# Linux下嵌入式Qt开发学习总结报告

# 第一章：基础

## 1.1 GUI程序原理分析

图形界面应用程序的特点Graphic Use Interface

- 是一种**基于消息驱动模型**的可执行程序
- 程序的执行**依赖于用户的交互过程**
- 程序执行过程中**实时响应用户操作**
- 一般情况下**程序执行后不会主动退出**

---

**图形界面应用程序的运行模式：**

```
main()
{
    定义主窗口 → 创建主窗口 → 创建主窗口中的元素 → 显示主窗口
    while(1)
    {
        → 进入消息循环
    }
}
```

---

图形界面应用程序的消息处理模型：

用户操作，操作系统会检测到事件，操作系统内核发送系统消息到应用程序，应用程序中调用消息处理函数

---

**图形界面应用程序适用于：**

- 多任务的场合
- 强用户交互的场合
- 非专业计算机用户

图形界面应用程序是当代计算机系统中的主要程序类型

---

命令行应用程序与图形界面应用程序区别

| 命令行应用程序 | 图形界面应用程序区别 |
| --- | --- |
| 基于顺序执行结构 | 基于消息驱动模型 |
| 弱交互执行 | 强交互执行 |
| 由用户触发运行 | 由用户触发运行 |
| 主动结束 | 由用户触发结束 |

---

现代操作系统提供原生SDK支持GUI程序开发

不同操作系统上的GUI SDK不同，但是GUI开发原理相同

GUI程序开发原理

- GUI程序正在运行时会**创建一个消息队列**
- 系统内核将**用户操作**翻译成为对应的**程序消息**
- 程序在运行过程中需要**实时处理队列中的消息**
- 当队列中没有消息时，程序将处于**停滞状态**

> 用户操作 → 操作系统内核 → 程序消息 → GUI应用程序

---

GUI 程序开发

1. 在代码中用程序创建窗口及窗口元素
2. 在消息处理函数中根据程序消息做出不同响应

> **经典GUI程序开发模式：可视化界面开发 + 消息映射**
>
> 1. 通过所见即所得的方式“画”出界面；开发环境自动生成对应的程序代码
> 2. 程序中将具体消息映射到指定函数；当消息触发时，函数被调用

## 1.2 Qt的诞生和本质

---

GUI用户界面是由固定的窗口元素所构成的

> 用户界面元素：主窗口、菜单栏、工具栏、标签、文本框、按钮等等

---

操作系统提供了创建用户界面元素所需的函数，各种函数依次调用，从而创建出界面元素

但是操作系统提供的**原生函数无法直接映射到界面元素**

因此使用**面向对象**的C++语言，面向对象方法学视角下

- 将**界面元素**定义为**类**
- 通过**抽象**和**封装**隐藏界面元素的细节
- 所有的**界面元素**都可以看作**实际的对象**
- GUI用户界面是**由各个不同的对象组成**
- 应用程序创建过程就是**组合不同界面元素对象**的过程

> 计算机图形框架的C++解决方案：Qt，MFC（Window专用）

---

**Qt的本质**

- 利用**面向对象**方法学开发的一套**GUI组件库**
- 将不同操作系统的**GUI细节封装于类的内部**
- 提供一套**跨平台的类**用于开发GUI抽象
- 遵循经典的GUI应用程序开发模式：**可视化界面开发 + 消息映射**

---

命令行编译Qt源程序（只有 .cpp .h 文件）步骤

```
1. qmake -project // 根据当前目录中的源码（.cpp）生成工程文件(.pro)
2. qmake    // 根据工程文件生成makefile文件
3. make    // 根据makefile进行编译，生成.exe和.o
```

**Qt Creator是一套可视化的集成开发环境**，可以打开.pro文件，以**工程项目的方式**对源码进行管理

Creator工程构成

- .pro 项目描述文件
- .pro.user 用户配置描述文件
- .h .cpp 源码

- .ui 界面描述文件
- 资源文件（图片、音频等）

在.pro文件中

| # | 注释起始符 |
| --- | --- |
| QT | 模块声明 |
| TARGET | 可执行文件名 |
| TEMPLATE | 程序模板声明 |
| SOURCES | 源码文件声明 |
| HEADERS | 头文件声明 |
| FORMS | 界面文件声明 |
| RC_FILE | 资源文件声明 |

.pro本质就是Qt中与平台无关的Makefile文件

.pro中的高级变量

| INCLUDEPATH | 头文件搜索路径 |
| --- | --- |
| CONFIG | 设定项目的**配置信息**和**编译选项** |
| LIBS | 添加第三方库文件 |
| DEFINES | 定义编译宏 |

---

CONFIG的常用选项

- debug
- release
- debug_and_release 同时构建debug版和release版本的可执行程序
- warn_on
- warn_off 不输出警告信息

```
CONFIG += warn_on debug
CONFIG(debug){
    DEFINES += DEBUG_LOG
    SOURCES += DebugLog.cpp
    HEADERS += DebugLog.h
}
```

跨平台开发原理

> .pro文件 → qmake → 生成Makefile（对于不同系统生成不同的）

一般使用相对路径，Creator在打开项目文件的同时会生成.pro.user文件（包含本地配置信息），因此，在不同计算机之间移动项目源码时，建议删除.pro.user

---

窗口组件和窗口类型

图形用户界面由窗口和窗口组件构成

Qt以组件对象（头文件中包含GUI模块）的方式构建图形用户界面

窗口：没有父组件的顶级组件

组件类型：

- 容器类（父组件）：用于包含其他界面组件，如 widget，frame
- 功能类（子组件）：用于实现特定的交互功能，如 label，pushbutton

---

窗口组件

QWidget类继承自QObject类和QPintDevice类

- QObject是所有支持Qt对象模型的基类
- QPaintDevice是Qt中所有可绘制组件的基类

> QLabel、QLineEdit、QPushButton等 →（继承自）→ QWidget → QObject、QPintDevice

QWidget能够绘制自己和处理用户的输入，是**所有窗口组件类的父类**，是**所有窗口组件的抽象**，每一个窗口组件都是一个QWidget

**窗口类型**

- Qt::Dialog 对话框类型
- Qt::Window 主窗口类型
- Qt::SplashScreen 用来做欢迎界面，启动动画类型

**窗口标志**

- Qt::WindowStaysOnTopHint 保持在顶部，相当于一直贴在屏幕上
- Qt::WindwoContextHelpButtonHint 帮助替换了最大最小化

```
QWidget w(NULL,Qt::SplashScreen);
QWidget w(NULL,Qt::Window | WindowStaysOnTopHint);
QWidget w(NULL,Qt::Dialog);
```

## 1.3 Qt的坐标系统

```
qDebug()<<"x = "<<w.x()<<"y="<<w.y();
qDebug()<<"width = "<<w.width()<<"y="<<w.height();
qDebug()<<"geometry x = "<<w.geometry().x()<<"geometry y="
<<w.geometry().y();
qDebug()<<"geometry width = "<<w.geometry().width()<<"geometry y="
<<w.geometry().height();
qDebug()<<"frameGeometry x = "<<w.frameGeometry().x()<<"y="
<<w.frameGeometry().y();
qDebug()<<"frameGeometry width = "<<w.frameGeometry().width()<<"y="
<<w.frameGeometry().height();
```

改变窗口部件的大小

- resize(in w, int h); resize(const QSize&)

改变位置

- move(in x, int y); move(const QPoint&)

---

快捷键

| ctrl + shift + r | 预览ui |
| --- | --- |
| ctrl + h | 水平布局 |
| ctrl + l | 垂直布局 |
| | |
| | |
| | |
| | |
| | |

## 1.4 软件开发中代码重构

重构Refactoring，对软件系统进行重写架构

以**改善代码质量**为目的代码重写

- 使其软件的**设计和框架更加合理**
- 提高软件的**扩展性**和**维护性**

---

重构与实现区别

代码实现：按照设计编程实现，**重心在于功能实现**，不考虑架构的好坏，只考虑功能的实现

代码重构：以提高代码质量为目的**软件架构优化**，不能影响已实现的功能，只考虑框架的改善

---

从工程角度的软件开发过程

> 需求分析 → 功能分解 → 功能实现 → 功能测试 → （重构 → 功能测试）→ 系统测试 → 最终发布

重构是维持代码质量在可接受范围内的重要方式

重构时间和时机：当重复的代码越来越多，代码功能越来越不清晰，代码离设计越来越远

---

## 二阶构造法

```
class Test9Calculator : public QWidget
{
	Q_OBJECT
public:
	~Test9Calculator();
	void show();
	static Test9Calculator* NewInstance();
private:
	Test9Calculator(QWidget *parent = 0);
	bool construct();
	QLineEdit* MyLineEdit = 0;
	QPushButton* MyBtn = 0;
};
```

```
#include "test9calculator.h"
Test9Calculator::Test9Calculator(QWidget *parent)
	:
QWidget(parent,Qt::WindowCloseButtonHint|Qt::WindowMaximizeButtonHint)
{
}
Test9Calculator::~Test9Calculator()
{

}

void Test9Calculator::show()
{
	this->move(300,300);
	this->setFixedSize(280,300);
	QWidget::show();
}

Test9Calculator* Test9Calculator::NewInstance()
{
	Test9Calculator* ret = new Test9Calculator();
	if( (ret == NULL) || (!ret->construct()) )
	{
		delete ret;
		ret = NULL;
	}
```

```
        return ret;
}
bool Test9Calculator::construct()
{
        bool ret = true;
        MyLineEdit = new QLineEdit(this);
        if(MyLineEdit != NULL)
        {
                MyLineEdit->resize(240,30);
        }
        else
        {
                ret = false;
        }
        MyBtn = new QPushButton(this);
        if(MyBtn != NULL)
        {
                MyBtn->setText(StrList[i]);
        }
        else
        {
             ret = false;
        }
        return ret;
}
```

## 1.5 消息处理

Qt遵循经典的GUI消息驱动事件模型，封装了具体操作系统的消息机制

> 用户事件 → 操作系统 → （翻译为应用程序消息） → 应用程序（消息处理函数）

信号到槽的连接必须发生在两个Qt对象之间

- **SIGNAL** 由操作系统产生的消息，用字符串（const char*）进行描述
- **SLOT** 程序中的消息处理函数
- **Connect** 将系统消息绑定到消息处理函数

信号槽依赖于 Q_OBJECT 宏，只有 Qt 类才能定义信号

---

信号与槽的对应关系

> 单个信号 → 单个槽（一对一）
>
> 一个信号 → 多个槽（一对多）
>
> 多个信号 → 单个槽（多对一）
>
> 单个信号 → 单个信号（转嫁）

---

信号本质是什么，如何发射的

- 信号只是一个**特殊的成员函数声明**（必须使用 signals 关键字进行声明，返回值是 void，只能说明不能定义）
- 函数访问属性**自动**被设置为 **protected**
- 只能通过 **emit** 关键字调用函数（发射信号）

信号槽军规

- Qt 类只能在头文件中声明
- 信号与槽原型应该完全相同，信号参数更多时，多余的参数被忽略
- 槽函数的返回值必须是 void
- 槽函数可以像普通函数一样被调用
- **信号与槽的访问属性对于 connect/disconnect 无效**

信号槽意义

- 最大限度**弱化**类之间的耦合关系，使得类间关系松散，提高类的可复用性
- 在设计阶段，可以减少不必要的接口类（抽象类）
- 在开发阶段，对象间的交互通过信号与槽动态绑定

---

连接方式：connect() 第五个参数，连接方式决定槽函数调用时候的相关行为

1. **立即调用**　Qt::DirectConnection
   直接在发送信号的线程中调用槽函数，等价于槽函数的**实时调用**（在 emit 信号处），**忽略掉线程依附性**
2. **异步调用**　Qt::QueuedConnection
   信号发送到目标线程的事件队列，由目标线程处理，当前线程继续向下执行
3. **同步调用**　Qt::BlockingQueuedConnection

信号发送到目标线程的事件队列，由目标线程处理；当前线程等待槽函数返回，之后继续向下执行

注意：**目标线程和当前线程必须不同**，如果相同，那当前发送信号的线程永远无法向下执行

4. **默认连接** Qt::AutoConnection，自动确定连接类型
   情况一：等价于 DirecConnection，发送线程= 接收线程
   情况二：等价于 QueuedConnection，发送线程 ！ = 接收线程
5. **单一连接** Qt::UniqueConnection

功能与AutoConnection相同，自动确定连接类型

同一个信号与同一个槽函数之间只有一个连接

默认情况：同一个信号可以多次连接到同一个槽函数，这时信号发送时，槽函数会被多次调用

---

- 每一个线程都有自己的事件队列
- 线程通过事件队列接收信号
- 信号在事件循环中被处理
- 发送信号，信号进入接收者对象所依附线程的事件队列

## 1.6 QString字符串类

**C缺陷：不支持真正意义上的字符串，用字符数组和一组函数实现字符串操作，不支持自定义类型，因此无法获得字符串类型**

C到C++的进化过程引入了自定义类型

STL是意义上需要与c++一同发布的标准库

STL是一套以模板技术完成的C++类库，包含常用的算法和数据结构，包含了字符串类

**STL的缺陷：具体实现依赖于编译器生产厂商**

STL的标准只是**其接口是标准的**：相同的全局函数，相同的算法类和数据结构类，相同的类成员函数

依赖于STL开发的C++程序在不同平台上的行为可能不同

因此，需要跨平台时，不使用STL库，而使用QString，更适合跨平台开发的场景，更强大易用

QString特点

- 采用Unicode编码，STL采用ASCII编码

- 使用隐式共享技术来节省内存和不必要的数据拷贝，是集合了深拷贝和浅拷贝的优点于一身
- 跨平台使用，不必考虑字符串的平台兼容性

支持字符串和数组的相互转换，支持字符串大小比较，支持不同字符编码间的相互转换，支持std::string和std::wstring的相互转换，支持正则表达式

### 1.6.1 符号上标

| 上标 | Unicode | |
|---|---|---|
| 0 | 2070 | C++ |
| 1 | 00B9 | \xE2\81\80 |
| 2 | 00B2 | \xC2\xB9 |
| 3 | 00B3 | \xC2\xB2 |
| 4 | 2074 | \xC2\xB3 |
| 5 | 2075 | \xE2\81\B4 |
| 6 | 2076 | \xE2\81\B5 |
| 7 | 2077 | \xE2\81\B6 |
| 8 | 2078 | \xE2\81\B7 |
| 9 | 2079 | \xE2\81\B8 |

```
// 方法一
QString str = QString::fromUtf8("m\xC2\xB2");
// 方法二
quint16 square = 0xB2;
QString paintStr = "m" + QString::fromUtf16(&square, 1);
```

## 1.7 设计原则

界面与逻辑

用户界面模块UI：接受用户输入及呈现数据

业务逻辑模块：根据用户需求处理数据

---

基本设计原则

> 功能模块之间需要进行解耦，**强内聚**，**弱耦合**

- 每个模块只实现**单一功能**
- 模块内部的**子模块只为整体的单一功能而存在**
- 模块之间通过约定好的**接口进行交互**

接口：（1）广义：一种**契约**（协议，语法，格式等）（2）狭义：面向对象，接口是**一组预定义的函数原型**；面向对象中接口是**纯虚类**

用户界面与业务逻辑，用户界面使用业务接口，业务逻辑实现业务接口

模块之间仅通过接口进行关联（有模块使用，也有模块实现该接口）

模块之间的关系是单项依赖的（避免模块间存在循环依赖的情况）

# 第二章：Qt

---

## 2.1 Qt对象间的父子关系

Qt对象（继承QObject）间存在父子关系：（1）每一个对象保存它所有子对象的指针（2）每个对象保存指向其父对象的指针

```
QObject* p = new QObject();
      QObject* c1 = new QObject(p);
      QObject* c2 = new QObject(p);
      qDebug()<<"p: "<<p;   // p:  QObject(0x55f32ce1eef0)
      qDebug()<<"c1: "<<c1;   //c1:  QObject(0x55f32ce1ef90)
      qDebug()<<"c2: "<<c2;   // c2:  QObject(0x55f32ce1f060)
      const QObjectList& list = p->children();
      for(int i = 0; i<list.length();++i)
     {
            qDebug()<<i<<": "<<list[i];   //0 :  QObject(0x55f32ce1ef90)  1
:  QObject(0x55f32ce1f060)
     }
      qDebug()<<"parent of c1: "<<c1->parent();   //parent of c1:
QObject(0x55f32ce1eef0)
      qDebug()<<"parent of c2: "<<c2->parent();   //parent of c2:
QObject(0x55f32ce1eef0)
```

---

当Qt对象被销毁时（1）将自己从父对象的 Children List移除，解除父子关系（2）将自己的Children List中的所有对象销毁

## 2.2 Dialog

---

对话框是与用户**简短交互**的顶层窗口

QDialog是所有对话框类的基类，QDialog继承于QWidget，是一种容器类型的组件

意义

- 作为一种**专用交互窗口**而存在
- **不能作为子部件**嵌入其它容器
- 是定制了窗口式样的特殊QWidget

1.模态对话框（QDialog::exec()）

- 显示后无法与父窗口交互
- 是一种阻塞式的对话框调用方式

常用于消息提示、文件选择、打印设置等

2.非模态对话框（QDialog::show()）

- 显示后独立存在可以同时和父窗口交互
- 非阻塞式的对话框调用方式

用于查找操作、属性功能设置等

在**栈**上创建**模态对话框**，堆上创建非模态

通过QDialog::setModal可以创建混合特性的对话框

**非模态对话框需要指定 Qt::WA_DeleteOnClose属性**

dialog->setAttribute(Qt::WA_DeleteOnClose);

```
QDialog* Dia = new Dialog(this);
Dia->setModal(this);   // 具有模态的特性
Dia->show()
```

Dialog::exec()的返回值为交互结果

自制控件（继承与Dialog），可以使用done(Accept);来指定返回，自定义返回done(100);

```
int r= Dlg.exec();
if(QDialog::Accepted == r){}
else if(QDialog::Rejected == r){}
return r;   // 不是 return a.exec();，因为 Dlg.exec()已经进入了消息循环
// 如果 return a.exec(); 那么关闭窗口，程序还在运行，因为进入了两次消息循环
```

---

可复用登录对话框

附加需求：随机验证码

```
name= LineEdit1.text.trimmed();   // 去除前后空格
```

---

Qt中的标准对话框

---

消息对话框 QMessageBox

```
QMessageBox msg(this);
msg.setStandardButtons(QMessage::Ok | QMessage::Cancel |
QMessage::YesToAll);
if(msg.exec() == QMessage::Ok){}
```

---

## 文件对话框 QFileDialog

```
QFileDialog dlg(this);
dlg.setAcceptMode(QFileDialog::AcceptOpen);   //QFileDialog::AcceptSave保存文件
dlg.setFileMode(QFileDialog::ExistingFile);   // 单选，多选ExistingFiles
if( dlg.exec() == QFileDialog::Accepted)
{
    QStringList fs = dlg.selectedFiles();
}
// 文件过滤
JPG   *.jpg*
Text(*.txt)
All   *.*
dlg.setFile
```

---

## 颜色对话框 QColorDialog

QColor支持多种颜色表示

- RGB：以红绿蓝为基准的三色模型
- HSV：以色调、饱和度、明度为基准的六角锥体模型
- CMYK：以天蓝、品红、黄色、黑为基准的全彩印刷色彩模型

```
QColorDialog dlg(this);
dlg.setCurrentColor(Qt::red);
if( dlg.exec() == QColorDialog::Accepted)
{
    qDebug()<<dlg.selectedColor();   // QColor(ARGB 1, 1, 0, 0) ，第一个1是透明度100%（即不透明），QColor.red()单个颜色
}
```

## 输入对话框 QInputDialog

```
QInputDialog dlg(this);
dlg.setInputMode(QInputDialog::IntInput);    // TextInput IntInput DoubleInput
if( dlg.exec() == QInputDialog::Accepted)
{
    qDebug()<<dlg.intValue();
}
```

实用函数 getInt

## 字体对话框 QFontDialog

```
QFontDialog dlg(this);
dlg.setCurrentFont(QFont("Kinnari",10,QFont::Bold));
if( dlg.exec() == QFontDialog::Accepted)
{
    qDebug()<<dlg.selectedFont();
}

// 实用函数
bool ok;
QFont font = QFontDialog::getFont(&ok,QFont("Times",12),this);
qDebug()<<"font:"<<font;
```

## 进度对话框 QProgressDialog

用于显示进度信息，用于需要用户等待的场合

```
 QProgressDialog dlg(this);
      dlg.setWindowTitle("progressdialog");
      dlg.setLabelText("downloading from server...");
      dlg.setMinimum(0);
      dlg.setMaximum(100);
      dlg.setValue(35);
      // create a new thread

      dlg.exec();
```

## 打印对话框 QPrintDialog

用于设置打印相关的参数信息

```
QPrintDialog dlg(this);
dlg.setWindowTitle("print dialog"));
if( dlg.exec() == QPrintDialog::Accepted)
{
    QPrinter* p = dlg.printer();
    QTextDocument td;
    td.setPlainText("printer object test.");
    td.print(p);
      td.setHtml("<h1>Print html object test</h1>");
      p->setOutputFileName("/test.pdf");
}
```

界面部件产生数据对象，业务逻辑中的其他对象使用数据对象，GUI界面与业务逻辑通过数据对象相连

## 2.3 布局管理器

在像素级指定位置和大小move，resize 等，是绝对定位，不会随着窗口的缩放而改变

**绝对定位的布局方式无法自适应窗口的变化**

解决方案：布局管理器，自动排列窗口组件，自动更新组件大小

不是界面部件，而是界面部件的定位策略

同一布局管理器中的组件拥有相同的父组件，组件间的父子关系是Qt内存管理的重要方式

QLayout是抽象基类

QBoxLayout（QVBoxLayout与QHBoxLayout）、QGridLayout、QFormLayout、QStackdLayout

```
QVBoxLayout* layout = new QVBoxLayout();
layout->setSpacing(30);
layout->addWidget(btn1);
layout->addWidget(btn2);
layout->addWidget(btn3);
layout->addWidget(btn);
setLayout(layout);    // 执行之后 btn 会成为 this 的子部件，同一布局管理器中的组件拥有相同的父组件
```

---

布局管理器可以相互嵌套，形成更加复杂的布局方式，自定义布局类可以达到个性化界面布局的效果

```
QVBoxLayout* layoutv1 = new QVBoxLayout();
layoutv1->setSpacing(30);
layoutv1->addWidget(btn1);
layoutv1->addWidget(btn2);

QVBoxLayout* layoutv2 = new QVBoxLayout();
layoutv2->setSpacing(30);
layoutv2->addWidget(btn);
layoutv2->addWidget(btn3);

QHBoxLayout* layouth = new QHBoxLayout();
```

```
layouth->setSpacing(30);
layouth->addLayout(layoutv1);
layouth->addLayout(layoutv2);
setLayout(layouth);
```

---

自定义组件大小更新时的比例系数

```
layout->setStretchFactor(btn1,1);
layout->setStretchFactor(btn2,2);
layout->setStretchFactor(btn3,1);
layout->setStretchFactor(btn,4);
```

---

QGridLayout 以网格（二维）的方式管理界面组件

```
QGridLayout* layout = new QGridLayout();
layout->setSpacing(10);
layout->addWidget(btn1,0,0);   //指定坐标（0，0），也可以指定大小 layout->addWidget(btn1,0,0,2,1);占两行一列
layout->addWidget(btn2,0,1);
layout->addWidget(btn,1,0);
layout->addWidget(btn3,1,1);
layout->setRowStretch(0,1);
layout->setRowStretch(1,2);   // 第一行：第二行 = 1：2
setLayout(layout);
```

---

QFormLayout 表单布局管理器 嵌入式中最常用

以表单form的方式管理界面组件

表单布局中的标签和组件是相互对应的关系 [标签1，组件1]

```
QFormLayout* layout = new QFormLayout();
layout->setSpacing(10);
layout->addRow("name:",lineedit1);
layout->addRow("age:",lineedit2);
layout->addRow("sex:",lineedit3);
setLayout(layout);
```

样式函数

```
layout->setRowWrapPolicy(QFormLayout::WrapAllRows);   // 分行显示，一行文字，一行输入框
layout->setLabelAlignment(Qt::AlignRight);   // 文本对齐
```

---

QStackedLayout 栈式布局管理器，不能嵌套其他布局管理器

所有组件垂直于屏幕的方向，每次仅能显示一个组件

组件大小一致，能够自由切换需要显示的组件

```
QStackedLayout* layout = new QStackedLayout();
	QHBoxLayout* layouth = new QHBoxLayout();
	QWidget* widget = new QWidget();

	btn->setParent(widget);
	btn1->setParent(widget);
	layouth->addWidget(btn);
	layouth->addWidget(btn1);
	widget->setLayout(layouth);

	layout->addWidget(widget);
	layout->addWidget(btn2);
	layout->addWidget(btn3);
	layout->setCurrentIndex(0);

	setLayout(layout);
```

---

QTimer 计时器

## 2.4 主窗口

封装有 菜单栏 MenuBar、工具栏 ToolBar、中心组件 CentralWidget、停靠组件 DockWidget、状态栏 StatusBar

QMenuBar 菜单栏，其中有 下拉菜单组 QMenu，菜单项 QAction

QToolBar 工具栏，容器类型组件，其中有QAction，QToolBar 中可以加任意widget组件，如pushbutton、label、lineedit

```
QToolBar* tb = addToolBar("Tool Bar");
tb->setFloatable(false);
tb->setMovable(false);
tb->setIconSize(QSize(16,16));
QAction* action = new QAction("",NULL);
action->setToolTip("Open");
action->setIcon(QIcon(":/res/open.png"));
tb->addAction(action);
```

QStatusBar 状态栏，容器类型组件，可以添加任意组件进去

输出简要信息的区域，一般位于最底部，消息类型一般为：

实时信息（显示在左边区域 addWidget）：如当前程序状态

永久消息（显示在右边区域 addPermanentWidget）：如程序版本号，机构名称

进度消息：如进度条提示，百分比提示

```
QStatusBar* sb = statusBar();
QLabel* label = new QLabel("label");
QLineEdit* ed = new QLineEdit();
QPushButton* btn = new QPushButton();
sb->addWidget(label);
sb->addPermanentWidget(ed);
sb->addPermanentWidget(btn);
label->setText("ZZH");
sb->show();
```

QCentralWidget 文本编辑组件

QLineEdit、QTextEdit（多行富文本、支持图片音频，html 等）、
QPainTextEdit（多行普通文本）

内置功能

右键弹出式菜单，快捷键功能（复制粘贴等）

```
QTextEdit* te = new QTextEdit(this);
te->insertPlainText("\n\nok");
QString str = "<img
src=\"D:\\File\\Ubuntu1804Shared\\DTSoftwareQt\\Test27\\Test27\\Icons\\got
o.png\" />";
te->insertHtml(str);

QPlainTextEdit* pe = new QPlainTextEdit(this);
pe->insertPlainText("\n\nok");
```

避免过度设计

## 2.5 文件操作

Qt中IO操作的处理方式

通过**统一的接口**简化了文件与外部设备的操作方式

文件被看作一种特殊的外部设备

Qt的文件操作和外部设备的操作相同

```
bool open(OpenMode)
QByteArray read(qint64 maxsize)
qint64 write(const QByteArray&)
void close()
```

IO设备的类型

1. **顺序读取设备**：只能从头开始顺序读写数据，不能指定数据的读写位置，如串口设备
2. **随机存取设备**：可以定位到任意位置进行数据的读写 seek function，如文件

文件 QFile，串口 QBuffer ，网络编程 QAbstractSocket，进程间多进程编程 QProcess 都是IO设备

```
file.open(QIODevice::WriteOnly | QIODevice::Text)    // 以文本的形式打开，默
认是以字节方式
// 没有文件则会创建文件，有该文件则会清空文件内容
file.
```

QTemporaryFile 临时文件操作类

- 安全地创建要给**全局唯一**的临时文件
- **当对象销毁时对应的临时文件将被删除**
- 临时文件的打开方式为 **QIODevice::ReadWrite**
- 临时文件常用于**大数据传递**或者**进程间通信**的场合

---

文本流和数据流

两个辅助类，用于IO设备的

人类角度文件类型

文本文件：内容是可读的文本字符

数据文件：内容是直接的二进制数据

QFile直接支持文本文件和数据文件，因为直接操作的都是基于字节

如何将浮动数据写入文本文件和数据文件

---

错误示例：不使用辅助类遇到的问题

```
QString str =
"/mnt/hgfs/Ubuntu1804Shared/DTSoftwareQt/Lesson29/Lesson29/demo.txt";
QFile file(str);
if(file.open(QIODevice::WriteOnly))      // 默认是写二进制文件
{
    QString dt = "zzh";
    double value = 3.14;
    file.write(dt);   // ERROR
    file.write(&value,sizeof(value));   // ERROR
    file.close();
}
// 以下才为正确
file.write(dt.toStdString().c_str()); // 先转化为std标准的字符串，再转化为二
进制数据，最后得到一个指针（指向该二进制数据）
file.write(reinterpret_cast<char*>(&value),sizeof(value)); //
// 打开文件会显示乱码，因为是二进制数据



/*--------------------------*/
if(file.open(QIODevice::ReadOnly))   // 默认是读二进制文件
{
    QString dt = "";
    double value = 0;
```

```
    dt = file.read(3);   //  == dt = (QString)file.read(3);  read 返回的是
QByteArray字节数据
    file.read(&value,sizeof(value));   // ERROR
    qDebug()<<"dt = "<<dt<<"value = "<<value;
}
 // 以下才为正确
 file.read(reinterpret_cast<char *>(&value),sizeof(value));
```

---

使用辅助类

QTextStream 写入的数据全部**转换为可读文本**，用于**文本数据的快速读写**

QDataStream 写入的数据根据类型**转换为二进制数据**，用于**二进制数据的快速读写**

IO设备使用辅助类的方式

```
// 1. 创建IO设备的对象，如创建QFile文件对象 file
// 2. 使用file 对象打开文件
// 3. 将数据写入文件
QXXXXStream out(&file);
out<<QString("zzh");
out<<QString("Result: ")<<3.14;

// 4. 将数据从文件读出
QXXXXStream in(&file);
in>>dt;
in>>result;
in>>value;
```

```
// 使用 QTextStream
void text_stream_test(QString f)
{
      QFile file(f);
      // 使用 QTextStream 写入
      if(file.open(QIODevice::WriteOnly | QIODevice::Text))   // 使用的是
QTextStream，因此必须加 QIODevice::Text
     {
            QTextStream out(&file);
            out<<QString("zzh")<<endl;
            out<<QString("Result:")<<endl;
            out<<5<<"*"<<6<<"="<<5*6<<endl;
            file.close();
     }
      // 使用 QTextStream 读取
```

```
        if(file.open(QIODevice::ReadOnly | QIODevice::Text ))
    {
            QTextStream in(&file);
            while(!in.atEnd())
            {
                QString line = in.readLine();
                qDebug()<<line;
            }
            file.close();
    }
}

// 使用 QDataStream
void data_stream_test(QString f)
{
     QFile file(f);
     // 使用 QDataStream 写入，写入的是二进制数据，因此是乱码的
     if(file.open(QIODevice::WriteOnly ))
    {
            QDataStream out(&file);
            out<<QString("zzh");
            out<<QString("Result:");
            out<<5.14;
            file.close();
    }
     // 使用 QDataStream 读取
     if(file.open(QIODevice::ReadOnly  ))
    {
            QDataStream in(&file);
            QString dt;
            QString result;
            double value;
            in>>dt;
            in>>result;
            in>>value;
            qDebug()<<"dt = "<<dt<<"result = "<<result<<"value = "<<value;
            file.close();
    }
}
```

---

注意：

不同Qt版本的数据流文件格式可能不同，**当数据流文件要在不同版本程序中传递数据时，需要考虑版本问题**

void setVersion(int v) // 设置读写版本号，如：
out.setVersion(QDataStream::Qt_4_7)  in.setVersion(QDataStream::Qt_4_7)

int version() const  // 获取读写版本号

## 2.5.1 缓冲区操作与目录操作

计算机中有外部缓冲区（位于外部设备中）和内部缓冲区

Qt中缓冲区的本质为**一段连续的存储空间**

**QBuffer**类，缓冲区可看作**一种特殊的 IO 设备**，文本辅助类可以直接用于操作缓冲区

```
QByteArray arr;
QBuffer buffer(&arr);
if(buffer.open(QIODevice::WriteOnly))
{
    QDataStream out(&buffer);
    out<<QString("ZZH");
    out<<3.14;
    buffer.close();
}
```

---

QBuffer 缓冲区的使用场合：

1. 在线程间进行不同类型的**数据传递**
2. **缓存外部设备**中的数据返回
3. **数据读取速度小于数据写入速度**

涉及到大量数据的读写时候

```
// 使用缓冲区读写不同类型数据
void WriteBuffer(int type,QBuffer& buffer)
{
    if(buffer.open(QIODevice::WriteOnly))
    {
        QDataStream out(&buffer);
        out<<type;
        if(0==type)
        {
            out<<QString("ZZH");
            out<<QString("3.1415926");
        }
```

```
            else{
                out<<3;
                out<<1415926;
            }

            buffer.close();
        }
    }
    void ReadBuffer(QBuffer& buffer)
    {
        if(buffer.open(QIODevice::ReadOnly))
        {
            int type = -1;
            QDataStream in(&buffer);
            in>>type
            if(0==type)
            {
                QString a,b;
                in>>a;
                in>>ab;
            }
            else{
                int a,b;
                in>>a;
                in>>b;
            }
            buffer.close();
        }
    }
    int main(){
        QByteArray arr;
        QBuffer buffer(&arr);
        WriteBuffer(1,buffer);
        ReadBuffer(buffer);
    }
```

---

目录操作 QDir ‘/’

能够对目录**进行任意操作**（创建、删除、重命名）

能够**获取指定目录中的所有条目**（文件和文件夹）

能够使用**过滤字符串**获取指定条目

能够获取系统的所有根目录

---

QFileSystemWatcher 文件系统监视器

用于监控文件和目录的状态变化

能够监控特定目录和文件的状态

能够同时对**多个目录和文件**进行监控

当目录或者文件**发生改变时将触发信号**

可以通过**信号和槽**的机制捕捉信号并作出响应

```
QFileSystemWatcher   m_wathcer;
m_wathcer.addPath(".///text.txt");   // Qt中 /
connect(&m_watcher,SIGNAL(fileChanged(const QString &)),this
,SLOT(statusChanged()));
connect(&m_watcher,SIGNAL(directoryChanged(const QString &)),this
,SLOT(statusChanged()));
```

## 2.5.2 文本编辑器中的数据存取

界面代码与功能代码分离开

尽量复用平台中提供的相关组件

大部分时间都是写槽函数，相应用户操作，具体功能的触发点

## 2.5.3 文本编辑器的功能交互

QPlainTextEdit 能够触发与编辑操作相关的信号

**void textChanged()** 用于检测数据变化

void cursorPositonChanged()

void copyAviable(bool)

void redoAviable(bool)

void undoAviable(bool)

功能的交互通过状态变量完成

## 2.5.4 QMap与QHash

两个非线性数据结构类，用于存储键值对的类模板

---

QMap 是一个以**升序键顺序**存储**键值对**的数据结构

原型为 class QMap<K,T> 模板

键值对根据 **Key （键大小)进行了排序**

QMap中的 Key 类型必须重载 operator<

使用方式一

```
QMap<QString,int> map;
map.insert("key 2",2);
map.insert("key 0",0);
map.insert("key 1",1);
for(int i=0;i<3;++i)
{
    qDebug()<<map.value("key "+QString::number(i));   // 0  1  2
}
QList<QString> list = map.keys();
for(int i = 0;i<list.count();++i)
{
    qDebug()<<list[i];   //"key 0"    "key 1"    "key 2"
}

QList<int> list2 = map.values();
for(int i = 0;i<list2.count();++i)
{
    qDebug()<<list2[i];   //  0  1  2
}
```

使用方式二

```
QMap<QString,int> map2;
map2["key 2"]=2;
map2["key 0"]=0;
map2["key 1"]=1;
for(int i=0;i<3;++i)
{
    qDebug()<<map2["key "+QString::number(i)];   // 0  1  2
}
```

```
QMapIterator<QString, int> it(map2);   // 初始时，it指向 map2 前一个位置
while(it.hasNext())
{
    it.next();
    qDebug()<<it.key()<<":"<<it.value(); // "key 0" : 0  "key 1" : 1  "key
2" : 2
}
```

插入键值对时

- 当Key存在：更新 Value的值
- 当Key不存在：插入新的键值对

通过Key获取Value时

- 当Key存在：返回对应的 Value
- 当Key不存在：放回类型值所对应的 0 值

---

QHash是哈希数据结构

- 原型为class QHash<K,T> 模板
- 键值对在内部**无序排序**
- Key 类型必须重载 **operator ==**
- Key 对象必须重载全局hash函数 **qHash()**

```
QHash<QString,int> hash;
hash.insert("key 2",2);
hash.insert("key 0",0);
hash.insert("key 1",1);
for(int i=0;i<3;++i)
{
    qDebug()<<hash.value("key "+QString::number(i));   // 0  1  2
}
QList<QString> list = hash.keys();
for(int i = 0;i<list.count();++i)
{
    qDebug()<<list[i];   //"key 1"    "key 0"    "key 2"
}

QList<int> list2 = hash.values();
for(int i = 0;i<list2.count();++i)
{
    qDebug()<<list2[i];   //    1  0    2
}
```

```
QHash<QString,int> hash2;
hash2["key 2"]=2;
hash2["key 0"]=0;
hash2["key 1"]=1;
for(int i=0;i<3;++i)
{
    qDebug()<<hash2["key "+QString::number(i)];   // 0  1  2
}
QHashIterator<QString, int> it(hash2);   // 初始时，it指向 map2 前一个位置
while(it.hasNext())
{
    it.next();
    qDebug()<<it.key()<<":"<<it.value(); // "key 1" : 1      "key 0" : 0
"key 2" : 2
}

QHash<QString,int>::const_iterator i;
for(i=hash2.constBegin();i!=hash2.constEnd();++i)
{
   qDebug()<<i.key()<<":"<<i.value();   //  "key 1" : 1        "key 0" : 0
"key 2" : 2
}
```

---

QMap 与 QHash 接口相同，不同点如下：

- QHash查找速度快于QMap
- QHash占用空间多于QMap
- QHash以任意方式存储元素，以Key顺序存储元素
- QHash的键类型必须提供 operator ==() 和 qHash(key) 函数
- QMap的键类型必须提供 operator<() 函数

## 2.6 事件处理

Qt 将系统产生的消息转换为 Qt 事件

- Qt事件是一个 **QEvent** 对象
- 用于**描述程序内部或外部发生的动作**
- 任意Object对象都具备事件处理的能力

如: QInputEvent  QDropEvent  QPaintEvent  QCloseEvent

Qt 事件的传递过程

> OS →(1.OS_MESSAGE) QApplication →(2.QEvent) QWidget(Child)
> 可能→(3.QEvent) QWidget(Parent)

事件被组件对象处理后**可能传递到其父组件对象**

---

GUI 应用程序的事件处理方式

1. Qt事件产生后立即被**分发到 QWidget 对象**
2. QWidget 中的 **event(QEvent*)** 进行事件处理 （ event() 函数是处理入口）
3. event() 根据事件类型**调用不同的事件处理函数**
4. 在事件处理函数中**发送 Qt 中预定义的信号**
5. 调用信号相关的**槽函数**

如QPushButton事件处理

1. 接收到鼠标事件
2. 调用 event(QEvent*) 成员函数
3. 调用 mouseReleaseEvent(QMouseEvent*) 成员函数
4. 调用 click() 函数
5. 触发信号 SIGNAL(clicked())

---

事件 QEvent 和信号 SIGNAL 不同

- **事件由QObject具体对象进行处理**
- **信号由QObject具体对象触发**
- 重写事件处理函数可能导致程序行为发生改变
- 信号是否存在，对应的槽函数不会改变程序行为
- 一般而言，信号在具体的事件处理函数中产生

事件处理的具体方法

QEvent 关键成员函数

```
void ignore()   // 接收者忽略当前事件，事件可能传递给父组件

void accept() // 接收者期望处理当前事件

bool isAccept()   // 判断当前事件是否被处理
```

不同的事件处理方式不同，方式非常多，如：有的事件 accept() 了，但是Qt还是要再处理一遍

### 2.6.1 事件过滤器

Qt 中的事件过滤器

- 可以**对其它组件接受到的事件进行监控**
- 任意的 QObject 对象都可以作为事件过滤器使用
- 事件过滤器对象**需要重写 eventFilter() 函数**

使用事件过滤器

- 组件通过 **installEventFilter()** 安装事件过滤器
- 事件过滤器**在组件之前接收到事件**，对事件进行监控
- 事件过滤器能够**决定是否将事件转发到组件对象**

```
/* 事件过滤器的 典型实现 */
// 返回 true 表示事件已经处理，无需传递给 obj
// 返回 false 则正常传递给 obj
bool Widget::eventFilter(QObject* obj,QEvent* e)
{
    if()   // 根据 obj 判断对象
    {
        if()   // 根据 e->type() 判断事件
        {
            // 事件处理逻辑
        }
    }
    // 调用父类中的同名函数
    return QWidget::eventFilter(obj,e);
}
```

案例：

```
mLineEdit->installEventFilter(this);

bool Test39::eventFilter(QObject* watched,QEvent* event)
{
      bool ret = true;
      if((watched == mLineEdit) && ( event->type() == QEvent::KeyPress))
     {
            qDebug()<<"eventFilter  MyLineEdit KeyPress";
            QKeyEvent* evt = dynamic_cast<QKeyEvent*>(event);
```

```
            switch(event->key)
            {
                    case Qt::Key_0:
                    case Qt::Key_1:
                    case Qt::Key_2:
                          ret = false;break;   // 仅当输入1，2，3时 才分发给
mLineEdit
                          default : break;
            }
      }
       else
      {
             ret = QWidget::eventFilter(watched,event);
      }
       return ret;
}
```

### 2.6.2 拖放事件

鼠标拖放文件事件：

- 拖放一个文件进入窗口时**触发拖放事件**
- 每一个 QWidget 对象都能够处理拖放事件
- 拖放事件的处理函数
  void dragEnterEvent(QDragEnterEvent* e)
  void dropEvent(QDropEvent* e)

---

拖放事件中的 QMimeData

- 是Qt中的多媒体数据类
- 拖放事件通过 QMimeData 对象传递数据
- QMimeData 支持多种不同类型的多媒体数据

常用 MIME 类型数据处理函数

---

自定义拖放事件：

1. 接收拖放事件的对象调用 **setAcceptDrops** 成员函数
2. 重写 **drapEnterEvent** 函数并判断 MIME 类型
   期望数据：e->acceptProposedAction();
   其它数据：e->ignore();
3. 重写 **dropEvent** 函数比判断MIME类型

期望数据：从事件对象中获取 MIME 数据并处理
其它：e->ignore();

### 2.6.3 编辑交互功能

编辑器中的常规编辑交互功能

复制 copy 、粘贴 psate 、剪切 cut 、撤销 undo、重做 redo 、删除 delete

QPlainTextEdit 中有

```
public slots:
    void copy();
    void undo();
signal:
    void copyAvailable(bool yes);
    bool undoChanged();
```

### 2.6.4 文本打印与光标定位

QPlainTextEdit 只负责界面形态的显示 内部

通过 QTextDocument 对象存储文本数据

- 设置文本属性：排版，字体，标题等
- 获取文本参数：行数，文本宽度，文本信息
- 实现标准操作：撤销，重做，查找，打印

QTextCursor 对象提供光标相关的信息

### 2.6.5 发送自定义事件

---

发送预定义事件

Qt可以在程序中主动发送事件

1. 阻塞型事件发送：事件发送后需要等待事件处理完成，完成后才返回
   bool sendEvent(QObject* receiver, QEvent* event)
   **同时支持栈事件对象和堆事件对象**的发送
2. 非阻塞型事件发送：事件发送后立即返回，事件被发送到事件队列中等待处理
   void postEvent(QObject* receiver, QEvent* event)
   **只能发送堆事件对象**，事件被处理后由 Qt 平台销毁

消息发送可以理解为：在 sendEvent() 内部直接调用Qt对象的 event()

```
void testSendEvent()
{
    QMouseEvent
dbcEvt(QEvent::MouseButtonDbClick,QPoint(0,0),Qt::LeftButton,Qt::NoButton,
Qt::NoModifier);

    QApplication::sendEvent(this,&dbcEvt);      // 发送给 this 一个事件
dbcEvt，该事件处理完才往下执行
}
void testPostEvent()
{
    QMouseEvent* dbcEvt = new
QMouseEvent(QEvent::MouseButtonDbClick,QPoint(0,0),Qt::LeftButton,Qt::NoBu
tton,Qt::NoModifier);

    QApplication::postEvent(this,dbcEvt);

}
```

案例：点击按钮，实现键盘的 delete 功能

```
void MainEditor::OnDeleteClicked()
{
    QKeyEvent keyPress(QEvent::KeyPress,Qt::Key_Delete,Qt::NoModifier);
  // NoModifier 是指该键没有和shift等组合按下
    QKeyEvent
keyRelease(QEvent::KeyRelease,Qt::Key_Delete,Qt::NoModifier);
    QApplication::sendEvent(&mainEditor,&keyPress);
    QApplication::sendEvent(&mainEditor,&keyRelease);
}
```

delete 可以删除空白行

---

## 发送自定义事件

自定义的事件类**必须继承自 QEvent**

自定义的事件类**必须拥有全局唯一的 Type 值**

程序中**必须提供处理自定义事件对象的办法**

```
class StringEvent : public QEvent
```

```
{
      QString str;
public:
    static const Type TYPE = static_cast<Type>(QEvent::User + 0xFF);
      QString data(){ return str; }
}

/* 发送 */
StringEvent strEvent("zzh");
QApplication::sendEvent(&mainEditor,&strEvent);

/* 处理 */
bool eventFilter(QObject* obj,QEvent* event)
{
      if( (obj == &mainEditor)&& (event->type() == StringEvent::TYPE) )
     {
            StringEvent* se = dynamic_cast<StringEvent*>(event);
            qDebug()<<"received:"<<se->data();
            return true;   // 表示该事件已被处理
     }
      return QWidget::eventFilter(obj,event)
}
```

---

事件的 Type 值

- 每个事件类拥有**全局唯一**的 Type 值
- **自定义事件类的Type 值也需要自定义**
- 自定义事件类使用 QEvent::User 之后的值作为 Type 值
- 程序中保证 **QEvent::User + VALUE 全局唯一**即可

---

处理自定义事件对象，两种方法

1. 将**事件过滤器**安装到目标对象
   在eventFilter() 中编写自定义事件处理逻辑
2. 在目标对象的类中**重写事件处理函数**
   在 event() 函数中编写自定义事件的处理逻辑

---

要自定义事件类的场景

- 需要**扩展**一个已有组件类的功能
- 需要开发一个**全新功能**的组件类
- 需要向一个**第三方的组件类**发送消息

## 2.7 调色板

调色板是存储组件颜色信息的数据结构，窗口组件内部都拥有 QPalette 对象

| | WindowText | ButtonText |
|---|---|---|
| **Active** | black | blue |
| **Inactive** | black | blue |
| **Disable** | gray | gray |

QPalette 类包含了组件状态和颜色组

三个状态的颜色描述

1. 激活颜色组 **Active**：组件**获得焦点**使用的颜色搭配方案
2. 非激活颜色组 **Inactive**：**失去焦点**使用的颜色方案
3. 失效颜色组 **Disable**：组件处于**不可用状态**使用的颜色方案

QPalette 的颜色组定义了组细节的颜色值

QPalette ::ColorRole 中的常量值用于标识组件细节

使用

```
p = ui->plainTextEdit->palette();

p.setColor(QPalette::Active,QPalette::Highlight,Qt::red);
p.setColor(QPalette::Inactive,QPalette::Highlight,Qt::red);
p.setColor(QPalette::Inactive,QPalette::HighlightedText,Qt::white);

ui->plainTextEdit->setPalette(p);
```

```
QPalette p = mainEditor.palette();

p.setColor(QPalette::Inactive,QPalette::Highlight,p.color(QPalette::Active
,QPalette::Highlight));
p.setColor(QPalette::Inactive,QPalette::HighlightedText,p.color(QPalette::
Active,QPalette::HighlightedText));

mainEditor.setPalette(p);
```

## 2.8 程序中的配置文件

应用程序在运行后都有一个初始化的状态（最近一次运行退出前的状态）

---

解决思路：

- 程序退出前保存状态参数到文件（数据库）
- 程序再次启动时读出状态参数并恢复

---

参数状态的存储方式：

- 文本文件格式（XML，JSon，等）
- 轻量级数据库（Access，SQLite 等）
- **私有二进制文件格式**

通过二进制数据流（QDataStream）将状态参数直接存储与文件中

优势：

- **参数的存储和读取简单高效**，易于编码实现
- 最终文件为二进制格式，**不易被恶意修改**

## 2.9 命令行参数的应用

---

如何保存主窗口的大小

应用程序退出的过程：

1. 收到关闭事件
2. **执行关闭事件处理函数**
3. 主窗口从屏幕上消失
4. 主窗口的析构函数执行

一般而言，受到关闭事件时进行状态参数的保存

Qt中：

1. 重写关闭事件处理函数
2. 在关闭处理函数中保存状态参数

---

每一个应用程序都能接收命令行参数

传统应用方式：在命令行启动 GUI 程序时传递参数 notepad text.txt

操作系统关联的方式：1. 在文件被双击时，操作系统根据文件后缀选择应用程序

3. 将文件绝对路径作为命令行参数启动应用程序

## 2.10多页面切换组件 QTabWidget

能够在同一个窗口中**自由切换不同页面**的内容

是一个**容器类型的组件**，同时提供友好的页面切换

使用

1. 创建容器类型的 QWidget 组件对象
2. 将多个子组件在容器对象中布局
3. 将容器对象加入QTabWidget中生成新的页面

```
QTabWidget m_tabWidget;
m_tabWidget.setParent(this);
m_tabWidget.setGeometry(10,10,200,200);

QPlainTextEdit* edit = new QPlainTextEdit(&m_tabWidget);
edit->insertPlainText("1st tab page");

m_tabWidget.addTab(edit,"1st");



QWidget* widget = new QWidget(&m_tabWidget);
QLabel* lbl = new QLabel(widget);
QPushButton* btn = new QPushButton(widget);
btn->setText("btn");

QVBoxLayout* layout = new QVBoxLayout();
layout->addWidget(lbl);
layout->addWidget(btn);
widget->setLayout(layout);


m_tabWidget.addTabWidget(widget,"2nd");
```

高级用法：

- 设置 Tab 标签的位置 North，South，West, East

- 设置 Tab 的外观 Rounded，Triangular
- 设置 Tab 的**可关闭模式**

预定义信号

```
void currentChanged(int index)
void tabCloseRequested(int index)
```

## 2.11 模型视图设计模式

核心思想

- 模型（数据）与视图（显示）相分离
- 模型提供对外标准接口存储数据（不关心数据如何显示）
- 视图自定义数据的显示模式（不关心数据如何组织存储）

---

工作机制

- 当数据发生变化时：模型发出信号通知视图
- 当用户与视图进行交互时：视图发出信号提供交互信息

---

Qt 内置了支持模型视图的开发方式

模型用于定义数据的显示方式，不关心数据的组织方式

模型如何为数据提供统一的访问方式

Qt中，**不管模型以什么结构组织数据，都必须为每一个数据提供独一无二的索引**；视图通过索引访问模型中的具体数据

```
m_treeView.setParent(this);
m_treeView.setGeometry(10,10,500,500);


m_fileSystemModel.setRootPath(QDir::currentPath());

m_treeView.setModel(&m_fileSystemModel);

m_treeView.setRootIndex(m_fileSystemModel.index(QDir::currentPath()));
```

- 模型**定义标准接口**（成员函数）对数据进行访问

- 视图**通过标准接口**获取数据并定义显示方式
- 模型使用**信号槽机制**通知视图数据变化
- 模型中的数据都是**以层次结构表示的**

模型索引是数据与视图分离的重要机制

QModelIndex 是 Qt 中的模型索引类：包含具体数据的访问途径，包含一个指向模型的指针

1. 线性模型可以使用 （row，column）作为数据索引

```
QModelIndex indexA = model->index(0,0);
```

2. 非线性模型
   树： （index，parent）作为索引 parent下编号为index的节点
3. 通用方式：**三元组**： （row，column，parent）
   当父节点为虚拟root节点时，可以使用空索引（直接调用 QModelIndex() 产生）作为父节点参数

特殊的模型可以自定义特殊的索引获取方式，如 QFileSystemModel

不同的视图如何显示同一模型的数据

Qt 标准模型定义

```
QStandardItem* root = m_model.invisibleRootItem();
QStandardItem* itemA = new QStandardItem();
QStandardItem* itemB = new QStandardItem();
QStandardItem* itemC = new QStandardItem();

itemA->setData("A1"); itemA->setData("A2"); itemA->setData("A3");
itemB->setData("B1"); itemB->setData("B2");
itemC->setData("C1");

root->setChild(0,0,itemA);
root->setChild(0,1,itemB);
root->setChild(0,2,itemC);

m_tableView.setParent(this);
m_tableView.setGeometry(10,10,400,400);

/* 连接模型与视图 */
m_tableView.setModel(&m_model);
```

数据角色

- 模型中的数据在视图中的用途（**显示方式**）可能不同
- 模型必须为数据设置特定数据角色（**数据属性**）
- **数据角色用于提示视图数据的作用**
- 数据角色是不同视图以**统一风格显示数据的标准**

常用数据角色

| **Qt::DisplayRole** | **用于以文本形式显示数据QString** |
| --- | --- |
| Qt::EditRole | 用于文本数据的编辑QString |
| Qt::ToolTipRole | 当鼠标处于选中的数据时，显示出数据的相关提示QString |
| Qt::SizeHintRole | 可以提示相应大小QSize |

意义

- 定义了数据在特定系统下的标准用途
- 不同的视图可以通过相同标准显示数据

## 2.11.1 自定义模型类

QStandardItemModel 是一个通用的模型类

- 能够以**任意的方式**组织数据（线性，非线性）
- 数据组织的基本单位为**数据项**（**QStandardItem**）
- 每一个数据项能够**存储多个具体数据**（附加数据角色）
- 每一个数据项能够**对数据状态进行控制**（可编辑，可选）

变体类型 QVariant

用于封装的类型，能够表示大多数常见的值类型，每次只能封装（保存）单一类型的值

意义在于能够设计“**返回类型可变的函数**”

---

工程中常用模型设计

- **解析数据源中的数据**（数据库、网络、串口等）
- 将解析后的数据存入 **QStandardItem** 对象中
- 根据数据间的关系在 **QStandardItemModel** 对象中组织数据项

- **选择合适的视图显示数据值**

---

工程中数据应用架构为4层结构（系统架构）

数据层 data source，如类DataSource，用于抽象表示数据的来源

数据表示层 data object，如ScoreInfo，

数据组织层 model，如ScoreInfoModel，用于从数据源获取数据并组织

数据显示层 view，如QTabView，用于显示模型中的数据

> 优点：易于扩展和维护

DataSource：设置数据源并读取数据，对数据进行解析后生成数据对象

ScoreInfo：封装数据源中的一组完整数据，提供返回具体数据值的接口函数

ScoreInfoModel：使用QStandardItemModel作为成员，以ScoreInfo类对象为最小单元进行数据组织

---

架构图：定义模块功能

类图：定义具体功能的接口

流程图：定义类对象间的交互

模块实现结束后需进行单元测试

---

鼠标右键上下文菜单的实现

1. 定义菜单对象 **QMenu**
2. 连接菜单中的 **QAction** 对象到槽函数
3. 定义事件过滤器，并处理 **ContextMenu** 事件
4. 在当前鼠标的位置打开菜单对象

fetchData() 只能取一次，取一次，清空一次

getData() 可以一直取

## 2.11.2 模型视图中的委托

传统的 MVC 设计模式：

Model 模型负责数据组织

View 视图负责数据显示

Controller 控制器负责用户输入

---

Qt中的模型视图设计模式借鉴了 MVC

模型负责组织数据，视图负责显示数据，如何编辑修改数据

视图中集成了处理用户输入的功能，视图将用户输入作为内部独立的子功能而实现（即将 View 与 Controller 结合了）

模型视图中的委托 Delegate

- 是视图中**处理用户输入的部件**
- 视图可以设置委托对象用于处理用户输入
- **委托对象负责创建和显示用户输入上下文**

标准委托功能：QItemDelegate

自定义委托：QCustomizeDelegate

```
qDebug()<<m_view.itemDelegate();    // 当前的视图的默认委托，是
QStyledItemDelegate类
```

---

委托中的编辑器：

- 提供编辑时需要的上下文环境（编辑器）
- 不同委托提供的编辑器类型不同（文本框，单选框等）
- 编辑器能够从模型获取数据，并将编辑结果返回模型

createEditor，创建编辑器组件

updateEidtorGeometry，更新编辑器组件大小

setEditorData，通过索引从模型获取数据

setModelData，将编辑后的数据返回模型

关键信号：

void closeEditor(QWidget* editor, QAbstractItemDelegate::EndEditHint hint)

void commitData(QWidget* editor)

委托的本质：

- 为视图提供数据编辑的上下文环境
- 产生界面元素的工厂类
- 能够使用和设置模型中的数据

---

## 2.11.3 自定义委托

当预定义的委托无法满足需求，就自定义委托类（重写以下虚函数）

1. createEditor
2. updateEditorGeometry
3. setEditorData
4. setModelData
5. paint(可选)

```
QWidget* createEditor() const   // 根据不同的数据类型创建不同的编辑器
{
    QWidget* ret = NULL;
    if(index.data().type == QVariant::Bool)
    {
        /* 创建 checkbox */
    }
    else if(index.data().type == QVariant::Char){   /* 创建 combobox */  }
    else ()

    return ret;
}
```

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

创建的是什么类型的编辑器，setEditorData设置编辑器的初始数据

```
void setEditorData(QWidget* editor,const QModelIndex& index) const
{
    if(index.data().type() == QVariant::Bool)
    {
        QCheckBox* cb = dynamic_cast<QCheckBox*>(editor);
    }
    else
}
```

根据参数中的数据索引更改模型中的数据

```
void setModel(QWdiget* editor,QAbstractItemModel *model,const QModelIndex&
index) const
{
    if(index.data().type() == QVariant::Bool)
    {
        QCheckBox* cb = dynamic_cast<QCheckBox*>(editor);
    }
    else
}
```

paint函数在每次视图刷新时都会被调用，paint 作用是 显示 模型中的所有数据

```
void paint(QPainter* painter,const QStyleOptionViewItem& option,const
QModelIndex& index) const
{
    if(){ /* 自定义绘图动作 */  }
    else{ QItemDelegate::paint(painter,option,index); }
}
```

在重写函数时，很多函数都是const，不允许修改成员变量，方法： mutable bool flag; 这样在 const 函数也可修改 flag 的值

自定义委托类需重写相应的成员函数

- 根据需要创建编辑组件并设置组件中的数据
- 编辑结束后将数据返回模型
- 成员函数的参数携带了数据存取时需要的信息

## 2.11.4 视图与委托

委托是视图的构成部分，那么委托需要承担数据显示的部分工作

- 视图负责确定**数据项的组织显示形式**（列表、树形、表格）
- 委托负责**具体数据项的显示和编辑**（数据值，编辑器）
- 视图和委托**共同完成**数据显示和编辑功能

---

自定义委托的默认数据显示方式，步骤

1. 重写 paint 成员函数
2. 在 paint中 **自定义数据显示方式**
3. 重写 editorEvent 成员函数
4. 在 editorEvent 中**处理交互事件**

---

案例1：bool 类型数据，用 checkBox 显示

```
if(index.data().type() == QVariant::Bool)
{
    bool data = index.model()->data(index,Qt::DisplayRole).toBool;

    QStyleOptionButton checkBoxStyle; // 组件绘制参数
    checkBoxStyle.state = data ? QStyle:State_On : QStyle:State_Off;
    checkBoxStyle.state |= QStyle::State_Enable;
    checkBoxStyle.rect = option.rect;
    checkBoxStyle.rect.setX(option.rect.X()+option.rect.width()/2-6);   //
checkBox 偏移

    // 根据参数绘制组件（数据项自定义显示方式）
    QApplication::style()-
>drawControl(QStyle::CE_CheckBox,&checkBoxStyle,painter);
   //QApplication::style() 返回一个类对象，里面存储了操作系统的风格
}
```

在 editorEvent 中

```
if(index.data().type() == QVariant::Bool)
{
    QMouseEvent* mouseEvent = dynamic_cast<QMouseEvent*>(event);
    if(event->type() == QEvent::MouseButtonPress &&
option.rect.contains(mouseEvent->pos()))
    {
        bool data = model->data(index,Qt::DisplayRole).toBool();
        model_setData(index,!data,Qt::DisplayRole;)
    }
}
```

按钮二：将 Progress 从**纯文本**的显示方式改为 **进度条+文本显示** 的方式

1. 自定义新的委托类
2. 在paint成员函数中绘制进度条显示方式
3. 在editorEvent成员函数中禁止数据编辑操作

在 paint 中

```
if(index.data().type() == QVariant::Int)
{
    int progress = index.model()->data(index,Qt::DisplayRole).toInt(); //
1. 根据索引参数获取模型中的数据

    QStyleOptionProgressBar progressStyle; // 2. 定义绘制参数对象
xxxxOption
    progressStyle.rect = option.rect;   // 3. 设置具体组件绘制参数 到
xxxxOption
    progressStyle.minimum = 0;
    progressStyle.maximum = 100;
    progressStyle.progress = progress;

    // 4. 根据参数对象 xxxxOption 绘制数据显示方式
    QApplication::style()-
>drawControl(QStyle::CE_ProgressBar,&progressStyle,painter);
   //QApplication::style() 返回一个类对象，里面存储了操作系统的风格
}
```

在 editorEvent 中，过滤掉鼠标双击事件

```
bool ret = true;
if(index.data().type() == QVariant::Int)
{
   if(event->type() != QEvent::MouseButtonDblClick )   // 禁用双击事件
  {
      ret = QItemDelegate::editorEvent(event,model,option,index);
  }
}
else
{
    ret = QItemDelegate::editorEvent(event,model,option,index);
}
return ret;
```

重写定义矩形范围

```
const int DELTA   = 4;
int top = option.rect.top()+DELTA;
int left = option.rect.left()+DELTA;
int width = option.rect.width()- 2*DELTA;
int height = option.rect.height()-2*DELTA;

progressStyle.rect = QRect(left,top,width,height);
```

---

任务进度模拟

1. 定义计时器用于模拟任务进度
- 定义计算器槽函数 void timerTimeout()
- 在槽函数中修改模型中的数据

```
QModelIndex p1 = m_model.index(3,0,QModelIndex());
QModelIndex p2 = m_model.index(3,1,QModelIndex());
QVariant v1 = (p1.data().toInt()+1)%100;
QVariant v2 = (p2.data().toInt()+3)%100;

m_model.setData(p1,v1,Qt::DisplayRole);
m_model.setData(p2,v2,Qt::DisplayRole);
```

在实际工程项目中，可使用后台线程根据实际的任务情况更新模型中的数据，从而更新数据的界面显示

## 2.12 基本图形绘制

QPainter ，拥有QPen（颜色，宽度，线风格），QBrush（填充风格，颜色），QFont（用于文本绘制，由字体属性组成）

QPaintDevice，是QPainter的绘画板，所有的QWidget类都继承自 QPaintDevice，因此，任意的 QWidget对象都能够作为画布绘制图形

只能在 **QWidget::paintEvent** 中绘制图形，是绘图上下文

工程中做法：改变绘图参数进行动态绘图

1. 根据需要确定参数对象(绘图类型，点坐标，角度等)
2. 将参数对象存入数据集合中，如链表
3. 在 paintevent 函数中遍历数据集合
4. 根据参数对象绘制图形 update()

坐标系

物理坐标系：设备坐标系，原点（0，0）在左上角

逻辑坐标系：

QPainter 使用逻辑坐标系绘制，默认情况下，Qt的逻辑坐标系与物理坐标系完全一致

定义视口 setViewport

定义窗口：setWindow

widget一运行就会触发绘图事件，绘图事件由 QPainter 中的 paintEvent(QPaintEvent *) 处理函数自动调用

```
// 首先，在 widget.h 的public 中增加 void paintEvent(QPaintEvent *)
#include <QPainter>
// void paintEvent(QPaintEvent *) 定义在 widget.cpp 中，如下
// 使用画笔 QcPen
QPainter painter(this);
QPen myPen(QClolor(255,0,0));
myPen.setWidth(3);
myPen.setStyle(Qt::DashDotLine);
painter.setPen(myPen);
painter.drawLine(QPoint(0,0),QPoint(100,100));

// 使用画刷 QBrush
QBrush myBrush(Qt::green,Qt::SolidPatten);
painter.setBrush(myBrush);
painter.setPen(myPen);
painter.drawRect(200,200,50,50);

//使用渐变色
QConicalGradient conicalGradient(QPointF(300,300),0);   // 从 0 度开始，逆时针
conicalGradient.setColorAt(0.2,Qt::gray);   // 0 ~ 0.2*360 度，渐变为 灰色
conicalGradient.setColorAt(0.5,Qt::green);
conicalGradient.setColorAt(0.7,Qt::blue);
conicalGradient.setColorAt(1,Qt::yellow);    // 0.7*360  ~ 360 度，渐变为黄色
```

```
QPainter painter(this);
QPen myPen(Qt::red);
myPen.setWidth(15);
painter.setPen(myPen);    //  == painter.setPen(QPen(Qt::red,15));  创建一个匿名对象
```

## 2.12.1 抗锯齿和坐标系变换

1. 抗锯齿 setRendeHints，坐标系平移 translate

```
QPinter painter(this);
QPen myPen(Qt::red);
myPen.setWidth(15);
painter.setPen(myPen);
painter.drawEllipse(QPoint(200,200),100,100);

painter.setRendeHints(QPainter::Antialiasing);   // 对第二个圆使用 抗锯齿 操作，使得表面平滑
painter.translate(200,0);   // 坐标系平移，x 向右平移 200，y 不变，取负值是向左平移
painter.drawEllipse(QPoint(200,200),100,100);
```

2. 坐标系旋转 rotate，使用时，一般先平移，在旋转坐标系

```
painter.save();   // 对坐标系进行操作之前，先保存当前的坐标系
painter.setPen(QPen(Qt::green,10));
painter.drawLine(QPoint(20,20),QPoint(100,100));
painter.rotate(90);   // 顺时针旋转 90 度
painter.setPen(QPen(Qt::blue,10));
painter.drawLine(QPoint(20,20),QPoint(100,100));

painter.restore();   // 恢复原来坐标系
```

3. 坐标系缩放 scale

```
painter.save();   // 对坐标系进行操作之前，先保存当前的坐标系
painter.setPen(QPen(Qt::green,5));
painter.setBrush(Qt::red);
painter.drawRect(300,300,50,50);
painter.scale(1.5,0.5);   // x 放大 1.5 倍， y放大 0.5 倍
painter.setBrush(Qt::green);
painter.drawRect(300,300,50,50);

painter.restore();   // 恢复原来坐标系
```

4. 坐标系扭曲 shear（不推荐使用）

```
painter.save();   // 对坐标系进行操作之前，先保存当前的坐标系
painter.setPen(QPen(Qt::green,5));
painter.setBrush(Qt::red);
painter.drawRect(300,300,50,50);
painter.shear(0.5,0)
painter.setBrush(Qt::black);
painter.drawRect(300,300,50,50);   // 扭曲后坐标变为 （450，300，50，50）变成平行四边形，每一个点的 x 坐标变成 1.5 倍
painter.restore();   // 恢复原来坐标系
```

## 2.12.2 手动调用绘图事件处理函数

widget一运行就会自动触发绘图事件 paintEvent(QPaintEvent *)，如何刷新绘图，如钟表，动画?

解决方法：手动调用 repaint() 或 update() 推荐使用 update()，会自动回调 paintEvent(QPaintEvent *)

```
QTimer *timerID1 = new QTimer(this);
timerID1->start(1000);
connect(timerID1, &QTimer::timeout,this,[&](){ update(); });   // 每隔 1s 调用 paintEvent 绘图
```

## 2.13.3 绘制文字和路径 path 为一个容器

```
QPainter painter(this);
QPainterPath path;
path.moveTo(50,250);
path.moveTo(50,200);
path.lineTo(100,100);
pathe.addEllipse(QPoint(100,100),30,30);
painter.setPen(Qt::red);
painter.drawPath(path);
path.translate(200,0);   // 坐标轴先 x 方向右平移 200
painter.setPen(Qt::blue);
painter.drawPath(path);   // 使用 path 容器，绘图
```

## 2.12.4 绘图设备_QPixmap_QImage

Qt 中绘图工具，QWidget，QPixmap，QImage，QBitmap，QPicture

QPixmap：图片类，主要用于显示图片，对于图片的显示做了优化处理，和平台有关，只能在主线程中

QImage：图片类，图依赖于平台的，多用于图片的传输，可以做像素级修改，可在主线程，多线程中

```
// 不需要paintEvent(QPaintEvent *)
#include <QPainter>
#include <QPen>
#include <QPixmap>
#include <QImage>

// 使用 QPixmap
QPixmap pix(400,300);   // 画布大小 400*300
pix.fill(Qt::white);    // 画布全白
QPainter painter(&pix);
painter.setPen(QPen(Qt::red,5));
painter.setRenderHints(QPainter::Antialiasing);
painter.drawEllipse(QPoint(100,100),50,50);
pix.save("D://mypix.jpg");

// 使用 QImage
QImage img(400,300);    // 画布大小 400*300
img.fill(Qt::white);    // 画布全白
QPainter painter2(&img);
painter2.setPen(QPen(Qt::green,5));
painter2.setRenderHints(QPainter::Antialiasing);
painter2.drawEllipse(QPoint(100,100),50,50);
```

```
pix.save("D://myImg.jpg");
```

使用 QImage 来修改图片中的像素点

```
QPainter painter(this);
QImage img;
img.load(":/image/image/demo.jpeg");
for(int i=0;i<50;i++)
{
      for(int j=0;j<100;j++)
      {
            QRgb rgb = qRgb(255,0,0);
            img.setPixel(i,j,rgb);
      }
}
painter.drawImage(0,0,img);
```

## 2.12.5 绘图设备_QPicture_QBitmap

QPicture：可视为一个绘图的容器，里面保存有绘图的记录和重绘制的命令，存储的形式是二进制形式，不方便打开(要用 Qt 代码打开显示)

QBitmap：黑白图片

1. 使用 QPicture

```
QPicture myPicture;
QPainter painter(&myPicture);   // == QPainter painter;
painter.begin(&myPicture);
painte.setPen(QPen(Qt::red,5));
painter.drawEllipse(QPoint(100,100),50,50);
painter.end();
myPicture.save("D://picture.jpg");   // 任意后缀文件
```

想要显示该图片，如

```
QPaint painter(this);
QPicture mypic;
mypic.load("D://picture.jpg");
painter.drawPicture(0,0,mypic);
```

2. 使用 QBitmap

```
QBitmap mybitmap(400,300);
QPainter painter2(&mybitmap);
painter2.setPen(QPen(Qt::green,5));
painter2.drawEllipse(QPoint(100,100),50,50);
mybitmap.save("D://bitmap.jpg");    // 图片为黑白的，并且会有很多不正常的像素点
```

使用 QBitmap打开图片，会自动转换为黑白图片，但是会出现很多不正常的像素点

```
QPaint painter(this);
QBitmap mybitmap;
mybitmap.load("D://demo.jpeg");
painter.drawPixmap(0,0,mybitmap);
```

### 2.12.6 绘制弧线

```
QPaint painter(this);
QPen mypen;
mypen.setColor(Qt::red);
my.setWidth(5);
painter.setPen(mypen);
painter.setRenderHints(QPainter::Antialiasing);
painter.drawArc(25,25,1550,1550,0,180*16);   // 在 （0，0）开始 长 1550，宽 1550的矩形中，画弧线，起始角度0，结束角度 180度
```

## 2.13 进程与线程

---

定义：

**程序**是计算机存储系统中的**数据文件**（静态）

源代码程序：文本文件，描述程序行为和功能

可执行程序：二进制文件，直接加载并执行

---

**进程**（动态）：广义：程序中关于某个数据集合的一次运行活动

狭义：程序被加载到内存中执行后得到进程

---

区别：

程序是硬盘中静态的文件（存储系统中的一段二进制表示），

**进程**是内存中动态的**运行实体**（数据段、代码段、PC指针等）

---

联系：

一个程序可能对应多个进程（一个程序多次运行，每次运行产生一个进程）

一个进程可能包含多个程序（一个程序依赖多个其它动态库）

---

在当代操作系统中，资源分配的基本单位是进程，而CPU调度执行的基本单位是线程，线程是进程使用CPU资源的基本单位

线程：

- **进程内的一个执行单元**
- 操作系统中一个可调度的实体
- 进程中相对独立的一个执行流序列
- 执行时的现场数据和其它调度所需的信息
- 进程中可以存在多个线程并行执行，**共享进程资源**（一个进程被加载，就会产生一个主线程）
- 线程是被调度的执行单元，而**进程不是调度单元**
- **线程不能脱离进程单独存在**，只能依赖于进程运行
- 线程有生命周期，有诞生和死亡
- **任意线程都可以创建其它新的线程**

---

总结：

- 程序是数据文件
- 进程是程序运行后得到的执行实体
- 线程是进程内部的具体执行单元
- 一个进程内部可以有多个线程存在
- 进程是操作系统资源分配的基本单位
- 线程是操作系统调度执行的基本单位

## 2.13.1 多线程编程

通过 QThread 直接支持多线程，是一个跨平台的多线程解决方案

Qt 中线程以对象的形式被创建和使用

每一个线程对应着一个 QThread 对象

关键函数

```
void run()   // 线程体函数，用于定义线程功能（执行流）
void start()   // 启动函数，将线程入口地址设置为 run 函数
void terminate() // 强制结束线程（不推荐）
```

示例

```
class MyThread:public QThread   // 创建线程类
{
    protected:
    void run()   // 线程入口函数
    {
        for(int i = 0; i<5; ++i)
        {
            qDebug()<<objectName()<<": "<<i;
            sleep(1);
        }
    }
};

MyThread t;   // 创建子线程
t.setObjectName("t");
t.start();   // 启动子线程
```

---

生命周期

自然死亡 run() 执行结束

非自然死亡 t.terminate()

> 在工程中，**terminate() 是禁止使用的**，会使得操作系统**暴力终止线程**，而不会考虑数据完整性，资源释放问题

---

如何终止线程

- **run() 函数执行结束**是优雅结束线程的唯一方式
- 在线程中增加标志变量 **m_toStop**(volatile bool)，必须使用 volatile，不允许编译器做优化
- 通过 m_toStop 的值判断是否需要从 run() 函数返回

```
class MyThread:public QThread   // 创建线程类
{
public:
    MyThread(){ m_toStop = false; }
    void stop() { m_toStop = true; }
protected:

    volatile bool m_toStop;

    void run()   // 线程入口函数
    {
        while(!m_toStop)
        {
            //
        }
    }
};
```

### 2.13.2 多线程间的同步

多线程编程的本质：**并发性**

在宏观上，所有线程并行执行，多个线程间相互独立，互不干涉

线程间总是完全独立毫无依赖的吗，在特殊情况下，多线程的执行在**时序上存在依赖**

同步：在特殊情况下，控制多线程间的相对执行顺序

```
bool QThread::wait(unsigned long time = ULONG_MAX)
```

示例

```
QThread t;
t.start();
t.wait();   // 等待子线程执行结束，默认等待时间无限长
```

### 2.13.3 多线程间的互斥

多线程间除了在时序上可能产生依赖，在

如生产消费者问题

临界资源 critical resource：每次只允许一个线程进行访问（读/写）的资源

线程间的互斥（竞争）：多个线程在同一时刻都需要访问临界资源

**QMutex** 类是一把线程锁，保证线程间的互斥

**利用线程锁能够保证临界资源的安全性**

关键成员函数

```
void lock()    //  当锁空闲时，获取锁并继续执行；当锁被获取时，阻塞并等待锁释放

void unlock()   // 释放锁（同一把锁的获取和释放必须在同一线程中成对出现）
```

示例

```
QMutex mutex;
mutex.lock();
// 使用临界资源

mutex.unlock();    // 如果mutex在调用unlock时是处于空闲状态，那么程序的行为是未定义的
```

---

线程死锁：线程间相互等待临界资源而造成彼此无法继续执行

发生死锁的条件：

系统中存在**多个临界资源**且**临界资源不可抢占**

线程需要多个临界资源才能继续执行

```
//  QThreadA 获得g_mutex_1，而QThreadB获得 g_mutex_2，则发生死锁
class ThreadA:public QThread
{
protected:
      void run()
     {
            int count = 0 ;
            while(true)
           {
                  g_mutex_1.lock();
                  g_mutex_2.lock();

                  qDebug()<<objectName()<<"is working";
```

```
                g_mutex_2.unlock();
                g_mutex_1.unlock();
            }
        }
};

class ThreadB:public QThread
{
protected:
      void run()
      {
            int count = 0 ;
            while(true)
            {
                g_mutex_2.lock();
                g_mutex_1.lock();

                qDebug()<<objectName()<<"is working";

                g_mutex_1.unlock();
                g_mutex_2.unlock();
            }
      }
};
```

---

死锁的避免：对临界资源进行编号

- 对所有的临界资源都分配一个唯一的序号（r1，r2，…，rn）
- 对应的线程锁也分配相同的序号（m1，m2，…，mn）
- **系统中的每个线程安装严格递增的次序请求资源**

方法二：只使用一把线程锁，把n个资源看成一整个集合（服务器不会怎么做，客户端可以）

---

信号量

- 信号量是特殊的线程锁
- 信号量允许 N 个线程同时访问临界资源
- Qt 中直接支持信号量 QSemaphore

```
QSemaphore sem(1);
sem.acquire();
// 使用临界资源
sem.release();
```

QSemaphore 对象中维护了一个整型值，acquire() 会使其减1，release() 会使其加1，当该值为0时，acquire() 函数将阻塞当前线程

### 2.13.4 多线程中的信号与槽

QThread 继承自 QObject，可以发送信号，定义槽函数

关键信号：

```
void start()          // 线程开始运行时发射该信号
void finished()       // 线程完成运行时发射该信号
void terminated()     // 线程异常终止时发射该信号
```

如果程序中有多个线程，槽函数是在哪个线程中执行的

- **进程中存在栈空间**（区别于栈数据结构）
- 栈空间专门用于函数调用（保存函数参数，局部变量等）
- **线程拥有独立的栈空间**（可调用其它函数）

只要函数体中没有访问临界资源的代码，同一个函数可以被多个线程同时调用，不会产生副作用

操作系统通过整型标识管理进程和线程

- 进程拥有全局唯一的ID值（PID）
- 线程有进程内唯一的ID值（TID）

QThread 中的关键静态成员函数

- QThread * currentThread()
- Qt::HANDLE currentThreadId()

对象依附于哪一个线程

线程的依附性与槽函数执行的关系

对象的依附性是否可以改变

默认情况下：**对象依附于自身被创建的线程**，例如对象在main函数（主线程）中被创建，则依附于主线程

默认情况下：槽函数在其所依附的线程中被调用执行

```
int main()
{
    QThread t;    // 依附于主线程
    MyObject m;   // 依附于主线程
    QObject::connect(&t,SIGNAL(started()),&m,SLOT(getStarted()));   // 槽函数都在主线程被调用执行
}
```

QObject::moveToThread() 用于改变对象的依附性，使得对象的槽函数在依附的线程中被调用执行

```
m.moveToThread(&t);   // 改变 m 的线程依附性
```

---

线程中的事件循环

- 信号与槽的机制需要事件循环支持
- QThread 类中提供的 **exec()** 函数用于**开启线程的事件循环**
- **只有事件循环开启**，依附在线程中的槽函数才能在信号发送后被调用

```
QObject::connect(&t,SIGNAL(started()),&m,SLOT(getStarted()));
m.moveToThread(&t); // 槽函数不会被调用执行（Qt4中）
```

> 研究槽函数的具体执行线程的意义：避开临界资源的竞争问题
>
> 当信号的发送与对应槽函数的执行在不同线程中，可能产生临界资源的竞争问题

无论事件循环是否开启，信号发送后会直接进入对象所依附线程的事件队列；然而，只有开启了事件循环，对应的槽函数才会在线程中被调用

使用 exec() 之后，线程进入事件循环

- 事件循环结束前，exec() 后的语句无法执行
- quit() 与 exit() 函数用于结束事件循环
- quit() → exit(0)，**exec()的返回值由 exit() 参数决定**

什么时候需要在线程中开启事件循环

设计原则：

> 事务性操作（间断性IO操作等）可以开启线程的事件循环，每次操作通过发送信号的方式使得槽函数在子线程中执行，将操作分摊给子线程

---

文件缓冲区：

- 默认情况下，文件操作时会开辟一段内存作为缓冲区
- 向文件中写入**数据会先进入缓冲区**
- 只有当**缓冲区满**或者**遇到换行符**才将数据写入磁盘
- 缓冲区的意义：减少磁盘的低级 IO 操作，提高文件读写效率

人为强制将缓冲区写入磁盘：file.flush()

Qt线程的使用模式：

- 无事件循环：后台执行长时间的耗时任务，如文件复制，网络数据读取等
- 开启事件循环模式：执行事务性操作，如文件写入，数据库写入等

工程开发中，多数情况不会开启线程的事件循环，线程多用于执行后台任务或耗时任务

### 2.13.5 线程的生命期

C++对象有生命周期

线程也有生命周期

工程实践中的经验准则：

> 线程对象生命周期 > 对应的线程生命周期

如果是 < ，那么线程对象销毁了，线程还在执行，线程执行时访问线程对象的成员变量时，就是非法的，因为线程对象的存储空间都没了

---

为确保 线程对象生命周期 > 对应的线程生命周期

解决方案：

1. 同步型线程设计：

   线程对象**主动等待**线程生命周期结束后才销毁

特定：

同时支持**栈**和**堆**中创建线程对象，对象销毁时确保线程生命周期结束

设计要点：

**在析构函数中先调用 wait()**，强制等到线程运行结束

使用场合：

**线程生命周期相对较短的情形**

```
SyncThread::~SyncThread()
{
    wait();
}
```

---

2. 异步型线程设计

概念：

线程生命周期结束时**通知销毁销毁对象**

特定：

只能在**堆**中创建线程对象，线程**对象不能被外界主动销毁**

要点：

**在 run() 的最后调用 deleteLater()**

线程体函数**主动申请**销毁线程对象

使用场合：

线程生命周期不可控，需要长时间运行于后台的情形

```
void AsyncThread::run()
{
    for(){}   // 执行线程任务
    deleterLater();
}
```

---

总结：

- 线程对象生命周期**必须大于**线程生命周期
- 同步型线程设计——**线程生命周期较短**
- 异步型线程设计——**线程生命周期不可控**
- 线程类的设计**必须适应**具体的场合；没有万能的设计，只有合适的设计

## 2.13.6 第二种线程创建方法

面向对象设计实践早期，工程中习惯通过继承的方式扩展系统功能

现代软件架构：尽量使用组合的方式实现系统功能，代码中仅体现需求中的继承关系

通过继承的方式实现新的线程类没有实际意义，就是为了重写 run()

ThreadOne, ThreadTwo,ThreadThree 就仅仅是 run() 不同

通过继承的方式实现多线程没有任何实际意义

- QThread 对应于操作系统中的线程
- QThread 用于充当一个线程操作的集合
- **应该提供灵活的方式指定线程入口函数**
- **尽量避免重写 void run()**

---

解决方案——信号与槽

1. 在类中定义一个槽函数 void tmain() 作为线程入口函数
2. 在类中( Qt 类就可)定义要给 QThread 成员对象 m_thread
3. 改变当前对象的线程依附性到 m_thread
4. 连接 m_thread 的 start() 信号到 tmain()

---

总结：

- 早期Qt只能通过继承的方式创建线程
- 现代软件技术提倡**以组合方式替代继承**
- QThread 应该作为**线程的操作集合**而使用
- 可以通过**信号与槽**的机制灵活指定线程入口函数

```
// 新版本的 QThread
class QThread:public QObject
{
    protected:
    virtual void run()
    {
        (void) exec();    // 提供默认实现形式，默认开启了事件循环
    }
}
```

## 2.13.7 多线程与界面组件的通信

是否可以在子线程中创建界面组件？ ————————不能

GUI系统设计原则：

界面组件对象必须依附于主线程

所有界面组件的操作都只能在主线程中完成，因此，主线程也叫 GUI 线程

---

子线程如何对界面组件进行更新：**子线程不能直接操作界面组件，但是可以通过信号与槽的机制间接操作界面组件**

解决方案1：信号与槽

1. 咋主线程类中**定义界面组件的更新信号** UpdateUI
2. 在主窗口类中**定义更新界面组件的槽函数** SetInfo
3. 使用**异步方式**连接更新信号到槽函数（UpdateUI → SetInfo）
   子线程通过发射信号的方式更新界面组件
   所有的界面组件对象只能依附于主线程

---

解决方案2：发送自定义事件

1. **自定义事件类**用于描述界面更新细节
2. 在主窗口类中**重写事件处理函数 event**
3. 使用 postEvent() 以**异步方式** 发送自定义事件类对象(发送事件对象有两种方式：sendEvent()、postEvent()发送的对象必须在堆上创建 )
   子线程指定接收消息的对象为主窗口对象（需要将主窗口设为父组件，这样主线程才能发送）
   在 event() 事件处理函数更新界面状态

事件对象是在主线程中被处理的，即 event() 的调用是在主线程中完成

## 2.13.8 软件开发流程

与开发技术无关，是开发团队必须遵守的一系列规则，在于保证产品的质量和进度，业界已经存在多种开发流程的模型

1. 即兴模型 Build and Fix Model
   与用户交流后即开始进行开发，没有需求分析和需求发掘过程，没有整体设计以及规划，没有软件文档，维护性差
2. 瀑布模型 Waterfall Model

自上而下的布局（需求分析→架构设计→开发实现→系统测试→最终发布），每个步骤都是不可逆的

3. 增量模型 Incremental Model
   将系统功能分解为互不重叠的子功能，并行实现子功能，每个子功能就是瀑布模型，全部完成后系统开发结束
4. 螺旋模型 Spiral Model
   采用一种迭代的方法，项目分解成多个不同的版本，每个版本的开发过程都需要用户参与（多次需求分析，多次开发）
5. **敏捷模型** Agile Modeling
   一切从简，拥抱变化，高效工作，持续开发

# 第三章：Proj

## 3.1计算器

---

计算器核心算法：

1. 中缀表达式进行**数字**和**运算符**的**分离**
2. 将中缀表达式转换为**后缀表达式**
3. 通过后缀表达式**计算最终结果**

---

1.分离算法

中缀表达式中包含

- 数字和小数点：0-9 或 .
- 符号位:+或-
- 运算符：+-*/
- 括号： ()

思想：以**符号作为标志**对表达式中的字符逐个进行访问

1. 定义累计变量num
2. 当前字符 exp[i] 为**数字**或**小数点**时：
   累计： num += exp[i]
3. 当前字符为**符号**时：
   num 为运算数，分离并保存
   若 exp[i] 为正负号：
   累计符号位+和-：num+=exp[i]

若 exp[i] 为运算符，分离并保存

如何区分正负号和加减号

- +和-在表达式的第一个位置
- 括号后的+和-
- 运算符后的+和-

正负号：看前一个字符，前一个字符为空、（、运算符

## 2.中缀转换为后缀表达式

类似编译过程

- 四则运算表达式中的括号必须匹配
- 根据运算符优先级进行转换
- 转换后的表达式中没有括号
- 转换后可以顺序的计算最终结果

转换过程：

1. 当前元素为**数字**：输出
2. 当前元素e为**运算符**：
   （1）与栈顶运算符进行优先级比较
   （2）小于等于：将栈顶元素输出，转（1）
   （3）大于：将当前元素入栈
3. 当前元素为 **(** ：入职
4. 当前元素为 **)** ：
   （1）弹出栈顶元素并输出，直至栈顶元素为左括号
   （2）将栈顶的左括号从栈中弹出

## 3.后缀表达式计算

遍历后缀表达式

- 当前元素为数字：进栈
- 当前元素为运算符：
  1.从栈中弹出右操作数
  2.从栈中弹出左操作数
  3.根据符号进行运算
  4.将运算结果压入栈中

遍历结束，栈中的唯一数字为结果

## 3.2 文本编辑器

文本编辑器需求固定，功能之间的耦合性弱，因此增量模型

Valgrind内存分析器

ui->addMenu(menu) // menu并没有成为ui的子对象

### 3.2.1 创建查找对话框

可复用对话框需求分析：

- 可复用软件部件
- 查找文本框中的指定字符串
- 能够指定查找方向
- 大小写敏感查找

QString 提供不同的子串查找方式 indexOf() 与 lastIndexOf()

QPointer 智能指针，当指向的Qt对象销毁时，该指针会自动设置为NULL，实现查找对话框与文本框的弱耦合关系

### 3.2.2 替换对话框

需求分析：

- 可复用软件部件
- 查找文本框中的指定字符串
- 替换单个指定字符串
- 替换所有指定字符串

具体实现

- 替换对话框的功能**涵盖**了查找对话框
- 替换对话框可以**继承自查找对话框**
- 替换功能的实现是**基于查找算法完成的**
- 查找对话框与主界面文本框是聚合关系

### 3.2.3 关于对话框

关于对话框用于标识软件自身的信息

软件 logo、项目名、版本号、开发者信息、版权信息、联系方式

布局

```
#include <QHBoxLayout>

QPushButton* submitBtn = new QPushButton(this);
QPushButton* cancelBtn = new QPushButton(this);
QPushButton* browserBtn = new QPushButton(this);
```

## 3.3 3D 动画

QOpenGL

查看版本

```
#include <QApplication>
#include <QOffscreenSurface>
#include <QOpenGLContext>
#include <QOpenGLFunctions>
#include <QDebug>


int main(int argc, char *argv[])
{
	QApplication a(argc, argv);
	QOffscreenSurface surf;
	surf.create();
	QOpenGLContext ctx;
	ctx.create();
	ctx.makeCurrent(&surf);

	GLint major, minor;
	ctx.functions()->glGetIntegerv(GL_MAJOR_VERSION, &major);
	ctx.functions()->glGetIntegerv(GL_MINOR_VERSION, &minor);
	qDebug()<<"OpenGL Version Info:" <<(const char *)ctx.functions()-
>glGetString(GL_VERSION);
	qDebug()<<"OpenGL Version Major:" <<major<<"OpenGL minor:"<<minor;
	ctx.doneCurrent();
	return a.exec();
}
/*
OpenGL Version Info: 3.3 (Compatibility Profile) Mesa 20.0.8
OpenGL Version Major: 3 OpenGL minor: 3  */
```

# 3.4 串口调试助手项目

## 3.4.1 自动识别串口号

使用定时器，定时 500ms

扫描串口

```
// widget.h 中
private:
    QTimer *timerID1;
// widget.cpp 中
#include <QtSerialPort/QtSerialPort>
bool Widget::event(QEvent *event)
{
    QStringList newPortStringList;
      static QStringList prePortStringList;
      newPortStringList.clear();
      foreach(const QSerialPortInfo &info, QSerialPortInfo::availablePorts)
            newPortStringList += info.portName;
      if(prePortStringList != newPortStringList)
     {
            ui->comboBox->clear();
            ui->comboBox->addItems(newPortStringList);
            prePortStringList = newPortStringList;
     }
}
```

## 3.4.2 设置串口参数

## 3.4.3 接收功能

## 3.4.4 发送功能

```
QByteArray byteArray;   // 用于发送 write 要发送 QByteArray 类型
sendText = ui->textEdit_2->toPlainText();
byteArray = sendText.toLation1();   // 将 QString 类型的 sendText 转换为
QByteArray 类型
sendText_byte += sendText.length();
ui->label_12->setText(QString::number(sendText_byte));
mySerial->write(byteArray);
```

# 3.5 STM32开发板Qt开发

## 3.5.1 交叉编译Qt项目

### （1）安装交叉编译器

在 ubuntu 中安装 .sh 脚本文件

```
chmod 777 ./*.sh
sudo ./*.sh
```

默认安装在 /opt

每次使用安装后的该脚本，都需要source添加环境变量

```
source
```

source是一个内置的Shell命令，它读取并执行当前Shell中文件的内容，仅对当前terminal 有效

source 之后 qmake-v 才有效

### （2）命令行交叉编译Qt项目

```
qmake -v
```

会打印 qmake 版本信息

在 Lesson55QUdp.pro 文件夹下

```
qmake Lesson55QUdp.pro
```

执行完会生成 Makefile 文件，以及一些隐藏文件（ls -a查看，ls -l显示全部文件的详细信息）

> make distclean  // 不用

会删除上述 qmake 生成的所有文件

> make -j 2    // 使用2两个线程进行编译，不能大于内核数量，实际是调用 gcc

根据makefile进行编译，执行完会生成 Lesson55QUdp 的可执行文件（ls -l可看到带x属性）

> make clean  // 需要用

只会删除一部分多余的么用文件

### （3）拷贝编译好的执行程序到开发板上

---

方法一：U盘

通过U盘，U盘在ubuntu目录 /media/zhihongzeng/ 下面，cp 拷贝，拷贝完 sync 同步数据

通过U盘拷贝可执行文件到开发板

需要先关闭开发板当前运行的程序，killall+程序名

然后修改默认启动的程序

---

方法二：通过网络

开发板与电脑处于同一网络

```
scp 文件 用户名@ip地址：路径
scp -r 文件夹 用户名@ip地址：路径

scp Lesson55QUdp root@开发板ip :/home/root
// 将 Lesson55QUdp 拷贝到 开发板的 /home/root 目录下，开发板的默认用户名为 root
```

### （4）STM32 交叉编译Qt项目

## 3.5.2 Qt点亮开发板的LED

Qt → 驱动提供接口 → 开发板硬件

---

1. 开发板的 Linux 中找到 LED

内核设备树路径 arch/arm/boot/dts/stm32mp157d-atk.dtsi

里面注册有硬件中的 LED，LED注册成了 gpio-led 类型设备，通过应用层接口操作 LED

```
#include "lesson58led.h"
#include "ui_lesson58led.h"
#include <QDebug>

Lesson58Led::Lesson58Led(QWidget *parent) :
      QWidget(parent),
      ui(new Ui::Lesson58Led)
{
      ui->setupUi(this);

      MyFile.setFileName("/sys/class/leds/sys-led/brightness");
      // 设置触发模式为 none
      system("echo none > /sys/class/leds/sys-led/trigger");
}

void Lesson58Led::on_pushButton_clicked(bool checked)
{
      if( !MyFile.exists() )
     {
            qDebug() <<" no led.";
            return;
     }
      else
     {
            if( !MyFile.open(QIODevice::WriteOnly) )
                  return;
            QByteArray buff[] = {"0","1"};
            if(checked)
           {
                  ui->pushButton->setText("off");
                  MyFile.write(buff[1]);
```

```
            }
             else
            {
                    ui->pushButton->setText("on");
                    MyFile.write(buff[0]);
            }

             MyFile.close();
      }
}
```

### 3.5.3 使用开发板的按键

Linux 系统中集成了键盘，因此可以设置按键为键盘中的某个键

Qt::Key_VolumeDown 的按键编号为 114

```
void Lesson60Btn::keyPressEvent(QKeyEvent *event)
{
      if( event->key() == Qt::Key_1 )
      {
            this->setStyleSheet("QWidget{background-color:red;}");
      }
      qDebug()<<"key pressed.";
}

void Lesson60Btn::keyReleaseEvent(QKeyEvent *event)
{
      if( event->key() == Qt::Key_1 )
      {
            this->setStyleSheet("QWidget{background-color:white;}");
      }
}
```

### 3.5.4 串口上位机

要在ubuntu中连接串口，需要 chmod 777 /dev/ttyUSB

### 3.5.5 嵌入式Qt嵌入式移植概述

---

为什么要移植：

自制系统，如BusyBox简易系统，希望能运行Qt

厂家提供的Qt库太大，太全，占空间

版本升级，想用其他好用的Qt版本

---

方法：

1. 编译Qt源码（在Qt官网的single中），得到Qt库文件（Linux中.so后缀，Windows中 .dll 后缀），部署到嵌入式系统中
2. **利用开源的嵌入式 Linux 系统自动构建框架 Buildroot** 或者 Yocto

**Buildroot：很方便**

Yocto：容易编译出错，而且文件巨大，不学

三种方法对比

| | **单独编译Qt源码（移植到BusyBox）** | **Buildroot构建Qt** | **Yocto构建Qt** |
|---|---|---|---|
| **难度** | 相当大，依赖第三方库 | 小，需要什么就勾选就行 | 大 |
| **Qt库完整性** | 不完整 | 完整 | 完整 |
| **复杂程度** | 复杂 | 一般 | 一般 |

| | 单独编译Qt源码（移植到BusyBox） | Buildroot构建Qt | Yocto构建Qt |
| --- | --- | --- | --- |
| **缺少的功能** | 播放媒体（Qt是调用gstreamer媒体去解码），网络功能（需要python）等 | 较完整，需要配置依赖编译 | 较完整，需要配置依赖编译 |
| **适用场合** | 运行简单的Qt图形界面 | 功能完善，媒体，网络等功能可用 | 功能完善，媒体，网络等功能可用 |

# 第四章：进阶

## 4.1 网络编程

对于不同人群

- 网络用户：资源库、虚拟世界、通讯媒体
- 应用程序开发者：数据收发的通道（open、send、receive、close）
- 网络设备开发者：数据收发器、设备连接方式、通信协议

---

网络形态：

1.局域网 LAN：物理上位置接近的设备通过交换机连接

2.广域网 WAN：不同的局域网通过路由器连接成广域网

---

网络基本概念

- MAC地址（硬件地址，给交换机用）：网络设备出厂时设定的全球唯一硬件地址
- 网络地址（软件地址即ip地址，给路由器用）：每一台网络主机都有唯一的地址（如192.168.12.1）
- 网络端口：每一台网络主机可以通过不同端口进行多路通信（如80端口）

1. 交换机：**端到端数据收发**（只做一次中转）
   基于硬件地址实现不同设备间的数据转发
   特定：工作层次低，转发速度快
   原理：有一张主机-MAC地址的映射表
2. 路由器：**决定数据转发路线，执行转发操作**
   基于软件地址实现不同网络间的数据转发

特点：能够选择数据通道，实现通信控制

原理：如何知道哪条路径发送时间最短：静态路由（管理员配置好的一张表），动态路由（算法）

交换机与路由器工作在不同的层次

网络协议：为数据交换而建立的规则、标准或约定的集合

协议栈：

> 应用层
>
> 传输层（TCP/UDP）
>
> 网络层（IP）（路由器）
>
> 接口层：1.数据链路层(交换机)　2.物理层

上层协议基于下层协议实现

---

TCP（传输控制协议）

基于连接的可靠传输协议，主要用于大量数据的场合，传输速度慢

TCP的3次握手（建立连接）

UDP（用户数据报协议）

非连接方式的传输协议，主要用于少了数据的场合，传输速度快

区别：TCP数据传输前需要建立连接，而UDP不需要

---

TCP和UDP是应用层协议的基础

应用层协议：

- HTTP：超文本传输协议，常用于游览器/Web服务器
- FTP：文件传输协议，常用于文件共享
- SMTP：邮件传输协议，常用于邮件发送
- Telent：远程登录协议，常用于终端远程登陆主机

### 4.1.1 客户端

Qt网络编程方式：

网络只是数据传输的通道，Qt提供了网络协议对应的类（封装了协议细节），使用类进行数据收发，从而进行网络应用开发

QTcpSocket、QTcpServer

```
connectToHost()     // 1.连接服务器主机
read()、write()     // 2
close()             // 3
```

默认情况下，QTcpSocket 使用**异步编程**的方式

1. 操作完成后立即返回

```
2.  通过**发送信号的方式返回操作结果**，可以在程序中将对应信号连接到槽函数，获取结果
```

3. 在GUI应用程序中，通常使用 QTcpSocket 的异步方式

同步编程：如函数调用，返回就是操作的结果

异步编程：函数返回时是代表操作的启动（返回 bool，判断是否启动成功）。通过回调拿到操作的结果（回调函数或者信号槽都可实现）

QTcpSocket 提供了辅助函数，可完成同步编程的方式

```
waitForConnected()        waritForDisconnected()
waitForBytesWritten()     waitForReadyRead()
```

示例：

```
void SyncClientDemo()
{
      QTcpSocket client;
      char buf[256] = {0};
      client.connectToHost("127.0.0.1",9999);
      qDebug()<<"connected:"<<client.waitForConnected();


      qDebug()<<"send byte:"<<client.write("zzh");
      qDebug()<<"send status:"<<client.waitForBytesWritten();


      qDebug()<<"data available:"<<client.waitForReadyRead();
      qDebug()<<"received data byte:"<<client.read(buf,sizeof(buf)-1);
```

```
        qDebug()<<"received data:"<<buf;

        client.close();

        //qDebug()<<"disconnected:"<<client.waitForDisconnected();
}
```

异步编程

```
connect()       disconnected()
readyRead()
bytesWritten(qint64)    //数据成功发送志系统
```

示例：

```
#include "clientdemo.h"

ClientDemo::ClientDemo(QObject *parent) : QObject(parent)
{
        connect(&m_client,SIGNAL(connected()),this,SLOT(onConnected()));

    connect(&m_client,SIGNAL(disconnected()),this,SLOT(onDisconnected()));
        connect(&m_client,SIGNAL(readyRead()),this,SLOT(onDataReady()));

    connect(&m_client,SIGNAL(bytesWritten(qint64)),this,SLOT(onBytesWritten(q
int64)));
}

void ClientDemo::connectTo(QString ip, int port)
{
        m_client.connectToHost(ip,port);
}

qint64 ClientDemo::send(const char *data, int len)
{
         return m_client.write(data,len);
}

qint64 ClientDemo::available()
{
        return m_client.bytesAvailable();
}

void ClientDemo::close()
```

```
{
    m_client.close();
}

void ClientDemo::onConnected()
{
    qDebug()<<"onConnected";
    qDebug()<<"local addr:"<<m_client.localAddress();
    qDebug()<<"local port:"<<m_client.localPort();
}

void ClientDemo::onDisconnected()
{
    qDebug()<<"onDisconnected";
}

void ClientDemo::onDataReady()
{
    char buf[256] = {0};
    qDebug()<<"onDataReady"<<m_client.read(buf,sizeof(buf)-1);
    qDebug()<<"data:"<<buf;
}

void ClientDemo::onBytesWritten(qint64 bytes)
{
    qDebug()<<"onBytesWritten"<<bytes;
}
```

### 4.1.2 服务器

服务器是为客户端服务的，服务的内容诸如向客户端提供资源，保存客户端数据，为客户端提供功能接口，等

---

1.C/S 网络结构 Client / Server

服务端公开网络地址，提供服务

客户端提供服务端网络地址主动连接服务端

特点:

- 服务端被动接受连接（服务端无法主动连接客户端）
- 服务端必须公开网络地址(容易受攻击)
- 客户端倾向于处理用户交互及体验

- 服务端倾向于用户数据的组织和存储（数据处理）

---

2. B/S 网络结构  Browser / Server

- 是一种**特殊的 C/S 网络架构**
- 客户端统一使用浏览器 **Browser**
- 客户端 GUI 通常使用 **HTML** 进行开发
- 客户端与服务端通常采用 **http** 协议进行通信

---

TCP 服务端编程

QTcpServer 类封装了 TCP 协议细节

QTcpServer 用于端口的连接监听，**监听到连接后，生成 QTcpSocket 对象与客户端通信**

每个QTcpSocket 对象同样需要连接信号到槽函数

---

使用方式：

1. 监听地址和端口 listen()
2. 通过信号通知客户端连接 newConnection()
3. 获取 QTcpSocket 通信对象  nextPendingConnection()
4. 停止监听 close()

---

注意：

- QTcpServer 用于处理客户端连接，不进行具体通信
- 监听的端口只用于响应连接请求
- 监听到连接后，生成QTcpSocket 对象与客户端通信

```
QTcpServer :   listen()   isListening()   close()
QTcpSocket :   connect()    read()   write()   close()
```

### 4.1.3 TCP 传输文件小案例

1. 客户端 client

```
QT += core ui network
```

```
#include <QTcpServer>
#include <QTcpSocket>
#incldue <QFileDialog>
   #include <QTimer>

// 定义套接字
QTcpSocket *tcpSocket;
QString fileName;
Qint64 fileSize;
Qfile file;
QTimer myTimer;

tcpSocket = new QTcpSocket(this);
myTimer =new QTimer(this);


connect( tcpSocket,&QTcpSocket::readyRead,this,[&](){
      QByteArray arr =tcpSocket->readAll();
      qDebug()<<"client receive: "<<arr;
     });
connect(myTimer,&QTimer::timeout,this,[&](){   // 20ms 的定时器，定时结束再开始真正发送文件
      myTimer->stop();
      qint64 len;
      qint64 sendSize;
      do{
            len = 0;
            char buf[4*4096]; // 4kb
            len = file.read(buf,sizeof(buf));   // 实际读出来的值为 len
            tcpSocket->write(buf,len);
            sendSize += len;      // 累计已经发送文件大小
            ui->progressBar->setValue(sendSize/1024);   // 为了让数字不那么大，/1024

     }while(len>0)
      if(sendSize == fileSize)
     {
            file.close();   // 关闭文件
            tcpSocket->disconnectFromHost();   // 断开服务器的连接
            tcpSocket->close();   // 关闭套接字
     }
});


void Widget::on_pushButton_clicked()   // 点击连接
{
```

```
        tcpSocket->connectToHost(ui->lineEdit->text(),ui->lineEdit_2->text().toInt());
}


void Widget::on_pushButton_2_clicked()
{
        QString filePath = QFileDialog::getOpenFileName(this);   // 打开要发送的文件

        QFileInfo fileData(filePath);
        fileName = fileData.fileName();   // 记录打开的文件名
        fileSize = fileData.size();          // 记录打开的文件大小
        qDebug()<<"name:"<<fileName<<", size:"<<fileSize;
        if(!filePath.isEmpty)
        {
                ui->lable_3->setText(filePath);
                file.setFileName(filePath);
                file.open(QIODevice::ReadOnly);

        }
        ui->progressBar->setMaximun(0);
        ui->progressBar->SetMinum(fileSize/1024);
        ui->progressBar->setValue(0);
}

void Widget::on_pushButton_3_clicked()   // 按键3 发送
{
        // 第一次发送信息，即文件的头信息 "文件名+###"+文件大小
        QString head = fileName+"###"+QString::number(fileSize);
        qint64 length = tcpSocket->write(head.toUtf8());
        if(length>0)
        {
                myTimer->start(20);   // 20ms 的定时器，定时结束再开始真正发送文件
                qDebug()<<"send head successfully.";
        }
        else
        {
                qDebug()<<"failed to send head.";
                file.close();
        }

}
```

2. 服务器端 server

```
// 定义套接字
QTcpServer *tcpServer;   // 用于监听
QTcpSocket *clientConnection = nullptr;    // 用于通信
bool headInfo= true;
QString fileName;
qint64 fileSize;
qint64 recvSize;
QFile file;

// 初始化
tcpServer = new QTcpServer(this);
clientConnection = new QTcpSocket(this);


void Widget::on_pushButton_clicked()   // 点击连接
{
      // 监听网络

    tcpServer->listen(QHostAddress::LocalHost,8080);

    connect(tcpServer,&QTcpServer::newConnection,this,[&](){   // 监听到客户端有请求到来
      clientConnection = tcpServer->nextPendingConnection();   // 产生新的套接字，用于通信
      clientConnection->write("welcome to connect to server.");
      ui->textEdit->append("new connection......");

      // 客户端接收信号
      connect(clientConnection,&QTcpSocket::readyRead,this,[&](){
            QByteArray arr = clientConnection->readAll();

        if(headInfo)   // 第一次接收到信息，即文件的头信息 "文件名+###"+文件大小
          {
                headInfo = false;
                recvSize = 0;
                fileName = QString(arr).section("###",0,0);
                fileSize = QString(arr).section("###",1,1).toInt();
                file.setFileName(filePath);
              file.open(QIODevice::WriteOnly);


                ui->progressBar->setMaximun(0);
            ui->progressBar->SetMinum(fileSize/1024);
            ui->progressBar->setValue(0);

          }
```

```
            else
            {
                qint64 length = file.write(arr);
                if(length > 0)
                {
                    recvSize += length;
                }
                ui->progressBar->setValue(recvSize/1024);   // 为了让数字
不那么大，/1024
                if(recvSize == fileSize)
                {
                    QMessageBox::information(this, “OK”,"reveived
completed.");
                    file.close();
                }
            }

    });
            ui->pushButton->setEnable(false);   // 第一次监听收到信号后，就不
能再点击
}
```

## 4.2 文本协议的设计与实现

缓冲区概念：一段暂存数据的内存

1.发送缓冲区：

- 数据先进入发送缓冲区，之后由操作系统送往远程主机
- flush() 强制发送缓冲区中的数据立即被送走

2.接受缓冲区：

- 远端数据被操作系统接收后放入接收缓冲区
- bytesAvailable() 获取接收缓冲区中数据的字节数

为什么需要缓冲区：因为应用程序不能直接操作硬件（网卡）

---

问题，write("A"); write("B"); write("C"); 但是由于缓冲区，接收端不知道 ABC是一次一次发送的，还是一次性发送到的

**通过建立协议解决数据粘连问题**

网络设计中的期望

- 每次发送一条**完整**的消息，每次接收一条**完整**的消息
- 即使缓冲区中有多条消息，也**不会出现消息粘连**
- 消息中涵盖了**数据类型**和**数据长度**等信息

---

应用层协议设计

协议是通信双方为数据交换而建立的规则、标准或约定的集合

协议对数据传输的作用

- 通信双方根据协议能够正确收发数据
- 通信双方根据协议能够解释数据的意义

---

目标：基于TCP设计可用于文本传输的协议

数据头：数据类型（即：数据区用途，固定长度）

数据长度：数据区长度（固定长度）

数据区：字符数据（变长）

| 数据类型 | 数据长度 | 数据区 |
| --- | --- | --- |
| 4个字符 | 4个字符 | n个字符（由数据长度标识） |

通过计算数据消息的总长度，能够避免数据粘连的问题

## 4.2.1 从数据流装配文本协议对象

如何将接收缓冲区的数据装配成为协议对象

缓冲区数据量充足，能够装配不止一个对象，如何处理剩余数据；

数据量不足，能否达到协议最小长度，达到最小长度，但是无法产生一个对象

解决方案：

- 定义一个类用于接收字节流并装配协议对象
- 类中提供容器（队列）暂存字节流
- 当容器中至少存在 8 个字节时开始装配
  1.首先装配协议中的 类型（type）和数据长度（length）
  2.根据数据区长度从容器中取数据装配协议数据（data）
  3.当协议数据装配完成时，创建协议对象并返回，否则返回 NULL

总结：

从连续的字节流装配协议对象是应用自定义协议的基础

装配类（TextMsgAssembler） **用于解析自定义协议**

装配列的实现关键是**如何处理字节数据不够的情况**

自定义协议列和装配类能够**有效解决数据粘连问题**

## 4.2.2 文本协议中中文处理

中文类型的宽字符编码 宽字符 >= 2 字节

```
TextMessage mesg ("demo","威威位");
QString d = mesg.serialize();

/* 方式一 */
TextMessage unknown;
unknown.unserialize(d);
qDebug()<<unknown.type()<<", "<<unknown.length()<<", "<<unknown.data();
// 能够正常的识别中文

/* 方式二 */
TextMsgAssembler as;
QSharedPointer<TextMessage*> p;
p = as.assembler(d.toStdString().c_srt(), d.length());
qDebug()<<p->type<<", "<<p->length<<", "<<p->data();
// 不能正常的识别中文
```

---

原文本协议只考虑ASCII码的情况，不支持中文。ASCII是一个字节，而宽字符 >= 2字节

协议设计改动：

- Type: 4 个ASCII 字符
- Length: 4 个ASCII 字符（存储数据区字节数）
- 数据区：使用 **UTF-8** 方式进行编码

---

编码格式

ASCII：最早统一编码标准，规定了 128 个字符编码（一个字节表示一个字符）

Unicode： 一个很大的字符集，规定了字符的二进制代码（编码标准）

UTF-8：使用最广泛的一种 Unicode 编码标准的实现

特点：一种边长的编码方式，使用1-4个字节表示一个字符

英文字母，UTF-8与 ASCII 相同

### 4.2.3 文本协议的网络应用

- 客户端提供发送 TextMessage 对象的成员函数
- 客户端和服务端均内置 TextMsgAssembler 对象，用于从网络字节流装配 TextMessage 对象
- 当成功收到 TextMessage 对象，使用 TextMsgHandler 接口进行异步通

## 4.3 Qt 数据库

### 4.3.1 创建数据库

依靠的是 SQL 模块

| 驱动 | 数据库 |
| --- | --- |
| QDB2 | IBM DB2(7.1及更新) |
| QIBASE | BOrland InterBase |
| QMYSQL | MySQL |
| QOCI | Oracle Call Interface Driver |
| QODBC | Open Database Connectivity |
| QPSQL | PostgreSQL(7.3及更新) |
| QSQLITE2 | SQLite2 |
| **QSQLITE** | **SQLite3** |
| QSYMSQL | 针对 Symbian 平台的 SQLite3 |
| QTDS | Sybase Adaptive Server(自 Qt4.7 废除) |

SQL 的分层

**数据库 -> 驱动层 -> SQL 接口层（QSQLDatabase）-> 用户接口层**
（QSQL....Model 作用是将数据库链接到窗口部件）

用户使用的是接口层，驱动层是桥梁，

- 使用SQLite3，经常用于嵌入式

```
QT          += core gui sql    // 要使用数据库，添加 sql

#include <QSqlDatabase>
#include <QSqlError>   // 数据库错误相关

QStringList drivers = QSqlDatabase::drivers();   // 查看支持哪些数据库
foreach(QString d,drivers)
      qDebug()<<d;
// 创建数据库
      QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("database.db");
    if(!db.open())
     {
            qDebug()<<"Failed to onnect."<<db.lastError();
     }
    else
     {
            qDebug()<<"Connect to database successfully.";
     }
```

**SQLite 数据库中有很多命令和指令 ，网上可以找到 sqlite 语句大全，学习数据库就是学习数据库指令，以及在Qt中调用这些命令**

## 4.3.2 插入数据库_遍历数据库

数据库中分为不同的表格table，数据要存入 table 中

```
#include <QSQlQuery>   // 数据库命令


QSqlQuery sql_query;
// create a table student(int id, varchar name(30), int age )
QString create_sql = "create a table student( id int, name varchar(30),
age int )"
sql_query.prepare(create_sql);
```

```
    if(!sql_query.exec()}
    {
        qDebug()<<"Failed to create table."<<sql_query.lastError();
    }
    else
    {
        qDebug()<<"Table created."
    }

    // 添加数据
    // "insert into student values( ?, ?, ?)"
    QString insert_sql = "insert into student values( ?, ?, ?)";
    sql_query.prepare(insert_sql);
    sql_query.addBindValue(1);   // 三个数据，因此要三次
    sql_query.addBindValue("Rick");
    sql_query.addBindValue(18);
    if(sql_query.exec())
        qDebug() <<"Insert student no.1 OK";

    QString insert_sql = "insert into student values( ?, ?, ?)";
    sql_query.prepare(insert_sql);
    sql_query.addBindValue(2);   // 三个数据，因此要三次
    sql_query.addBindValue("Lihua");
    sql_query.addBindValue(19);
    if(sql_query.exec())
        qDebug() <<"Insert student no.2 OK";

    // 显示数据库
    // "select * from student"
    QString select_all_sql = "select * from student";
    sql_query.prepare(select_all_sql);
    if(sql_query.exec())
    {
        qDebug() <<"show database:\n";
        while(sql_query.next())
        {
            int id = sql_query.value(0).toInt();
            QString name = sql_query.value(1).toString();
            int age = sql_query.value(2).toInt();
            qDebug() << "id: "<<id<<", name:"<<name<<", age:"<<age;
        }
    }
```

### 4.3.3 更新_删除数据库

```
// 删除数据库
 “delete from student”
QString clear_sql =  “delete from student”
sql_query.prepare(clear_sql);
if(!sql_query.exec())
         qDebug() <<"failed to clear table.";
else
{
    qDebug() << "Table cleared";
}

//  更新数据库
"update student set name = :nm where id =n"
QString update_sql = "update student set name = :nm where id = :n";
sql_query.prepare(update_sql);
sql_query.bindValue(":nm","Michael");
sql_query.bindValue(":n","1");   //修改 id 为1的那一行 name 为 "Michael"
if(!sql_query.exec())
         qDebug() <<"failed to update table.";
else
    qDebug() << "Table updated";

 // 删除某一行
 “delete form student where id = n”
QString delete_sql =  “delete form student where id = ?” ;
sql_query.prepare(update_sql);
sql_query.bindValue("2");   // 删除 id 为2的那一行
if(!sql_query.exec())
     qDebug() <<"failed to delete table row NO.2.";
else
    qDebug() << "Table row NO.2 deleted";
```

### 4.3.4 数据库数据窗体访问，修改以及查找

```
QT          += core gui sql    // 要使用数据库，添加 sql

#include <QSqlDatabase>
#include <QSqlError>   // 数据库错误相关
#include <QSQlQuery>   // 数据库命令
```

以上是 库提供SQL 接口层，

**用户接口层（QSQL....Model 作用是将数据库链接到窗口部件）**

```
#include <QSqlQueryModel>
#include <QSqlTableModel>
#include <QSqlRelationModel>

#include <QTableView>   // ui 界面中的表格，用于显示数据库

QSqlTableModel *model;   // .h 中


// .c 中

model = new QSqlTableModel(this);
ui->tableView->setModel(model);
model->setTable(student);
model->select();
// 将 id，name，age 修改为中文，仅仅是显示改了
model->setHeaderData(0,Qt::Horizontal,"序号");
model->setHeaderData(1,Qt::Horizontal,"姓名");
model->setHeaderData(2,Qt::Horizontal,"年龄");


model->setEditStrategy(QSqlTableModel::OnManualSubmit);   // 修改后需要
submit 才会生效

connect(ui->pushButton1,&QPushButton::clicked(),this,[&](){model-
>submit()});   // 修改后点击按键提交修改
connect(ui->pushButton2,&QPushButton::clicked(),this,[&](){model-
>revertAll()});   // 撤销修改


// 查找

connect(ui->pushButton3,&QPushButton::clicked(),this,[&](){
      QString name = ui->lineEdit->text();
      QString nm = QString("name='%1' ").arg(name);
      // " ame = 'Tom' "
      model->setFilter(nm);   //
      model->select()
});
```

QString nm = QString("name='%1' ").arg(name);

将字符串 name 内容替换掉 %1