实验二　窗口、控件及基本绘图实验

一、 实验目的

了解使用QtCreator进行程序开发和调试的基本方法；

初步掌握窗口组件的创建、布局方法；

理解Qt的绘图系统和坐标系统；

正确处理鼠标事件；

掌握使用QPainter实现二维图形绘制的基本过程和基本方法。

二、 实验软件

- Qt 5.7.0
- MinGW 5.3.0 32bit
- Windows 7 系统及以上

三、 实验任务

编写一个具备基本绘图功能的简单绘图程序，用户可以选择不同的画笔进行绘制，也可以选择不同的形状绘制。功能如图1所示：

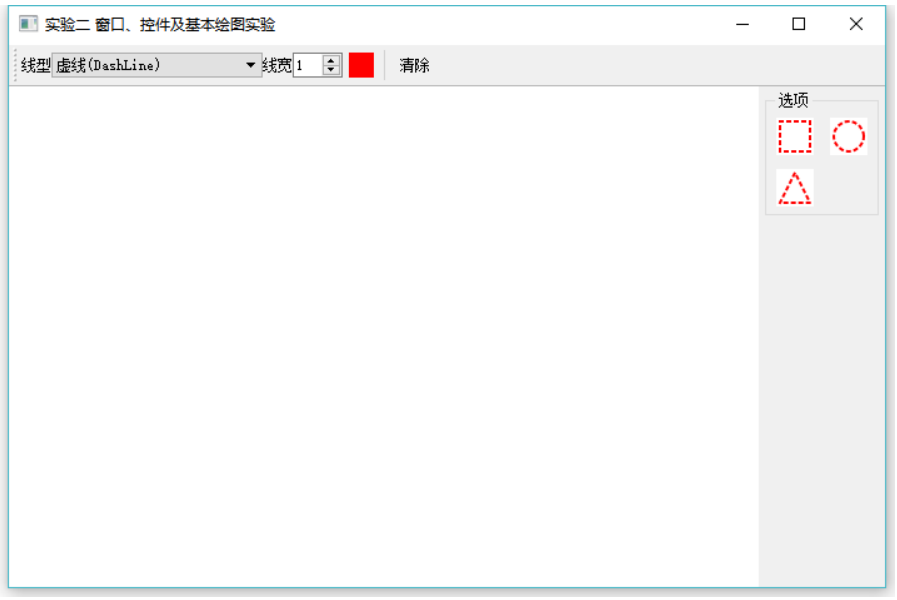


图 1 基本绘图程序

四、 实验原理与说明

1. 窗口组件的创建、布局方法

一般来说，Qt的基本窗口控件创建过程包括三部分：控件对象定义、创建控件并设置属性、将该控件加入到特定的布局Layout中。在上述过程中，正确合理设置属性（包括外观、显示文本以及信号槽的连接等）是控件使用的重要内容，本实验以QGroupBox、QPushButton、QLabel、QToolButton以及QCombobox等控件为例说明Qt简单窗口控件的基本使用方法。

根据需要，将已经创建完成的控件显示在合理的位置是布局的主要工作。Qt中的布局主要有水平布局、垂直布局、网格布局、窗体布局和堆栈布局5类，前三种布局较为常用，基本根据名称就能理解其功能。窗体布局用于类似网页表格界面的布局，堆栈布局用于容纳多个子窗口。一般情况下，很少直接使用堆栈布局，其功能在很多情况下可以用QStackedWindget替代。

```
// 矩形按钮
btnRect = new QPushButton(group);
btnRect->setToolTip("绘制矩形");
btnRect->setCheckable(true);
btnRect->setIconSize(p.size());
btnRect->setIcon (QIcon(p));   //p 是一个 QPixmap 对象
connect(btnRect,&QPushButton::clicked,
        this,&CenterFrame::on_btnRectClicked);
…
// 其他控件创建代码
…
// 选项 Group 布局
QGridLayout *gridLayout = new QGridLayout();
gridLayout->addWidget(btnRect,0,0);
…
// 在 gridLayout 布局中添加其他控件或布局
…

group->setLayout(gridLayout);  // group 是布局的容器父对象
```

上述代码首先创建了一个 QPushButton 对象并将其指针赋予 btnRect，随后几行代码设置了该按键的工具提示(Tool Tip)、可切换以及图标和图标的尺寸。Connect 语言将该按键的 clicked 信号与 on_btnRectClicked 槽函数连接起来。在布局的最后，将布局赋给容器父对象。

对于复制界面的设计，可以将上述方法嵌套使用，从而设计完成一个较为美观的用户界

面。

2. Qt 的绘图系统

Qt 的绘图系统允许使用统一的函数接口在屏幕或其它可绘制设备上进行绘图操作。整个绘图系统基于QPainter，QPainterDevice和QPaintEngine三个类，三者关系如图2所示。

图 2 绘图系统层次

QPainter是执行图像绘制操作的类；QPaintDevice是一个二维空间抽象绘图对象。

- QPainter：用于执行绘图操作。
- QPaintDevice：二维空间的抽象层，是QPainter的绘制对象。
- QPaintEngine：提供了统一的接口，用于QPainter在不同的设备上进行绘制。

可以把QPainter理解成画笔；把QPaintDevice理解成使用画笔的地方，比如纸张、墙壁等。使用Qt进行二维图形的绘制工作，也就是使用QPainter对象在QPaintDevice上的二维空间内进行绘图的过程。因为不同的绘图对象（QPaintDevice）的不同，绘制工具也会相应的不同（在纸上绘图用铅笔，在墙壁上绘图可能需要使用铁凿）。为统一QPainter与QPaintDevice之间的调用，Qt设计了一个中间层QPaintEngine类。该类提供了QPainter在不同的设备上进行绘制的统一的接口。QPaintEngine类位于QPainter和QPaintDevice之间，对开发人员来说是透明的。除非需要针对自定义设备进行绘图，否则一般情况下不用关心。

3. Qt 的画笔和画刷。

Qt 绘图系统定义了两个绘制时使用的关键类：画刷和画笔。前者使用QBrush描述，大多用于填充；后者使用QPen描述，大多用于绘制轮廓线。

QBrush定义了QPainter的填充模式，具有样式、颜色、渐变以及纹理等属性。

画刷的style()定义了填充的样式，使用Qt::BrushStyle枚举，默认值是Qt::NoBrush，也就是不进行任何填充。图2显示了不同画刷填充样式的区别：

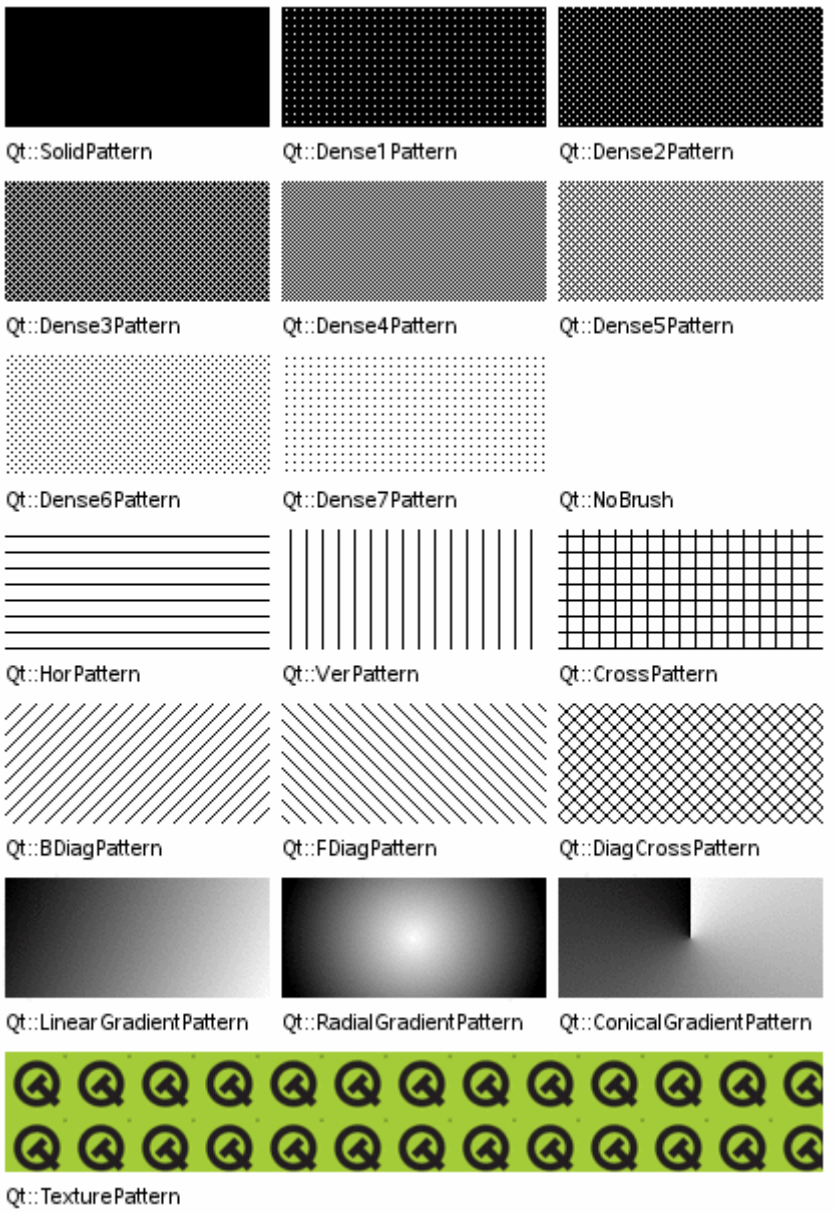


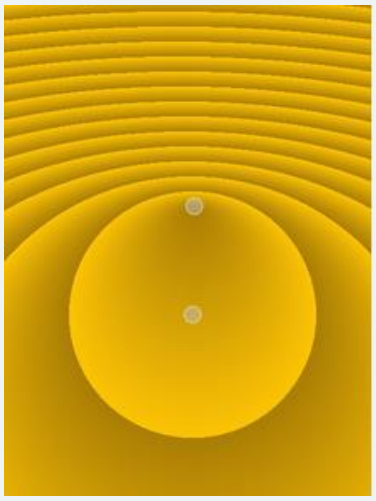
图 2 画刷类型

画刷的color()定义了画刷的填充颜色(QColor对象)。 Qt 预定义了部分QColor颜色常量，使用枚举类型Qt::GlobalColor表示，如Qt::red、Qt::blue等。

画刷的gradient()定义了渐变填充。这个属性只有在样式是Qt::LinearGradientPattern、Qt::RadialGradientPattern或者Qt::ConicalGradientPattern之一时才有效。渐变可以由QGradient对象表示。Qt 提供了三种渐变：QLinearGradient、QConicalGradient和QRadialGradient，它们都是QGradient的子类。我们可以使用如下代码片段来定义一个渐变的画刷：

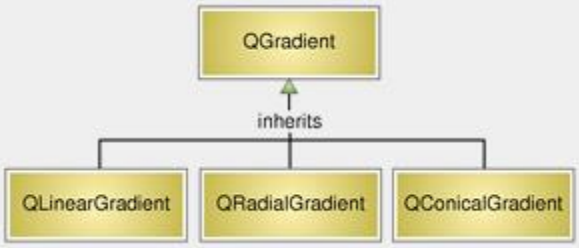
```
QRadialGradient gradient(50, 50, 50, 50, 50);
gradient.setColorAt(0, QColor::fromRgbF(0, 1, 0, 1));
gradient.setColorAt(1, QColor::fromRgbF(0, 0, 0, 0));
```

```
QBrush brush(gradient);
```



QGradient

QGradient 类用于指定画刷 QBrush 的渐变填充。



Qt 目前支持 3 种渐变：线性渐变在开始和结束点线型插值，径向渐变在焦点和结束点之间的环形平面内插值，锥形渐变以一个中心点进行插值。

图 3 Qt 中的渐变类型

QPen定义了QPainter画线或轮廓线的样式。画笔具有样式、宽度、画刷、笔帽样式和连接样式等属性。画笔的样式style()定义了线的样式。画刷brush()用于填充画笔所绘制的线条。笔帽样式capStyle()定义了使用QPainter绘制的线的末端；连接样式joinStyle()则定义了两条线如何连接起来。画笔宽度width()或widthF()定义了画笔的宽。注意，画笔宽度通常至少是 1 像素，不存在宽度为 0 的线。即使定义 width 为 0，QPainter依然会绘制出一条线宽为1像素的线。

```
QPainter painter(this);
QPen pen;                                   // 创建一个画笔

pen.setStyle(Qt::DashDotLine);              // 画笔风格为点划线
pen.setWidth(3);                            // 画笔宽度为 3px
pen.setBrush(Qt::green);                    // 画笔颜色为绿色
pen.setCapStyle(Qt::RoundCap);              // 画笔笔帽为圆形
pen.setJoinStyle(Qt::RoundJoin);            // 画笔连接使用圆形连接

painter.setPen(pen);                        // 设置 painter 的画笔
painter.setRenderHint(QPainter::Antialiasing);    //抗锯齿渲染
```

画笔的样式如图4所示：

图 4 画笔类型

笔帽定义了画笔末端的样式，如图5所示。

图 5 笔帽类型

画笔的连接样式定义了两端线之间的连接方式，如图6所示。

图 6 画笔连接方式

4. 绘图设备（Paint Device）

Qt 中 QPainter 类能够使用、进行绘制操作的绘图设备都是派生于 QPaintDevice 的子类。也就是说，在 Qt 程序中，除自定义绘图设备外，QPainter 能够绘制的对象是 QWidget、QImage、QPixmap、QPicture、QPrinter 和 QOpenGLPaintDevice 等类。

注意：当绘图设备(QPaintDevice)是 QWidget 对象时, QPainter 仅能在 paintEvent()函数或由 paintEvent()函数调用的函数内使用。

5. 坐标系统及渲染

QPaintDevice类是可以被绘制的对象的基类，它的绘图功能由QWidget、QImage、QPixmap、QPicture和QOpenGLPaintDevice继承。默认坐标系统位于设备的左上角，即坐标原点(0, 0)。X轴由左向右增加，Y轴由上向下增加。在基于像素的设备上（比如：显示器），坐标的默认单位是1像素；在打印机上则是1点（1/72 英寸）。

QPainter对象内部的坐标称为逻辑坐标，QPaintDevice内部设备的坐标称为物理坐标，逻辑坐标和屋里坐标之间的映射，由QPainter的变换矩阵（transformation matrix）、视口（viewport）和窗口（window）完成。默认情况下，物理坐标与逻辑坐标系统是重合的，QPainter支持坐标转换，例如：旋转、缩放。

(1) 图形的渲染

逻辑层面上，一个图形图元的大小（宽度和高度）总是对应于它的数学模型，而忽略绘制时画笔的宽度，如图 7 所示。此时逻辑矩形以数学模型为精确坐标点。

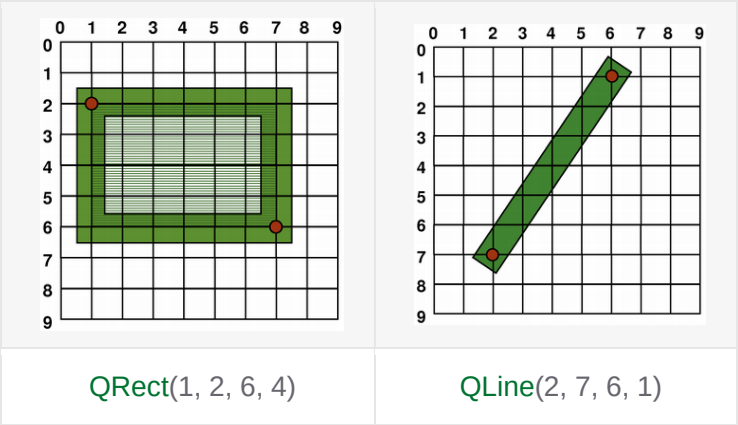


图 7 绘图时的逻辑表示

但是在物理设备上绘图时，由于像素点的不可分割性，默认情况下，QPainter 绘制的图像在渲染时会出现锯齿。带锯齿绘制图形时遵循以下规则：

- 如果画笔的宽度像素是偶数，则实际绘制会包裹住逻辑坐标值
- 如果是奇数，则是包裹住逻辑坐标值，再加上右下角1个像素的偏移。

图 8 显示了一个矩形在不同画笔宽度情况下的渲染结果。

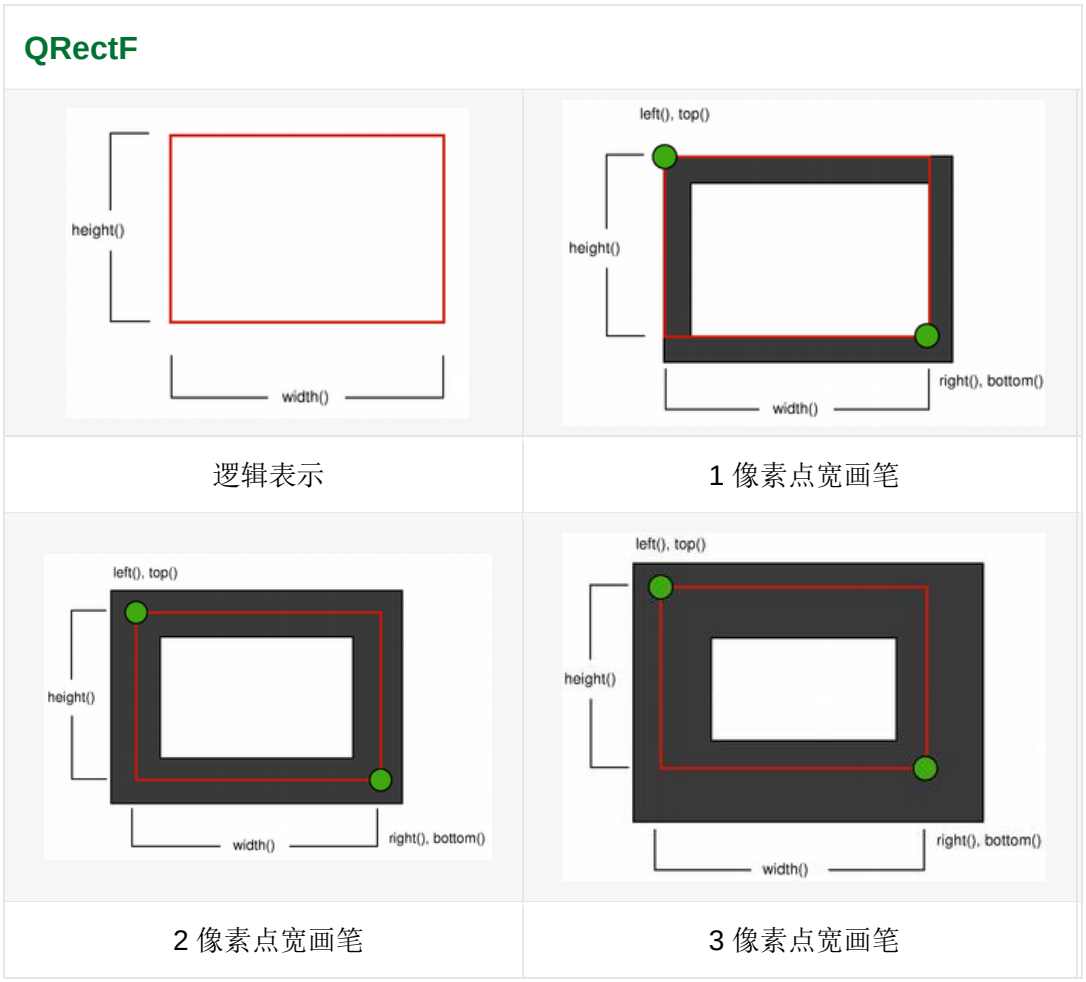


图 8 带锯齿时绘图渲染规则

注意：由于历史原因，QRect::right()和 QRect::bottom()的返回值并不是矩形右下角的真实坐标值。

QRect::right()返回：left() + width() – 1；QRect::bottom()返回：top() + height() – 1。上图中右下角的绿色点指出了这两个函数返回的坐标值。

为避免这个问题，建议是使用 QRectF。QRectF 使用浮点精度的坐标来定义一个平面矩形（QRect 则使用整形坐标）。函数 QRectF::right()和 QRectF::bottom()会返回真正的右下角坐标值。

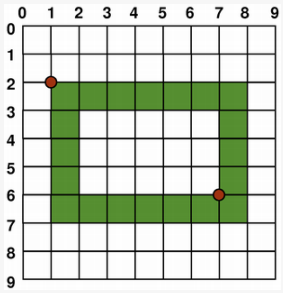
如果要使用 QRect，可以利用 x() + width()和 y() + height()来替代 right()和 bottom()。

如当用 1 像素宽的画笔绘制时，像素会被绘制在右下角。如图 9 所示。

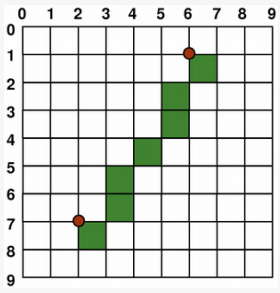
RenderHint 枚举变量指定了 QPainter 可以使用的渲染标志。如果希望 QPainter 使用抗锯齿模式进行图形绘制，可以使用 QPainter::setRenderHint 函数来设置渲染标志。如使用抗锯齿模式渲染：QPainter::Antialiasing，此时像素会均匀地绘制在两侧，通过使用不同的颜色亮度让图像边缘平滑，如图 10 所示。以下代码说明了如何设置 QPainter 的渲染模式：

```
    QPainter painter(this);

    // 抗锯齿必须在 painter 激活后，也就是绘制对象确定后设置
    painter.setRenderHint(QPainter::Antialiasing);
```

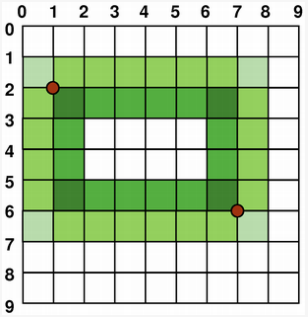


```
QPainter painter(this);
painter.setPen(Qt::darkGreen);
painter.drawRect(1, 2, 6, 4);
```

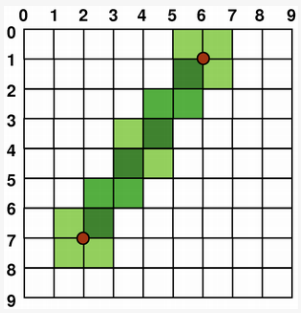


```
QPainter painter(this);
painter.setPen(Qt::darkGreen);
painter.drawLine(2, 7, 6, 1);
```

图 9　1 像素宽图像渲染结果



```
QPainter painter(this);
painter.setRenderHint(
        QPainter::Antialiasing);
painter.setPen(Qt::darkGreen);
painter.drawRect(1, 2, 6, 4);
```



```
QPainter painter(this);
painter.setRenderHint(
        QPainter::Antialiasing);
painter.setPen(Qt::darkGreen);
painter.drawLine(2, 7, 6, 1);
```

图 10　使用抗锯齿模式渲染图像

6.　事件的响应与处理

事件是 Qt 框架的核心机制之一，非常重要但又比较复杂。事件涉及到的函数很多，其处理方法也各有不同，让人难以学习和掌握选择。通常来说，Qt 有很多诸如鼠标事件、键盘事件、尺寸变化事件、绘图事件等事件。对这些事件的处理，Qt 提供了一对一的处理函数，只需要重载父类相应的事件处理函数，然后该重载函数内执行相应的处理命令即可。例如，假设由于外部原因（如遮挡）或内部原因（如调用 update 函数），用户界面需要更新时，

系统会触发一个 paintEvent 事件，表示程序界面需要进行绘制，那么只需要在需要绘制的 QWidget 对象中重载 paintEvent 虚函数，并在其中执行相应的绘制命令即可实现绘图功能。

本实验需要重载的事件主要有 2 大类：窗口绘制事件和鼠标事件。对鼠标事件而言又可以分为鼠标按下事件、鼠标移动事件、鼠标释放事件，分别对应 mousePressEvent、mouseMoveEvent 和 mouseReleasEvent。用户的一次绘图过程中，包括按下鼠标左键选定起点、拖动鼠标移动至目标点、在目标点释放鼠标左键等三个动作，因此程序需要处理上述三个事件。以下代码说明了鼠标事件的处理：

```
void DrawWidget::mousePressEvent (QMouseEvent *e)
{
    if(e->button() == Qt::LeftButton){
        startpos = e->pos();
        canDraw = true;
    }
}

void DrawWidget::mouseReleaseEvent(QMouseEvent *e)
{
   if (e->button() == Qt::LeftButton){
        endpos = e->pos();
        drawShape(startpos,endpos,shapeType);
        update();
        canDraw = false;
   }
}
```

mousePressEvent 函数首先根据 QMouseEvent 参数，判断按下的按键是否为左键，然后获取按下点的坐标，同时设置绘图标志，表示绘图动作的开始。mouseReleaseEvent 函数刚好相反，获得结束点坐标之后，开始绘制图形，具体的图形绘制工作在函数 drawShape 中完成。绘图完成后，必须调用 update 函数以更新用户界面，同时设置绘图标志为 false，表明本次绘图完成。

update 函数会发送一个界面更新事件，系统会自动调用 paintEvent 事件处理函数，重新绘制用户界面。在本实验中，绘图实际上并不是直接绘制在窗口中，而是绘制在一个非可见的 QPixmap 对象上。如果希望看到绘制的图形，每次绘图动作完成（用户松开鼠标左键）后，需要将 QPixmap 对象上的内容及时显示到用户界面上，因此调用 update 函数，要求系统更新界面。在界面更新的 paintEvent 函数中，将 QPixmap 对象的内容复制到窗口里来。

如果将 mouseReleaseEvent 函数中的 update 函数删去，那么绘图完成后窗口内的图像并不改变。只有窗口尺寸变化或被其他程序遮挡、界面更新后，才能看到刚才绘制的图像。

五、 系统设计

本实验作为一个小型程序，有多种实现方案，以下方案仅作为参考方案。

用户界面划分为 3 个部分，一个主窗口 MainWindow，一个容器窗口 CenterFrame，一个绘图区域窗口 DrawWidget。三个窗口分别派生于 QMainWindow、QFrame 和 QWidget，其派生关系如图 11 所示。

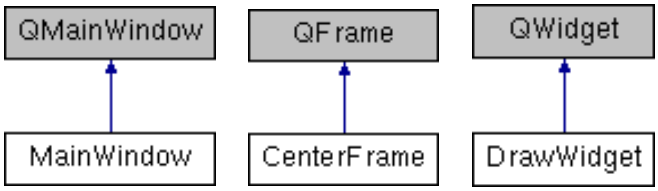


图 11 窗口类派生关系

MainWindow 作为与用户交互的主要窗口，负责工具栏菜单等窗口控件的创建；CenterFrame 作为一个容器，包含了一个快捷功能区和一个绘制区；绘制区 DrawWidget 主要工作是处理绘图任务，三个类共同完成程序的功能。

1. MainWindow 类的构造及析构函数

```
MainWindow::MainWindow (QWidget *  parent = 0)
```

参数:

parent	父窗口指针

主窗口构造包括工具栏和绘图窗口的创建， 绘图窗口是一个QFrame派生类，该QFrame子类对象相当于一个窗口容器，其内部包括了一个绘图区和一个工具面板。

```
MainWindow::~MainWindow ()
```

析构函数 在销毁主窗口时释放相应的资源

2. MainWindow 类的成员函数

```
void MainWindow::createToolBar ()
```

创建工具栏按键菜单

在该函数内部，创建了2个Label、2个下拉框、1个颜色选择按钮和1个清除按钮，QToolButton在使用时类似QPushButton，两者的区别在于以下2点： 1) QToolButton多与QToolBar结合使用，其布局由QToolBar的Layout控制 QPushButton多作为单独按键使用，（QToolButton也可以作为单独控件使用，与QPushButton一样） 2) 两者触发信号不同：

QPushButton仅有一个clicked信号， QToolButton除clicked信号外，还有一个triggled信号（用于QAction的响应），

```
void MainWindow::penColorChangged ()[slot]
```

处理用户选择不同的颜色

该函数在用户选取不同的颜色时，将画笔设置为用户选择的新颜色，

```
void MainWindow::penStyleChangged (int  index = 0)[slot]
```

处理用户选择不同的画笔风格

参数:

index	画笔风格索引

该函数在用户选取不同的画笔风格时，读取用户选取项的用户数据(userData)， 并依据该数据设置不同的画笔风格。

3. MainWindow 类的成员变量

```
CenterFrame* MainWindow::centerFrame[private]
```

用户界面中心窗口

```
QToolButton* MainWindow::clearBtn[private]
```

“清除”工具栏按键

```
QToolButton* MainWindow::colorBtn[private]
```

颜色选择工具栏按键

```
QComboBox* MainWindow::styleComboBox[private]
```

画笔风格下拉框

```
QSpinBox* MainWindow::widthSpinBox[private]
```

画笔线宽spinbox

```
QLabel* MainWindow::styleLabel[private]
QLabel* MainWindow::widthLabel[private]
```

4. MainWindow 的完整实现代码

```
MainWindow::MainWindow(QWidget *parent)
  : QMainWindow(parent)
{
```

```
  centerFrame = new CenterFrame; //新建 CenterFrame 对象
  //新建的 centerFrame->insideWidget()对象作为主窗口的中央窗口
  setCentralWidget (centerFrame);
  createToolBar (); //创建一个工具栏
  setMinimumSize (600, 400); //设置主窗体的最小尺寸
  setWindowTitle(tr("实验二 窗口、控件及基本绘图实验"));

  //初始化线型，设置控件中当前值作为初始值
  penStyleChangged(styleComboBox->currentData().toInt());

  centerFrame->setPenWidth (widthSpinBox->value ()); //设置初始线宽
  centerFrame->setPenColor (Qt::red); //设置初始颜色
}

MainWindow::~MainWindow()
{
}

void MainWindow::createToolBar ()
{
  QToolBar *toolBar = addToolBar (tr("Tool")); //为主窗口创建一个工具栏对象

  // 线型选择下拉框
  styleLabel = new QLabel(tr("线型"));
  styleComboBox = new QComboBox;
  styleComboBox->setToolTip(tr("选择画笔线型"));
  // 此处在下拉框的每个 Item 里，设置了一个 item 的 userData，
  // 该用户数据作为用户选择的标志
  styleComboBox->addItem (tr("实线(SolidLine)"),
        static_cast<int>(Qt::SolidLine));
  styleComboBox->addItem (tr("虚线(DashLine)"),
        static_cast<int>(Qt::DashLine));
  styleComboBox->addItem (tr("点线(DotLine)"),
        static_cast<int>(Qt::DotLine));
  styleComboBox->addItem (tr("点划线(DashDotLine)"),
        static_cast<int>(Qt::DashDotLine));
  styleComboBox->addItem (tr("连点划线(DashDotDotLine)"),
        static_cast<int>(Qt::DashDotDotLine));
   connect (styleComboBox, static_cast<void(QComboBox::*)(int)>
        (&QComboBox::currentIndexChanged),
          this, &MainWindow::penStyleChangged);
  styleComboBox->setCurrentIndex(1);

  // 线宽选择框
```

```
  widthLabel = new QLabel(tr("线宽"));
  widthSpinBox = new QSpinBox;
  widthSpinBox->setToolTip(tr("选择画笔线宽"));
  widthSpinBox->setRange(1,50); //线宽范围
  connect (widthSpinBox, static_cast<void(QSpinBox::*)(int )>
       (&QSpinBox::valueChanged),
  centerFrame, &CenterFrame::setPenWidth);
  widthSpinBox->setValue(1);

  // 颜色选择框
  colorBtn = new QToolButton;
  QPixmap pixmap(20, 20);
  pixmap.fill (Qt::red);
  colorBtn->setIcon (QIcon(pixmap));
  colorBtn->setToolTip(tr("选择画笔颜色"));
  connect (colorBtn, &QToolButton::clicked,
       this, &MainWindow::penColorChangged);

  // 创建清除工具栏
  clearBtn = new QToolButton;
  clearBtn->setText (tr("清除"));
  clearBtn->setToolTip(tr("清除当前画板"));
  connect (clearBtn, &QToolButton::clicked,
        centerFrame, &CenterFrame::clearPaint);

  // 向工具栏上添加各个控件
  toolBar->addWidget (styleLabel);
  toolBar->addWidget (styleComboBox);
  toolBar->addWidget (widthLabel);
  toolBar->addWidget (widthSpinBox);
  toolBar->addWidget (colorBtn);
  toolBar->addSeparator();
  toolBar->addWidget (clearBtn);
}

void MainWindow::penStyleChangged (int index)
{
  Q_UNUSED(index)
  centerFrame->setPenStyle(styleComboBox->currentData().toInt ());
}

void MainWindow::penColorChangged ()
{
  QColor color = QColorDialog::getColor (static_cast<int>(Qt::red),
```

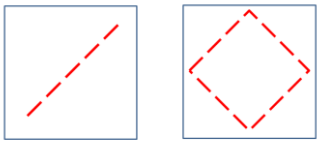
```
                                        this, tr("选取画笔颜色"));

  //判断颜色是否有效
  if(color.isValid ())
  {
       centerFrame->setPenColor (color);
       QPixmap p(20, 20);
       p.fill (color);
       colorBtn->setIcon (QIcon(p));
  }
}
```

其他代码不再赘述，请参考工程文档。

六、 预习及实验要求

1. 认真阅读本说明及参考示例；
2. 阅读 Qt 文档：Layout Management 和 The Event System，并将其按照原文档格式译为中文（代码不需翻译）。
3. 模仿示例代码，在绘图区右侧的功能窗口上增加 2 个新按钮，分别是绘制直线功能和绘制菱形功能，示例图标如下：



4. 参考 Qt 文档，添加一个显示文本的按键，当用户按下后，出现一个文本框，在鼠标点击拖动并释放后，在鼠标拖动矩形中心位置处显示出用户输入的文本。
5. 本实验不得使用界面编辑器制作用户界面。