第13篇 (2D绘图) 坐标系统

代码地址：https://github.com/Lornatang/QtStartQuicklyTutorial/tree/main/Painter02

目录

第一部分 Qt坐标系统应用

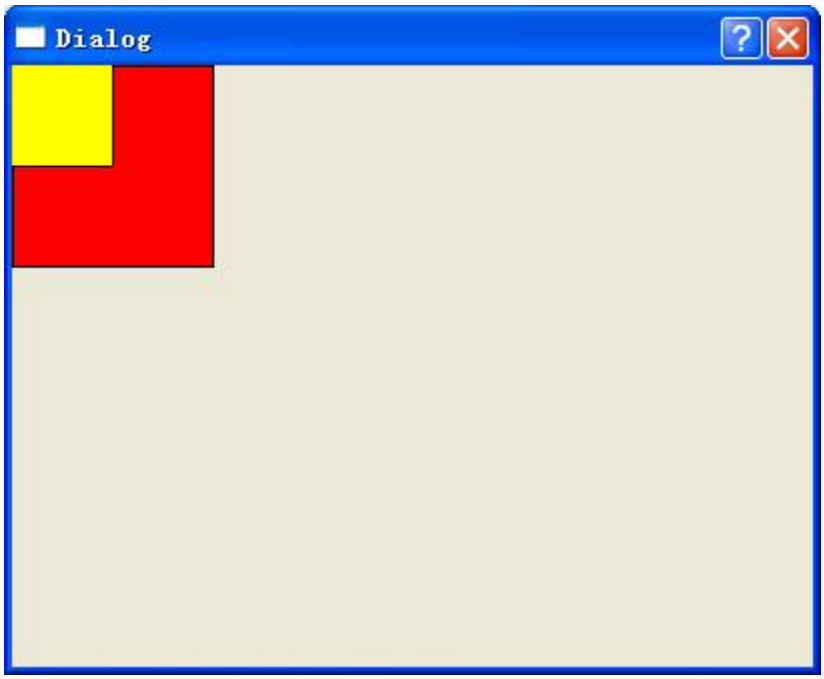
一、坐标系统简介

Qt中每一个窗口都有一个坐标系统，默认的，窗口左上角为坐标原点，水平向右依次增大，水平向左依次减小，垂直向下依次增大，垂直向上依次减小。原点即为`(0, 0)`点，以像素为单位增减。

下面仍然在上一节的程序中进行代码演示，更改`paintEvent()`的内容如下：

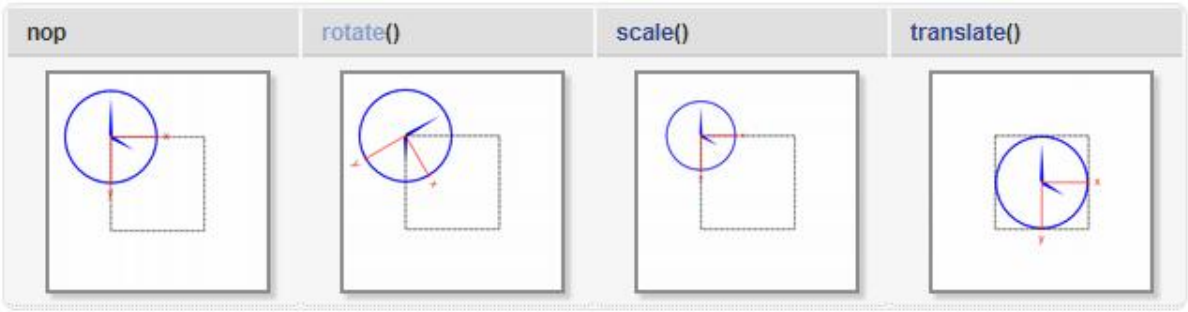
```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setBrush(Qt::red);
    painter.drawRect(0, 0, 100, 100);
    painter.setBrush(Qt::yellow);
    painter.drawRect(-50, -50, 100, 100);
}
```

我们先在原点 `(0, 0)` 绘制了一个长宽都是100像素的红色矩形，又在 `(-50, -50)` 点绘制了一个同样大小的黄 色矩形。可以看到，我们只能看到黄 色矩形的四分之一部分。运行程序，效果如下图所示。



二、坐标系统变换

默认的， `QPainter` 在相关设备的坐标系统上进行绘制，在进行绘图时，可以使用 `QPainter::scale()` 函数缩放坐标系统；使用 `QPainter::rotate()` 函数顺时针旋转坐标系统；使用 `QPainter::translate()` 函数平移坐标系统；还可以使用 `QPainter::shear()` 围绕原点来扭曲坐标系统。如下图所示。

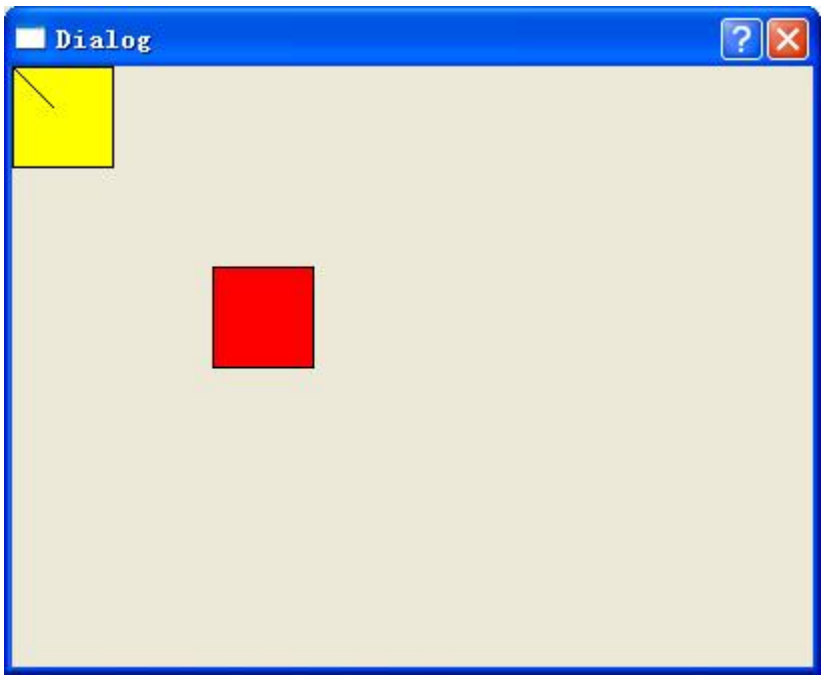


坐标系统的2D变换由`QTransform`类实现，我们可以使用前面提到的那些便捷函数进行坐标系统变换，当然也可以通过`QTransform`类实现。

1．平移变换。将`paintEvent()`函数内容更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    // 平移变换
    QPainter painter(this);
    painter.setBrush(Qt::yellow);
    painter.drawRect(0, 0, 50, 50);
    painter.translate(100, 100);  //将点（100，100）设为原点
    painter.setBrush(Qt::red);
    painter.drawRect(0, 0, 50, 50);
    painter.translate(-100, -100);
    painter.drawLine(0, 0, 20, 20);
}
```

运行程序，效果如下图所示。

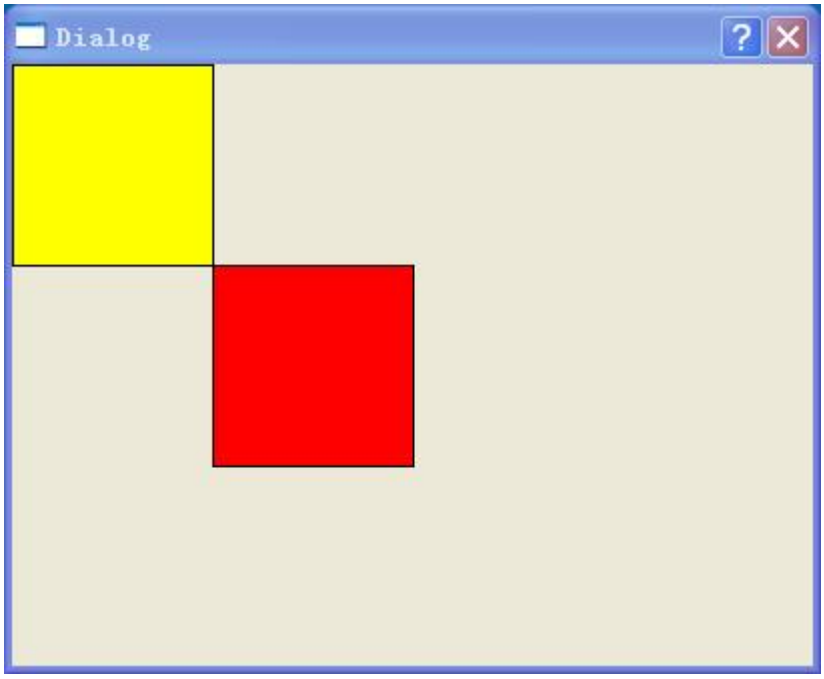


这里先在原点`(0, 0)`绘制了一个宽、高均为50的正方形，然后使用`translate()`函数将坐标系统进行了平移，使`(100, 100)`点成为了新原点，所以我们再次进行绘制的时候，虽然`drawRect()`中的逻辑坐标还是`(0, 0)`点，但实际显示出来的却是在`(100, 100)`点的红色正方形。可以再次使用`translate()`函数进行反向平移，使原点重新回到窗口左上角。

2．缩放变换。将`paintEvent()`函数中的内容更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    // 缩放
    QPainter painter(this);
    painter.setBrush(Qt::yellow);
    painter.drawRect(0, 0, 100, 100);
    painter.scale(2, 2); //放大两倍
    painter.setBrush(Qt::red);
    painter.drawRect(50, 50, 50, 50);
}
```

运行程序，效果如下图所示。



可以看到，当我们使用`scale()`函数将坐标系统的横、纵坐标都放大两倍以后，逻辑上的`(50, 50)`点变成了窗口上的`(100, 100)`点，而逻辑上的长度50，绘制到窗口上的长度却是100。

3．扭曲变换。将`paintEvent()`函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
            // 扭曲
            QPainter painter(this);
            painter.setBrush(Qt::yellow);
            painter.drawRect(0, 0, 50, 50);
            painter.shear(0, 1); //纵向扭曲变形
            painter.setBrush(Qt::red);
            painter.drawRect(50, 0, 50, 50);
}
```

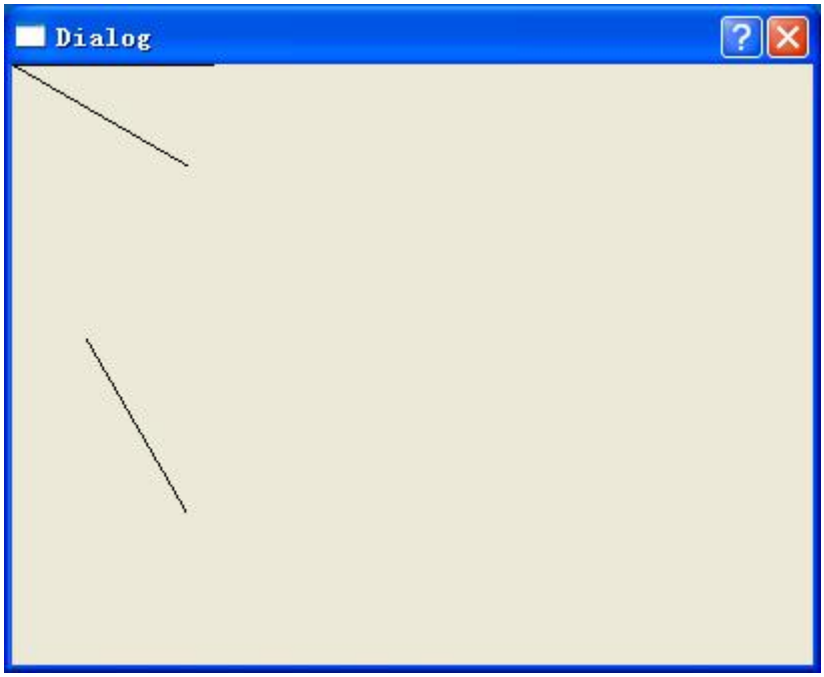
运行程序，效果如下图所示。

`shear()`有两个参数，第一个是对横向进行扭曲，第二个是对纵向进行扭曲，而取值就是扭曲的程度。比如程序中对纵向扭曲值为1，那么就是红色正方形左边的边下移一个单位，右边的边下移两个单位，值为1就表明右边的边比左边的边多下移一个单位。大家可以更改取值，测试效果。

4．旋转变换。将`paintEvent()`函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    // 旋转
    QPainter painter(this);
    painter.drawLine(0, 0, 100, 0);
    painter.rotate(30); //以原点为中心，顺时针旋转30度
    painter.drawLine(0, 0, 100, 0);
    painter.translate(100, 100);
    painter.rotate(30);
    painter.drawLine(0, 0, 100, 0);
}
```

运行程序，效果如下图所示。

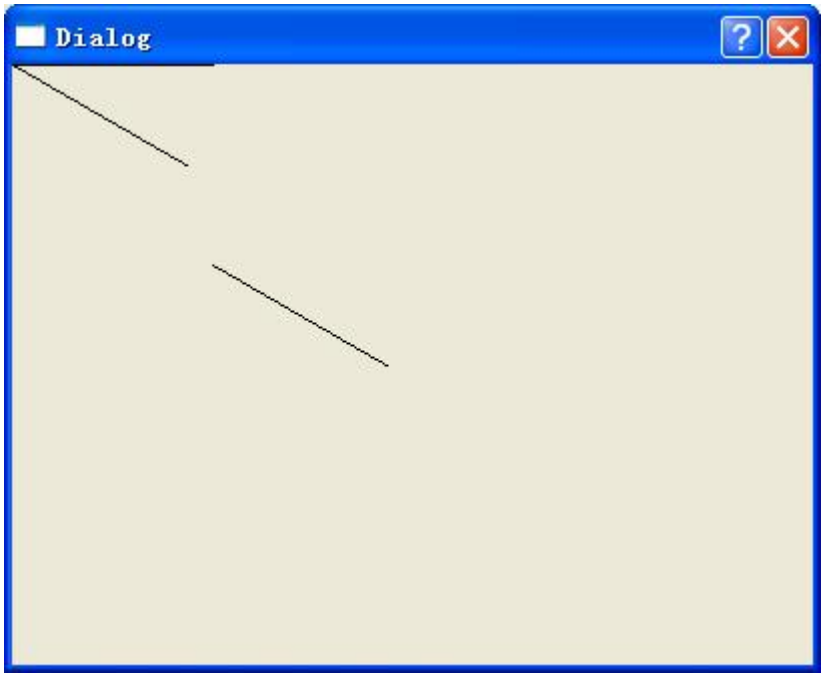


这里先绘制了一条水平的直线，然后将坐标系统旋转了30度，又绘制了一条直线。可以看到，默认是以原点`(0, 0)`为中心旋转的。如果想改变旋转中心，可以使用`translate()`函数，比如这里将中心移动到了`(100, 100)`点，然后旋转了30度，又绘制了一条直线。我们的本意是想在新的原点从水平方向旋转30度进行绘制，可是实际效果却超过了30度。这是由于前面已经使用`rotate()`函数旋转过坐标系统了，后面的旋转会在前面的基础上进行。

下面我们再次更改`paintEvent()`函数：

```
void Dialog::paintEvent(QPaintEvent *)
{    // 旋转
      QPainter painter(this);
      painter.drawLine(0, 0, 100, 0);
      painter.rotate(30); //以原点为中心，顺时针旋转30度
      painter.drawLine(0, 0, 100, 0);
      painter.rotate(-30); // 反向旋转
      painter.translate(100, 100);
      painter.rotate(30);
      painter.drawLine(0, 0, 100, 0);
}
```

运行程序，效果如下图所示。



这次我们在移动原点以前先将坐标系统反向旋转，可以看到，第二次旋转也是从水平方向开始的。

其实，前面讲到的这几个变换函数都是如此，他们改变了坐标系统以后，如果不进行逆向操作，坐标系统是无法自动复原的。针对这个问题，下面我们将讲解两个非常实用的函数来实现坐标系统的保存和还原。

三、坐标系统的保存

我们可以先利用`save()`函数来保存坐标系现在的状态，然后进行变换操作，操作完之后，再用`restore()`函数将以前的坐标系状态恢复，其实就是一个入栈和出栈的操作。下面来看一个具体的例子，更改`paintEvent()`函数如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.save(); //保存坐标系状态
    painter.translate(100,100);
    painter.drawLine(0, 0, 50, 50);
    painter.restore(); //恢复以前的坐标系状态
    painter.drawLine(0, 0, 50, 50);
}
```

运行程序，效果如下图所示。利用好这两个函数，可以实现坐标系快速切换，绘制出不同的图形。

第二部分 坐标系统深入研究

在第一部分，我们主要学习了常用的一些坐标变换，虽然在编程中，这些变换已经可以满足大部分的应用需求。不过，大家是否也感觉到现在对坐标的变换依然很模糊，没有一个透彻的认识。下面咱们就一点一点来研究一下坐标系统的变换。

一、获得坐标信息

前面图形的变换都是我们眼睛看到的，为了更具有说服力，下面将获取具体的坐标数据，通过参考数据来进一步了解坐标变换。

1．首先在`dialog.h`文件中添加头文件包含：

```
#include <QMouseEvent>
```

然后添加一个`protected`鼠标事件处理函数声明：

```
void mousePressEvent(QMouseEvent *);
```

2．到`dialog.cpp`文件中，先添加头文件包含：

```
#include <QDebug>
```

然后添加函数定义：

```
void Dialog::mousePressEvent(QMouseEvent *event)
{
        qDebug() << event->pos();
}
```

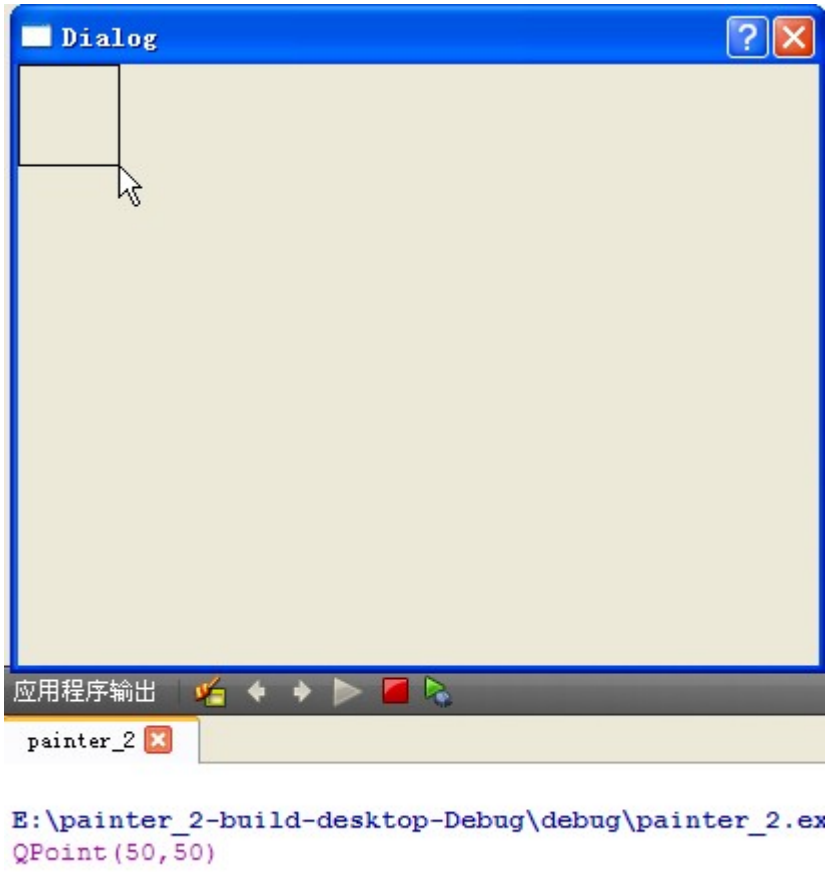
这里应用了`qDebug()`函数，该函数可以在程序运行时将程序中的一些信息输出到控制面板，在QtCreator中会将信息输出到其下面的“应用程序输出”窗口。这个函数很有用，在进行简单的程序调试时，都可以利用该函数进行。我们这里利用它将鼠标指针的坐标值输出出来。

3．将`paintEvent()`函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
```

```
    painter.drawRect(0, 0, 50, 50);
}
```

现在运行程序，然后将鼠标在绘制的正方形右下角顶点处点击，在QtCreator的应用程序输出窗口就会输出相应点的坐标信息。如下图所示。大家也可以点击一下其他的地方，查看输出信息。

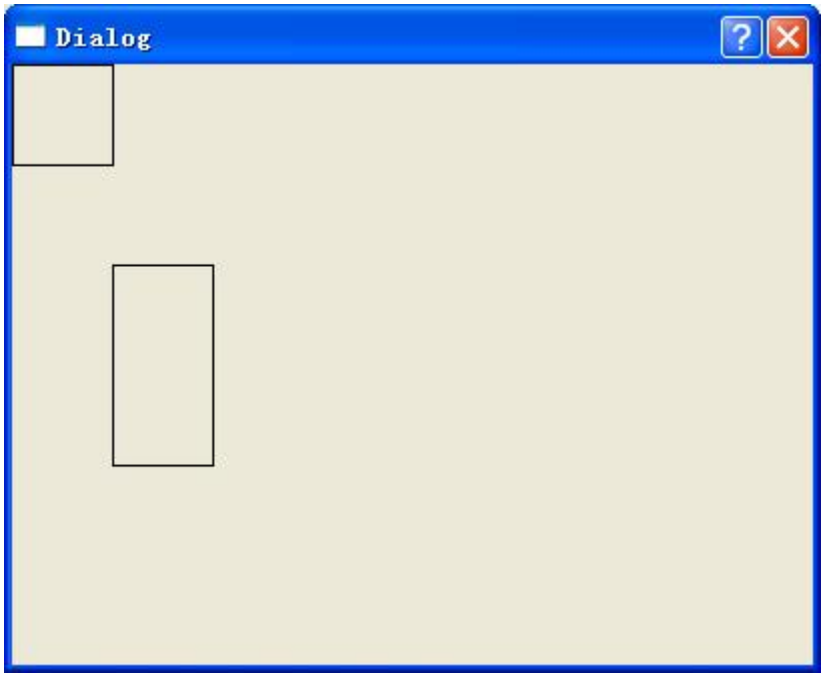


二、研究变换后的坐标系统

1．首先研究放大后的坐标系统，将`paintEvent()`函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    // 放大
    QPainter painter(this);
    painter.drawRect(0, 0, 50, 50);
    painter.scale(1, 2);
    painter.drawRect(50, 50, 50, 50);
}
```

这里，我们将纵坐标放大了两倍，而横坐标没有改变。运行程序，效果如下图所示。

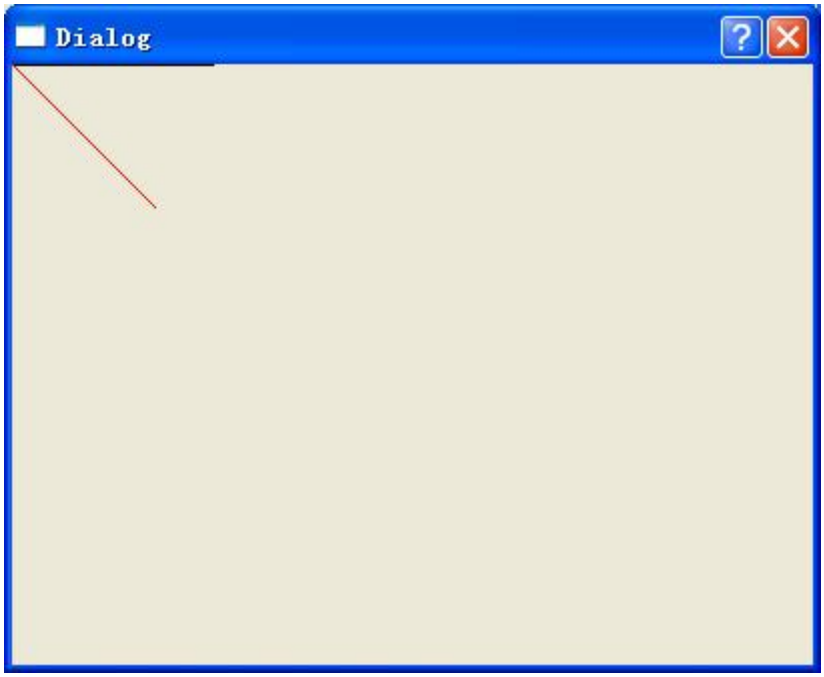


大家可以查看一下第二个矩形的各个顶点的坐标，左上角是`(50, 100)`也就是说纵坐标扩大了两倍，查看其它点，会发现左右两条边长都变成了100。

2．研究旋转后的坐标系统。修改`paintEvent()`函数如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawLine(0, 0, 100, 0);
    painter.rotate(45);
    painter.setPen(Qt::red);
    painter.drawLine(0, 0, 100, 0);
}
```

这里我们先绘制了一条水平的直线，然后将坐标系统旋转45度，再次绘制了一条相同的红色直线。运行程序，效果如下图所示。



大家可以查看一下各处的坐标，虽然旋转后直线位置发生了变化，但是坐标其实是没有变化的。 我们也可以利用这种方法来测试一下应用其他变换函数后坐标的变化，这里就不再赘述。

三、研究绘图设备的坐标系统

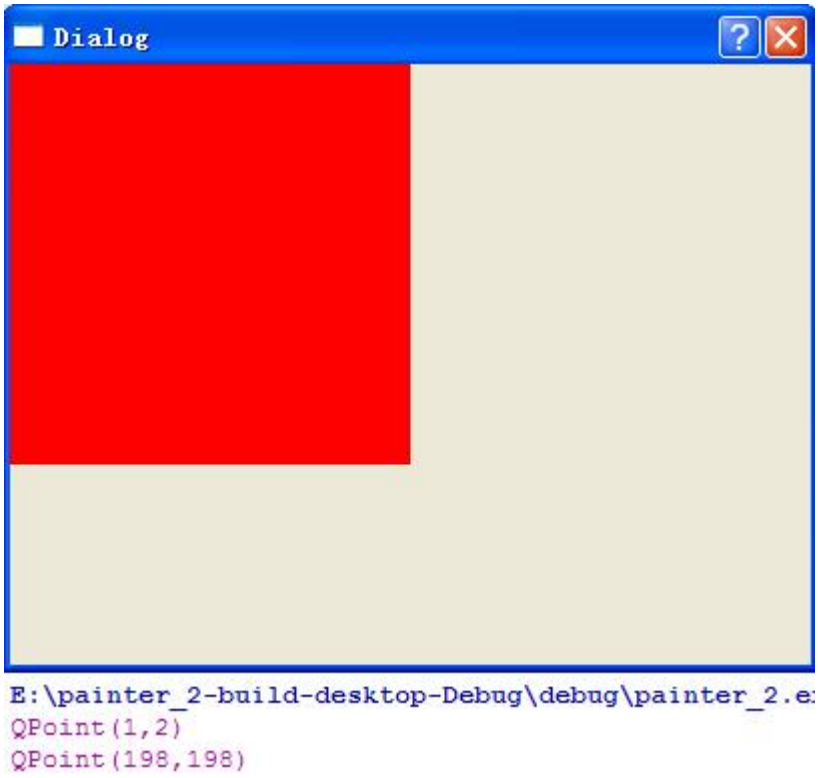
除了可以在`QWidget`等窗口部件上进行绘制以外，还可以在`QPixmap`、`QImage`等上面进行绘制，这些均称为绘图设备。下面我们就以`QPixmap`为例，来研究一下它的坐标系统。

1．首先更改`paintEvent()`函数如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200, 200);
    pix.fill(Qt::red);   //背景填充为红色
    painter.drawPixmap(0, 0, pix);
}
```

在前面我们已经讲过，`QPixmap`可以用来显示图片。其实`QPixmap`本身就是一个绘图设备，可以在它上面直接绘图。这里先生成了一个宽和高都是200像素的`QPixmap`类对象（注意，必须在构建时指定其大小），然后为其填充了红色，最后在窗口的原点进行了绘制。为了表述方面，下面将`QPixmap`对象称为画布，这里就是先绘制了一个红色画布。

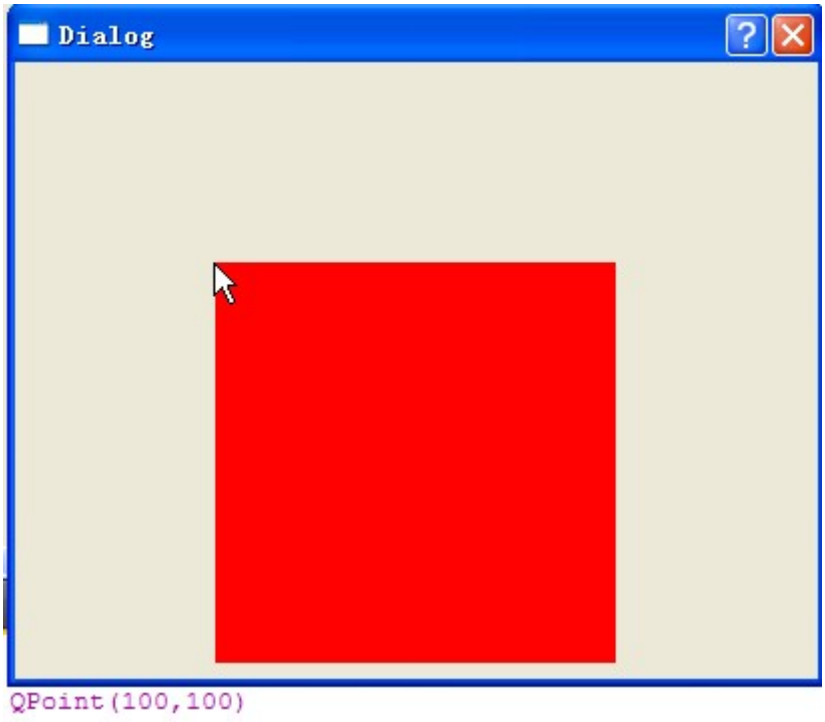
我们运行程序，并在红色画布的左上角和右下角分别点击，查看输出的坐标。如下图所示。因为点击位置的误差，所以两个点可能不是顶点。



2．下面我们接着更改`paintEvent()`的代码：

```
void Dialog::paintEvent(QPaintEvent *)
{
      QPainter painter(this);
      QPixmap pix(200, 200);
      pix.fill(Qt::red);   //背景填充为红色
      painter.drawPixmap(100, 100, pix);
}
```

这次我们在`(100, 100)`点重新绘制了画布，现在运行程序，发现画布左上角坐标确实为`(100,100)`，这个就是我们窗口中的坐标，是没有什么疑问的。效果如下图所示。

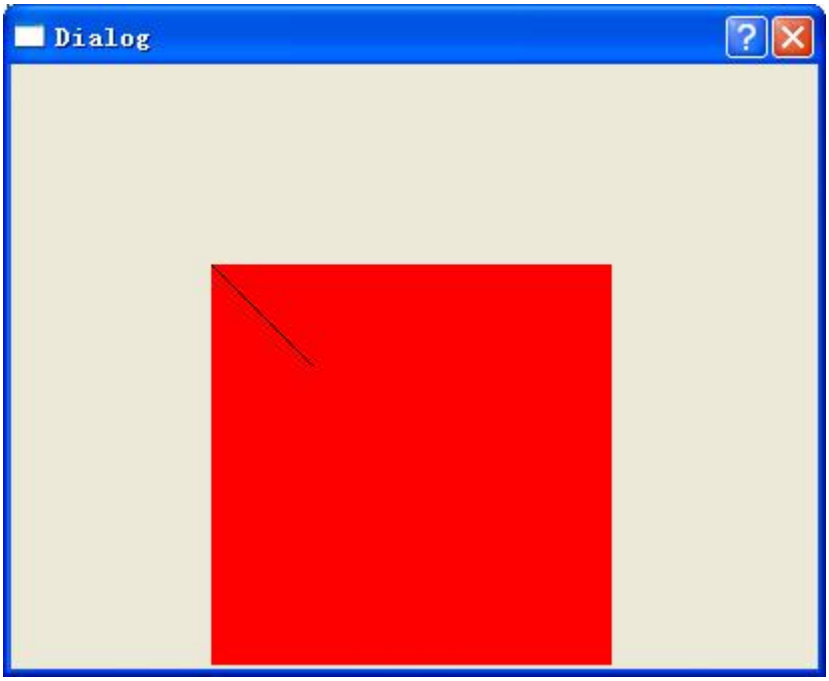


窗口和画布都是绘图设备，那么画布本身有没有自己的坐标系统呢？我们接着研究！

3．我们继续更改`paintEvent()`函数：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200, 200);
    pix.fill(Qt::red);
    //新建QPainter类对象，在pix上进行绘图
    QPainter pp(&pix);
    //在pix上的（0，0）点和（50，50）点之间绘制直线
    pp.drawLine(0, 0, 50, 50);
    painter.drawPixmap(100, 100, pix);
}
```

这里我们为画布`pix`创建了一个`QPainter`对象`pp`，注意这个`pp`只能在画布上绘画，然后我们在画布上绘制了一条从原点`(0, 0)`开始的直线。运行程序，效果如下图所示。



可以看到，直线是从画布的左上角开始绘制的，也就是说，画布也有自己的坐标系统，坐标原点在画布的左上角。

下面补充说明一下：`QPainter painter(this)`，`this`就表明了是在窗口上进行绘图，所以利用`painter`进行的绘图都是在窗口部件上的，`painter`进行的坐标变换，是变化的窗口的坐标系；而利用`pp`进行的绘图都是在画布上进行的，如果它进行坐标变化，就是变化的画布的坐标系。

而通过坐标数值，我们可以得出下面两条结论：

第一，`QWidget`和`QPixmap`各有一套坐标系统，它们互不影响。可以看到，无论画布在窗口的什么位置，它的坐标原点依然在左上角，为`(0,0)`点，没有变。

第二，我们所得到的鼠标指针的坐标值是窗口坐标系统的，不是画布的坐标。

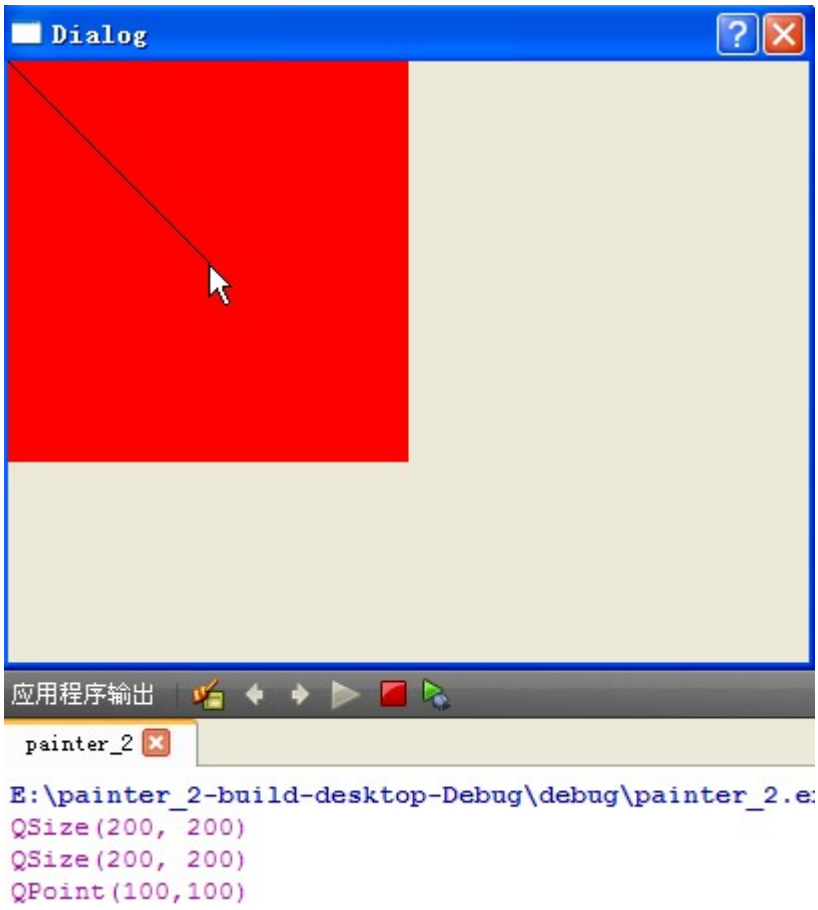
4．下面这个例子将对比分析扩大窗口坐标或画布坐标的异同。

首先将`paintEvent()`函数更改如下：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200,200);
    //放大前输出画布的大小
    qDebug() << pix.size();
    pix.fill(Qt::red);
    QPainter pp(&pix);
    //画布的坐标扩大2倍
    pp.scale(2, 2);
    //在画布上的（0，0）点和（50，50）点之间绘制直线
    pp.drawLine(0, 0, 50, 50);
```

```
    //放大后输出画布的大小
    qDebug() << pix.size();
    painter.drawPixmap(0, 0, pix);
}
```

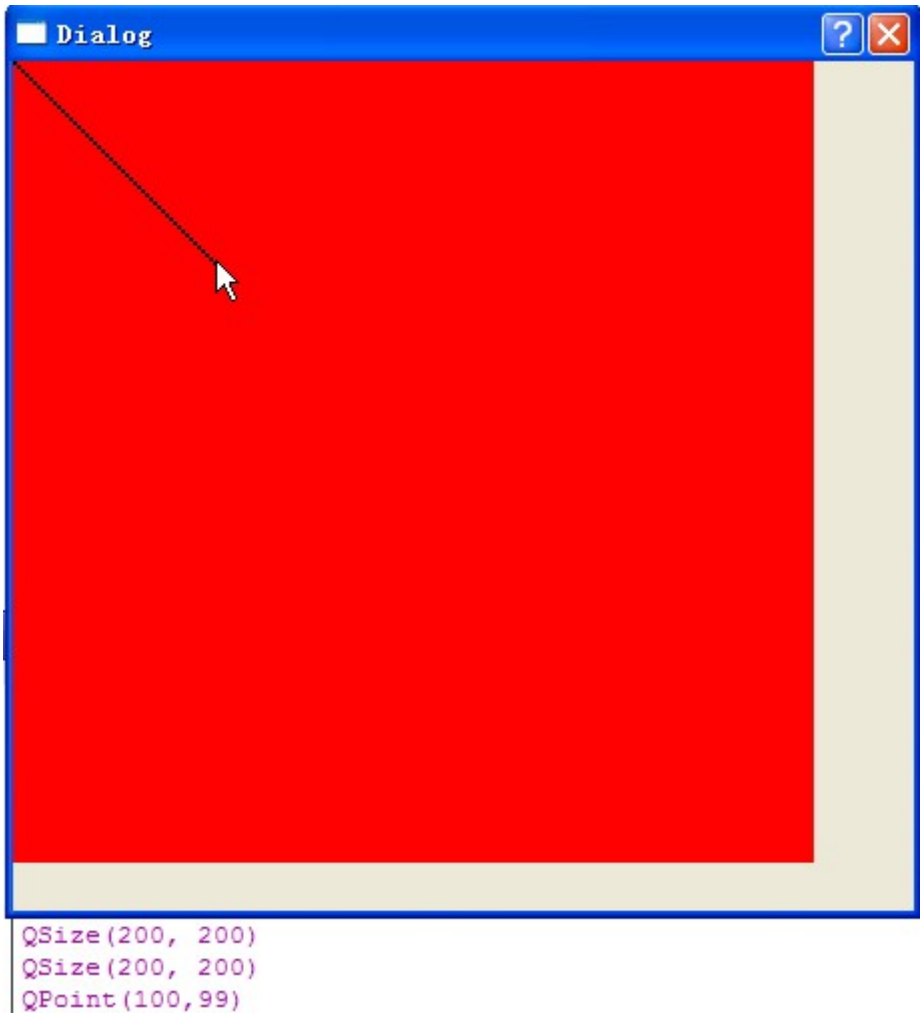
这里我们将画布坐标系统放大了两倍，然后从原点开始绘制了一条直线，并分别输出了画布放大前后的大小。运行程序，效果如下图所示。



下面再次更改`paintEvent()`函数：

```
void Dialog::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QPixmap pix(200,200);
    qDebug() << pix.size();
    //窗口坐标扩大2倍
    painter.scale(2,2);
    pix.fill(Qt::red);
    QPainter pp(&pix);
    pp.drawLine(0, 0, 50, 50);
    qDebug() << pix.size();
    painter.drawPixmap(0, 0, pix);
}
```

这里与前面唯一的不同是：这里放大了窗口的坐标系统，而前面放大的是画布的坐标系统。运行程序，效果如下图所示。



可以看到，整个画布的可见面积变大了。直线虽然长度依然是100，但是这次的效果跟前面明显不同，因为是窗口坐标变大，所以在上面绘出的线条有了明显的颗粒感。

上面两个程序虽然最终输出的数据是一样的，但实际效果还是有很大不同的。大家可以根据需要进行选择性应用。