放源代码的一大好处就是可以给那些渴望学习技术的人学习技术的机会。既然有这机会一窥这样一个成熟的框架，我们何不去试试呢？

好，说干就干。可是Qt这样大的库，我们从哪里入手去看它的内部情况呢？其实，踏破铁鞋无觅处，得来全不费功夫。我们想知道Qt一步一步都做了什么，跟着我们刚刚的Hello World一步一步走不久可以了？调用了什么函数，执行了哪些操作，岂不一目了然？那我们现在就开始。

（ 我们使用Qt 5.0.2版本的源代码。为使逻辑更清晰，无关的代码我们都直接忽略 ）
首先，执行的是QApplication a(argc, argv);这是Hello World的起点，也是我们这趟深入Qt之旅的起点。

```
//qtbase/src/widgets/kernel/qapplication.h
class Q_WIDGETS_EXPORT QApplication : public QGuiApplication
{
    //...

public:
    QApplication(int &argc, char **argv, int = ApplicationFlags);
    //从这里继续深入
    virtual ~QApplication();

    //...
}

//qtbase/src/widgets/kernel/qapplication.cpp
QApplication::QApplication(int &argc, char **argv, int _internal)
    : QGuiApplication(*new QApplicationPrivate(argc, argv, _internal))
{
    QApplicationPrivate * const d = d_func()
    d->construct();
}
//...
```

然后转入QApplication的父类QGuiApplication的构造函数

```
//qtbase/src/gui/kernel/qguiapplication.h
//...
class Q_GUI_EXPORT QGuiApplication : public QCoreApplication
{
public:
    QGuiApplication(int &argc, char **argv, int = ApplicationFlags);
    //从这里继续
    virtual ~QGuiApplication();

    //...
}
```

```
//qtbase/src/gui/kernel/qguiapplication.cpp
QGuiApplication::QGuiApplication(int &argc, char **argv, int flags)
    : QCoreApplication(*new QGuiApplicationPrivate(argc, argv, flags))
{
    d_func()->init();

    QCoreApplicationPrivate::eventDispatcher->startingUp();
}
```

和QApplication的定义相当相似，我们继续进入到QCoreApplication中一探究竟。

```
//qtbase/src/corelib/kernel/qcoreaapplication.h
//...

class Q_CORE_EXPORT QCoreApplication : public QObject
{
    //...

    QCoreApplication(int &argc, char **argv, int = ApplicationFlags);
    ~QCoreApplication();

    //...

private:
    void init();

    //...
}

//qtbase/src/corelib/kernel/qcoreapplication.cpp
//...

QCoreApplication::QCoreApplication(int &argc, char **argv, int _internal)
    : QObject(*new QCoreApplicationPrivate(argc, argv, _internal))
{
    init();
    QCoreApplicationPrivate::eventDispatcher->startingUp();
}
```

终于找到有实际意义的函数QCoreApplication::init()了，几经辗转啊！

```
//qtbase/src/corelib/kernel/qcoreapplication.cpp
void QCoreApplication::init()
{
    QCoreApplicationPrivate * const d = d_func()

    //初始化Loacle
    QCoreApplicationPrivate::initLocale();

    //判断是否已经创建过一个应用程序对象
 Q_ASSERT_X(!self, "QCoreApplication", "there should be only one application object");
    QCoreApplication::self = this;

    //使用app程序员创建的事件分发器(如果有的话)
    if (!QCoreApplicationPrivate::eventDispatcher)
    QCoreApplicationPrivate::eventDispatcher = d->threadData->eventDispatcher;
    //如果没有，自动创建一个默认的
    if (!QCoreApplicationPrivate::eventDispatcher)
        d->createEventDispatcher();
    Q_ASSERT(QCoreApplicationPrivate::eventDispatcher != 0);

    if (!QCoreApplicationPrivate::eventDispatcher->parent()) {
    QCoreApplicationPrivate::eventDispatcher->moveToThread(d->threadData->thread);
        QCoreApplicationPrivate::eventDispatcher->setParent(this);
    }

   d->threadData->eventDispatcher = QCoreApplicationPrivate::eventDispatcher;

#ifndef QT_NO_LIBRARY
    if (coreappdata()->app_libpaths)
        d->appendApplicationPathToLibraryPaths();
#endif

#if defined(Q_OS_UNIX) && !(defined(QT_NO_PROCESS))
    // Make sure the process manager thread object is created in the main
    // thread.
    QProcessPrivate::initializeProcessManager();
#endif

#ifdef QT_EVAL
    extern void qt_core_eval_init(uint);
    qt_core_eval_init(d->application_type);
```

```
    #endif

        //处理命令行参数
        d->processCommandLineArguments();

        //
        qt_startup_hook();
    }
    //
```

这么复杂的一堆代码我们自然不可能都看完。着重看一下我们一直关注的和事件处理相关的部分吧。我们来看看Qt是怎样创建默认的事件分发器的。透过这里的代码，我们应该可以简单地推测一下Qt是如何跨平台的。

```
    //qtbase/src/corelib/kernel/qcoreapplication.cpp
    void QCoreApplicationPrivate::createEventDispatcher()
    {
        Q_Q(QCoreApplication);

    //为Unix类平台创建事件分发器
    #if defined(Q_OS_UNIX)
    #  if defined(Q_OS_BLACKBERRY)
        //黑莓平台
        eventDispatcher = new QEventDispatcherBlackberry(q);
    #  else
    #  if !defined(QT_NO_GLIB)
     if (qEnvironmentVariableIsEmpty("QT_NO_GLIB") && QEventDispatcherGlib::versionSupported())
            //Glib事件分发器
            eventDispatcher = new QEventDispatcherGlib(q);
        else
    #  endif
        //UNIX事件分发器
        eventDispatcher = new QEventDispatcherUNIX(q);
    #  endif
    #elif defined(Q_OS_WIN)
        //windows事件分发器
        eventDispatcher = new QEventDispatcherWin32(q);
    #else
        //没有发现该使用何种事件分发器
    #  error "QEventDispatcher not yet ported to this platform"
```

```
    #endif
    }
```

在这里为不同的平台选择了不同的事件分发器。看来Qt使用了QAbstractEventDispatcher这个抽象类来抽象不同平台的差异，并为不同平台实现对应的子类。我们来看看QEventDispatcherWin32类。

```
   bool QEventDispatcherWin32::processEvents(QEventLoop::ProcessEventsFlags flags)
    {
        Q_D(QEventDispatcherWin32);

        if (!d->internalHwnd)
            createInternalHwnd();

        d->interrupt = false;
        emit awake();

        bool canWait;
        bool retVal = false;
        bool seenWM_QT_SENDPOSTEDEVENTS = false;
        bool needWM_QT_SENDPOSTEDEVENTS = false;
        do {
            DWORD waitRet = 0;
            HANDLE pHandles[MAXIMUM_WAIT_OBJECTS - 1];
            QVarLengthArray<MSG> processedTimers;
            while (!d->interrupt) {
                DWORD nCount = d->winEventNotifierList.count();
                Q_ASSERT(nCount < MAXIMUM_WAIT_OBJECTS - 1);

                MSG msg;
                bool haveMessage;

                        if (!(flags & QEventLoop::ExcludeUserInputEvents) && !
d->queuedUserInputEvents.isEmpty()) {
                    // 处理队列中的用户输入事件
                    haveMessage = true;
                    msg = d->queuedUserInputEvents.takeFirst();
                    } else if(!(flags & QEventLoop::ExcludeSocketNotifiers) && !
d->queuedSocketEvents.isEmpty()) {
```

```
                    // 处理队列中的socket事件
                    haveMessage = true;
                    msg = d->queuedSocketEvents.takeFirst();
                } else {
                    //调用win32api读取windows的消息队列
                    haveMessage = PeekMessage(&msg, 0, 0, 0, PM_REMOVE);
                  if (haveMessage && (flags & QEventLoop::ExcludeUserInputEvents)
                      && (//笔者吐槽：这对括号中间包含的东西也太多了吧，差点没区
分出来
                        (msg.message >= WM_KEYFIRST
                        && msg.message <= WM_KEYLAST)
                        //WM_KEYFIRST和WM_KEYLAST可作为过滤值取得所有键盘消息--源
自百度百科

                        || (msg.message >= WM_MOUSEFIRST
                            && msg.message <= WM_MOUSELAST)
                            //常数WM_MOUSEFIRST和WM_MOUSELAST可用来接收所有的鼠标
消息--还是来自百度百科

                        || msg.message == WM_MOUSEWHEEL
                        || msg.message == WM_MOUSEHWHEEL
                        //鼠标滚轮消息

                        || msg.message == WM_TOUCH
                    #ifndef QT_NO_GESTURES
                        || msg.message == WM_GESTURE
                        || msg.message == WM_GESTURENOTIFY
                    #endif

                        || msg.message == WM_CLOSE)
                        ) {
                        // 将输入的消息插入队列中
                        haveMessage = false;
                        d->queuedUserInputEvents.append(msg);
                    }

                  if (haveMessage && (flags & QEventLoop::ExcludeSocketNotifiers)
                && (msg.message == WM_QT_SOCKETNOTIFIER && msg.hwnd == d->internalHwnd)) {
                        //将socket事件插入队列中
                        haveMessage = false;
                        d->queuedSocketEvents.append(msg);
```

```
            }
        }
        if (!haveMessage) {
            // 没有消息，检查发出信号的对象
            for (int i=0; i<(int)nCount; i++)
                pHandles[i] = d->winEventNotifierList.at(i)->handle();
   waitRet = MsgWaitForMultipleObjectsEx(nCount, pHandles, 0, QS_ALLINPUT, MWMO_ALERTABLE)
            if ((haveMessage = (waitRet == WAIT_OBJECT_0 + nCount))) {
                // 一条新消息到达，处理它
            continue;
        }
    }
    if (haveMessage) {

   if (d->internalHwnd == msg.hwnd && msg.message == WM_QT_SENDPOSTEDEVENTS) {
            if (seenWM_QT_SENDPOSTEDEVENTS) {
        // when calling processEvents() "manually", we only want to send posted
                // events once
                needWM_QT_SENDPOSTEDEVENTS = true;
                continue;
            }
            seenWM_QT_SENDPOSTEDEVENTS = true;
        } else if (msg.message == WM_TIMER) {
        // avoid live-lock by keeping track of the timers we've already sent
            bool found = false;
            for (int i = 0; !found && i < processedTimers.count(); ++i) {
                const MSG processed = processedTimers.constData()[i];
        found = (processed.wParam == msg.wParam && processed.hwnd == msg.hwnd && processed.lPa
            }
            if (found)
                continue;
            processedTimers.append(msg);
        } else if (msg.message == WM_QUIT) {
            if (QCoreApplication::instance())
                QCoreApplication::instance()->quit();
            return false;
        }

   if (!filterNativeEvent(QByteArrayLiteral("windows_generic_MSG"), &msg, 0)) {
            //终于出现了，win32程序消息处理函数中的常客
            TranslateMessage(&msg);
```

```
                    DispatchMessage(&msg);
                }
            } else if (waitRet - WAIT_OBJECT_0 < nCount) {
           d->activateEventNotifier(d->winEventNotifierList.at(waitRet - WAIT_OBJECT_0));
            } else {
                // 无事可做，退出
                break;
            }
            retVal = true;
        }

        // 仍然无事可做，等待消息或对象发出信号
        canWait = (!retVal
        && !d->interrupt
        && (flags & QEventLoop::WaitForMoreEvents));
        if (canWait) {
            DWORD nCount = d->winEventNotifierList.count();
            Q_ASSERT(nCount < MAXIMUM_WAIT_OBJECTS - 1);
            for (int i=0; i<(int)nCount; i++)
                pHandles[i] = d->winEventNotifierList.at(i)->handle();

            emit aboutToBlock();
        waitRet = MsgWaitForMultipleObjectsEx(nCount, pHandles, INFINITE, QS_ALLINPUT, MWMO_ALERTABLE | MWMO_I
            emit awake();
            if (waitRet - WAIT_OBJECT_0 < nCount) {
           d->activateEventNotifier(d->winEventNotifierList.at(waitRet - WAIT_OBJECT_0));
                retVal = true;
            }
        }
        } while (canWait);

      if (!seenWM_QT_SENDPOSTEDEVENTS && (flags & QEventLoop::EventLoopExec) == 0) {
            // when called "manually", always send posted events
            sendPostedEvents();
        }

        if (needWM_QT_SENDPOSTEDEVENTS)
            PostMessage(d->internalHwnd, WM_QT_SENDPOSTEDEVENTS, 0, 0);

        return retVal;
    }
```

```
    //
```

长长一大片代码，虽然有些杂乱无章，不过还是看到了我们想要看的东西：PeekMessage()这是windows平台上读取消息用的一个函数，说明我们已经成功地找到了Qt获取并分发消息的地方。现在，让我们来简单整理一下思路。Qt程序的跨平台方案是这样的：Qt程序在编译时通过Q_OS_UNIX、Q_OS_WIN32等宏判断目标平台，然后选择创建对应的事件分发器。所有的事件分发器均基于QAbstractEventDispatcher抽象类。这样，不同平台的差异性就被抽象了出来，具体实现由QEventDispatcherWin32等等子类实现，这使得Qt的平台相关的部分与那些与平台无关的部分分离。

那么，最后，我们进入exec()函数，也即进入事件循环。具体查找Qt源代码的过程和上面相仿，我们直接给出结果，就不再赘述无关紧要的细节了。

```
    //qtbase/src/corelib/kernel/qcoreapplication.cpp
    int QCoreApplication::exec()
    {
        if (!QCoreApplicationPrivate::checkInstance("exec"))
            return -1;

        QThreadData *threadData = self->d_func()->threadData;
        if (threadData != QThreadData::current()) {
            //检查是否是从主线程调用的，如果不是，则报错
        qWarning("%s::exec: Must be called from the main thread", self->metaObject()->className());
            return -1;
        }
        if (!threadData->eventLoops.isEmpty()) {
            //检查是否已经运行事件循环了
           qWarning("QCoreApplication::exec: The event loop is already running");
            return -1;
        }

        threadData->quitNow = false;
        QEventLoop eventLoop;
        self->d_func()->in_exec = true;
        self->d_func()->aboutToQuitEmitted = false;

        //笔者吐槽：封装也太厚了，完全超乎笔者的想象啊。
        //从这里移交给QEventLoop的exec()
        int returnCode = eventLoop.exec();

        threadData->quitNow = false;
```

```
        if (self) {
            self->d_func()->in_exec = false;
            if (!self->d_func()->aboutToQuitEmitted)
                emit self->aboutToQuit(QPrivateSignal());
            self->d_func()->aboutToQuitEmitted = true;
            sendPostedEvents(0, QEvent::DeferredDelete);
        }

        return returnCode;
    }
```

好吧，又调用了QEventLoop的exec()函数，真是层层转接啊。没办法，打开eventloop的源文件看：

```
    //qtbase/src/corelib/kernel/qeventloop.cpp
    int QEventLoop::exec(ProcessEventsFlags flags)
    {
        Q_D(QEventLoop);
        //这里是为了线程安全做的保护
        //we need to protect from race condition with QThread::exit
     QMutexLocker locker(&static_cast<QThreadPrivate *>(QObjectPrivate::get(d->threadData->thread))->mutex);
        if (d->threadData->quitNow)
            return -1;

        if (d->inExec) {
         qWarning("QEventLoop::exec: instance %p has already called exec()", this);
            return -1;
        }

        struct LoopReference {
            QEventLoopPrivate *d;
            QMutexLocker &locker;

            bool exceptionCaught;
        LoopReference(QEventLoopPrivate *d, QMutexLocker &locker) : d(d), locker(locker), exceptionCaught(true
            {
                d->inExec = true;
                d->exit = false;
                ++d->threadData->loopLevel;
```

```
                d->threadData->eventLoops.push(d->q_func());
                locker.unlock();
            }

            ~LoopReference()
            {
                if (exceptionCaught) {
              qWarning("Qt has caught an exception thrown from an event handler. Throwing
\n"
                    "exceptions from an event handler is not supported in Qt. You must
\n"
                    "reimplement QApplication::notify() and catch all exceptions there.
\n");
                }
                locker.relock();
                QEventLoop *eventLoop = d->threadData->eventLoops.pop();
           Q_ASSERT_X(eventLoop == d->q_func(), "QEventLoop::exec()", "internal error");
                Q_UNUSED(eventLoop); // --release warning
                d->inExec = false;
                --d->threadData->loopLevel;
            }
        };
        LoopReference ref(d, locker);

        // 当进入一个新的事件循环时，移除所有已经发送的quit事件
        QCoreApplication *app = QCoreApplication::instance();
        if (app && app->thread() == thread())
            QCoreApplication::removePostedEvents(app, QEvent::Quit);

         //笔者吐槽：经历了千沟万壑，终于看到事件循环了。但，怎么这么短啊！太令人
失望了
        //如果没有被要求退出，就一直循环并不断处理消息
        while (!d->exit)
            processEvents(flags | WaitForMoreEvents | EventLoopExec);

        ref.exceptionCaught = false;
        return d->returnCode;
    }
```

然后，我们看一看处理消息的函数：

```
bool QEventLoop::processEvents(ProcessEventsFlags flags)
{
    Q_D(QEventLoop);
    if (!d->threadData->eventDispatcher)
        return false;
    return d->threadData->eventDispatcher->processEvents(flags);
}
```

这个函数就很简单了，调用我们之前已经创建好的事件分发器的事件处理函数而已。至此，Qt程序运行的基本轮廓就已经被勾勒完全了。

Qt程序运行基本轮廓

在繁琐的跟踪过程之后，我们简单整理一下我们所看到的内容：

1. 对locale等等内容进行初始化
2. 创建对应平台的事件分发器
3. 调用exec()进入事件循环，直到接受到退出消息。

其中，事件循环是通过在QEventLoop中不断调用QAbstractEventDispatcher的子类的processEvents()实现的。

Chapter 4

Qt学习备忘录3：基本Qt部件、窗体、动作以及Qt的事件处理机制

写出Hello World仅仅是我们学习之路的开始。它代表着我们已经出发了，带着对Qt的兴奋与好奇，让我们一起前行。那么我们从哪里开始呢？当然是最基础的东西。可什么东西是最基础的呢？Qt Core模块？Qt GUI模块？稍微有些不知所措了。重新回忆一下我们学习Qt的初衷：写跨平台GUI程序。很好。那么GUI程序都需要什么呢？窗体、菜单、工具栏、对话框……一堆我们熟悉的GUI元素浮现在我们脑海中。那么，我们就从这里开始吧。一点一点地探索如何创建并使用这些GUI元素。

针对桌面的Qt程序一般使用Qt Widgets编写。Qt Widgets是一组用于创建经典桌面样式用户界面的UI元素。每一种GUI组件都是一个放置在用户界面窗口内或显示在独立窗口中的部件(widget)。每一种部件都是QWidget的一个子类，而QWidget是QObject的一个子类。

QWidget并不是一个抽象类，它可以作为其它部件的容器。它还可以被继承以创建新的自定义部件，这只需要极小的努力就能完成。QWidget一般用于创建放置其他部件的窗体。

作为一个QObject，QWidget可以设置父对象（parent object），这样，它就可以在它不被使用的时候被自动删除。对于部件，这种父母和孩子的关系(parent-child relationships)有更多的意义：所有的子部件都被显示在父部件使用的屏幕区域内。这就意味着，如果你删除了一个窗口部件，所有的子部件都会被删除。

Chapter 5

一种典型的main()函数写法

Qt5的文档中提供了一种典型的Qt程序的主函数的写法，和我们之前的Hello World程序基本是一致的。

```
#include <QtWidgets>

// Include header files for application components.
// ...

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

    // 设置并显示部件
    // ...

    return app.exec();
}
```

具体的含义在我们解析Hello World程序时已经解说过了，相信大家现在都能看懂。

Chapter 6

创建最简单的窗口

这里有一个创建窗口的例子，这个例子非常简单，可以使我们了解QWidget的最基本的使用。

```
#include <QtWidgets>

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

    QWidget window;
    window.resize(320, 240);
    window.show();
    window.setWindowTitle(
        QApplication::translate("toplevel", "Top-level widget"));

    return app.exec();
}
```

运行效果：

一行一行地解释

我们从创建QWidget这一行开始，前面的初始化过程之前已经解释过，以后就不再说明了。`QWidget window`一句用于创建一个QWidget对象。由于我们没有制定window的父对象是谁，所以默认为NULL，也就是没有父对象。所以，window被作为最顶级的窗口。

Figure 6.1: show result

接下来的window.resize(320,240)设置窗口尺寸为320*240。

而后调用window.show()来显示窗口。

最后，调用setWindowTitle()函数设置窗口标题．QApplication::translate()函数与国际化有关，先不管它，以后再学。

Chapter 7

关于QWidget类

QWidget可以说是Qt Widget模块的核心。这个类是所有用户界面对象的基类。

部件(widget)是用户界面的最基本元素（Qt5的文档中使用atom，也即原子这个词来描述它）：它从窗口系统接收鼠标、键盘以及其他事件，并把它自己绘制在屏幕上。所有的部件都是矩形的，按Z轴的顺序排列。一个部件的遮挡由其父对象和它前面的部件决定。

没有父部件的部件被称作窗口(window)。一般情况下，窗口有一个边框和一个标题栏。当然，使用适当的窗口标记(window flags)来创建一个没有这些东西的窗口也是可行的。在Qt中，最常见的窗口类型就是QMainWindow以及众多的QDialog的子类。

所有部件的构造函数可以接受一个或两个标准的参数：

1. `QWidget *parent = 0`代表新部件的父部件。如果它为0(默认值)，新部件会成为一个窗口。如果不为0,则它会成为parent的一个孩子，会受parent的设置影响。

2. `Qt::windowFlags f = 0`用于设置窗口标记。默认值适用于几乎所有窗口。但如果你想实现一些特殊的窗口，例如没有边框的窗口，你需要使用特殊的标记。

顶级窗口和子部件

一个没有父部件的窗口会成为一个独立的窗口（顶级部件）。对于这些部件，你可以使用`setWindowTitle()`和`setWindowIcon()`独立地设置标题栏和图标。

子部件没有窗口。它们会在其父部件中被显示。Qt中的大部分都主要作为子部件使用。例如：你可以使用一个按钮作为顶级部件，但大多数人都使用它作为QDialog这样的部件的子部件（即把它放在一个对话框中）。之前我们的Hello World中的QLabel部件就是一个特殊的QLabel部件作为顶级部件使用的例子。

自定义部件及绘图

由于QWidget是QPaintDevice类的一个子类，它可以使用一系列QPainter实例的绘图操作来显示自定义内容。所有的绘图操作会在paintEvent()函数中完成，这个函数会在部件需要重绘或应用程序发出重绘请求时被自动调用。

Chapter 8

项目开工！从QMainWindow开始

在本备忘录的序中我们曾提及，我们准备开展一个IDE项目。从这节之后，我们将一步一步地按照我们学习Qt的顺序，将所学应用与项目中，并以此为例子，记录Qt中的一些概念和用法。

建立主窗口——QMainWindow的使用

之前我们已经说过，没有父部件的QWidget会成为一个独立的窗口。但一般情况下，很少有人直接使用QWidget类作为主窗口。因为主窗口往往需要菜单栏(Menu Bar)、状态栏(Status Bar)、工具栏(Toolbars)等等很多元素，QWidget显得太过简陋了。

幸运的是，Qt为我们准备好了一个功能强大的专门用于创建主窗口的框架。这个框架包含QMainWindow类以及很多相关的类(QToolBar、QDockWidget等等)。它们可以用于管理主窗口。其中，QMainWindow是主窗口的核心。

QMainWindow有它自己的布局，你可以向其中添加QToolBar,QDockWidget,QMenuBar和QStatusBar。布局中还有一个中央部件(Center Widget)，任何部件都可以成为中央部件。下面的图片展示了QMainWindow的布局。

注意：QT不支持创建一个不含中央部件(center widget)的主窗口。你必须有中央部件，即使那只是一个占位符。

创建主窗口部件

一个中央部件一般为标准Qt部件，如QTextEdit或QGraphicsView等。当然，自定义部件也是可以的。你可以使用`setCentralWidget()`函数设置中央部件。很好，那么我们来试一试。作为文本编辑器，最核心的部件当然是编辑器了。所以我们来把一个QTextEdit设置中央部件。

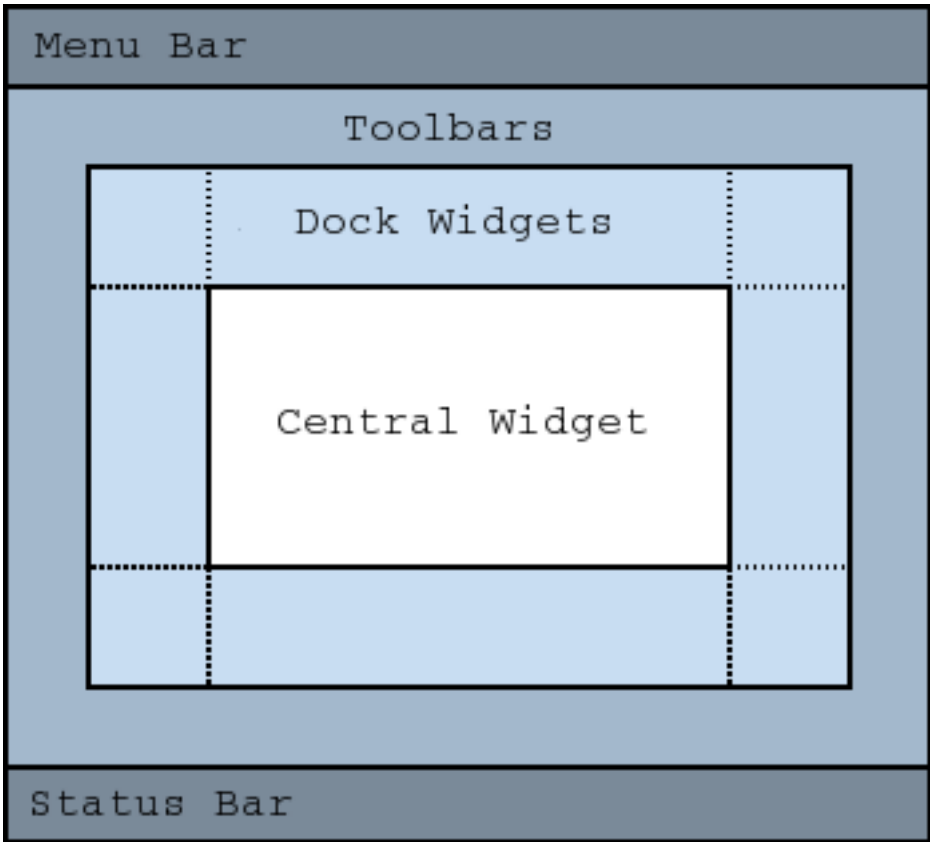

Figure 8.1: mainwindow layout

首先，我们知道，主窗口必然是需要自定义的，你见过有哪两个程序的的主窗口是完全一样的？所以，我们需要继承QMainWindow以实现自己的主窗口——一个有一个TextEdit的主窗口。

```
#include <QMainWindow>
#include <QTextEdit>

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);

private:
    QTextEdit *editor;

};
```

然后，我们在构造函数中设置我们的editor为主窗口。

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    editor = new QTextEdit(this);
    setCentralWidget(editor);
}
```

这样，我们就把一个TextEdit设置为中央部件了，很简单，不是吗？

创建菜单栏

接下来，我们来试着创建一个菜单栏。

Qt通过QMenu实现菜单。QMainWindow会将菜单放置在菜单栏。菜单中的每一个QAction都会以菜单项的形式显示。Qt的文档读到这里，我们会产生一个疑问，什么是QAction呢？

- 关于QAction

  QAction提供一个可以被插入到部件中的抽象用户界面动作（abstract user interface action）。在应用程序中，很多公共的命令可以通过菜单、工具栏和键盘快捷键调

用。由于用户希望同一个命令执行相同的操作，无论是从菜单启动还是从工具栏调用，所以，将每个命令抽象成一个动作(action)是很有必要的。

动作可以被添加到菜单中或者工具栏中，它会自动同步状态。例如，在一个字处理软件中，用户点击了加粗按钮，那么菜单中的加粗和工具栏中的加粗都会被自动设置为选中状态。

动作可以作为一个独立的对象常见，也可以在创建菜单的时候创建。QMenu类包含一组方便的函数用于创建适用于菜单项的动作。

一个QAction可以拥有图标，菜单文本(Menu Text)，快捷键，状态信息(status text)，“What’s this?(这是什么)”文本和一个工具提示(tooltip)。这些大多可以通过构造函数设置。也可以通过setIcon(),setText(),setIconText(),setShortcut(),setStatusTip(),setWhats些函数单独设置。对于菜单项，可以通过调用setFont()设置单独的字体。

动需可以通过QWidget::addAction()或QGraphicsWidget::addAction()函数添加到部件。注意，一个动作必须在添加到部件上之后才能使用，当添加全局快捷键是也是如此。

一旦一个QAction被创建，它应当被添加到相关的菜单和工具栏中，然后和对应的槽相连接。信号和槽是Qt很特别的一个机制，我们会在稍后研究。这里有一个源自Qt5Doc的例子：

openAct = new QAction(QIcon(“:/ images/ open.png”), tr(“&Open…”), this); openAct->setShortcuts(QKeySequence::Open); openAct->setStatusTip(tr(“Open an existing file”)); connect(openAct, SIGNAL(triggered()), this, SLOT(open()));

fileMenu->addAction(openAct); fileToolBar->addAction(openAct);

Qt建议将动作创建为使用它们的窗口的孩子(children of the window they are used in)。大多数情况下，动作会成为程序主窗口的孩子。

知道了什么是QAction，我们开始创建菜单。QMainWindow提供了menuBar()这个给力的函数。这个函数会返回当前程序的菜单栏的指针(QMenuBar*)，如果当前还没有创建菜单栏，它会自动创建一个。然后，我们就可以通过QMenuBar::addMenu()来添加菜单了。

QMainWindow有默认的菜单栏，但如果你想设置一个你自己的菜单栏，可以使用setMenuBar()如果你希望实现一个自定义的菜单栏(不使用QMenuBar部件)，你可以使用setMenuWidget()来设置它。这里有一个简单的例子(Qt5Doc)：

```
void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAct);
    fileMenu->addAction(openAct);
    fileMenu->addAction(saveAct);
}
```

创建工具栏

工具栏通过QToolBar类实现。你可以通过addToolBar()函数向主窗口添加工具栏。工具栏的初始位置可以通过设置具体的Qt::ToolBaarArea来控制。通过addToolBarBreak()或insertToolBarBreak(),你可以插入一个工具栏分隔(就像文本编辑中的分隔行一样)来分隔区域。你还可以通过QToolBar::setAllowedAreas()和QToolBar::setMovable()函数限定工具栏的位置。

工具栏图标的大小可以通过iconSize()函数设定。尺寸是平台独立的。你可以调用setIconSize()设置一个固定的大小。你还可以使用setToolButtonStyle()函数修改所有工具栏按钮的外观(alter the appearance of all tool buttons)。

一个创建工具栏的例子如下(Qt5Doc):

```
void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(newAct);
}
```

创建状态栏

创建状态栏和创建菜单栏类似，你可以使用statusBar()，它会返回当前的状态栏，如果没有则会创建一个。当然，也可以通过setStatusBar()设置一个。

亲自试试

在这份备忘录的开头，我们提过，这份备忘录是我们学习Qt并尝试编写IDE的过程中产生的。所以，我们现在来亲自试试创建一个IDE（或许目前的情况叫文本编辑器更合适）所需的菜单栏和状态栏。

```
//version 0.0.0.1
//mainwindow.h
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);

private:
```

```
    enum MainWindowActions
    {
        new_file,
        open,
        save,
        close_file,
        close_all,
        quit,
        about,
        act_count   //the number of actions.
    };

    QTextEdit *editor;
    QAction *act[act_count];

    void createActions();
    void createMenu();
    void createStatusbar();
};
```

我们需要创建多个QAction．因此我们需要多个QAction指针变量，一个一个地创建过于麻烦，所以我们直接创建一个QAction*数组。为了方便以后增加新的action，我们使用一个枚举来实现。如你所见，这个枚举中包含了所有我们需要创建的动作。枚举的最后一项act_count是枚举中的最后一个值，所以正好等于我们所需要创建的QAction*数组的大小。这样做便于维护。

接着，我们开始分别实现创建动作、菜单和状态栏的函数。

```
void MainWindow::createActions()
{
    //new file
    act[new_file]=new QAction(tr("new file"),this);

    //open
    act[open]=new QAction(tr("open"),this);

    //save
    act[save]=new QAction(tr("save"),this);

    //save_as
    act[save_as]=new QAction(tr("save as"),this);
```

```
    //save_all
    act[save_all]=new QAction(tr("save all"),this);

    //close
    act[close]=new QAction(tr("close"),this);

    //closs_all
    act[close_all]=new QAction(tr("close all"),this);

    //quit
    act[quit]=new QAction(tr("quit"),this);

    //redo
    act[redo]=new QAction(tr("redo"),this);

    //undo
    act[undo]=new QAction(tr("undo"),this);

    //cut
    act[cut]=new QAction(tr("cut"),this);

    //copy
    act[copy]=new QAction(tr("copy"),this);

    //paste
    act[paste]=new QAction(tr("paste"),this);

    //about
    act[about]=new QAction(tr("about"),this);

    //about_qt
    act[about_qt]=new QAction(tr("about Qt"),this);
}

void MainWindow::createMenu()
{
    int i;
    QMenuBar *_menubar=menuBar();

    //file menu
```

```
    menu[file] = _menubar->addMenu(tr("file"));
    //from new_file to quit add into file menu
    for(i=new_file;i<=quit;i++)
        menu[file]->addAction(act[i]);

    //edit menu
    menu[edit]= _menubar->addMenu(tr("edit"));
    //from redo to paste add into edit menu
    for(i=redo;i<=paste;i++)
        menu[edit]->addAction(act[i]);

    //help menu
    menu[help]= _menubar->addMenu(tr("help"));
    //from about to about_qt add into help menu
    for(i=about;i<=about_qt;i++)
        menu[help]->addAction(act[i]);

}

void MainWindow::createStatusbar()
{
    QStatusBar *statusbar=statusBar();

    //...
}
```

然后，在MainWindow的构造函数中调用它们，这样当主窗口被创建时，它们也会被创建。

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    editor = new QTextEdit(this);
    setCentralWidget(editor);

    createActions();
    createMenu();
    createStatusbar();
}
```

至此，我们的编辑器的第一个版本就完成了。它有一个独一无二的功能——有一组没有用的菜单和一个状态栏，开个玩笑，呵呵。运行结果：

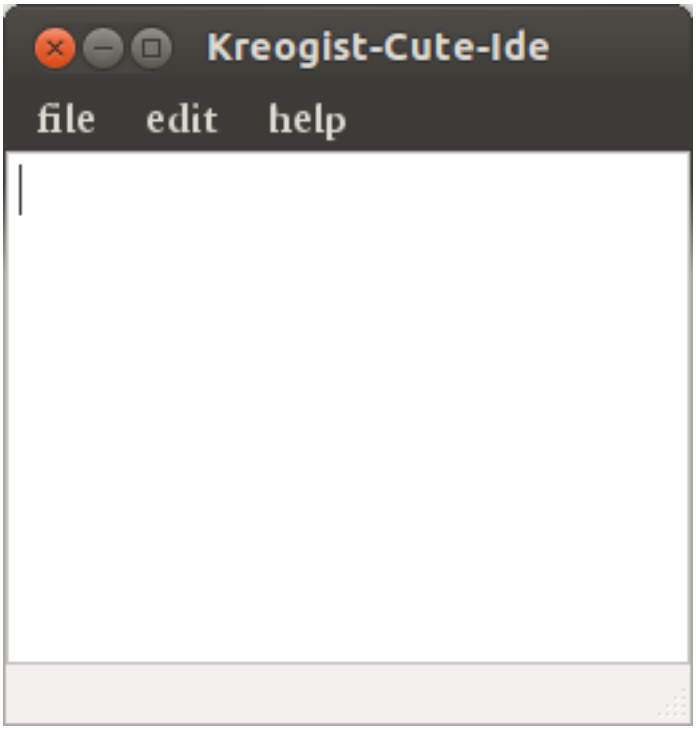

Figure 8.2: ch3-1result

Qt比我们想象得简单得多，不是吗？

Chapter 9

给程序以生命之一：Qt的信号与槽机制

之前我们曾开玩笑，我们的文本编辑器有一个独一无二的功能，就是有一组无法响应用户操作的菜单。说笑归说笑，我们总得想办法让它响应用户操作。有了对用户操作的响应，GUI程序才具有了生命。Qt提供了两种机制用于处理事件。一种是Qt的事件系统。这是一个底层的以QObject为基类的对象间通信的机制。还有一种更常用的高层的机制叫做信号与槽。我们先从最常用的信号与槽机制介绍起。

什么是信号和槽？

信号与槽用于对象间通信。信号和槽机制是Qt的一个核心特性，而且这是Qt最不同于其它的框架的部分，可以说是Qt的一个标志。

在GUI编程中,当我们改变一个部件时,我们总是想让其它部件得到通知。从更广义的角度讲，我们希望任意类型的对象间能够互相通讯。例如，如果用户点击关闭按钮,我们希望窗口的`close()`函数被调用。也即我们希望按钮和窗口这两个对象间可以通讯：关闭按钮被点击后，按钮通知窗口到了该关闭的时候了。然后窗口关掉自己。老式的工具包(toolkit)或框架使用回调(callback)来实现这个功能。一个回调是一个指向函数的指针。如果你希望一个处理函数通知你一些事件发生了，你需要传递一个指向另一个函数(回调)的指针给处理函数。然后，处理函数会在适当的时候调用回调函数。

- 什么是回调

  回调是指通过传递一个函数指针给另外一个函数，使得在特定的事件或条件发生时，可以由对方调用，以响应相应的事件或条件。

  我们在设计软件时常会遇到这样一种情况：某个模块检测到某个事件发生，然后，需要通知其它模块去处理。但设计这个模块时，我们并不知道具体是哪个模块去处理这个事件。例如：我们设计一个计时器，当时间到了的时候，需要通知其它模块处理。

但具体哪个模块在使用计时器是不确定的，因为很多模块都可以使用它。这种情况下，就可以使用回调。一般在C语言中，使用函数指针实现回调。在前面的例子中，就是把一个处理函数的指针传给计时器。当计时器发现时间到了的时候，通过指针调用处理函数。这样，就降低了计时器和处理函数的耦合性。计时器不需要知道具体是哪个函数处理，处理函数也不必在意是哪个计时器调用它。双方都只需要关系自己的处理逻辑即可，这样他们间的关联性就降低了很多。

最常见的例子是C语言标准库中的快排函数。它的函数原型如下：

void qsort (void* base, size_t num, size_t size, int (compar)(const void,const void*));

最后传入的compar函数的指针就是一个回调。这个函数会在qsort比较两个元素大小时被自动调用。这样，qsort就只需要关心排序逻辑，而不需要关心它排的数据的具体类型是int,double还是一个结构体。

Qt的开发者们认为，回调有两个根本缺陷：首先，它们不是类型安全的。我们不能保证处理函数会以正确的参数调用回调函数。其次，回调与处理函数是强耦合的，因为处理函数必须知道它需要调用的回调函数是哪个。

于是在Qt中,使用了另一种回调技术——信号和槽(signals and slots)。一个特定的事件(event)发生时，会发出一个信号(signal)。Qt的部件有许多预定义的信号，我们可以通过继承它们来添加我们自己的信号到这些部件上。一个槽是一个响应特定信号的函数。Qt的部件有许多预定义的槽。但如果我们希望自定义对一些信号的处理方法，我们可以继承这些窗口部件并添加自己的槽，这样就可以处理那些我们感兴趣的信号了。

信号和槽机制是类型安全的：一个信号的参数必须与接收信号槽的参数类型相匹配。(事实上，一个槽的接受的参数数量可以比它接收的信号提供的参数量更少，因为它可以忽略额外的参数)。信号和槽是松耦合的：一个类发出一个信号后，它既不需要知道这个信号发给谁，也不必关心接收到信号的槽是什么。Qt的信号与槽机制确保了，当你连一个信号到一个槽时，槽将在正确的时机以信号发出的参数被调用。信号与槽可以包含任意类型、任意数量的变量。他们是完全类型安全。

所有继承自QObject的类(如。,QWidget)都可以包含信号和插槽。当一个对象改变它的状态且有其它对象对此感兴趣时，一个信号会被发出。这就是Qt中的对象进行通信的方式。它不知道也不关心谁会会接收它发出的信号。这是真正的对信息的封装。这确保了对象(objects)可以作为一个组成软件的组件来使用。

槽可以用来接收并处理信号。同时，它们也普通成员函数。就像对象不需要知道谁会接收它发出的信号一样，槽也不必知道是否有信号连接到它。这个机制保证了Qt可以创造出真正独立的组件。

只要你愿意，你可以连接任意多的信号到同一个槽。同样，只要你需要，一个信号也可以连接到任意多的槽上。甚至直接连接一个信号到另一个信号也是可行的。(这将使得第一个信号发出后，第二个信号立即被发出。)

信号和槽组合在一起，构成了Qt强大的组件编程机制。（ Together, signals and slots make up a powerful component programming mechanism. ）

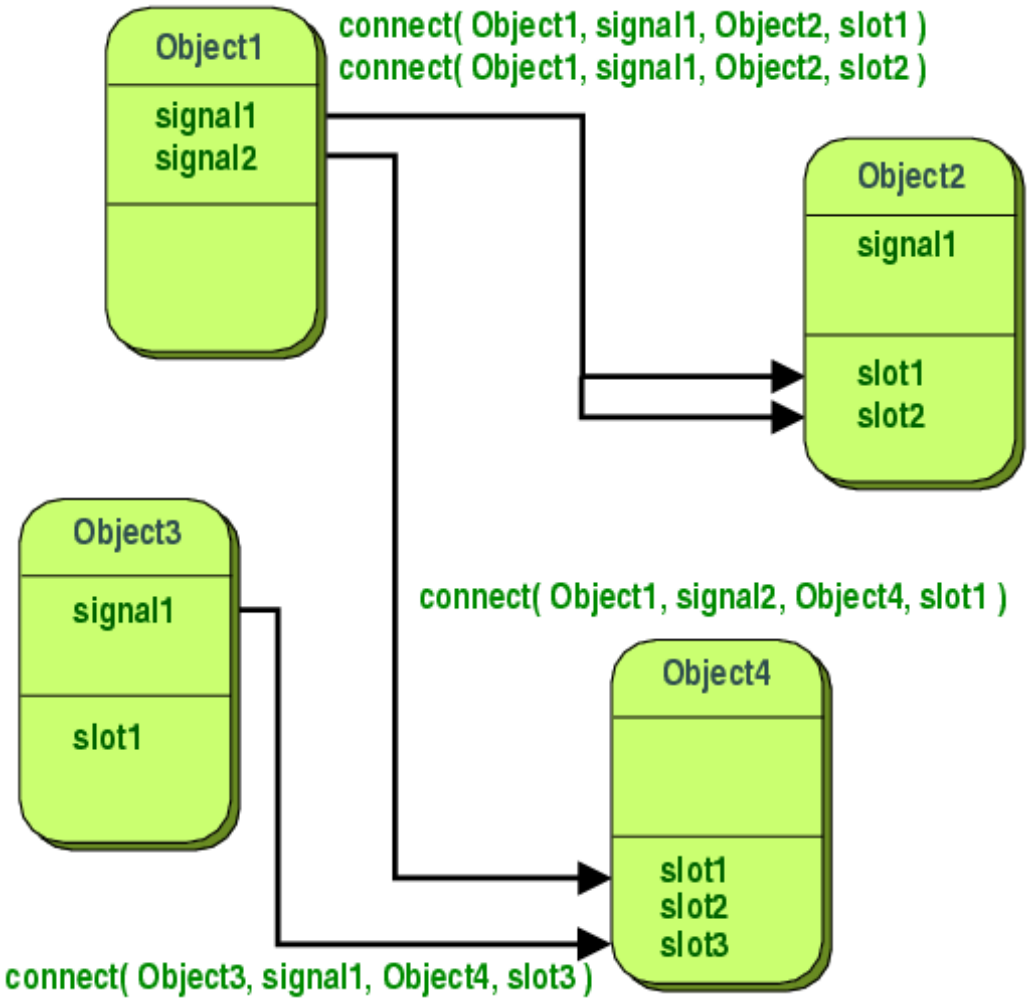

Figure 9.1: connections

一个源自Qt5Doc的简单例子

一个简单的C++版本计数器如下：

```
class Counter
{
public:
    Counter() { m_value = 0; }

    int value() const { return m_value; }
    void setValue(int value);

private:
    int m_value;
};
```

而这段代码基于QObject技术实现的版本如下：

```
#include <QObject>

class Counter : public QObject
{
    Q_OBJECT

public:
    Counter() { m_value = 0; }

    int value() const { return m_value; }

public slots:
    void setValue(int value);

signals:
    void valueChanged(int newValue);

private:
    int m_value;
};
```

这个使用QObject的版本有同样的内部状态，并提供公共方法来访问状态，但它同时也支持使用信号和槽进行组件编程。这个类可以通过发射一个valueChanged()信号告诉其它模块，它的值已经改变了。它有一个槽，其他对象可以发送信号到这个槽。

注意：所有需要定义信号或槽的类都必须在类的声明的顶部包含Q_OBJECT宏。同时，它们也必须直接或间接继承QObject类

通过对比以上两个Counter类的设计，我们可以发现使用QObject实现的Counter类更像一个独立的组件。它更容易和别的组件协同工作。打个比方，QObject版本就像一个独立的具有计数技能的人，他会高声宣布自己的计数改变了，也可以在听到他所关心的人高声宣布的信息是改变自己的计数。但至于是吼出信息的人是男的女的老的少的，他好不关心，他只关心信号本身所携带的信息。同时，他也不关系到底谁关注他吼的信息，他只需吼出来就可以。因而他可以和任何人一起工作。

组件的槽由程序员实现。例如，我们可以这样实现setValue(int value)槽：

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

emit valueChanged(value)这一行以value的值为参数发出valueChanged()信号。在下面这段代码中，我们建立两个Counter对象，然后把第一个对象的valueChanged()信号连接到另一个对象的setValue()槽上。

```
Counter a, b;
QObject::connect(&a, &Counter::valueChanged,
                 &b, &Counter::setValue);

a.setValue(12);     // a.value() == 12, b.value() == 12
b.setValue(48);     // a.value() == 12, b.value() == 48
```

调用a.setValue(12)会发射一个valueChanged(12)信号。然后b的setValue()槽会接收到它。换句话说，b.setValue(12)会在此时被调用。接着，b也会发射valueChanged()信号，但由于没有槽和b的信号相连，b的信号会被自动无视掉。

需要注意的是，setValue()设置value后，只有当value不等于m_value时才会发射信号。这是为了避免循环连接导致的无限循环。(例如：b.valueChanged()再被连接到a.setValue())。也就是说， Qt会自动处理循环连接的问题。

默认情况下，你每形成一个连接，就会发出一个信号。重复的连接会发出两个信号。你可以通过调用disconnect()函数断开连接。如果你向connect()传入Qt::UniqueConnection参数，那么只有当以前没有建立过连接时，连接才会被建立。如果连接已经重复了(同一个信号连到了同一个对象的同一个槽上多次)，连接会失败，然后返回false。

这个例子是为了说明，QObject对象可以在不需要知道其它QObject对象任何信息的情况下和其一起工作。你只需要将它们连接在一起就可以实现让它们协同工作。想象一下，只需要调用几个connect函数就可以让许多强大的Qt部件协同工作，完成一个软件是多么惬意的一件事。

连接信号和槽

我们接着回到我们之前的项目中。我们现在使用信号和槽机制为我们的编辑器增加退出功能。退出也就是当用户出发退出动作后，主窗口自动关闭。也即退出动作发出一个表明自己被触发了的信号后，主窗口自动关闭即可。

也就是说，我们只需要把退出动作被触发的信号连接到主窗口的close()槽上即可。

```
//quit
act[quit]=new QAction(tr("quit"),this);
connect(act[quit],SIGNAL(triggered()),this,SLOT(close()));
```

只需要这样改写一下，再编译运行，即可实现退出功能。

如我们的例子中所见，连接信号和槽的函数叫connect。那么，这个函数具体有什么作用，又有那些用法。让我们来进入Qt5的文档一探究竟。

connect函数一共有五种形式：

```
QMetaObject::Connection QObject::connect(
    const QObject * sender, const char * signal,
    const QObject * receiver, const char * method,
    Qt::ConnectionType type = Qt::AutoConnection) [static]

QMetaObject::Connection QObject::connect(
    const QObject * sender, const QMetaMethod & signal,
    const QObject * receiver, const QMetaMethod & method,
    Qt::ConnectionType type = Qt::AutoConnection) [static]

QMetaObject::Connection QObject::connect(
    const QObject * sender, const char * signal, const char * method,
```

```
    Qt::ConnectionType type = Qt::AutoConnection) const

QMetaObject::Connection QObject::connect(
    const QObject * sender, PointerToMemberFunction signal,
    const QObject * receiver, PointerToMemberFunction method,
    Qt::ConnectionType type) [static]

QMetaObject::Connection QObject::connect(
   const QObject * sender, PointerToMemberFunction signal, Functor functor) [static]
```

下面让我们来一个一个的学习。