

QT4.7 界面设计指南

作者:向仔州

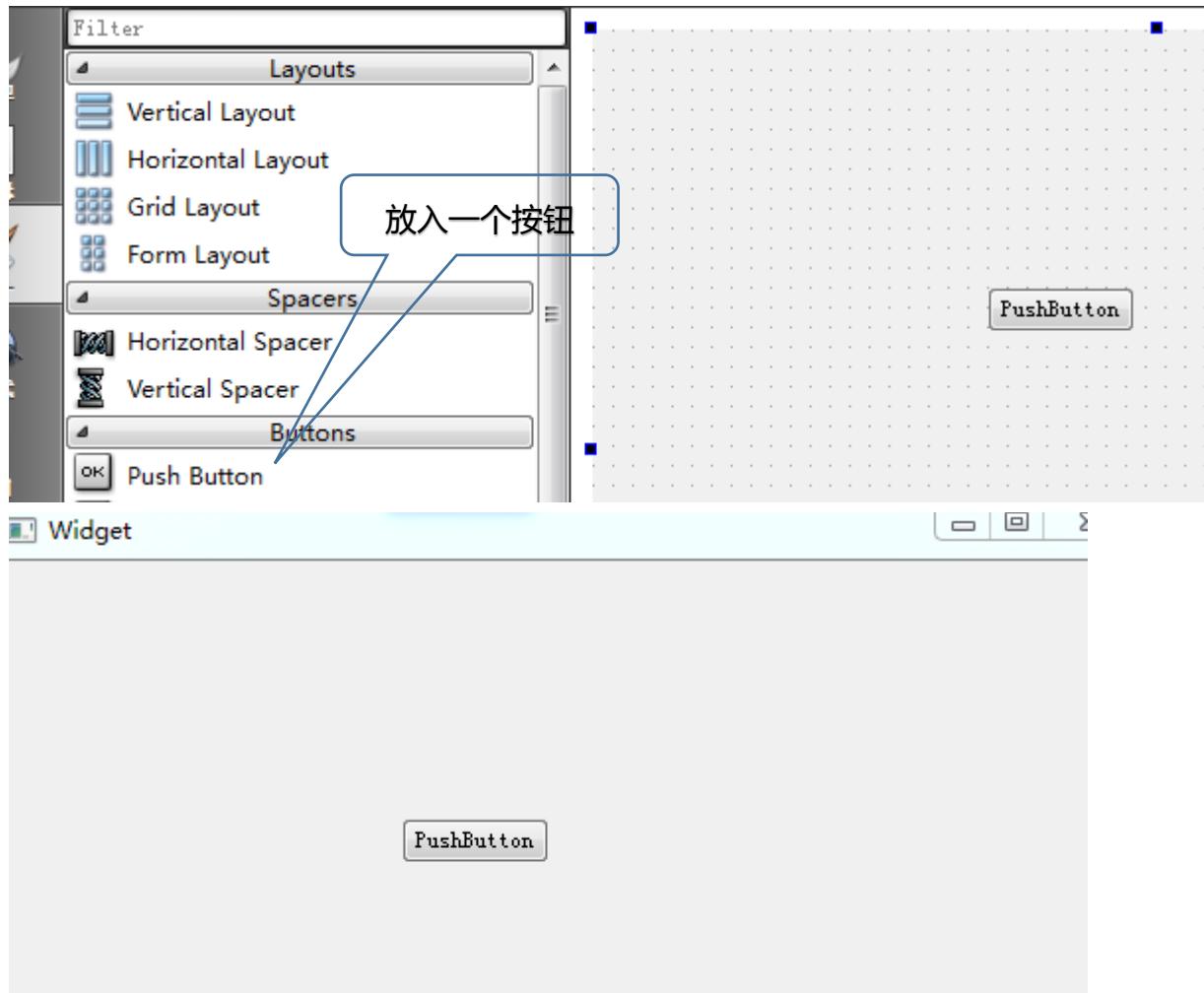
按钮界面设计.....	3
代码实现按钮外形样式.....	5
代码实现按钮外形样式.....	6
用 QSS 按钮样式模拟软件来生成按钮的样式代码.....	11
QLineEdit 输入字符对话框样式设置 , (重点介绍盒子模型).....	12
padding 的使用.....	15
QLabel 界面设计.....	16
QInputDialog 输入对话框界面设计.....	19
QPainter 在界面上画图.....	20
QPainter 画图片.....	23
用绘图设备画出来的图形保存在电脑上.....	25
用绘图设备画出来的图形保存在电脑上.....	25
给每个主窗口的背景图片.....	26
设置主窗口背景图片透明.....	28
点击按钮播放音效 , 或者打开软件启动背景音乐.....	29
设计按钮点击时候产生抖动效果.....	29
如何设计主界面切换窗口.....	30
控件显示字符无法实现中文编码问题.....	34
按钮按下时和松开时按钮图形变化,不用 QSS , 而是用 QT 原生鼠标事件来做.....	35
设置一个控件的背景透明色图片.....	38
QLabel 产生多图切换动画效果 , 当然其它控件也可以参考这个思路.....	40
用 QMovie 类来替代定时器执行 gif 动画.....	42
让一个图片突然滑动出现的效果.....	42
无边框窗口实现 , 偏平化界面设计.....	45
qlineargradient 线性渐变使用介绍.....	48
用 border 设置 widget 窗口的边框来达到阴影和透明.....	49
QT4.7 曲线库 qcustomplot 移植.....	50
设置曲线颜色.....	53
设置曲线下方和 x 轴之间用透明颜色填充.....	53
我想增加一条或者两条以上的曲线.....	54
增加鼠标移动曲线和缩放功能.....	55
如何让曲线窗口尺寸随着我 widget 主窗口尺寸变大缩小.....	55
QCPGraph 类在曲线上打点.....	55
设置曲线形状.....	56

QCPGraph 类两条曲线之间产生阴影，这样看起很舒服.....	57
QCPBars 类用来画条型柱状图.....	59
如何在坐标图上画两个柱状图？	60
设置坐标图的背景颜色和坐标 xy 轴的刻度线风格.....	62
有时候曲线的某个点的值在坐标图上看起不太舒服，改成鼠标指到曲线某个点就输出某个点坐标... ...	63
我们其实可以将鼠标 x 轴和 y 轴的数字放在鼠标箭头上跟着鼠标移动显示.....	65
QCPIItemText 类使用方法.....	68
在曲线图中实现移动十字线功能.....	71
下面实现水平和垂直的十字线实时移动.....	73
柱状图叠加图形设计.....	75
设置坐标轴刻度线，和坐标轴的其它一些属性.....	81
电池库使用.....	83
音量条显示库使用.....	84
圆形计量计库使用.....	86
仪表盘库使用.....	88
垂直液位表库使用.....	89
3D 液位条库使用.....	91

按钮界面设计

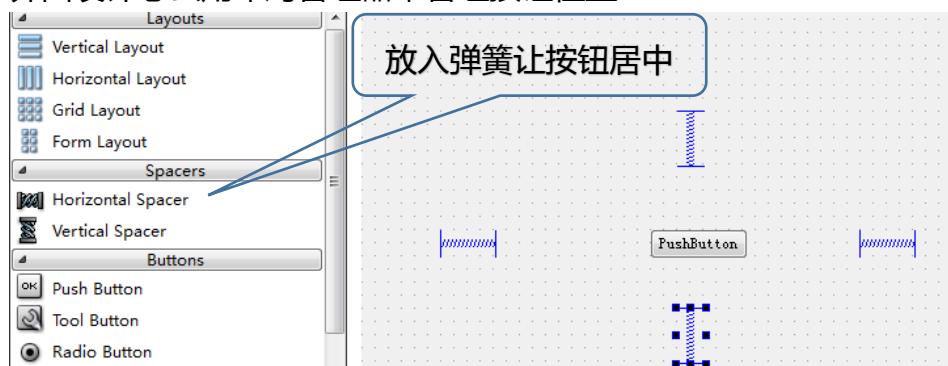
▶ 头文件
▶ 源文件
 main.cpp
 widget.cpp
▶ 界面文件
 widget.ui

进入 ui 界面



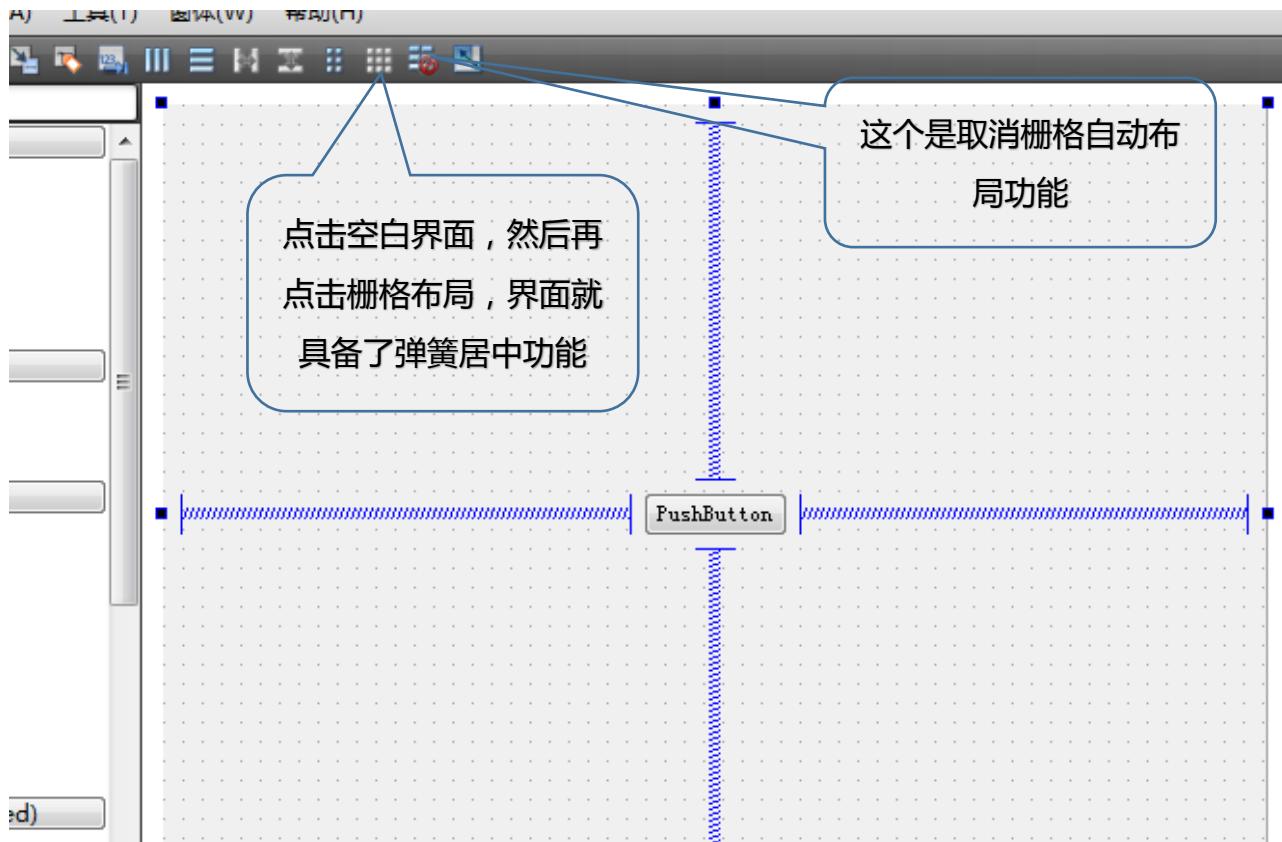
运行之后发现按钮并没有居中

界面设计怎么用布局管理器来管理按钮位置？

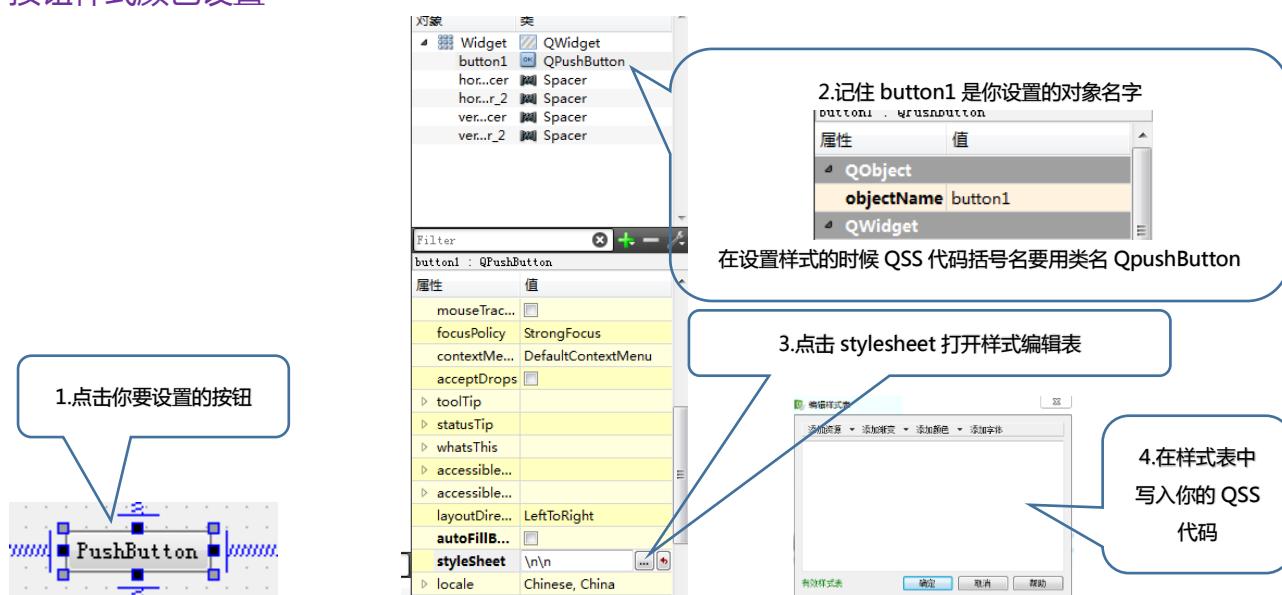


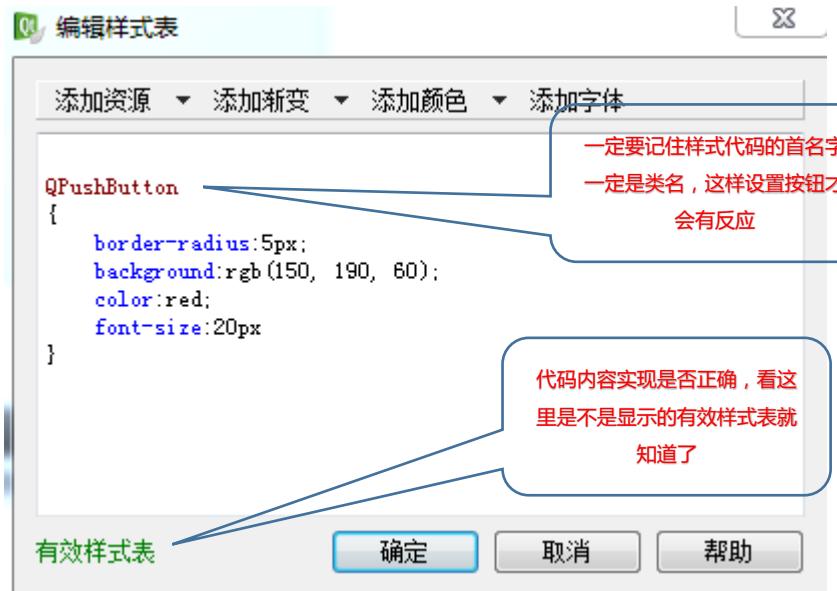
但是我放入弹簧怎么感觉按钮还是没有居中呢？

这是因为你没有设置界面的布局方式



按钮样式颜色设置





其实界面设计不好用还是代码好用。

代码实现按钮外形样式

和 QT4.7 程序设计指南一样，要先给按钮加资源文件，资源文件里面放图片

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    QPushButton button1(&w);

    button1.setText("xzz"); //其实可以不用这个字体
    button1.move(100,100);
    button1.resize(100,100); //设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大
    button1.setStyleSheet("QPushButton{border-image:url(:/xzz/png/01.png);}"
                         "QPushButton:hover{border-image:url(:/xzz/png/02.png);}"
                         "QPushButton:pressed{border-image:url(:/xzz/png/03.png);}");
    button1.show();
    w.show();

    return a.exec();
}
```

按钮添加方法和 QT4.7 程序设计指南代码一样，只不过这里多加了一个 `setStyleSheet` 函数，用来添加 url，就是给按钮添加图片路径

这里一定指明是 QPushButton 类，这是按钮类

这是按钮本身初始形状

这是鼠标点击按钮的形状

这是鼠标移动到按钮的形状

```

button1.setText("xzz");//其实可以不用这个字体
button1.move(100,100);
button1.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大
button1.setFlat(true);//实现按钮透明，用PNG图片时很有用
button1.setStyleSheet("QPushButton{border-image:url(:/xzz/png/01.png);}"
                     "QPushButton:hover{border-image:url(:/xzz/png/02.png);}"
                     "QPushButton:pressed{border-image:url(:/xzz/png/03.png);}")
button1.show();
w.show();

```

实现按钮透明

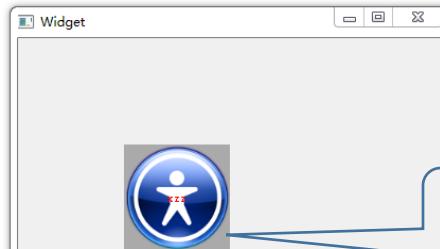
用 QSS 设置按钮颜色样式

```

button1.setStyleSheet("QPushButton{border-image:url(:/xzz/png/01.png);}" //鼠标没有
                     "QPushButton:hover{border-image:url(:/xzz/png/02.png);}" //hover鼠
                     "QPushButton:pressed{border-image:url(:/xzz/png/03.png);}"
                     "QPushButton{color:red;background-color:#aaaaaa}");//设置按钮字体为红色
button1.show();
w.show();

return a.exec();

```



虽然是红色了，但是按钮的背景区域被显示出来了，看着不舒服，如何处理？

这是因为你 QSS 的 background-color 加的按钮背景颜色是#aaaaaa

```

button1.setStyleSheet("QPushButton{border-image:url(:/xzz/png/01.png);}" //鼠标
                     "QPushButton:hover{border-image:url(:/xzz/png/02.png);}" //hover
                     "QPushButton:pressed{border-image:url(:/xzz/png/03.png);}"
                     "QPushButton{color:red;background-color:transparent;}");//设置按钮与
button1.show();
w.show();

return a.exec();
}

```



这些按钮背景色问题就解决了哦

如果是多个按钮就要建立多个 set.StyleSheet(...)

```

button1.setText("xzz");//其实可以不用这个字体
button1.move(100,100);
button1.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大

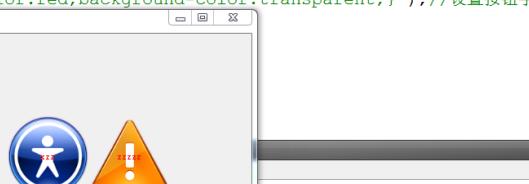
button1.setStyleSheet("QPushButton{border-image:url(:/xzz/png/01.png);}" //鼠标
                     "QPushButton:hover{border-image:url(:/xzz/png/02.png);}" //hover
                     "QPushButton:pressed{border-image:url(:/xzz/png/03.png);}"
                     "QPushButton{color:red;background-color:transparent;}");//设置按钮字
button1.show();

button2.setText("zzzz");//其实可以不用这个字体
button2.move(200,100);
button2.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大

button2.setStyleSheet("QPushButton{border-image:url(:/xzz/png/04.png);}" //鼠标
                     "QPushButton:hover{border-image:url(:/xzz/png/05.png);}" //hover
                     "QPushButton:pressed{border-image:url(:/xzz/png/06.png);}"
                     "QPushButton{color:red;background-color:transparent;}");//设置按钮字
button2.show();

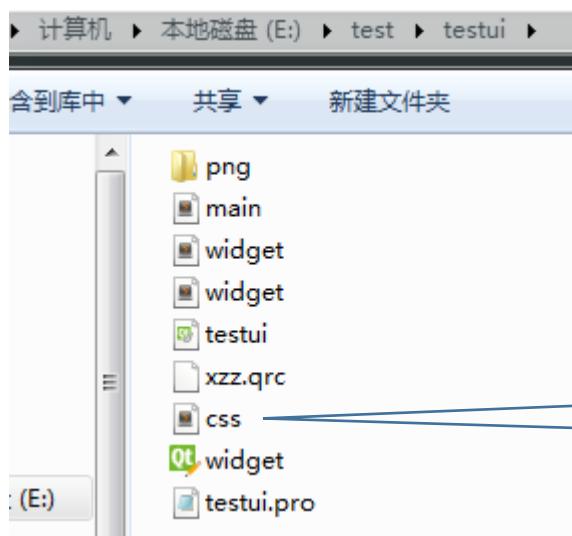
w.show();

```



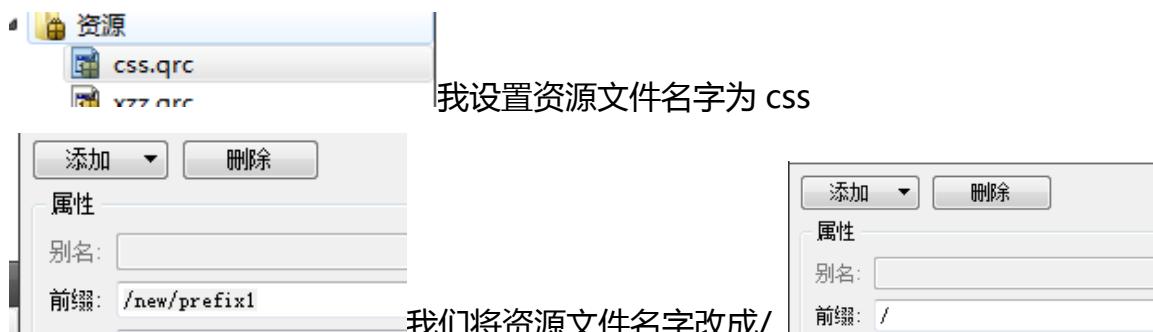
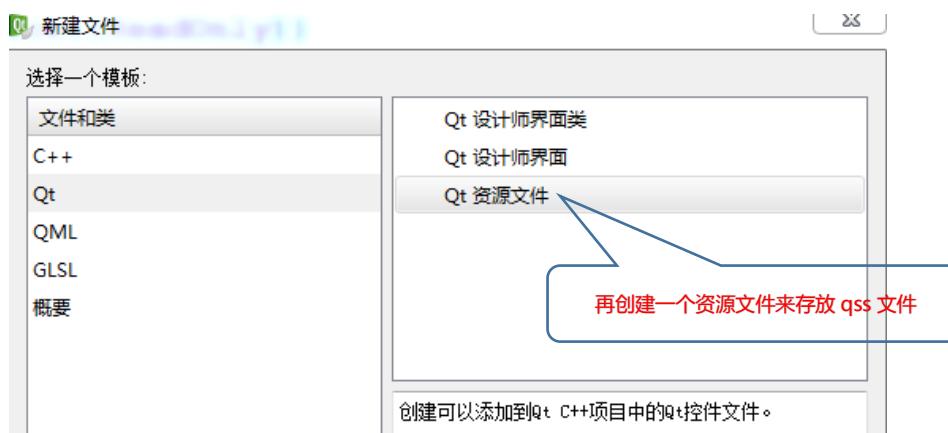
这对多个按钮不同按钮样式有用。如果我多个按钮样式一样，我就要写多个样式吗？

我们可以单独建立一个 QSS 文件，然后把所有一样样式的按钮放在 QSS 文件里面设置

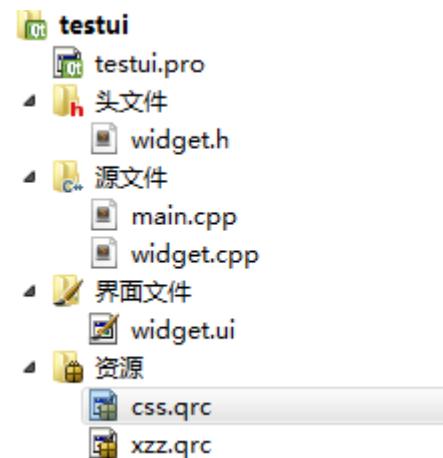


```
1 QPushButton
2 {
3     color:red;
4 }
```

A screenshot of a code editor showing a file named 'css.qss'. The content is a single-line style rule for a QPushButton. A callout bubble points to the code with the text: '在 qss 文件里面直接写 QPushButton 这种类名 (叫做类选择器)，表示设置的颜色属性是所有按钮都会设置上，说白了就是按钮类产生的按钮都会是同一个样式属性'.



方便代码 QFile 好找 QSS 文件的路径



在源代码中创建 file 类倒入 QSS 文件

```

1  QApplication a(argc, argv);
2  QFile qss(":/css.qss");
3  if(qss.open(QIODevice::ReadOnly))
4  {
5      a.setStyleSheet(QLatin1String(qss.readAll()));
6      qss.close();
7      qDebug() << "QSS success";
8  }
9  else
10 {
11     qDebug() << "QSS error";
12 }

13 Widget w;
14 QPushButton button1(&w);
15 QPushButton button2(&w);

16 //stylesheet.open(QIODevice::ReadOnly);

17 button1.setText("xzz");//其实可以不用这个字体
18 button1.move(100,100);
19 button1.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大

20 button1.show();
21 /**
22 *   qDebug() << "QSS error";
23 */
24 Widget w;
25 QPushButton button1(&w);
26 QPushButton button2(&w);

27 //stylesheet.open(QIODevice::ReadOnly);

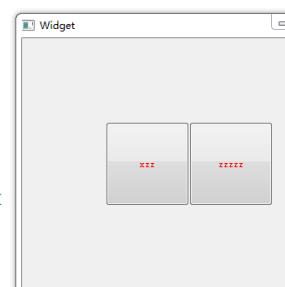
28 button1.setText("xzz");//其实可以不用这个字体
29 button1.move(100,100);
30 button1.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大
31 button1.show();

32 button2.setText("zzzzz");//其实可以不用这个字体
33 button2.move(200,100);
34 button2.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大
35 button2.show();

36 w.show();
37
38
39

```

将 QSS 文件写入
QApplication a
这个对象，因为
你的按钮都是建
立在这个对象下
面的，所以最后
QSS 文件的内容
都会影响到 a 对
象下面的内容



你看我设置的按钮 QSS 文件 color:red;样式实现了，而且是每个按钮都是这种样式。

假如样式没有生效，按钮字符没有变成红色，那就是 qss 文件的编码格式不对，改成 utf-8 就是了

如果我不想每个按钮都是这种颜色，那么我怎么在 QSS 文件里面指定按钮设置颜色呢？

我们要用 ID 选择器

```
9  
0  
1  
2 button1.setText("xzz");//其实可以不用这个字体  
button1.move(100,100);  
button1.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大  
button1.setObjectName("button1ID");//指定按钮1的ID号，QSS文件根据ID取设置某个按钮颜色  
button1.show();  
  
5 button2.setText("zzzzz");//其实可以不用这个字体  
button2.move(200,100);  
button2.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大  
button2.show();  
  
w.show();  
1
```

我指定了 button1 的 ID 为 button1ID

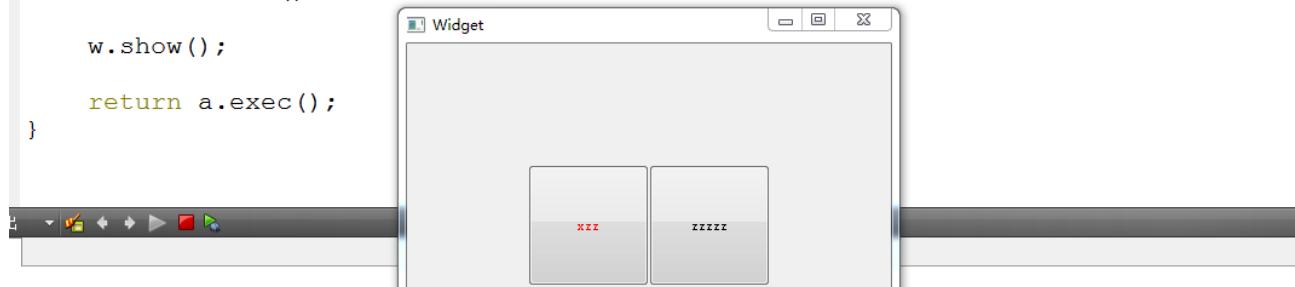
Button2 这个按钮 ID 没有指定，所以 QSS 不会设置它

```
css.qss *  
QPushbutton#button1ID  
{  
    color:red;  
}
```

在 QSS 文件里面用#表示 ID 选择，我这里选择的是 button1ID，也就是 button1 按钮

这样 C++ 文件和 QSS 文件的按钮 ID 对应上了

```
button1.setText("xzz");//其实可以不用这个字体  
button1.move(100,100);  
button1.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大  
button1.setObjectName("button1ID");//指定按钮1的ID号，QSS文件根据ID取设置某个按钮颜色  
button1.show();  
  
button2.setText("zzzzz");//其实可以不用这个字体  
button2.move(200,100);  
button2.resize(100,100);//设置按钮大小，图片不管多大，最后大小都会变成和按钮一样大  
button2.show();  
  
w.show();  
  
return a.exec();  
}
```



你看指定某个按钮颜色的 QSS 文件实现方法就成功了。

按钮多种样式代码参考

```
css.qss
QPushButton#button1ID
{
    border-image:url(:/xzz/png/01.png);
}

QPushButton:hover#button1ID
{
    border-image:url(:/xzz/png/02.png);
}

QPushButton:pressed#button1ID
{
    border-image:url(:/xzz/png/03.png);
}
```

第一个按钮，QSS 鼠标点击图片更换特效

```
QPushButton#button2ID
{
    color: #666;
    background-color: #aaa;
    border-color: #eee;
    font-weight: 300;
    font-size: 16px;
}

QPushButton#button3ID
{
    text-shadow: 0 1px 0 rgba(255, 255, 255, 0.3);
    text-decoration: none;
    background-color: #eeeeee;
    border-color: red;
    color: red;
    -webkit-transition-duration: 0s;
           transition-duration: 0s;
    -webkit-box-shadow: inset 0 1px 3px rgba(0, 0, 0, 0.2);
           box-shadow: inset 0 1px 3px rgba(0, 0, 0, 0.2);
}
```

第二个按钮样式

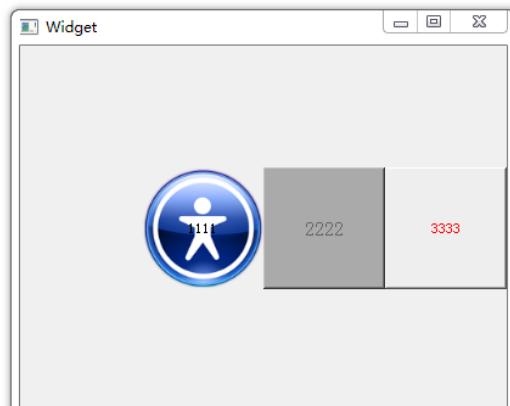
第三个按钮样式

```
button1.setText("1111");
button1.move(100,100);
button1.resize(100,100);
button1.setObjectName("button1ID");//指定按钮1的ID号, QSS文件根据ID取设置某个按钮颜色
button1.show();

button2.setText("2222");
button2.move(200,100);
button2.resize(100,100);
button2.setObjectName("button2ID");
button2.show();

button3.setText("3333");
button3.move(300,100);
button3.resize(100,100);
button3.setObjectName("button3ID");
button3.show();

w.show();
```



```

css.qss
1 QPushButton#button1ID
2 {
3     background-color: #473C8B;
4     border-style: outset;
5     border-width: 2px;
6     border-radius: 5px;
7     border-color: #8B7355;
8     font: bold 14px;
9     min-width: 2em;
10    color: white;
11    padding: 5px;
12    font-size: 50px; /* 设置字体大小 */
13    font-style: oblique; /* 设置字体倾斜 */
14    font-family: "Times New Roman", Georgia, serif; /* 设置字体样式，比如微软雅黑 */
15 }
16 QPushButton:hover#button1ID
17 {
18     background-color: rgb(0, 150, 0);
19 }
20 QPushButton:pressed#button1ID
21 {
22     background-color: #1E90FF;
23     border-style: inset;
24 }

```

```

Widget w;
QPushButton button1(&w);

button1.setText("1111");
button1.move(100, 100);
button1.resize(100, 100);
button1.setObjectName("button1ID");
button1.show();

w.show();

```



用 QSS 按钮样式模拟软件来生成按钮的样式代码

运行 ButtonStyleDemo 软件  这是 QT 实现的，QT 编译一下运行

The screenshot shows the Qt Creator interface with the 'ButtonStyleDemo' project open. On the left, the project tree shows files like 'ButtonStyleDemo.pro', '头文件', '源文件', and '界面文件'. In the center, the 'Button Style' dialog is displayed, allowing users to customize button styles through various settings like 'Edge Settings', 'Font', and 'Gradient'. A preview window shows a button with the text 'PushButton'. A green callout box points from the preview area to the text: '调试出来的按钮会生成 QSS 代码' (The debugged button will generate QSS code). Another green callout box points from the bottom right of the dialog to the code editor: '将代码复制到你的工程下 QSS 文件' (Copy the code to your project's QSS file). The code editor on the right contains the generated QSS code:

```

1 QPushbutton#button1ID
2 {
3     color: #ff007f;
4     border: 1px solid #888888;
5     font: 13pt "Aharoni";
6     border-radius: 13px;
7     padding: 5px;
8     background: qradialgradient(cx: 0.3, cy: -0.4,
9         fx: 0.3, fy: -0.4,
10        radius: 1.35, stop: 0 #fff, stop: 1 #888);
11    min-width: 80px;
12 }
13 QPushbutton:hover#button1ID
14 {
15     color: #ff007f;
16     border: 1px solid #888888;
17     font: 13pt "Aharoni";
18     border-radius: 13px;
19     padding: 5px;
20     background: qradialgradient(cx: 0.3, cy: -0.4,
21         fx: 0.3, fy: -0.4,
22        radius: 1.35, stop: 0 #fff, stop: 1 #888);
23 }
24 QPushbutton:pressed#button1ID
25 {
26     color: #ff007f;
27     border: 1px solid #888888;
28     font: 13pt "Aharoni";
29     border-radius: 13px;
30     padding: 5px;
31     background: qradialgradient(cx: 0.4, cy: -0.1,
32         fx: 0.4, fy: -0.1,
33        radius: 1.35, stop: 0 #fff, stop: 1 #add);
34 }
35 QPushbutton:hover#button1ID
36 {
37     background: qradialgradient(cx: 0.3, cy: -0.4,
38         fx: 0.3, fy: -0.4,
39        radius: 1.35, stop: 0 #fff, stop: 1 #888);
40 }
41 QPushbutton:pressed#button1ID
42 {
43     background: qradialgradient(cx: 0.3, cy: -0.4,
44         fx: 0.3, fy: -0.4,
45        radius: 1.35, stop: 0 #fff, stop: 1 #888);
46 }
47 QPushbutton:hover#button1ID
48 {
49     background: qradialgradient(cx: 0.3, cy: -0.4,
50         fx: 0.3, fy: -0.4,
51        radius: 1.35, stop: 0 #fff, stop: 1 #888);
52 }

```

最后生成你想要的按钮

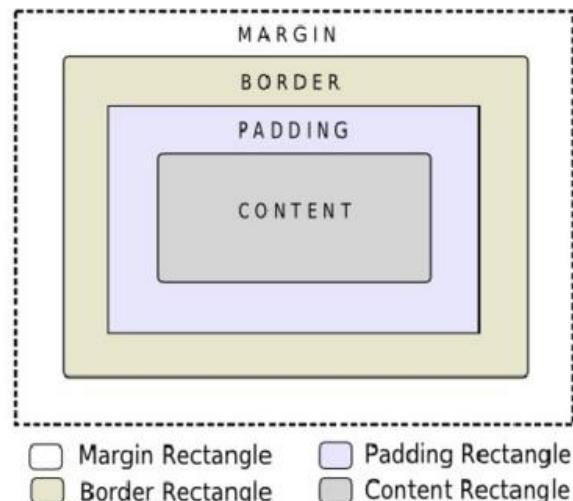
QLineEdit 输入字符对话框样式设置，(重点介绍盒子模型)

盒子模型



1. 样式表原理：方箱模型

对于任何一个控件或者控件里的子部件，可以这样理解，相对于右图，一个空间可以分为四个区域边框：



Margin : 控件最外围的空白区域，总是透明的

Border : 控件的外边框

Padding: 控件的外边空到内显示区域的空白区域

Content: 控件的最内显示区域



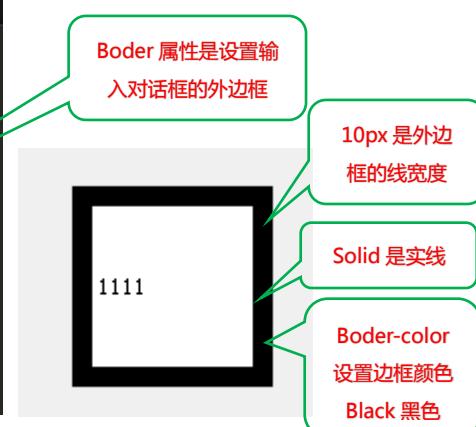
```
Widget w;
```

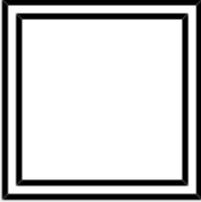
```
QLineEdit lineEdit(&w);
lineEdit.move(250,100);
lineEdit.resize(100,100);
lineEdit.setObjectName("lineEdit");

w.show();
```

C++ 创建一个输入对话框

```
css.qss
1
2 QLineEdit#lineEdit
3 {
4     border: 10px solid;
5     border-color:black;
6 }
```





```
QLineEdit#lineEdit
{
    border: 10px double;
    border-color:black;
}
```

double 双线
边框



```
QLineEdit#lineEdit
{
    border: 10px groove;
    border-color:red;
}
```

Groove 设置立体感 3D 凹槽边框，为了看清楚效果我们用红色



```
QLineEdit#lineEdit
{
    border: 10px ridge;
    border-color:red;
}
```

ridge 立体感 3D 凸槽边框，反向显示



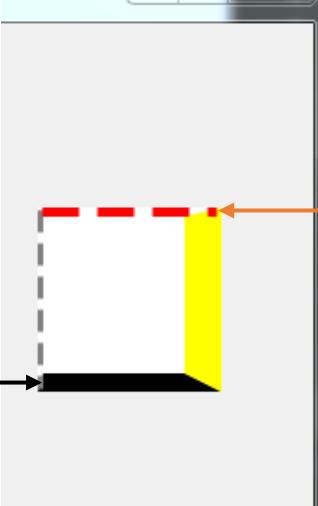
```
QLineEdit#lineEdit
{
    border: 10px inset;
    border-color:red;
}
```

inset 设置边框为 3D 凹边形状



```
1
2 QLineEdit#lineEdit
3 {
4     border: 10px outset;
5     border-color:red;
6 }
7
```

outset 设置 3D 凸边形状反向



```

QLineEdit#lineEdit
{
    border-color:green;
    border-top:5px dashed red;
    border-bottom:10px solid black;
    border-right:20px solid yellow;
    border-left:3px dashed gray;
}

```

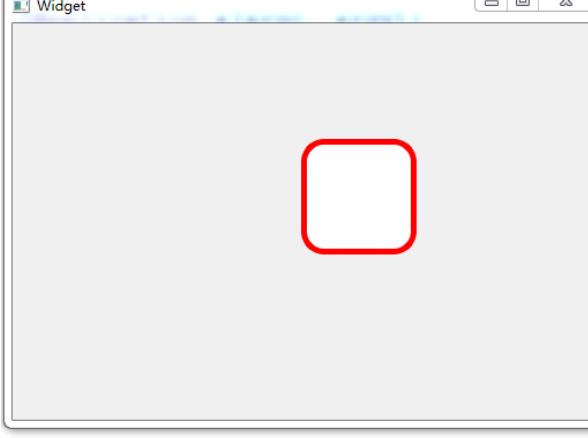
boder 是设置边框 4 边都一样的颜色和样式，如果你指定了边框 top(顶部) , bottom(底部) , 那么 border 和 border-color 将失效

border-top 设置上边框样式

border-bottom 设置下边框样式

border-right 设置右边框样式

border-left 设置左边框样式



```

Widget
E:\test\testui\css.qss - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
OPEN FILES
> CSS.QSS
1
2 QLineEdit#lineEdit
3 {
4
5     border: 5px solid red;
6     border-radius:20%;
7 }
8
9
10
11
12
13
14

```

方形输入框倒圆角
这 20%什么意思

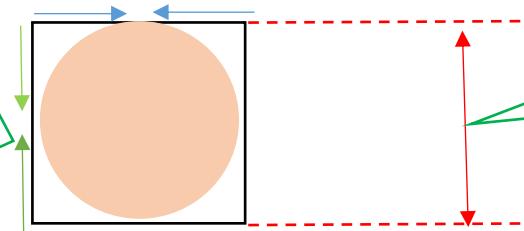
```

QLineEdit lineEdit(&w);
lineEdit.move(250,100);
lineEdit.resize(100,100); ←
lineEdit.setObjectName("lineEdit");

```

其实这 20%在我的 html_guidew 文档 CSS 部分有讲过

经过每条边向里面缩短 50% ,也就是 20px ,但是还要保持每条边相连 ,就形成了圆形了



Border-radius 是四边都倒相同的角 , 本来输入对话框是方形的 , 长 100px , 宽 100px

所以 border-radius 是让边框直线向里面缩短多长 , 然后还要保持横线和竖线链接 , 那么就只有倒圆角能保持横线竖线链接。



```

2 QLineEdit#lineEdit
3 {
4
5     border: 5px solid red;
6     border-top-left-radius:20%;
7
8

```

Border-top-left-radius: 表示左上
角边框横竖线长度向里面缩短 20%



```

2 QLineEdit#lineEdit
3 {
4
5     border: 5px solid red;
6     border-bottom-left-radius:20%;
7 }
8

```

Border-bottom-left-radius: 表示左下
角边框横竖线长度向里面缩短 20%



```

1
2 QLineEdit#lineEdit
3 {
4
5     border: 5px solid red;
6     border-top-left-radius:30%;
7     border-bottom-left-radius:30%;
8     border-top-right-radius:30%;
9     border-bottom-right-radius:30%;
10 }

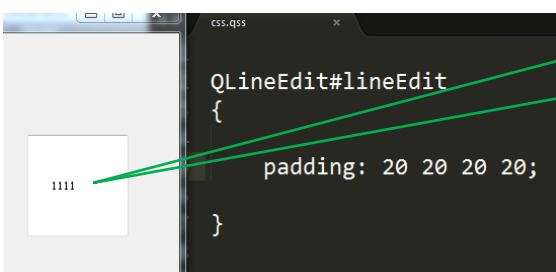
```

Border-bottom-right-radius: 右下角
Border-top-right-radius : 右上角

padding 的使用

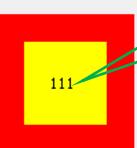


没有 padding 字符是挨着边框的



有了 padding 字符
输入的四周距离边框 20 个像素

这样看不是很清楚

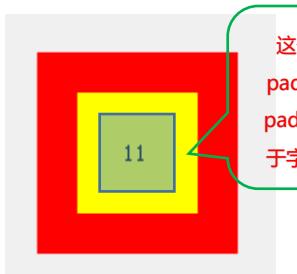


Pading 是不能用颜色填充的，但是我们能看出字符的区域和 border 区域被 padding 区域隔开了

```

QLineEdit#lineEdit
{
    background-color:yellow;
    border:20px solid red;
    padding: 20 20 20 20;
}

```



这个区域就是 padding，所以 padding 一般用于字体图像居中

QLabel 界面设计

```
QLabel label(&w);
label.setText("xxxxzzzz");
label.move(100,100);
```

设置字体大小

```
4 QLabel
5 {
6     font: 9pt;
7 }
```

xxxxzzzz

```
QLabel
{
    font: 9pt;
}
```

XXXXXXZ

```
3
4 QLabel
5 {
6     font: 100pt;
7 }
8
```

XXXXZZZZ

```
QLabel
{
    font: 20pt;
}
```

XXXXZZZZ

字体加粗

```
QLabel
{
    font: 20pt;
    font-family:"Vrinda";
}
```



```
XXXXZZZZ
```

```
QLabel
{
    font: 20pt;
    font-family:"Vrinda";
    color: rgb(0, 120, 0);
}
```

rgb 设置字体红色深度，绿色深度，和蓝色深度来配比不同颜色



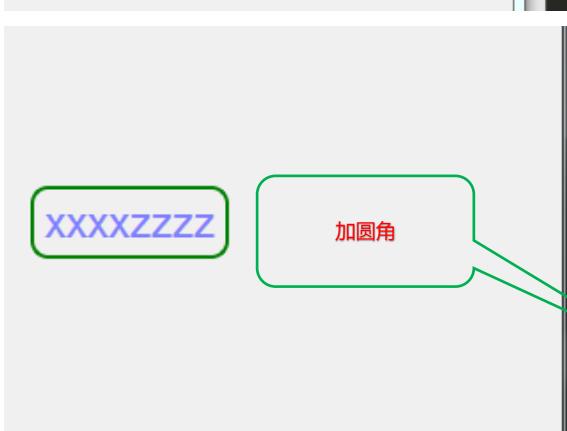
```
XXXXZZZZ
```

```
QLabel
{
    font: 20pt;
    font-family:"Vrinda";
    color: rgb(128, 128, 255);
}
```



```
XXXXZZZZ
```

```
QLabel
{
    font: 20pt;
    font-family:"Vrinda";
    color: rgb(128, 128, 255);
    border: 2px solid green;
}
```



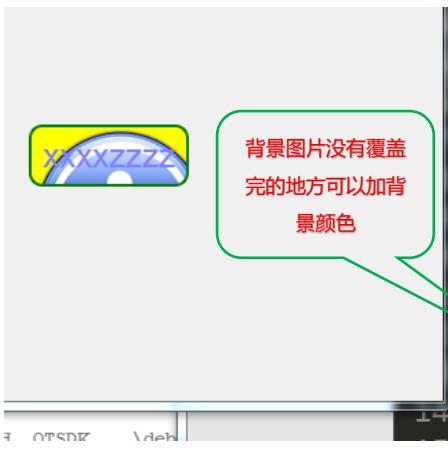
```
XXXXZZZZ
```

```
QLabel
{
    font: 20pt;
    font-family:"Vrinda";
    color: rgb(128, 128, 255);
    border: 2px solid green;
    border-radius: 10px;
}
```

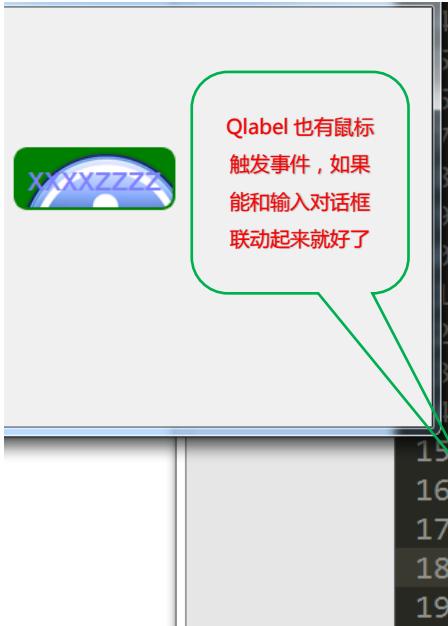


```
XXXXZZZZ
```

```
QLabel
{
    font: 20pt;
    font-family:"Vrinda";
    color: rgb(128, 128, 255);
    border: 2px solid green;
    border-radius: 10px;
    padding: 2px;
    background-image: url(:/xzz/png/01.png);
}
```



```
QLabel
{
    font: 20pt;
    font-family:"Vrinda";
    color: rgb(128, 128, 255);
    border: 2px solid green;
    border-radius: 10px;
    padding: 2px;
    background-image: url(:/xzz/png/01.png);
    background-color:yellow;
}
```



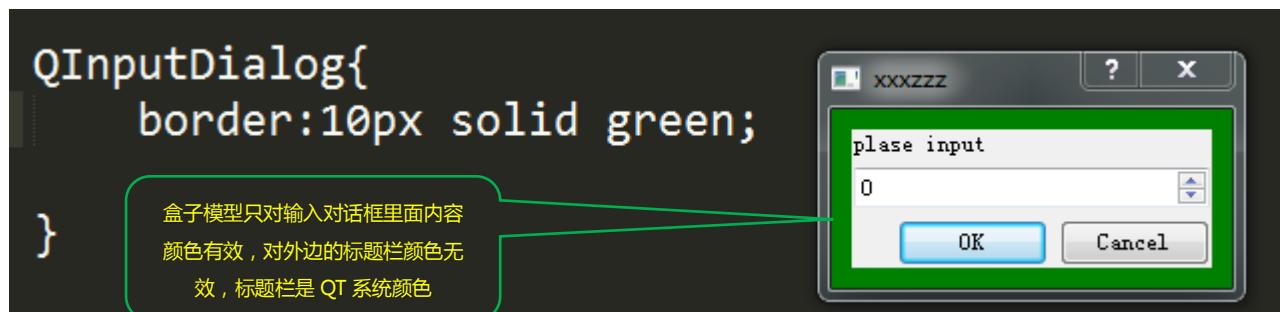
```
QLabel
{
    font: 20pt;
    font-family:"Vrinda";
    color: rgb(128, 128, 255);
    border: 2px solid green;
    border-radius: 10px;
    padding: 2px;
    background-image: url(:/xzz/png/01.png);
    background-color:yellow;
}

15
16 QLabel:hover
17 {
18     background-color:green;
19 }
```

QInputDialog 输入对话框界面设计

```
QInputDialog input(&w);
input.setWindowTitle("xxxxzz");
input.setLabelText("please input");
input.setInputMode(QInputDialog::IntInput);
input.move(100, 100);
input.exec();
```

输入对话框不属于任何父窗口，它是单独弹出来的窗口，但是好像可以嵌入其它窗口。



QPainter 在界面上画图

```
10 class Widget : public QWidget
11 {
12     Q_OBJECT
13
14 public:
15     explicit Widget(QWidget *parent = 0);
16     ~Widget();
17
18     void paintEvent(QPaintEvent *); //这是系统给你的默认函数，名字形参必须一样就是paintEvent
19
20 private:
21     Ui::Widget *ui;
22 };
23 }
```

头文件只需要创建一个系统随时都会自动调用的画板函数，这个 paintEvent 不是我自己取的函数名，是系统指定必须这么取函数名，到时候系统回去自动调用它

```
#include "widget.h"
#include "ui_widget.h"
#include <QPainter>/> //画图类
void Widget::paintEvent(QPaintEvent *)
{
    QPainter xzzpainter(this);
    xzzpainter.drawLine(QPoint(0, 0), QPoint(100, 100));
}
```

在对应的 C 文件 include 一个画图类，因为这个画图类创建的对象才有画图能力

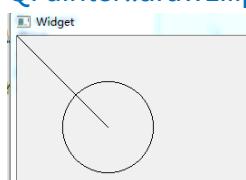
创建一个画家对象 xzzpainter，这个画家对象在哪里画图呢？注意 this 意思就是指定了画家在当前对象画图，也就是 Widget 对象的窗口中画图，你也可以把这个 this 改成其它对象，那么画家就在其它对象的窗口中画图了

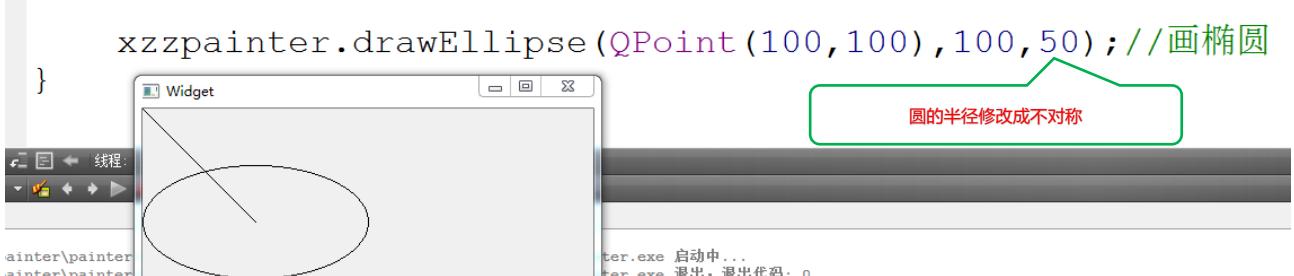
QPainter::drawLine(开始坐标，结束坐标) //这是划线函数，开始坐标和结束坐标构成一条线。



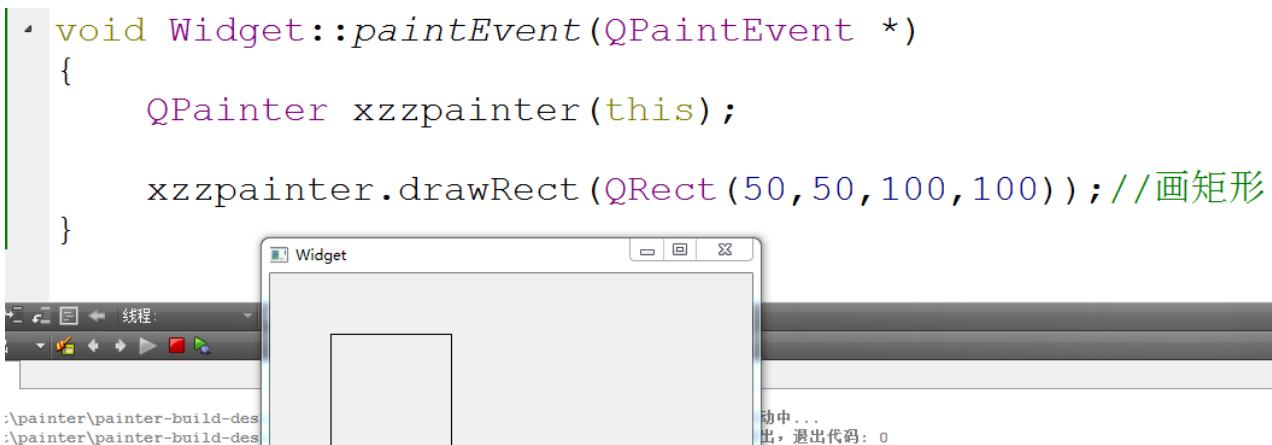
```
void Widget::paintEvent(QPaintEvent *)
{
    QPainter xzzpainter(this);
    xzzpainter.drawLine(QPoint(0, 0), QPoint(100, 100)); //画线
    xzzpainter.drawEllipse(QPoint(100, 100), 50, 50); //画圆
}
```

QPainter::drawEllipse(圆心坐标，圆上下半径，圆左右半径) //这是画图，这个函数也可以画椭圆





只需要把圆的左右半径和上下半径改得不对称就是椭圆了。



QPainter::drawRect(用 QRect 来指定矩形坐标)

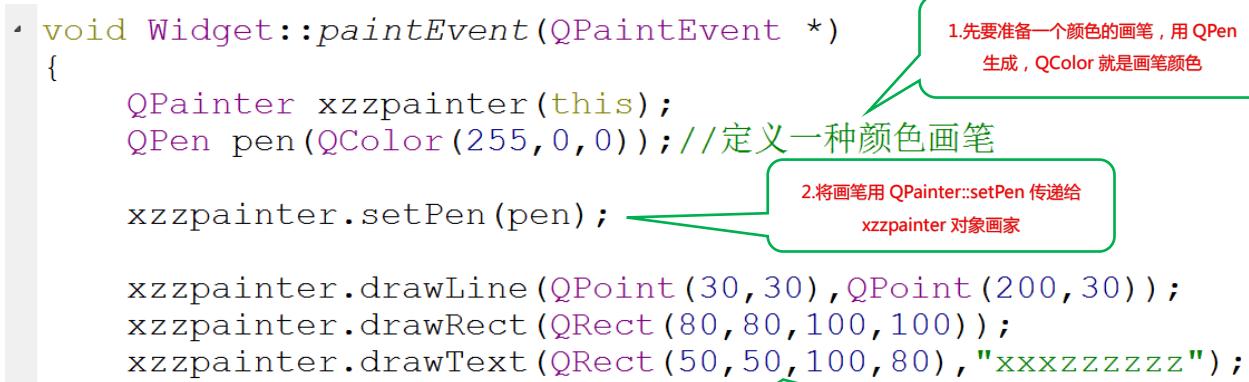
QRect(矩形开始的 xy 位置 , 矩形长度 , 矩形宽度)



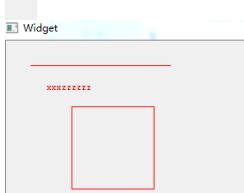
QPainter::drawText(用 QRect 指定矩形坐标位置和大小，在矩形里面写字符串)

因为没有加入中文支持，所以只能写英文字符

下面我们来给这些图形上颜色



3.一切用 xzzpainter 画家画的图都是这种颜色，
这里 QColor 定义的是红色



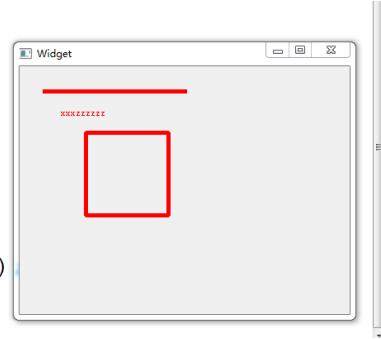
这个代码顺序一定是画笔先定义，然后才能用 draw...函数去画图

```

void Widget::paintEvent(QPaintEvent *)
{
    QPainter xzzpainter(this);
    QPen pen(QColor(255, 0, 0)); // 定义一种颜色画笔
    pen.setWidth(5); // 设置画笔粗细
    xzzpainter.setPen(pen);

    xzzpainter.drawLine(QPoint(30, 30), QPoint(200, 30));
    xzzpainter.drawRect(QRect(80, 80, 100, 100));
    xzzpainter.drawText(QRect(50, 50, 100, 80), "xxxxxxxx");
}

```



Widget

```

void Widget::paintEvent(QPaintEvent *)
{
    QPainter xzzpainter(this);
    QPen pen(QColor(255, 0, 0)); // 定义一种颜色画笔
    pen.setWidth(5); // 设置画笔粗细
    pen.setStyle(Qt::DotLine); // 设置画笔风格
    xzzpainter.setPen(pen);

    xzzpainter.drawLine(QPoint(30, 30), QPoint(200, 30));
    xzzpainter.drawRect(QRect(80, 80, 100, 100));
    xzzpainter.drawText(QRect(50, 50, 100, 80), "xxxxxxxx");
}

```

设置画笔风格，这里 Qt::DotLine 是点状，其它形状网上查

Widget

```

void Widget::paintEvent(QPaintEvent *)
{
    QPainter xzzpainter(this);
    QPen pen(QColor(255, 0, 0)); // 定义一种颜色画笔
    pen.setWidth(5); // 设置画笔粗细
    pen.setStyle(Qt::DotLine); // 设置画笔风格
    xzzpainter.setPen(pen);

    QBrush xbrush(Qt::cyan); // 定义一个画刷
    xzzpainter.setBrush(xbrush); // 把画刷拿给画家

    xzzpainter.drawLine(QPoint(30, 30), QPoint(200, 30));
    xzzpainter.drawRect(QRect(80, 80, 100, 100));
    xzzpainter.drawText(QRect(50, 50, 100, 80), "xxxxxxxx");
}

和 QPen 一样，  
QBrush 就是填充封闭形状的颜色  
Qt::cyan 就是青色

```

QBrush 填充封闭图形的颜色，像直线这种不封闭的图形无法填充

QPainter::setBrush(放入设置好填充颜色的画刷)

Widget

```

void Widget::paintEvent(QPaintEvent *)
{
    QPen pen(QColor(255, 0, 0)); // 定义一种颜色画笔
    pen.setWidth(5); // 设置画笔粗细
    pen.setStyle(Qt::DotLine); // 设置画笔风格
    xzzpainter.setPen(pen);

    QBrush xbrush(Qt::blue); // 定义一个画刷
    xbrush.setStyle(Qt::Dense1Pattern); // 设置画刷风格
    xzzpainter.setBrush(xbrush); // 把画刷拿给画家

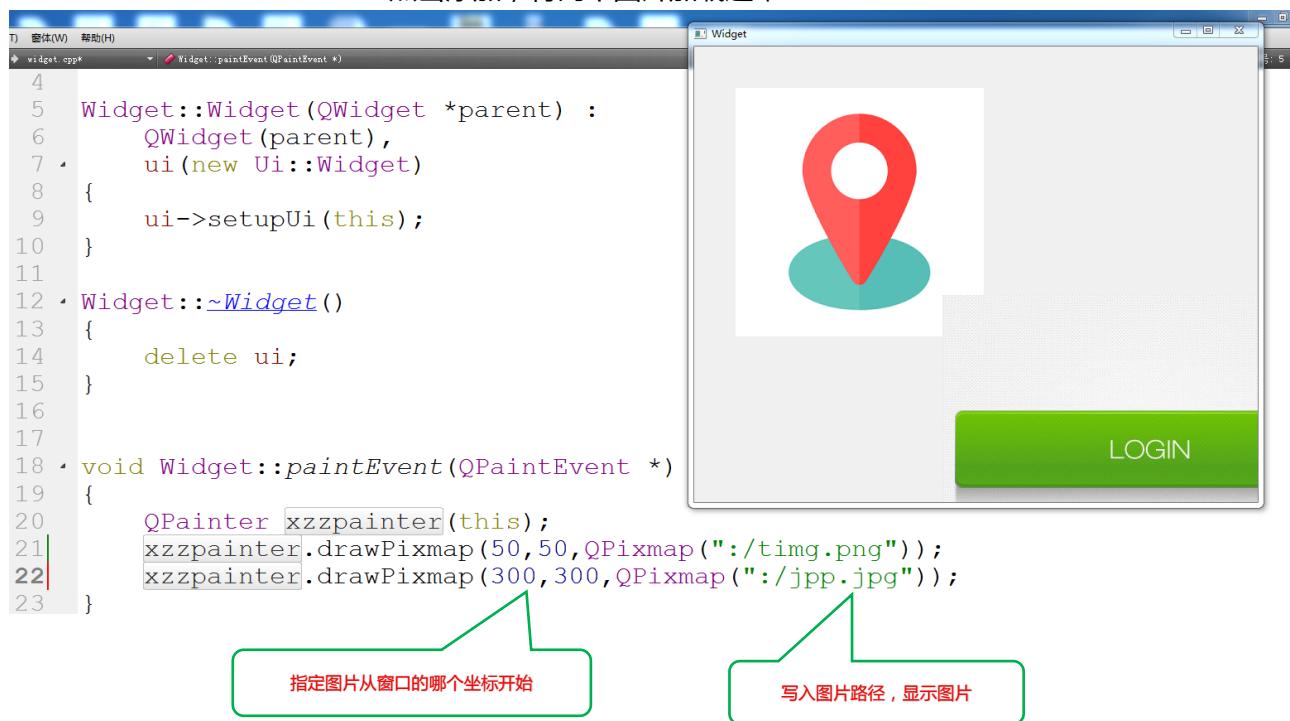
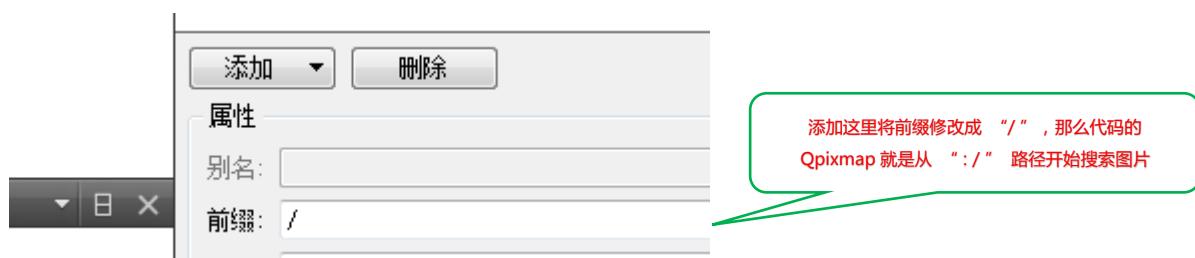
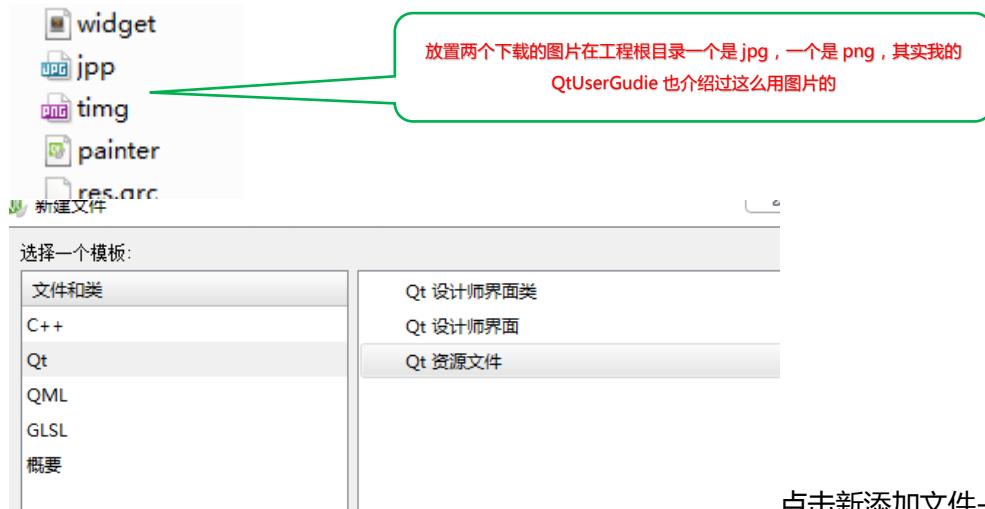
    xzzpainter.drawLine(QPoint(30, 30), QPoint(200, 30));
    xzzpainter.drawRect(QRect(80, 80, 100, 100));
}

QBrush::style 就是设置画刷的风格，我这里设置的是 Qt::Dense1Pattern

```

QBrush::style(选择画刷风格值)

QPainter 画图片



现在让图片根据按钮来移动

```
18 void paintEvent(QPaintEvent *);  
19 //这是系统给你的默认函数，名字形参必须一样就是paintEvent  
20  
21 private:  
22     Ui::Widget *ui;  
23     int posX; //定义一个类里面的全局变量  
24 private slots:  
25     void Xslot();  
26  
27 };  
28  
29 }
```

在类头文件中创建一个全局坐标变量，到时候槽函数会去修改坐标数据

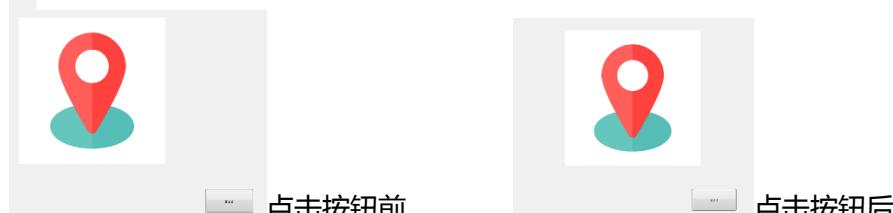
按钮点击后槽函数就会执行

```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
    QPushButton *button = new QPushButton(this);  
    button->setText("xzz");  
    button->resize(100, 50);  
    button->move(400, 400);  
  
    posX = 10; // 初始化图片的坐标  
    connect(button, SIGNAL(clicked()), this, SLOT(Xslot()));  
}  
  
void Widget::Xslot()  
{  
    posX += 10;  
    update(); // update是QT自带的触发paintEvent函数运行1次  
              // 执行一次update，类下面的paintEvent函数执行1次  
}  
  
void Widget::paintEvent(QPaintEvent *)  
{  
    QPainter xzzpainter(this);  
    xzzpainter.drawPixmap(posX, 50, QPixmap(":/timg.png"));  
}
```

在 cpp 文件中创建按钮，因为程序运行的时候就会最先执行构造函数，但是构造函数执行完了才会执行其它的函数，这里按钮必须在堆空间创建才不会因为构造函数执行完而消失

链接按钮信号到槽，在 QT5 中 connect 可以在后面用 [=] 这种符号来增加信号发生后就执行想要的程序，但是 QT4.7 就没有这个功能，只有自己建立槽函数，然后槽函数去执行更新图片坐标函数

PaintEvent 并不是一直死循环在执行，而是执行一次 update 去触发 PaintEvent 执行一次

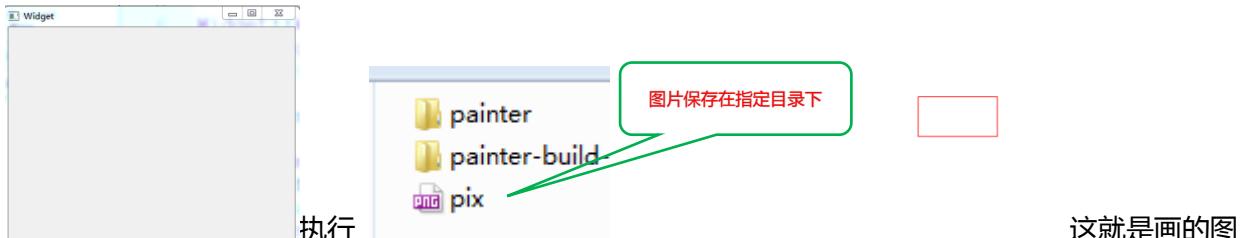


这里重点说一下 paintEvent 绘图函数，paintEvent 要执行有三种情况：1. 程序最先开始运行的时候，系统会去执行一次 paintEvent。2. 手动执行 update 的时候系统会去执行 paintEvent 像上面这样。3. 就是系统遇到特殊情况下会去执行 paintEvent，这个特殊情况就看是什么函数。

用绘图设备画出来的图形保存在电脑上

```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    QPixmap pix(300, 300); // 定义一张300X300大小的纸在窗口上  
    pix.fill(Qt::white); // 这种纸背景色为白色  
    // QPainter xzzpainter(this); // 给纸申请一个画家, 以前画家是给窗口画图用的this  
    QPaint xzzpainter(&pix); // 现在画家是给pix这张纸画图, 所以对象是pix  
    xzzpainter.setPen(QPen(Qt::red)); // 给画家设置红色的画笔  
    xzzpainter.drawRect(QRect(100, 100, 100, 50)); // 在pix纸上画矩形  
  
    pix.save("E://QTtest/painter/pix.png"); // 将画的图形保存在电脑路径下  
  
}  
  
void Widget::paintEvent(QPaintEvent *)  
{
```

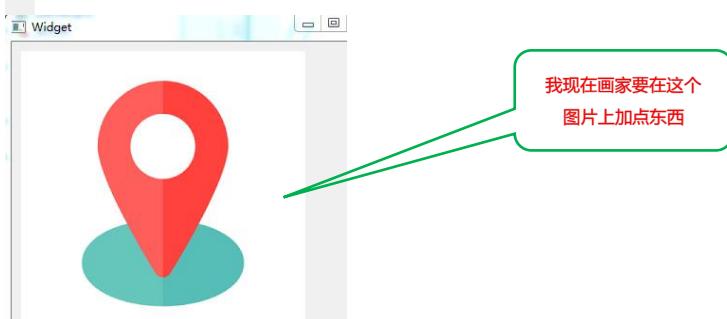
为了简单，没有在 paintEvent 函数去执行这些画图函数，所以窗口看不出来，但是确实在对应的磁盘下保存图片了。



如果你想画图在窗口上也能看见，你可以修改以上代码，修改到头文件用指针或者全局变量什么的。

修改图片，对图片进行标注什么的

```
void Widget::paintEvent(QPaintEvent *)  
{  
    QImage img;  
    img.load(":/timg.png"); // 加载资源文件里面的图片  
    QPaint xzzpainter(this); // 申请画家  
    xzzpainter.drawImage(10, 10, img); // 画家拿去img对象加载的图片在指定坐标显示  
}
```

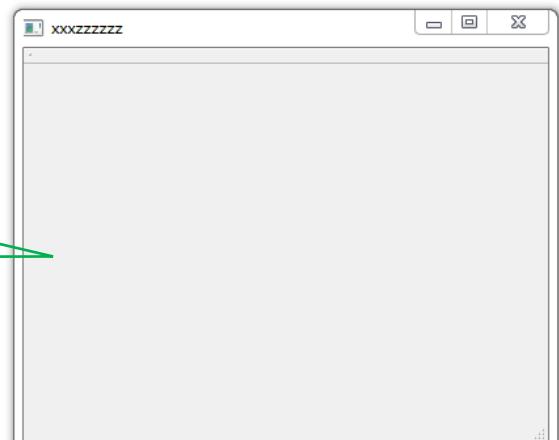


QImage::setPixel 可以在加载的图片上画想要的图形或者像素点
未完待续.....

给每个主窗口的背景图片

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    //w.setFixedSize(500,500); //这是固定窗口大小我不喜欢用
    w.setWindowTitle("xxxxxxxx");
    w.show();
    return a.exec();
}
```

给 MainWindow 主窗口加背景图片
有很多种方法，比如在窗口前面加个
QLabel，然后在 QLabel 里面加图片，
或者用 QSS 设置主窗口背景。但是我
这里用上面几节学的 QPainter 画图工
具来设置窗口背景图片



```
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
    void paintEvent(QPaintEvent *); //用来做窗口背景图片
private:
    Ui::MainWindow *ui;
};
```

头文件加 paintEvent 画
图事件的回调函数

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainter xzzpainter(this);
    QPixmap pix;
    pix.load(":/res/hua.jpg");
    xzzpainter.drawPixmap(0,0,pix);
}
```

用 QPixmap 获取图片

用 QPainter::drawPixmap 在窗口上画图



这种窗口无法适应图片的大小就很难看，有两种方法可以解决。

第1中: 用固定窗口大小的方法

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setFixedSize(1024,819); //这是固定窗口大小我不喜欢用
    w.setWindowTitle("xxxxxxxx");
    w.show();
}
```

这就是把主窗口大小固定成和图片大小一样大



这种 setFixedSize 的方法可以让窗口和图片一样大，但是有时候我想拖动窗口缩小和放大就不得行。窗口尺寸定死了不灵活

第2种: 图片自动适应窗口大小

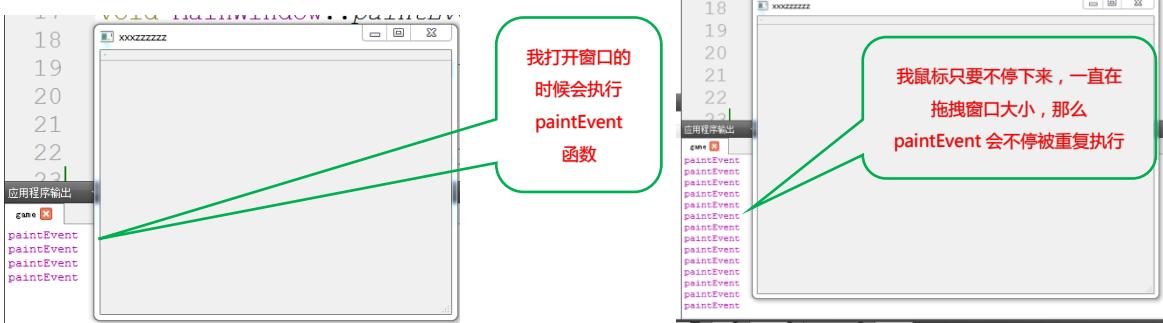
首先我们要知道在鼠标拖拽窗口大小的时候，paintEvent 会被触发执行

```
void MainWindow::paintEvent(QPaintEvent *)
{
    QPainter xzzpainter(this);

    QPixmap pix;
    pix.load(":res/hua.jpg");

    qDebug() << "paintEvent";
}
```

我打开窗口的时候会执行 paintEvent 函数



所以我前面几章说了 paintEvent 有三种情况会被执行:

1. 其它函数调用 update() , paintEvent 会被执行 ,
2. 打开软件时 paintEvent 会被执行一次 ,
3. 这种拖拽窗口的方式就是一种特殊情况 , paintEvent 也会执行。

```
void MainWindow::paintEvent(QPaintEvent *)  
{  
    QPainter xzzpainter(this);  
  
    QPixmap pix;  
    pix.load(":/res/hua.jpg");  
  
    xzzpainter.drawPixmap(0, 0, this->width(), this->height(), pix);  
}
```



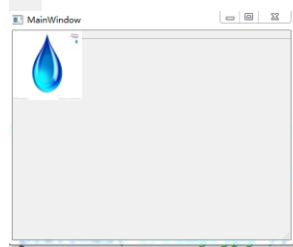
这样背景图片就会按着窗口大小来变化尺寸

因为拖拽窗口 paintEvent 会不停的执行，那么我们就不停的去修改图片的大小就是了，用 this 获取窗口长宽，然后填入 drawPixmap 设置图片长宽的参数项就行了

设置主窗口背景图片透明

```
void MainWindow::paintEvent(QPaintEvent *)  
{  
    QPainter xzzpainter(this);  
    QPixmap pix;  
  
    pix.load(":/timg.jpg");  
    xzzpainter.setOpacity(0.5); //透明度设置  
    xzzpainter.drawPixmap(0, 0, this->width(), this->height(), pix);  
}
```

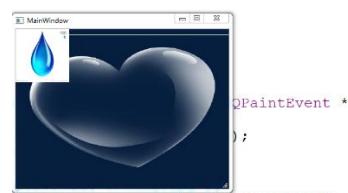
只需要在主窗口加载图片的 paintEvent 函数里面，在加载图片前设置图片透明度就是了，0 是完全透明，1 是不透明，调整参数在 0~1 之间



设置 0 图片完全透明



设置 0.5 图片半透明



设置 1 图片不透明

点击按钮播放音效，或者打开软件启动背景音乐

```
#include <QtGui/QApplication>
#include "mainwindow.h"
#include <QSound>/>音频支持类

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

    QSound *xzzsound = new QSound("E:/QTtest/play/play/xzz115.wav", &w);
    xzzsound->play();

    w.show();
    return a.exec();
}
```

QSound::stop() 停止音乐播放

QSound::setLoops(5) 参数 5 就表示音乐循环播放 5 次

QSound::setLoops(-1) -1 表示音乐无限重复播放

在实际应用中你可以在按钮点击槽函数里面执行 QSound 对象播放音乐,也可以在打开软件主窗口运行的时候执行 QSound 对象播放音乐

注意:QSound 只支持 wav 文件播放,不支持 mp3 格式播放

设计按钮点击时候产生抖动效果

```
9 private:
0     Ui::Widget *ui;
1     QPushButton *button;
2
3 private slots:
4     void buttonslots();
```

在头文件定义一个类全局的按钮和槽函数

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

    button = new QPushButton(this);

    button->setText("xxxxxxxx");
    button->move(100, 100);
    button->resize(100, 50);
    button->show();

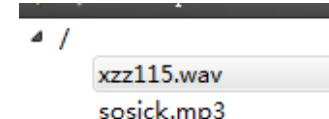
    connect(button, SIGNAL(clicked()), this, SLOT(buttonslots()));
}
```

包含 QSound 头文件,这个头文件需要 multimedia 库,所以要在 pro 编译链接文件

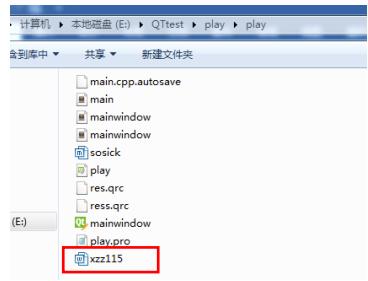
```
中      添      加      该      库
QT      += core gui multimedia
TARGET = play
TEMPLATE = app
```

这里将 QSound 对象嵌入到主窗口,主窗口运行才能播放

创建 QSound 对象,这个音乐路径不是资源文件加载的路径



这种路径是不行的,一定要到 wav 文件所在的 windows 目录路径下找



添加 windows 的音频文件路径才对

在 CPP 文件创建堆对象按钮,用 this
将按钮加入进 widget 窗口对象

当我们点击按钮的时候在槽
函数实现按钮动画

```

void Widget::buttonsslots()
{
    QPropertyAnimation *animation = new QPropertyAnimation(button, "geometry");
    animation->setDuration(200); //动画时间间隔200ms
    animation->setStartValue(QRect(button->x(), button->y(), button->width(), button->height()));
    //setStartValue是设置按钮动画开始坐标位置,我这里设置的是按钮当前坐标位置
    animation->setEndValue(QRect(button->x(), button->y() + 10, button->width(), button->height()));
    //在当前按钮的位置向y方向移动10个像素
    animation->setEasingCurve(QEasingCurve::OutBounce); //设置按钮向下滑10像素过程中的效果
    animation->start(); //开始动画
}

```

QPropertyAnimation 就是动画类

按钮动画播放的每一帧时间

填入按钮对象,让动画类操作它, geometry 是选择按钮动画的效果类型, 这里是矩形效果

这是按钮向下滑的效果,有很多选择,我选择的是 QEasingCurve::OutBounce

Widget

Widget

Widget

QEasingCurve::OutInBack 36

Easing curve for a back (overshooting cubic easing: $(s+1)^3 - 3s^2 - s$)

QEasingCurve::InBounce 37

Easing curve for a bounce (exponentially decaying parabolic bc)

QEasingCurve::OutBounce 38

Easing curve for a bounce (exponentially decaying parabolic bc)

QEasingCurve::InOutBounce 39

Easing curve for an in-out bounce (exponentially decaying parabolic bc)

这些都是按钮向下动的感觉选择,我选择的是 OutBounce,你可以查表选择其它参数试试效果

按钮回弹和按钮向下移动的类和函数是一样的,我这里新建立一个回弹动画类

我感觉按钮不能一直向下啊,应该按一下按钮向下动一下后就回弹,下面实现回弹功能

```

QPropertyAnimation *animation = new QPropertyAnimation(button, "geometry");
QPropertyAnimation *animation22 = new QPropertyAnimation(button, "geometry"); //建立回弹类

animation->setDuration(200); //动画时间间隔200ms
animation->setStartValue(QRect(button->x(), button->y(), button->width(), button->height()));
//setStartValue是设置按钮动画开始坐标位置,我这里设置的是按钮当前坐标位置
animation->setEndValue(QRect(button->x(), button->y() + 10, button->width(), button->height()));
//在当前按钮的位置向y方向移动10个像素
animation->setEasingCurve(QEasingCurve::OutBounce); //设置按钮向下滑10像素过程中的效果

animation->start(); //开始动画

```

让按钮向上移动一次就是了,启动第二次按钮动画

```

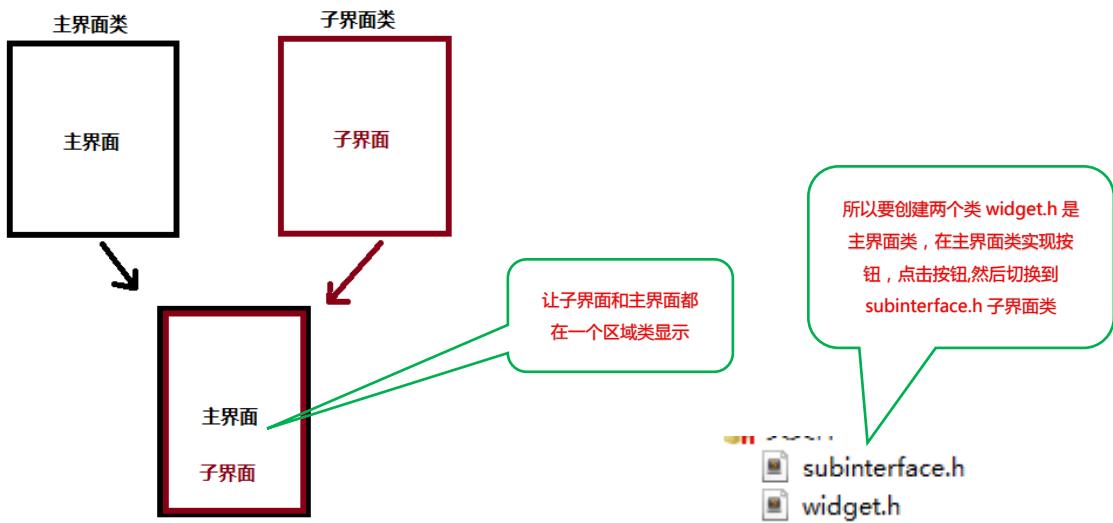
animation22->setDuration(200); //回弹动画时间间隔200ms
animation22->setStartValue(QRect(button->x(), button->y() + 10, button->width(), button->height()));
//setStartValue是回弹开始坐标为按钮已经向下移动过一次的坐标
animation22->setEndValue(QRect(button->x(), button->y(), button->width(), button->height()));
//从移动过一次的坐标开始向按钮最先初始化时的坐标移动,也就是向上移动10像素
animation22->setEasingCurve(QEasingCurve::OutBounce); //设置按钮向下滑10像素过程中的效果
animation22->start(); //开始动画

```

效果截图看不出来,自己运行程序看

如何设计主界面切换窗口

这种情况我要建立两个类，一个是主界面类，一个是子界面类



```
1 class Widget : public QWidget
2 {
3     Q_OBJECT
4
5     public:
6         explicit Widget(QWidget *parent = 0);
7         ~Widget();
8
9     private:
10        Ui::Widget *ui;
11        QPushButton *button;
12        subinterface *xzzsubinterface;
13
14     private slots:
15         void buttonslots();
16 };
17
18
19 Widget::Widget(QWidget *parent) :
20     QWidget(parent),
21     ui(new Ui::Widget)
22 {
23     ui->setupUi(this);
24
25     button = new QPushButton(this);
26
27     button->setText("xxxxzzzz");
28     button->move(100, 100);
29     button->resize(100, 50);
30     button->show();
31
32     xzzsubinterface = new subinterface;
33
34     connect(button, SIGNAL(clicked()), this, SLOT(buttonslots()));
35 }
```

主界面出了创建按钮，还有在槽
函数调用子界面指针

主界面创建按钮，然后还要创
建子界面，到时候在槽函数好
调用子界面

```

void Widget::buttonslots()
{
    QPropertyAnimation *animation = new QPropertyAnimation
    QPropertyAnimation *animation22 = new QPropertyAnimati

    animation->setDuration(200); //动画时间间隔200ms
    animation->setStartValue(QRect(button->x(), button->y(),
    //setStartValue是设置按钮动画开始坐标位置,我这里设置的是按钮当
    animation->setEndValue(QRect(button->x(), button->y())+10
    //在当前按钮的位置向y方向移动10个像素
    animation->setEasingCurve(QEasingCurve::OutBounce); //设

    animation->start(); //开始动画
}

animation22->setDuration(200); //回弹动画时间间隔200ms
animation22->setStartValue(QRect(button->x(), button->y(
//setStartValue是回弹开始坐标为按钮已经向下移动过一次的坐标
animation22->setEndValue(QRect(button->x(), button->y()),
//从移动过一次的坐标开始向按钮最先初始化时的坐标移动,也就是向上
animation22->setEasingCurve(QEasingCurve::OutBounce); //设
animation22->start(); //开始动画

this->hide(); //将主界面隐藏掉,这样子界面才能看到起
xzzsubinterface->show(); //显示子界面
}

```



现在有两个问题:

1. 主界面按钮点击没有抖动, 我们上一章节都抖动了的, 同样的代码。
2. 子界面尺寸大于主界面, 而且看不出界面切换效果。

答 1: 按钮没有抖动是因为没有加定时器, 在页面切换这种项目按钮必须加定时器, 定时多少 ms 之后才允许弹出子页面。

答 2: 子页面比主页面大的问题, 主要是主页面和子页面没有用 setFixedSize 函数, 让两个页面窗口设置成一样大。

但是我不喜欢 setFixedSize 这种定死窗口大小的方法。

解决第 1 个问题，切换界面点击按钮无抖动

```
#include <QTimer> //添加定时器

QTimer::singleShot(500, this, SLOT(timetrigc())); ←
this->hide(); //将主界面隐藏掉，这样子界面才能看到起
xzzsubinterface->show(); //显示子界面

 QTimer::singleShot(延时时间，哪一个窗口延时，时间到执行哪一个槽函数) →
```

我们在按钮按下后就执行动画了，因为没有延时所以直接跳过按钮动画进入子界面，这里我们延时，让按钮有 500ms 时间执行动画，500ms 之后执行槽函数里面的界面切换函数

```
void Widget::timetrigc()
{
    this->hide(); //将主界面隐藏掉，这样子界面才能看到起
    xzzsubinterface->show(); //显示子界面
}

所以我把界面切换函数放进了一个新的槽函数
timetrigc 槽函数
```

```
private slots:
    void buttonslots();
    void timetrigc();
```

在头文件增加槽函数给 QTimer 触发使用

下面是修改后的代码层次

```
animation22->setDuration(200); //回弹动画时间间隔 200ms
animation22->setStartValue(QRect(button->x(), button-
//setStartValue 是回弹开始坐标为按钮已经向下移动过一次的坐标
animation22->setEndValue(QRect(button->x(), button->
//从移动过一次的坐标开始向按钮最先初始化时的坐标移动，也就是向
animation22->setEasingCurve(QEasingCurve::OutBounce
animation22->start(); //开始动画
```

```
QTimer::singleShot(500, this, SLOT(timetrigc()));
```

```
}
```

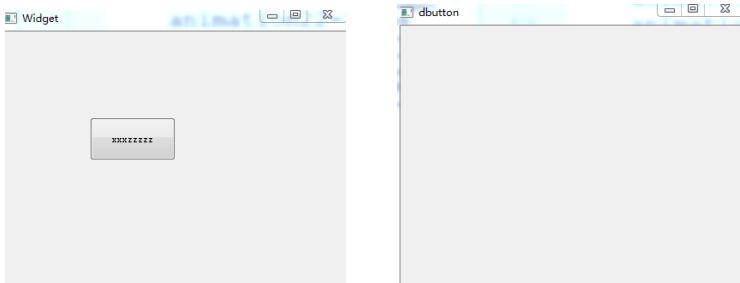
```
void Widget::timetrigc()
{
    this->hide(); //将主界面隐藏掉，这样子界面才能看到起
    xzzsubinterface->show(); //显示子界面
}
```

界面切换函数修改了位置
将 this->hide 隐藏主界面函数和
xzzsubinterface->show 显示子界面函数
放进了槽函数

这下我点击按钮，按钮动画就显示了，显示按钮动画之后跳转到新界面。这里就不掩饰了。

第2个问题，解决切换界面后，子界面和主界面一样大，我这里不用 setFixedSize

```
void Widget::timetrige()
{
    this->hide(); //将主界面隐藏掉，这样子界面才能看到起来
    xzzsubinterface->resize(this->width(), this->height());
    xzzsubinterface->show(); //显示子界面
}
```



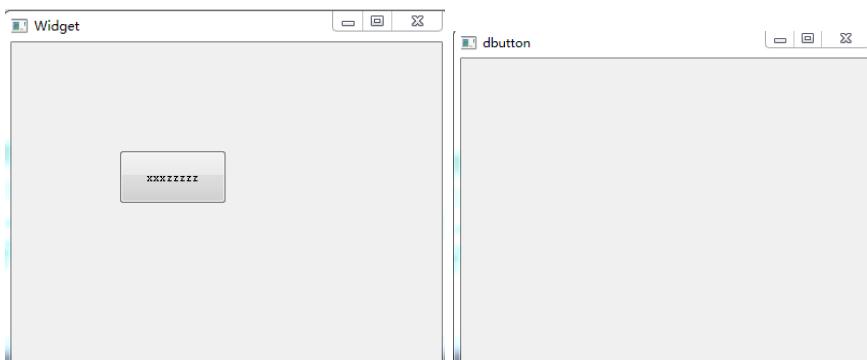
子界面和主界面一样大

但是还是有个问题，切换到子界面后，子界面坐标和主界面坐标不在同一个位置，比如子界面再屏幕右边显示，主界面在屏幕左边显示，感觉窗口切换有错位的感觉。

```
void Widget::timetrige()
{
    this->hide(); //将主界面隐藏掉，这样子界面才能看到起来
    xzzsubinterface->setGeometry(this->geometry());
    // xzzsubinterface->resize(this->width(), this->height());
    xzzsubinterface->show(); //显示子界面
}
```

geometry 函数就是获取当前窗口的开始坐标和大小尺寸

setGeometry 就是设置当前窗口的开始坐标和大小尺寸，比 resize 更高级一点，有开始坐标设置功能

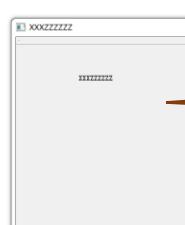


这样看起来主窗口和子窗口切换位置是一样的。

控件显示字符无法实现中文编码问题

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setWindowTitle("XXXXXXXXXX");
    QLabel label("XXXXXXXXXX", &w);
    label.setGeometry(100, 50, 160, 30);
    w.show();

    return a.exec();
}
```



取英文字符给控件没有问题

```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setWindowTitle("真全栈少年");

    QLabel label("真全栈少年", &w);
    label.setGeometry(100, 50, 160, 30);
    w.show();

    return a.exec();
}

```



取中文字符给控件就出现乱码

```

#include <QLabel>
#include <QString>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setWindowTitle(QString::fromLocal8Bit("真全栈少年"));

    QLabel label(QString::fromLocal8Bit("真全栈少年"), &w);
    label.setGeometry(100, 50, 160, 30);
    w.show();

    return a.exec();
}

```



我们可以用 QString 里面的 fromLocal8Bit 将中文转换一下写入控件

中文字符可以正常显示了

按钮按下时和松开时按钮图形变化,不用 QSS ,而是用 QT 原生鼠标事件来做
里面还涉及了 QMouseEvent 事件处理,就是鼠标移动,按下都会执行的回调函数

```

private:
    Ui::MainWindow *ui;

    QPushButton button; 在头文件创建按钮

    void mousePressEvent(QMouseEvent *event);
    //这是系统自带的函数和QPaintEvent一样,点击鼠标就会执行
    void mouseReleaseEvent(QMouseEvent *event);
    //这是系统自带的函数和QPaintEvent一样,松开鼠标就会执行

private slots:
    void btnEvent(); 按钮按下执行槽函数

```

鼠标事件是系统自带的,按下鼠标系统会自动调用该接口函数,我们只需要实现该函数就是了

在 CPP 文件添加按钮实现

```

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

    button.setParent(this); //因为不是用指针,所以必须用setParent将按钮嵌入进父窗口
    // button.setFixedSize(pixel.width(), pixel.height()); //如果用这种方法按钮就很大因为图片大
    button.setFixedSize(100, 50); //设置按钮固定大小
    button.setIcon(QIcon(":/timg1")); //加载按钮默认图片
    // button.setIconSize(QSize(pixel.width(), pixel.height())); //如果用这种方法按钮就很大因为图片大
    button.setIconSize(QSize(100, 50)); //设置图片大小

    connect(&button, SIGNAL(clicked()), this, SLOT(btnEvent()));
}

```

```

void MainWindow::mousePressEvent(QMouseEvent *event)
{
    button.setFixedSize(100, 50); //设置按钮固定大小
    button.setIcon(QPixmap(":/timg2")); //加载按钮默认图片
    button.setIconSize(QSize(100, 50));
    qDebug() << "Press";
}

```

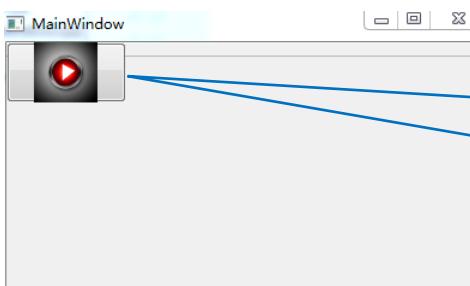
鼠标按下，执行该函数，这时候我们在该函数里面加载按钮按下图片

```

void MainWindow::mouseReleaseEvent(QMouseEvent *event)
{
    button.setFixedSize(100, 50); //设置按钮固定大小
    button.setIcon(QPixmap(":/timg1")); //加载按钮默认图片
    button.setIconSize(QSize(100, 50));
    qDebug() << "release";
}

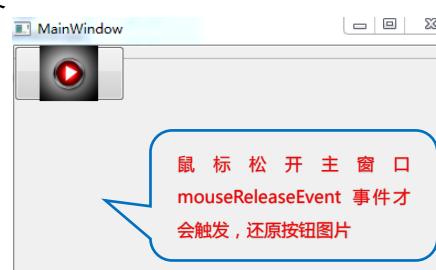
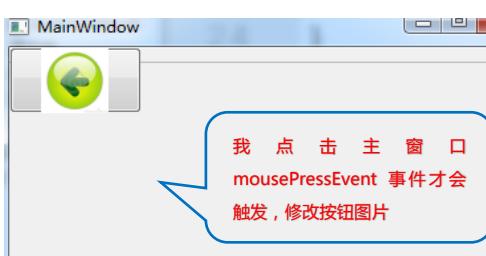
```

鼠标松开后，执行该函数，我们将按钮没按下的图片加载回来就是



运行的时候点击按钮发现图片没有变化，这是因为你的
mousePressEvent 和 mouseReleaseEvent 函数都是以
MainWindow 主窗口作为事件对象的

所以你要点击主窗口 Mouse 鼠标的事件才会触发



```

void MainWindow::mousePressEvent(QMouseEvent *event)
void MainWindow::mouseReleaseEvent(QMouseEvent *event)

```

所以问题就是这两个函数都继承自 MainWindow 主窗口造成的

下面我们解决以上问题，修改成点击按钮和松开按钮切换按钮图片，而不是点击主窗口来切换按钮图片

头文件建立按钮按下和松开的槽函数

```
private slots:  
    void btnEvent(); //按钮按下  
    void btnEventexit(); //按钮松开
```

CPP 文件实现信号与槽

```
connect(&button, SIGNAL(pressed()), this, SLOT(btnEvent())); //Pressed是鼠标按下按钮信号  
connect(&button, SIGNAL(released()), this, SLOT(btnEventexit())); //released是鼠标松开信号
```

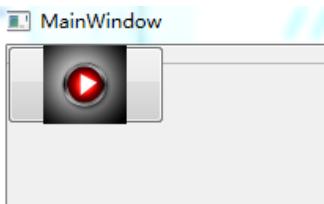
connect 因为要使用按钮按下和松开事件，所以必须发 pressed 信号和 released 信号，而不是 clicked 这种点击按钮就直接执行槽函数的信号。

```
void MainWindow::btnEvent()  
{  
    button.setFixedSize(100, 50); //设置按钮固定大小  
    button.setIcon(QPixmap(":/timg2")); //加载按钮默认图片  
    button.setIconSize(QSize(100, 50));  
    qDebug() << "Press";  
  
}  
  
void MainWindow::btnEventexit()  
{  
    button.setFixedSize(100, 50); //设置按钮固定大小  
    button.setIcon(QPixmap(":/timg1")); //加载按钮默认图片  
    button.setIconSize(QSize(100, 50));  
    qDebug() << "release";  
}
```

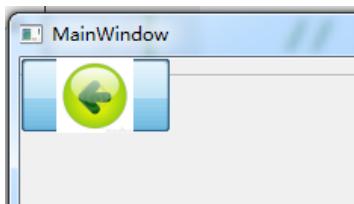
按钮按下执行的槽函数

按钮松开执行的槽函数

所以按钮事件无法用 Mouse 鼠标事件来直接继承，而是要用按钮里面自带的信号来实现。



点击按钮的图片

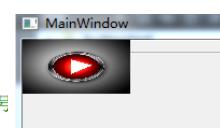


松开按钮的图片

下面我们来解决按钮和图片尺寸不兼容的问题

```
button.setParent(this); //因为不是用指针，所以必须用setParent将按钮嵌入进父窗口  
// button.setFixedSize(pixel.width(), pixel.height()); //如果用这种方法按钮就很大因为图片大  
button.setFixedSize(100, 50); //设置按钮固定大小  
button.setIcon(QPixmap(":/timg1").scaled(button.width(), button.height())); //在给按钮加载图片的时候就让QPixmap设置图片的同时用scaled修改图片大小和按钮大小一样  
  
button.setIconSize(QSize(100, 50)); //设置图片大小  
  
connect(&button, SIGNAL(pressed()), this, SLOT(btnEvent())); //Pressed是鼠标按下按钮信号  
connect(&button, SIGNAL(released()), this, SLOT(btnEventexit())); //released是鼠标松开信号
```

主要修改按钮加载图片这里



设置一个控件的背景透明色图片

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4     MainWindow w;
5
6     QLabel label(&w);
7
8     label.setGeometry(0, 0, 100, 100); //设置标签的开始位置和长宽
9
10    QPixmap pix;
11    pix.load(":/timg2.jpg");
12    label.setPixmap(pix); //加载图片给标签
13    label.setScaledContents(true);
14    //因为图片比标签大,所以用setScaledContents来让图片大小变成标签大小
15
16    w.show();
17
18    return a.exec();
19 }
```



我们平常写的 QLabel 加载图片无法实现图片透明功能，图片透明功能和 QMainWindow 一样需要在绘图事件(也就是 paintEvent)函数里面去设置 QLabel 图片透明

```
#include <QWidget>
#include <QLabel>
class myQlabel : public QLabel
{
    Q_OBJECT
public:
    explicit myQlabel(QWidget *parent = 0);

signals:

private:
    void paintEvent(QPaintEvent *e);
```

因为图片透明度要在 paintEvent 里面实现,所以我必须建立个类 ,这个类继承 QLabel ,这样 paintEvent 里面的功能是针对 QLabel 对象的

```
1 #include "myqlabel.h"
2 #include <QPainter>
3
4 myQlabel::myQlabel(QWidget *parent) :
5     QLabel(parent)
6 {
7 }
8
9 void myQlabel::paintEvent(QPaintEvent *e)
10 {
11     QPainter p(this);
12     QPixmap pix;
13     pix.load(":/timg2.jpg");
14     setGeometry(0, 0, 100, 100); //设置 QLabel 图片的大小
15     p.setOpacity(0.3); //设置 QLabel 图片的透明度
16     p.drawPixmap(0, 0, this->width(), this->height(), pix);
17 }
```

PaintEvent 里面就负责加载图片路径 ,然后用 setOpacity 设置图片透明度。最后用 drawPixmap 重新绘制 QLabel

```

#include "myqlabel.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;

    myQlabel mylabel(&w);

    w.show();

    return a.exec();
}

```

在主函数把这个自定义的 QLabel 包含进来，建立对像使用，以后这个 myqlabel 类就可以用来创建带透明色的图片或者 QLabel 背景



我们下面来测试下自定义的带图片透明的 QLabel 和系统原生的 QLabel 是否会冲突。

```

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

    myQlabel mylabel(&w);

    QLabel label(&w);
    label.setGeometry(150, 0, 100, 100); // 设置标签的开始位置和长宽

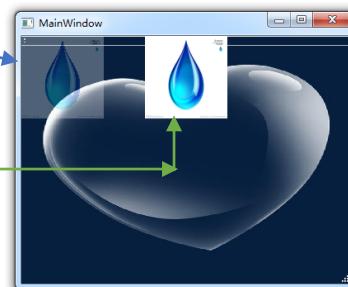
    QPixmap pix;
    pix.load(":/timg2.jpg");
    label.setPixmap(pix); // 加载图片给标签
    label.setScaledContents(true);
    // 因为图片比标签大，所以用setScaledContents来让图片大小变成标签大小

    w.show();
}

```

自己做的透明 QLabel

用系统原生做的 QLabel



测试没问题，系统原生的 QLabel 和我自定义的 QLabel 没有冲突，随意使用。

设置 QLabel 前景色和后景色来做的图片看起有叠层效果

```

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

    QLabel mylabel(&w);
    mylabel.resize(100, 100); // 设置 QLabel 大小，这样样式表加载图片也会按照这种大小来显示
    mylabel.setStyleSheet("border-image:url(:/timg2.jpg);color:rgb(0,0,0);");
    mylabel.setAlignment(Qt::AlignCenter); // 文字居中显示
    mylabel.setText("xxxxxxxx"); // 在图片前面加入文字

    w.show();

    return a.exec();
}

```

文字放在前景色

用上面的 paintEvent 的方式无法实现前景色和背景色，所以现在用 QSS 方式来做

你看前景色和后景色效果出来了，文字在前面，图片在后面

其实这根本不是前景色和后景色，文字和前景色图片在同一个图层的，只是文字覆盖图片而已。

```

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

    myQlabel mylabel(&w); // 背景透明图片的 QLabel 给主窗口
    mylabel.resize(500, 500); // 设置 QLabel 大小

    QLabel syslabel(&mylabel);
    syslabel.resize(50, 50); // 设置前景图片大小
    syslabel.move(20, 20); // 设置前景图片位置
    QPixmap pix;
    pix.load(":/timg2.jpg"); // 加载前景图片
    syslabel.setScaledContents(true);
    // 和 QPushButton 不一样，// QLabel 才用的是 setScaledContents 让图片适应 QLabel 大小

    syslabel.setPixmap(pix); // 显示图片
    w.show();

    return a.exec();
}

```

还是需要两个 QLabel，一个 QLabel 负责背景图片，一个 QLabel 负责前景图片

把前景图片嵌入到支持背景图片的 QLabel 上

你看问题解决了

如何让背景图片和前景图片四边角变成圆型

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

    myQlabel mylabel(&w); //背景透明图片的 QLabel 给主窗口
    mylabel.resize(500, 500); //设置 QLabel 大小

    QLabel syslabel(&mylabel);
    syslabel.resize(50, 50); //设置前景图片大小
    syslabel.move(20, 20); //设置前景图片位置

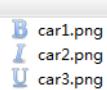
    syslabel.setScaledContents(true);
    //和 QPushButton 不一样,
    //QLabel 才用的是 setScaledContents 让图片适应 QLabel 大小
    syslabel.setStyleSheet("QString(\"border-image: url(:/timg2.jpg);border-radius:10%;\")");
    w.show();
    return a.exec();
}
```

用 QPixmap 或者 paintEvent 处理起来很麻烦，所以直接用 QSS，这里增加图片路径

Border-radius 就是倒圆角，这里倒圆角弧度为 10%，原理前面 QSS 有讲

QLabel 产生多图切换动画效果，当然其它控件也可以参考这个思路

其实 QT 的动画就是用 QTimer 定时器的信号与槽来实现的，定时器时间到一次，执行一次 QTimer 的槽函数，在槽函数里面切换图片。



准备三张图片来切换

```
#include <QTimer>
#include <QLabel>
namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
    QTimer *time1; // 定义一个定时器来执行槽函数切换图片
    QLabel *label; // 定义一个 QLabel 来显示不停切换的图片
    int xzzflag; // 这个变量是用来选择哪一张图片来切换
private slots:
    void cartoontest(); // 这就是切换图片的 QTimer 槽函数
};
```

头文件定义

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

    time1 = new QTimer(this);
    // 定义一个 QTimer 在堆中，因为构造函数执行完了要继续用这个 QTimer，嵌入给主窗口

    connect(time1, SIGNAL(timeout()), this, SLOT(cartoontest()));

    label = new QLabel(this);
    // QLabel 要在堆中定义，因为构造函数执行完了还要继续显示图片
    label->resize(100, 100);
    label->move(50, 50);
    label->setScaledContents(true);

    xzzflag=1;
    // 这里本来该初始化为 0，但是我的 png 图片是从 1 开始命名的，所以只有写 1 开始

    time1->start(300);
    // 启动定时器，因为定时器在堆中，所以没消失，那么每自动执行 300ms 就去触发一次槽函数
}
```

这是在 CPP 文件初始化图片显示的 QLabel 和定时器

```

void MainWindow::cartoontest()
{
    QPixmap pix;
    QString str = QString(":/car%1.png").arg(xzzflag++);
    pix.load(str);
    label->setPixmap(pix);

    qDebug() << "enter cartoon xzzflag = " << xzzflag;
}

```

每定时时间到一次，执行一次这个槽函数

这个%符号是我们不用每次都写 switch 去判断 xzzflag 现在等于几，然后去执行 switch 里面对应的图片字符串，这里用%就是使用 arg 获取 xzzflag 的值，然后把数字拼接在 car 后面，这样定时器触发槽函数一次，那么 car 字符串就会改变，就成了 car1，car2，car3 这种自动增加字符串尾数值，然后取寻找对应的图片来切换。这样就很方便了。



这就是动态切换图片的效果，我直接每一帧截图出来的

应用程序输出

```

cartoon
E:\QTtest\cartoon\cartoon-build-desktop-Qt_4_
enter cartoon xzzflag = 2
enter cartoon xzzflag = 3
enter cartoon xzzflag = 4
enter cartoon xzzflag = 5

```

你看 xzzflag 值在不断的增加，也就是不断的找图片，但是我的图片只有三张，所以加到第 4 张的时候就没有显示了

你可以做 if 判断不停的来回切换图片播放，因为 QTimer 定时器启动之后，必须用 QTimer::stop 函数才能停止定时器，如果不使用 Stop 函数，那么定时器会一直执行。

你可以多加些动态图片切成每一张的图片，然后把 start 时间改小点，就可以看到动画效果了。

所以一般工程当中我会把这个动画功能做成一个类，方便调用，记得程序执行完后要释放堆空间哦...

用 QMovie 类来替代定时器执行 gif 动画

如果你不想加载这么多图片组成动画的话，或者写一堆代码和定时器去组成动画。那么你就用 QMovie 类去获取 gif 格式的动画。

```
#include "mainwindow.h"
#include <QMovie> //QMovie类就是支持gif动画的
#include <QLabel>

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

    QMovie *move = new QMovie(":/timggg.gif"); //将gif动画图片给QMovie对象
    QLabel label(&w); //QLabel用来显示gif动画
    label.resize(100,100);
    label.move(100,100);
    label.setMovie(move); //将QMovie接受的gif动画赋值给QLabel
    label.setScaledContents(true);
    //因为gif动画尺寸比我们定义的QLabel窗口尺寸大,
    //所以将gif动画缩小到适合QLabel窗口的尺寸
    move->start();

    w.show();

    return a.exec();
}
```

gif 动画图片还是放在资源文件里面，然后 QMovie 去获取运行。



用 QMovie 和上一节用 QTimer 去切图片制作动画有什么区别呢？

区别在于 QTimer 制作动画灵活一些，你可以用软件指定动画的部分暂停和开始，而且还可以扩展其它更有意思的动画细节。而 QMovie 就只能播放 gif 动画，不能做其它的功能。所以根据需求来决定用哪一种播放动画的方法。

让一个图片突然滑动出现的效果

在头文件定义按钮和静态图片显示的 QLabel

```
16 public:
17     explicit MainWindow(QWidget *parent = 0);
18     ~MainWindow();
19
20 private:
21     Ui::MainWindow *ui;
22     QPushButton *button; //点击按钮弹出滑动图片
23     QLabel *label; //弹出的滑动图片在 QLabel 显示，也就是 QLabel 在滑动
24
25 private slots:
26     void buttonslot(); //槽函数用来执行滑动图像
27 };
```

```

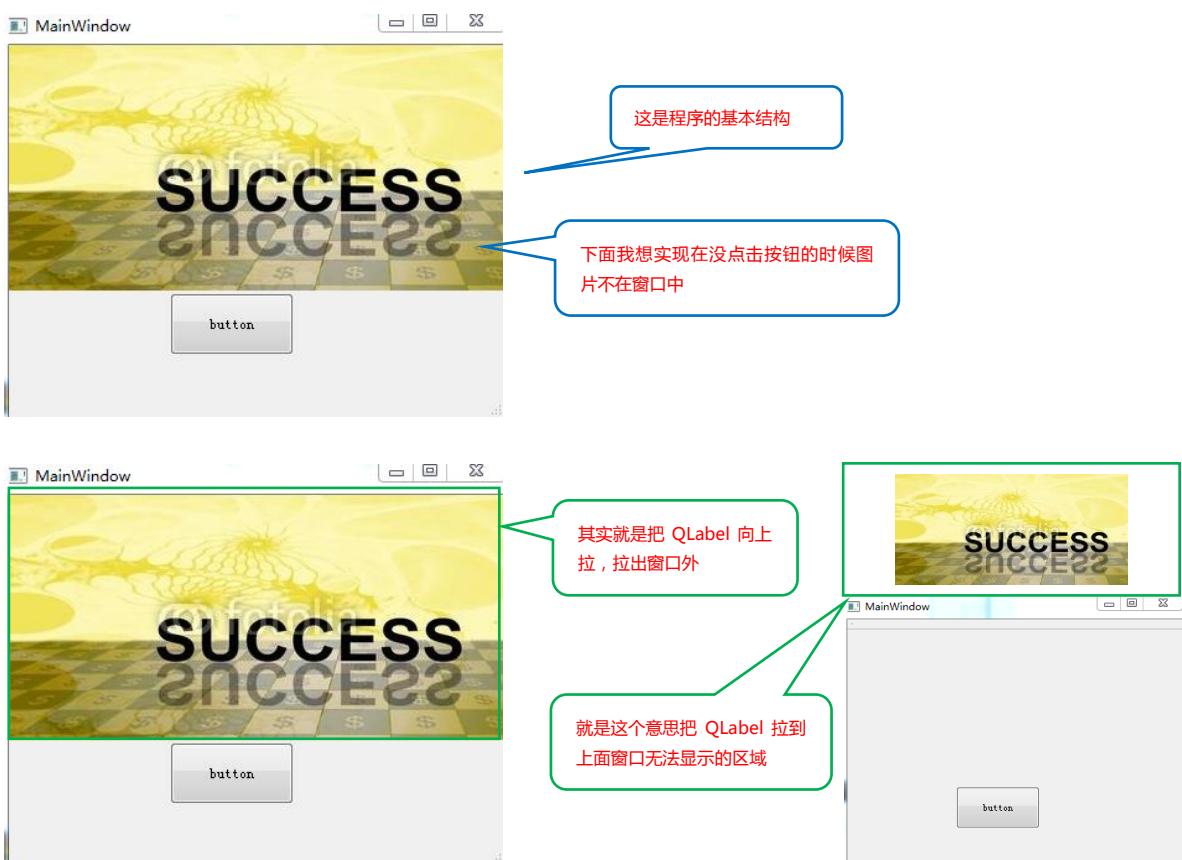
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    button = new QPushButton(this); //建立一个按钮，在槽函数图片画下来
    button->resize(100, 50);
    button->move(130, 200);
    button->setText("button");

    connect(button, SIGNAL(clicked()), this, SLOT(buttonslot()));

    label = new QLabel(this);
    QPixmap pix;
    pix.load(":/success.jpg");
    label->setGeometry(QRect(0, 0, pix.width(), pix.height()));
    //setGeometry是让QLabel框和图片大小一样
    label->setPixmap(pix);
}

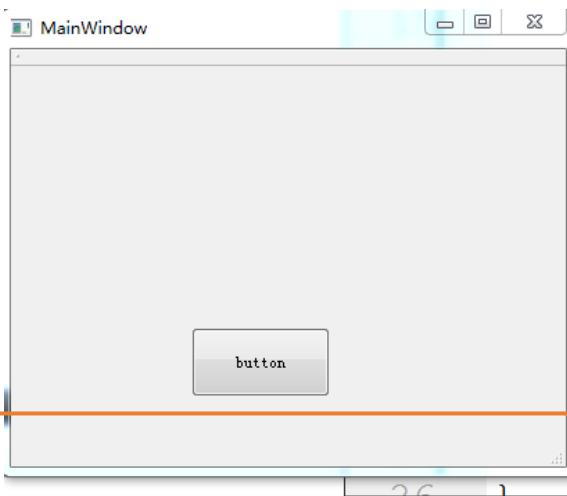
```





```
connect(button, SIGNAL(clicked()), this, S
label = new QLabel(this);
QPixmap pix;
pix.load(":/success.jpg");
label->setGeometry(QRect(0, 0, pix.width(),
//setGeometry是让QLabel框和图片大小一样
label->setPixmap(pix);
label->move(0, -100);
```

我修改了一下 QLabel 初始化
坐标 , y=-100 就是向窗口上
移动了 100 个像素 你看图片
向上隐藏了



```
connect(button, SIGNAL(clicked(
label = new QLabel(this);
QPixmap pix;
pix.load(":/success.jpg");
label->setGeometry(QRect(0, 0, p
//setGeometry是让QLabel框和图片大
label->setPixmap(pix);
label->move(0, -200);
```

我改成-200 让图
片向上隐藏完

下面在槽函数实现按钮点击后图片从上向下滑下来

```
#include <QPropertyAnimation> //动画类
```

这个产生抖动的图片是在 QLabel , 所以这里的父类指定 QLabel , 千万不要写成 this , 否则就是 MainWindow 主窗口抖动了 , 没有意义

```
void MainWindow::buttonslot()
{
    QPropertyAnimation *animation = new QPropertyAnimation(label, "geometry");
    animation->setDuration(1000); //时间间隔1000ms
    animation->setStartValue(QRect(label->x(), label->y(), label->width(), label->height()));
    animation->setEndValue(QRect(label->x(), label->y() + 200, label->width(), label->height()));
    animation->setEasingCurve(QEasingCurve::OutBounce); //设置缓和曲线
    animation->start();
}
```

→ setStartValue 是写图片滑到起始位置的坐标: 很明显现在获取的 label->x 和 y 都是 0,-200 的坐标

label->move(0, -200); 这是我们前面将图片拉上去的位置

→ setEndValue 是写图片滑到什么位置结束的坐标 :

setEasingCure 是缓和曲线 , 按钮抖动章节有讲

start() 执行后才能将图片滑动下来

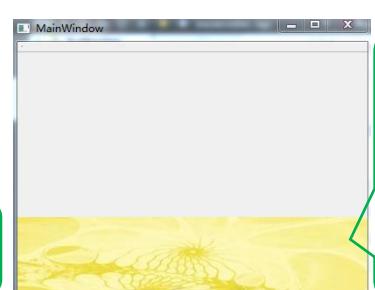
和上面按钮抖动章节一样 , 要
设置动画类



没有点
击按钮

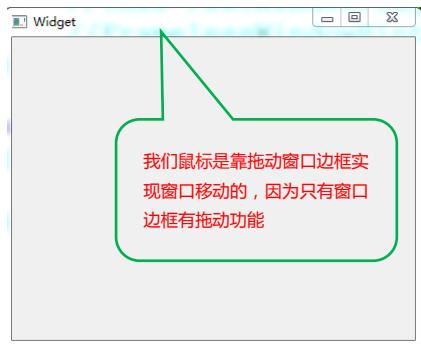


点击按钮图
片滑下来了



再多点击
几下按钮 ,
图片继续
向下移动
了 ,这种情
况自己修
改代码逻
辑解决

无边框窗口实现，扁平化界面设计



```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
    this->setWindowFlags(Qt::FramelessWindowHint);  
    //FramelessWindowHint就是将窗口边框去掉  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}  
  
    窗口边框去掉之后，鼠标是无法拖动窗口中央的，因为前面说了窗口边框才有  
    拖拽功能
```

setWindowFlags 就是设置窗口状态

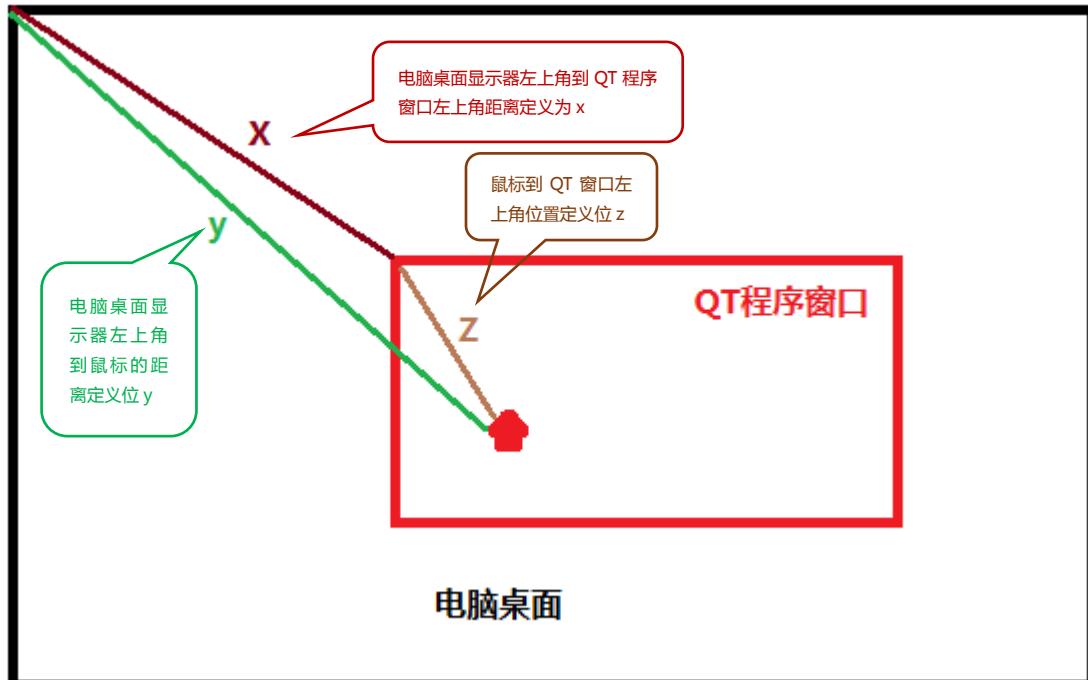
我们只有用鼠标点击事件，移动事件来获取坐标，然后去设置主窗口更新坐标显示。记住主窗口是不支持鼠标拖拽的哦，我是用鼠标事件去间接执行窗口 move 函数修改窗口位置。

```
public:  
    explicit Widget(QWidget *parent = 0);  
    ~Widget();  
  
private:  
    Ui::Widget *ui;  
    void mouseMoveEvent(QMouseEvent *event);  
    void mousePressEvent(QMouseEvent *event);  
    void mouseReleaseEvent(QMouseEvent *event);  
    QPoint z;  
};  
  
#endif // WIDGET_H
```

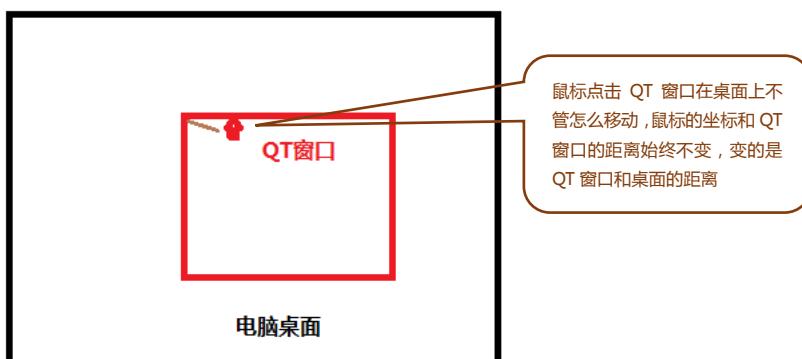
头文件建立鼠标回调函数来获取坐标，然后设置窗口新坐标

这个 z 变量是点击鼠标，移动鼠标，释放鼠标要计算的定值

鼠标坐标更新窗口坐标的理论讲解



公式满足 $x = y - z$ (x 线长度=y 长度-z 长度)



所以 z 值就是鼠标和 QT 窗口左上角的距离值，该值不变

```

void Widget::mouseMoveEvent (QMouseEvent *event)
{
    QPoint y = event->globalPos(); //globalPos获取鼠标相对于桌面左上角位置
    QPoint x = y - this->z;
    this->move(x); //移动窗口
}

void Widget::mousePressEvent (QMouseEvent *event)
{
    QPoint y = event->globalPos(); //globalPos获取鼠标相对于桌面左上角位置
    QPoint x = this->geometry().topLeft(); //this获取主窗口相对于桌面左上角位置
    this->z = y - x; //公式x = y - z //得到鼠标在QT窗口的位置
}

void Widget::mouseReleaseEvent (QMouseEvent *event)
{
    this->z = QPoint(); //释放鼠标把z值清空
}

```

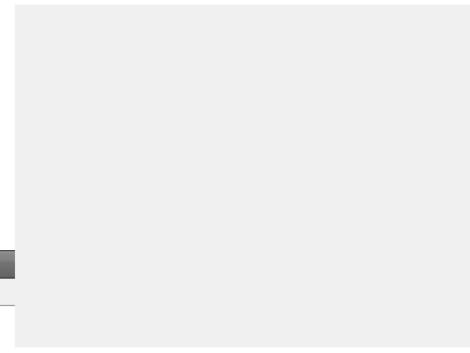
第3步，移动鼠标，马上获取鼠标与桌面左上角的新位置

第4步鼠标和桌面的新位置 - 鼠标到窗口的位置就是窗口相对于桌面改移动的位置

第1步，鼠标点击窗口，马上获取鼠标到桌面左上角距离，和窗口到桌面左上角距离

第2步得到 z 值

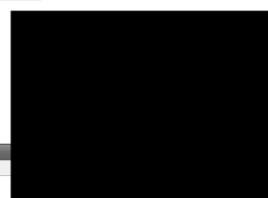
第5步释放鼠标，清空 z 值，下次点击鼠标再计算 z 值



编译运行，鼠标在拖动这个无边框窗口移动。

我感觉窗口没什么立体感，没有阴影效果，下面我们来加窗口阴影

```
3 #include <QMouseEvent> //该类里面有获取鼠标当前坐标函数
4 #include <QGraphicsDropShadowEffect> //这是添加阴影的类
5
6 Widget::Widget(QWidget *parent) :
7     QWidget(parent),
8     ui(new Ui::Widget)
9 {
10     ui->setupUi(this);
11     this->setWindowFlags(Qt::FramelessWindowHint);
12     //FramelessWindowHint就是将窗口边框去掉
13
14     QGraphicsDropShadowEffect *shaw = new QGraphicsDropShadowEffect(); //创建个阴影
15     shaw->setBlurRadius(5); //阴影的圆角大小
16     shaw->setColor(Qt::black); //阴影颜色
17     shaw->setOffset(0); //阴影和主窗口的距离
18
19     this->setGraphicsEffect(shaw); //将阴影赋值给主窗口，这样主窗口才能显示
20 }
```



这就是给窗口增加阴影

编译运行之后整个窗口都是黑色的，这是为什么呢？

其实代码是没有问题的，只是阴影不能加在主窗口上，程序第一个 widget 窗口是不能加阴影的，那么只有建立子窗口来覆盖主窗口加阴影

但是 QgraphicsDropShadowEffect 对 QWidget 窗口无效，我用 QSS 实现

```
ui->setupUi(this);
this->setWindowFlags(Qt::FramelessWindowHint);
//FramelessWindowHint就是将窗口边框去掉

this->setAttribute(Qt::WA_TranslucentBackground); //隐藏主窗口

w2 = new QWidget(this);
w2->resize(this->width(), this->height());
w2->move(this->x(), this->y());

w2->setStyleSheet("QWidget{background-color:gray;\n    border-top-left-radius:15px;\n    border-top-right-radius:15px;\n    border-bottom-right-radius:15px;\n    border-bottom-left-radius:15px;\n    border-right: transparent;\n    border-top: 20px solid \\\n        qlineargradient(y0:0, y1:1,stop: 0 #ececef, stop: 1 black);\n    border-left: 20px solid\\\n        qlineargradient(x0:0, x1:1,stop: 0 #ececef, stop: 1 black);\n    border-bottom: 20px solid\\\n        qlineargradient(y0:0, y1:1,stop: 0 white, stop: 1 black);\n    border-right: 20px solid \\\n        qlineargradient(x0:0, x1:1,stop: 0 white, stop: 1 black); }");
```

编译运行



阴影效果有了，但是看起太夸张了，下面是实现效果的 QSS 代码

```
setStyleSheet("QWidget {background-color:gray;\\
    border-top-left-radius:15px;\\
    border-top-right-radius:15px;\\
    border-bottom-right-radius:15px;\\
    border-bottom-left-radius:15px;\\
    border-right: transparent;\\
    border-top: 20px solid \\
    qlineargradient(y0:0, y1:1, stop: 0 #ececef, stop: 1 black);\\
    border-left: 20px solid\\
    qlineargradient(x0:0, x1:1, stop: 0 #ececef, stop: 1 black); \\
    border-bottom: 20px solid\\
    qlineargradient(y0:0, y1:1, stop: 0 white, stop: 1 black);\\
    border-right: 20px solid \\
    qlineargradient(x0:0, x1:1, stop: 0 white, stop: 1 black); }");
```

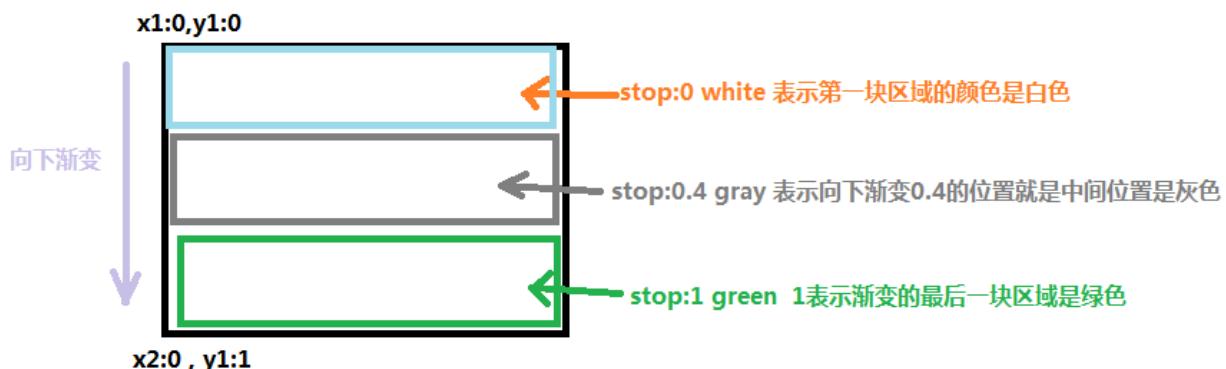
qlineargradient 线性渐变使用介绍

```
w2 = new QWidget(this);
w2->resize(this->width(),this->height());
w2->move(this->x(),this->y());

w2->setStyleSheet("QWidget{background-color:gray;\\
    border-top-left-radius:15px;\\
    border-top-right-radius:15px;\\
    border-bottom-right-radius:15px;\\
    border-bottom-left-radius:15px;\\
    border-right: transparent;\\
background: qlineargradient(x1:0, y1:0, x2:0, y2:1,stop:0 white, stop: 0.4 gray, stop:1 green);
}");
```

这句就是给窗口加渐变颜色

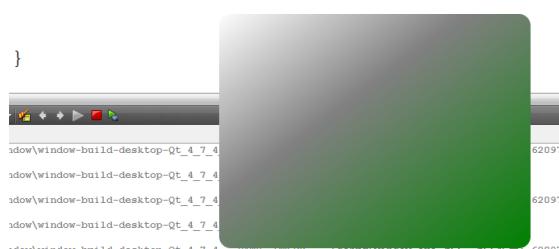
QSS 代码 qlineargradient 原理



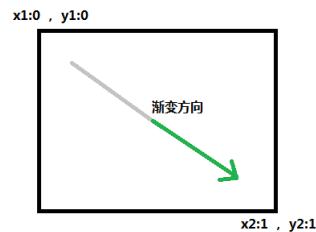
和我设想的结果是一样的

修改 x2:1,y2:1

```
border-right: transparent;\nbackground: qlineargradient(x1:0, y1:0, x2:1, y2:1, stop:0 white, stop: 0.4 gray, stop:1 green);\n}");
```



修改 x2 , y2 的值

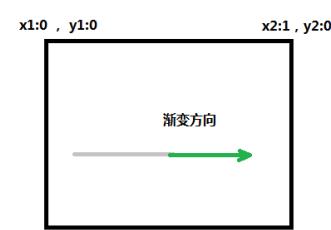


修改 x2:1 , y1:0

```
background: qlineargradient(x1:0, y1:0, x2:1, y2:0, stop:0 white, stop: 0.4 gray, stop:1 green);\n}");
```



你看起始位置是 x1:0 , y1:0,但是终止位置是 x2:1 , y2:0



用 border 设置 widget 窗口的边框来达到阴影和透明

```
w2->setStyleSheet("QWidget{background-color:gray;\nborder-top-left-radius:15px;\nborder-top-right-radius:15px;\nborder-bottom-right-radius:15px;\nborder-bottom-left-radius:15px;\nborder:10px solid rgba(91,79,96,50%);}\");
```

Border 才能用 solid 线把窗口圆角包裹起来，像 border-left: 这种是边框直线无法包裹，建议不要用

给边框设置 rgba 颜色和透明度，
50%就是透明度，100%不透明，
0%完全透明



用 border 方式达到边框透明窗口立体感，但是用阴影立体感还是没有解决

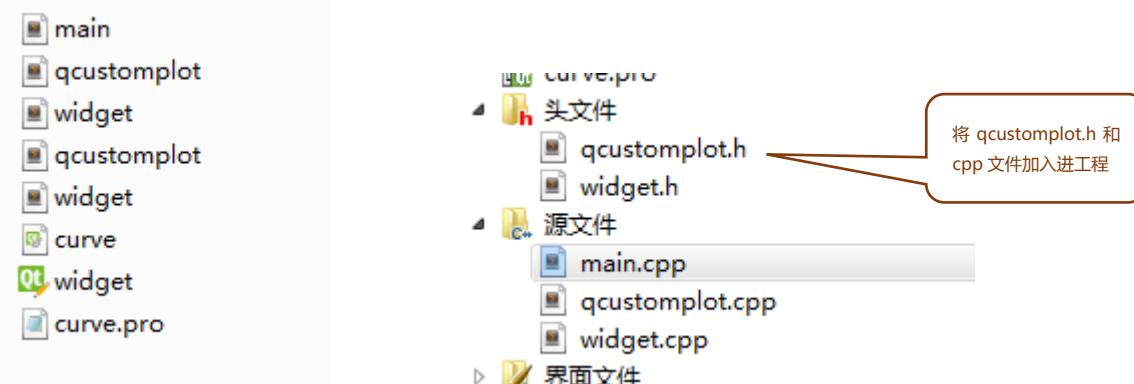
QT4.7 曲线库 qcustomplot 移植



去 qcustomplot 官方下载曲线库

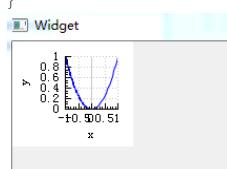
examples	2019/6/5 23:41	文件夹
changelog	2018/6/25 7:04	文本文档
GPL	2011/9/4 11:42	文本文档
qcustomplot	2018/6/25 7:05	CPP 文件
qcustomplot	2018/6/25 7:05	H 文件

将 qcustomplot.cpp 和 qcustomplot.h 拷贝到自己 Qt Creator 建立的工程下



在工程下写一个范例运行

```
#include <QtGui/QApplication>
#include "widget.h"
#include <QVector>
#include "qcustomplot.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.resize(400,400);
    QCustomPlot *customPlot = new QCustomPlot(&w); //建立个曲线库嵌入主窗口 QWidget 对象
    // generate some data:
    QVector<double> x(300), y(300); // initialize with entries 0..100
    for (int i=0; i<101; ++i)
    {
        x[i] = i/50.0 - 1; // x goes from -1 to 1
        y[i] = x[i]*x[i]; // let's plot a quadratic function
    }
    // create graph and assign data to it:
    customPlot->addGraph();
    customPlot->graph(0)->setData(x, y);
    // give the axes some labels:
    customPlot->xAxis->setLabel("x");
    customPlot->yAxis->setLabel("y");
    // set axes ranges, so we see all data:
    customPlot->xAxis->setRange(-1, 1);
    customPlot->yAxis->setRange(0, 1);
    customPlot->replot();
    w.show();
    return a.exec();
}
```



运行效果已经有了，证明这个曲线库没有问题，但是感觉曲线窗口好小

```

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.resize(400,400);

    QCustomPlot *customPlot = new QCustomPlot(&w);
    customPlot->resize(400,400);
    // generate some data:
    QVector<double> x(101), y(101);
    for (int i=0; i<101; ++i)
    {
        x[i] = i/50.0 - 1; // x goes from -1 to 1
        y[i] = x[i]*x[i]; // let's plot a parabola
    }
    // create graph and assign data
    customPlot->addGraph();
    customPlot->graph(0)->setData(x, y);
    // give the axes some labels:
    customPlot->xAxis->setLabel("x");
    customPlot->yAxis->setLabel("y");
    // set axes ranges, so we see all data:
    customPlot->xAxis->setRange(-1, 1);
    customPlot->yAxis->setRange(0, 1);
}

```

将qcustomplot也有设置曲线窗口大小的函数 resize ,和 Qt 基本窗口大小设置函数一样

QCustomplot 创建曲线图表用法案例

给一组已经收集完成的大量数据，数据用 x , y 表示

```

#include "qcustomplot.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.resize(400,400);

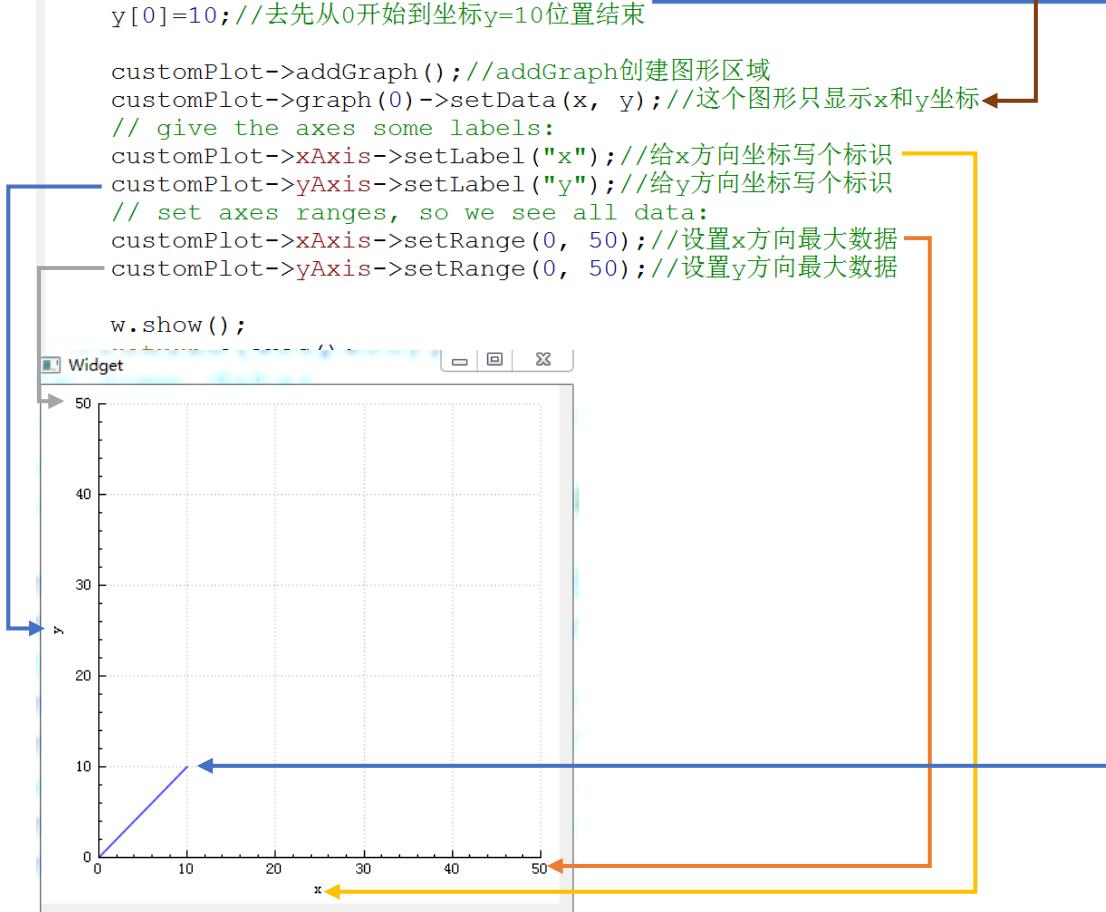
    QCustomPlot *customPlot = new QCustomPlot(&w); //创建曲线框类
    customPlot->resize(400,400);
    // generate some data:
    QVector<double> x(101), y(101); // x和y数组可以放100个数据
    x[0]=10;//曲线从0开始到坐标x=10结束
    y[0]=10;//去先从0开始到坐标y=10位置结束

    customPlot->addGraph(); //addGraph创建图形区域
    customPlot->graph(0)->setData(x, y); //这个图形只显示x和y坐标
    // give the axes some labels:
    customPlot->xAxis->setLabel("x"); //给x方向坐标写个标识
    customPlot->yAxis->setLabel("y"); //给y方向坐标写个标识
    // set axes ranges, so we see all data:
    customPlot->xAxis->setRange(0, 50); //设置x方向最大数据
    customPlot->yAxis->setRange(0, 50); //设置y方向最大数据
}

w.show();

```

说白了就是把现在
x 和 y 的当前值放进图形里面显示



我们用计时器让这个数据曲线动起来

先讲讲 QVector 和 C++ 标准 Vector 有什么区别

比如我们想要一个 int 类型数组，我们原先会写 `int array[10]`，我们在 Qt 里可以写 `QVector<int> array(10)`

赋值的时候，我们依然可以照旧 `array[5]=4`; 想获取某一项的值也还可以 `array[9]`，也就是说，原来的特性我们还可以用。

原先我们必须预定义好大小，而用 QVector 我们虽然最好也先定义好大小，但是预先不定义也可以。

我们可以使用 `append` 函数来不停的临时增加 QVector 定义的数组大小，需要给 QVector 定义的变量增加个数据就执行 `append(变量)`，类似 C++ vector 的 `push_back`，看我 C++ coding 文档就知道了。

```
public:  
    explicit Widget(QWidget *parent = 0);  
    ~Widget();  
  
private:  
    Ui::Widget *ui;  
    QVector<double> x, y; // x和y数组可以放100个数据  
    QCustomPlot *customPlot;  
    QTimer *timer;  
private slots:  
    void grap_update();  
};
```

这里不像上面指定 QVector x(101), y(101)
数组的大小，我们后面用 append 来加数据

因为只能在槽函数里面实时更新曲线，所以
这里把图像变量和数据变量定义到头文件

这是计时器时间到了的槽函数，
到时候在这里更新曲线数据

```
int i;//定义个全局变量来增加数据  
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    i=0;//一定要初始化，QT的变量默认值都很大  
    customPlot = new QCustomPlot(this);//创建曲线框类  
    customPlot->resize(400,400);  
    // generate some data:  
    customPlot->addGraph();//addGraph创建图形区域  
    // give the axes some labels:  
    customPlot->xAxis->setLabel("x");//给x方向坐标写个标识  
    customPlot->yAxis->setLabel("y");//给y方向坐标写个标识  
    // set axes ranges, so we see all data:  
    customPlot->xAxis->setRange(0, 50);//设置x方向最大数据  
    customPlot->yAxis->setRange(0, 50);//设置y方向最大数据  
  
    QTimer *timer = new QTimer();  
    connect(timer, SIGNAL(timeout()), this, SLOT(grap_update()));  
    timer->start(1000);
```

这是增加曲线的数据

这里没有 `graph()->setData` 函数 因为这个函数是显示 x 和 y 曲线的当前值，要放在槽函数
实时使用才行

其余设置曲线窗口
xy 坐标最大值和曲
线窗口标识，这些设
置方法不变

这个计时器 1 秒钟更
新一次曲线图

```

void Widget::grap_update()
{
    i++;
    x.append(i);
    y.append(i);
    customPlot->graph(0)->setData(x, y);
    customPlot->replot();
    qDebug() << "timer = " << i;
}

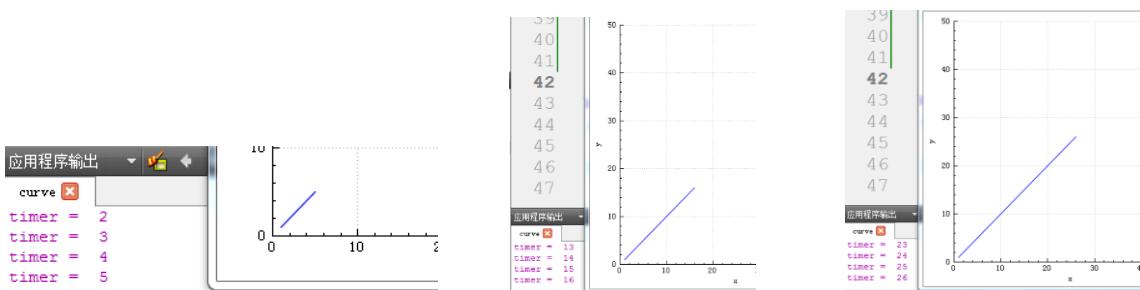
```

这就是每秒在槽函数更新的 xy 值

把 i 用 append 加进 xy 数组，我上一页说过，这种就是给 QVector 数组临时增加数据

这就是必须把更新的 x, y 值放在 graph->setData 里面

然后 replot 每次去 setData 取数据来更新曲线界面



这就是曲线数据在随着 timer 不停变化的效果。

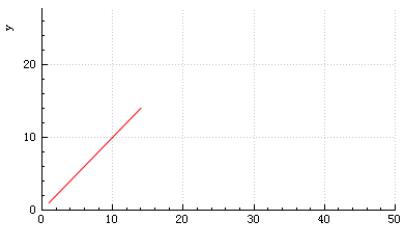
记得释放 new 分配的内存，还有这里的 i 值是无限增长的，记得优化

设置曲线颜色

```

customPlot->addGraph(); //addGraph 创建图形区域
customPlot->graph(0)->setPen(QPen(Qt::red)); //设置曲线颜色为红色

```

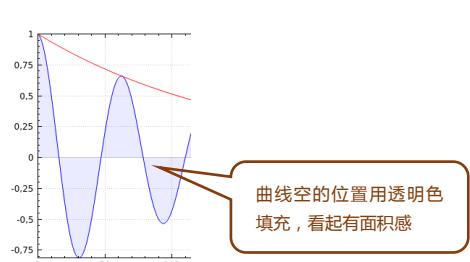
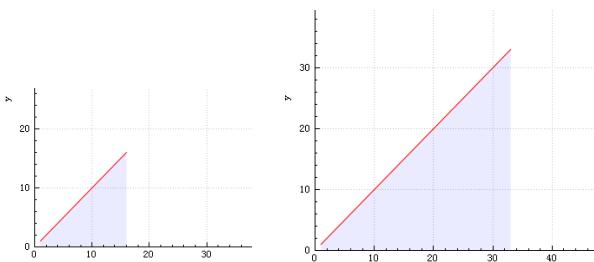


设置曲线下方和 x 轴之间用透明颜色填充

```

customPlot->addGraph(); //addGraph 创建图形区域
customPlot->graph(0)->setPen(QPen(Qt::red)); //设置曲线颜色为红色
customPlot->graph(0)->setBrush(QBrush(QColor(0, 0, 255, 20))); //曲线下面用透明颜色填充

```



我想增加一条或者两条以上的曲线

```
private:  
    Ui::Widget *ui;  
    QVector<double> x, y;  
    QVector<double> x2, y2;  
    QCustomPlot *customPlot;  
    QTimer *timer;  
  
private slots:  
  
    customPlot->addGraph(); // addGraph 创建图形区域  
    customPlot->graph(0)->setPen(QPen(Qt::red)); // 设置曲线颜色为红色  
  
    customPlot->addGraph();  
    customPlot->graph(1)->setPen(QPen(Qt::blue));  
  
void Widget::graph_update()  
{  
    i++;  
  
    x.append(i);  
    y.append(i);  
    x2.append(i+1);  
    y2.append(i+5);  
    customPlot->graph(0)->setData(x, y);  
    customPlot->graph(1)->setData(x2, y2);  
    customPlot->replot();  
    qDebug() << "timer = " << i;  
}
```

在头文件必须增加第 2 个 QVector 变量，因为一个 QVector 变量管理一条曲线

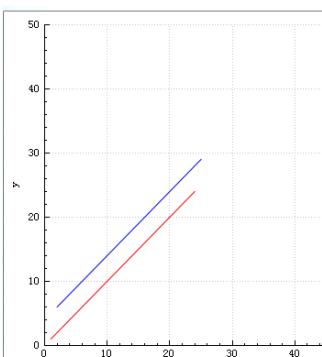
在 CPP 文件已经有一条曲线了，红色曲线

在给 graph 写 1 的时候，必须再先执行一次 addGraph()，否则会程序崩溃

graph(1) 表示新添加一条曲线，QPen 设置曲线为蓝色

在槽函数给第 2 条曲线增加数据

把第 2 条曲线的 x2y2 值写入第 2 个曲线图形



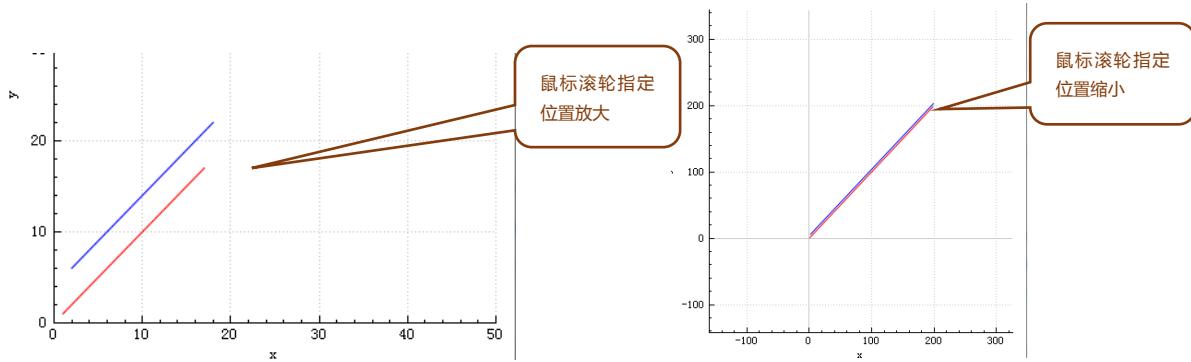
你看两条曲线就有了 如果还要加第 3 条曲线 就再增加 addGraph 和 graph(2)

增加鼠标移动曲线和缩放功能

```
customPlot->xAxis->setLabel("x");//给x方向坐标写个标识  
customPlot->yAxis->setLabel("y");//给y方向坐标写个标识  
customPlot->xAxis->setRange(0, 50);//设置x方向最大数据  
customPlot->yAxis->setRange(0, 50);//设置y方向最大数据  
  
customPlot->graph(0)->rescaleAxes();  
  
QTimer *timer = new QTimer();  
connect(timer, SIGNAL(timeout()), this, SLOT(grap_update()));  
timer->start(100);  
  
customPlot->setInteractions(QCP::iRangeDrag | QCP::iRangeZoom | QCP::iSelectPlottables);
```

把曲线窗口建立完成后，在构造函数最后增加 setInteractions 函数就可以实现了

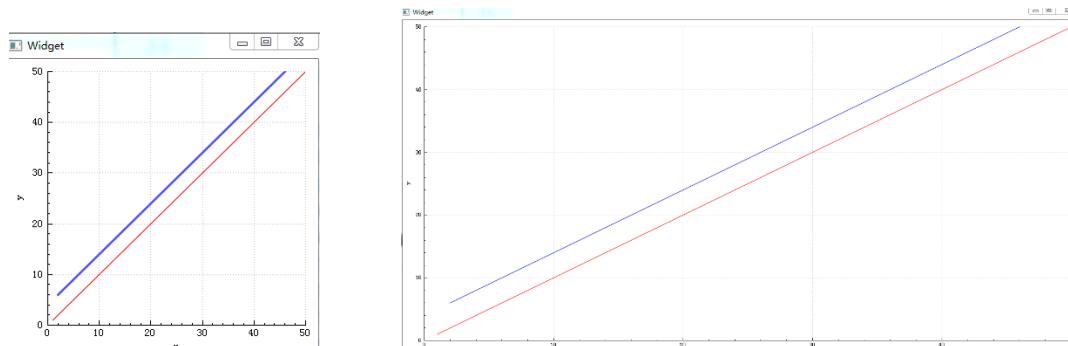
```
customPlot->setInteractions(QCP::iRangeDrag | QCP::iRangeZoom | QCP::iSelectPlottables);
```



如何让曲线窗口尺寸随着我 widget 主窗口尺寸变大缩小

```
void Widget::grap_update()  
{  
    i++;  
    x.append(i);  
    y.append(i);  
    x2.append(i+1);  
    y2.append(i+5);  
    customPlot->graph(0)->setData(x, y);  
    customPlot->graph(1)->setData(x2, y2);  
    customPlot->replot();  
    customPlot->resize(this->width(), this->height());  
    qDebug() << "timer = " << i;  
}
```

和前面章节《如何设计主界面切换窗口的内容》一样，将曲线构造函数对象的 resize 函数放在槽函数设置曲线窗口大小就行了，但是这种槽函数必须是实时都在更新的才行，



缩小放大 xy 的的最大格数都是 50，除非你滚轮滑动。

QCPGraph 类在曲线上打点

```
customPlot = new QCustomPlot(this); // 创建曲线框类  
customPlot->resize(500, 500);  
  
QVector<double> x1(20), y1(20);  
  
for (int i=0; i<x1.size(); ++i)  
{  
    x1[i] = i/(double)(x1.size()-1)*10;  
    y1[i] = qCos(x1[i]*0.8+qSin(x1[i]*0.16+1.0))*qSin(x1[i]*0.54)+1.4;  
}
```

老规矩先创建曲线框图

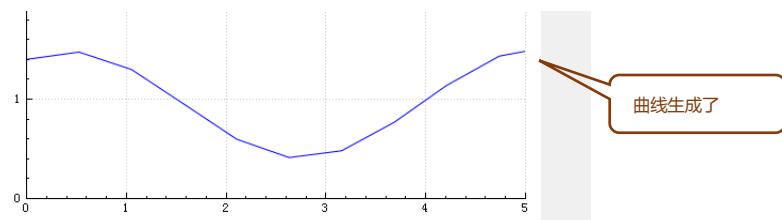
设置一条曲线的数据数组

给数组里面添加曲线每个点数据，创建一条 x1 , y1 曲线数据

下面正题来了

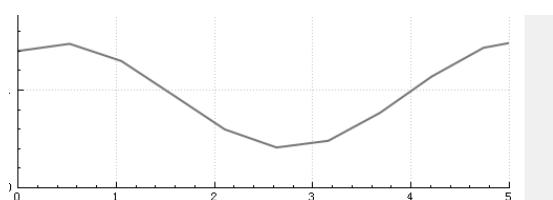
```
// create and configure plottables:  
QCPGraph *graph1 = customPlot->addGraph();  
graph1->setData(x1, y1);
```

用 QCPGraph 生成曲线



给曲线上色

```
QCPGraph *graph1 = customPlot->addGraph();  
graph1->setData(x1, y1);  
graph1->setPen(QPen(QColor(120, 120, 120), 2));
```



设置曲线形状

曲线形状像*、+、x、o 等等

QCPGraph::setScatterStyle 该函数设置曲线形状

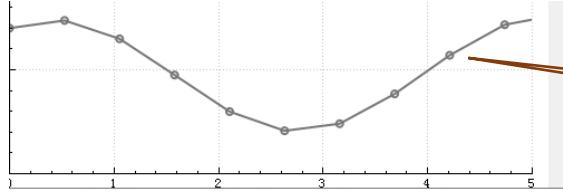
```
QCPGraph::setScatterStyle(QCPScatterStyle(QCPSscatterStyle::ssCircle));
```

设置曲线形状为圆点，就是 o

```

QCPGraph *graph1 = customPlot->addGraph();
graph1->setData(x1, y1);
graph1->setPen(QPen(QColor(120, 120, 120), 2));
graph1->setScatterStyle(QCPScatterStyle::ssCircle));

```



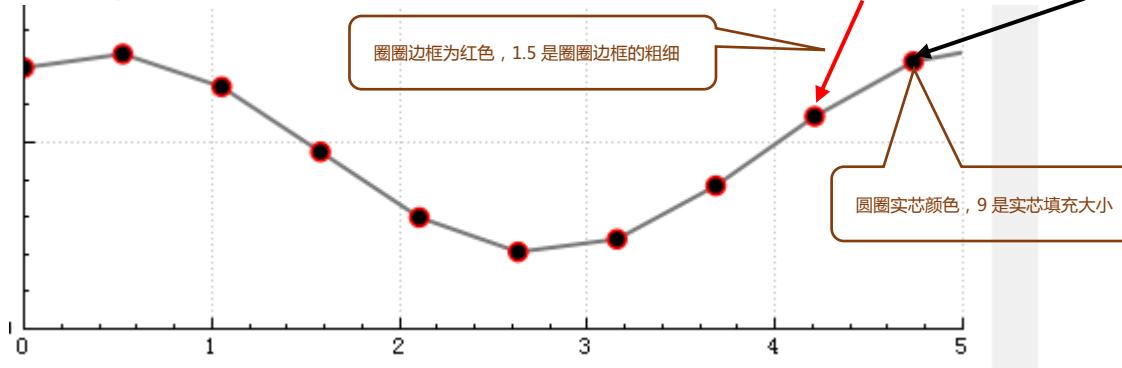
这就是设置曲线形状实例

我觉得这个圈圈好小哦，我想加大这个圈圈

```

QCPGraph ::setScatterStyle(QCPScatterStyle(QCPScatterStyle::ssCircle, QPen(Qt::red, 1.5), QBrush(Qt::black), 9));

```



圈圈边框为红色，1.5 是圈圈边框的粗细

圆圈实芯颜色，9 是实芯填充大小

```

QCPGraph *graph1 = customPlot->addGraph();
graph1->setData(x1, y1);
graph1->setPen(QPen(QColor(120, 120, 120), 2));
graph1->setScatterStyle(QCPScatterStyle(QCPScatterStyle::ssCircle, QPen(Qt::red, 1.5), QBrush(Qt::black), 9));

```

这就是完整代码

这句函数

QCPGraph 类两条曲线之间产生阴影，这样看起很舒服

```

customPlot = new QCustomPlot(this); // 创建曲线框类
customPlot->resize(500, 500);

```

```

// prepare data:
 QVector<double> x1(20), y1(20);
 QVector<double> x2(100), y2(100);
for (int i=0; i<x1.size(); ++i)
{
    x1[i] = i/(double)(x1.size()-1)*10;
    y1[i] = qCos(x1[i]*0.8+qSin(x1[i]*0.16+1.0))*qSin(x1[i]*0.54)+1.4;
}
for (int i=0; i<x2.size(); ++i)
{
    x2[i] = i/(double)(x2.size()-1)*10;
    y2[i] = qCos(x2[i]*0.85+qSin(x2[i]*0.165+1.1))*qSin(x2[i]*0.50)+1.7;
}

```

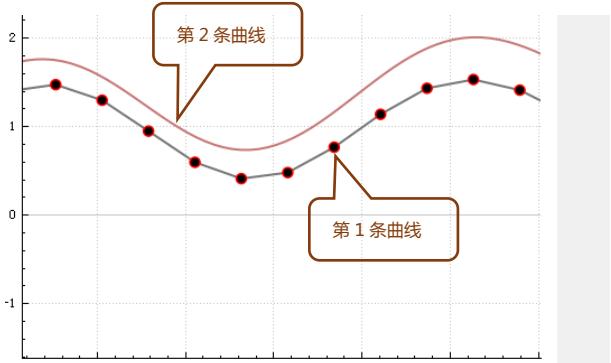
以上是设置曲线窗口，增加两个曲线的数据存放数据，绘制两条曲线 x1, y1, x2, y2 的数据，这些都是常规操作不用多解释

```
// create and configure plottables:  
QCPGraph *graph1 = customPlot->addGraph();  
graph1->setData(x1, y1);  
graph1->setPen(QPen(QColor(120, 120, 120), 2));  
graph1->setScatterStyle(QCPScatterStyle::ssCircle, QPen(Qt::r
```

第 1 条曲线绘
制上页讲过

```
QCPGraph *graph2 = customPlot->addGraph();  
graph2->setData(x2, y2);  
graph2->setPen(QPen(QColor(200, 120, 120), 2));
```

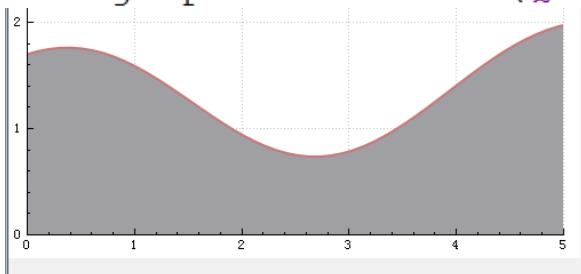
这是第 2 条曲线



现在我想让第 2 条曲线用颜色面积填充到第 1 条曲线

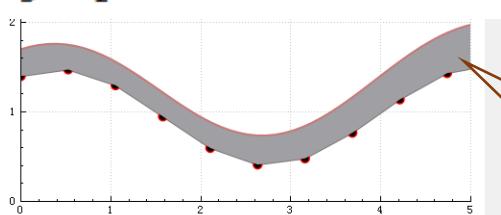
```
QCPGraph *graph2 = customPlot->addGraph();  
graph2->setData(x2, y2);  
graph2->setPen(QPen(QColor(200, 120, 120), 2));  
graph2->setBrush(QColor(Qt::gray));
```

第 2 条曲线用灰色面积填充到第 1 条曲
线，发现不行，setBrush 是填充从当前
曲线开始到下面 x 横坐标结尾



```
QCPGraph *graph2 = customPlot->addGraph();  
graph2->setData(x2, y2);  
graph2->setPen(QPen(QColor(200, 120, 120), 2));  
graph2->setBrush(QColor(Qt::gray));  
graph2->setChannelFillGraph(graph1);
```

SetChannelFillGraph 函数就是要求
graph2 对象也就是第 2 条曲线，填充到
graph1 对象结束，也就是第 1 条曲线

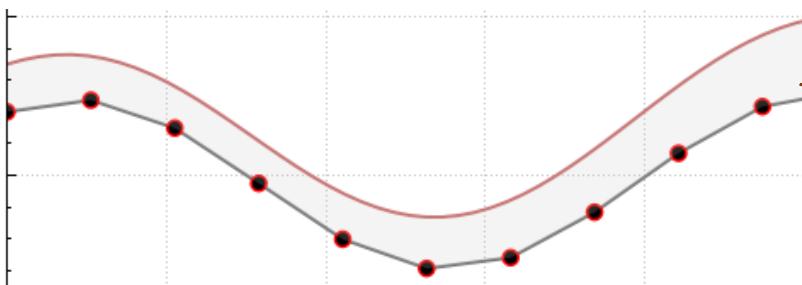


我感觉填充颜色是完全的灰色不好看，我喜欢带透明感的填充色

```

QCPGraph *graph2 = customPlot->addGraph();
graph2->setData(x2, y2);
graph2->setPen(QPen(QColor(200, 120, 120), 2));
graph2->setBrush(QColor(200, 200, 200, 50));
graph2->setChannelFillGraph(graph1);

```



我们把 Qt::gray 改成用数字表示颜色，最后这个 50 是颜色透明度

你看阴影部分曲线就出来了，很漂亮

QCPBars 类用来画条型柱状图

```

customPlot = new QCustomPlot(this); // 创建曲线框类
customPlot->resize(500, 500);

// prepare data:
QVector<double> x3(20), y3(20);

x3[0] = 0; // 第 1 个柱状在坐标轴 x 方向上，第几个坐标点开始显示
y3[0] = 5; // y 是柱状高度

QCPBars *bars1 = new QCPBars(customPlot->xAxis, customPlot->yAxis);
bars1->setWidth(1);
bars1->setData(x3, y3);
bars1->setPen(QPen(Qt::red));
bars1->setBrush(QColor(10, 140, 70, 160));

```



QCPBars::setWidth 表示柱状宽度，这里是 1 格

柱状高度 y3[0] = 5 表示柱状上升 5 格

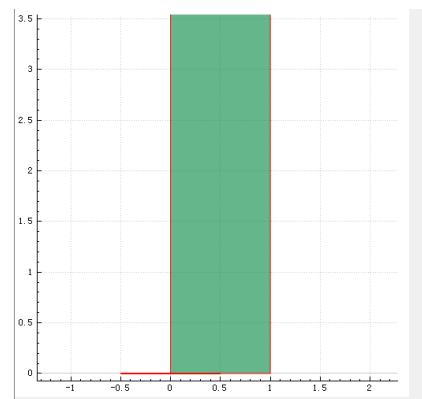
QCPBars::setPen 表示柱状边框颜色，这里是红色

QCPBars::setBrush 表示柱状填充颜色，这里是绿色

柱状高度 x3[0] = 0 表示柱状从 0 点开始

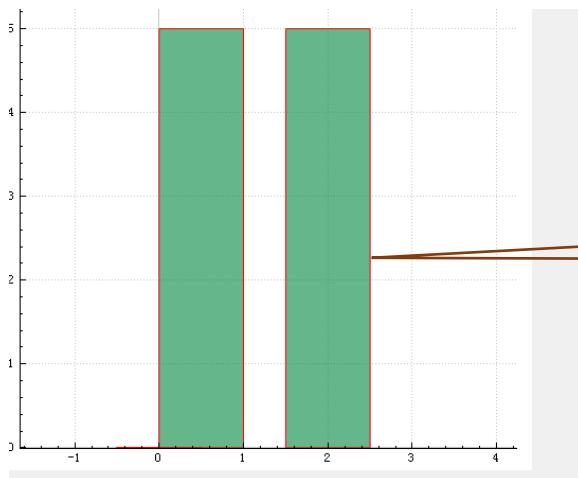
现在有个问题，我的柱状图是从 0 轴的中间开始的，感觉横跨了-0.5 到+0.5 看起来不舒服能不能让柱状图就在 0~1 格之间显示

```
x3[0] = 0.5;  
y3[0] = 5;  
  
QCPBars *bars1 = new QCPBars(customPlot->xAxis, customPlot->yAxis);  
bars1->setWidth(1);  
bars1->setData(x3, y3);  
bars1->setPen(QPen(Qt::red));  
bars1->setBrush(QColor(10, 140, 70, 160));
```

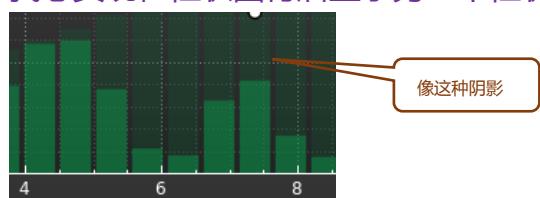


如何在坐标图上画两个柱状图？

```
QVector<double> x3(20), y3(20);  
  
x3[0] = 0.5;  
y3[0] = 5;  
  
x3[1] = 2;  
y3[1] = 5;  
  
QCPBars *bars1 = new QCPBars(customPlot->xAxis, customPlot->yAxis);  
bars1->setWidth(1);  
bars1->setData(x3, y3);  
bars1->setPen(QPen(Qt::red));  
bars1->setBrush(QColor(10, 140, 70, 160));
```



我想实现在柱状图背后显示另一个柱状图阴影



```

QVector<double> x3(20), y3(20);
QVector<double> x4(20), y4(20);

x3[0] = 0.5;
y3[0] = 5;

x4[0] = 0.5;
y4[0] = 8;

```

增加阴影柱状图的位置和高度数据

位置和 x3 柱状图一样在横着 0.5 刻度

高度为 8，这样 x4y4 柱状图阴影看起来就比 x3y3 柱状图高出 3 个刻度

```

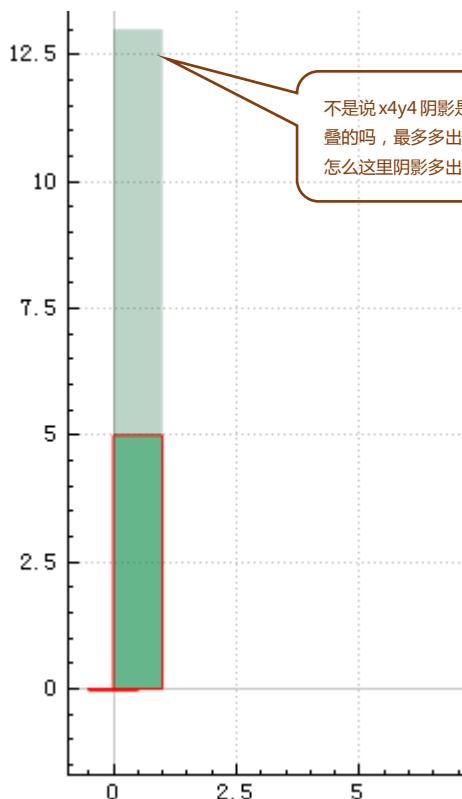
QCPBars *bars1 = new QCPBars(customPlot->xAxis, customPlot->yAxis);
bars1->setWidth(1);
bars1->setData(x3, y3);
bars1->setPen(QPen(Qt::red));
bars1->setBrush(QColor(10, 140, 70, 160));

QCPBars *bars2 = new QCPBars(customPlot->xAxis, customPlot->yAxis);
bars2->setWidth(1);
bars2->setData(x4, y4);
bars2->setPen(Qt::NoPen);
bars2->setBrush(QColor(10, 100, 50, 70));
bars2->moveAbove(bars1);

```

其实就是两个柱状图的重叠，所以你必须让 x4y4 柱状图有透明效果这样才能和前面的 x3y3 分离出来，透明度 70

QCPBars::moveAbove 就是让 x4y4 的柱状图对象 bars2 重叠在 x3y3 柱状图之上，这里要注意



不是说 x4y4 阴影是和 x3y3 重叠的吗，最多多出 3 个刻度，怎么这里阴影多出了 8 个刻度

确实多出了 8 个刻度，这 8 个刻度是叠加在 x3y3 上的，真正问题就在这里，不是重叠而是叠加

```

x4[0] = 0.5;
y4[0] = 8;

```

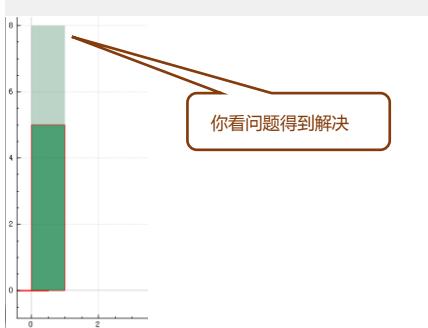
```

QCPBars *bars1 = new QCPBars(customPlot->xAxis, customPlot->yAxis);
bars1->setWidth(1);
bars1->setData(x3, y3);
bars1->setPen(QPen(Qt::red));
bars1->setBrush(QColor(10, 140, 70, 160));

QCPBars *bars2 = new QCPBars(customPlot->xAxis, customPlot->yAxis);
bars2->setWidth(1);
bars2->setData(x4, y4);
bars2->setPen(Qt::NoPen);
bars2->setBrush(QColor(10, 100, 50, 70));
// bars2->moveAbove(bars1);

```

把这里取消就是了，这里是要求你 x4y4 的对象 bars2 在 x3y3 的高度上再增加

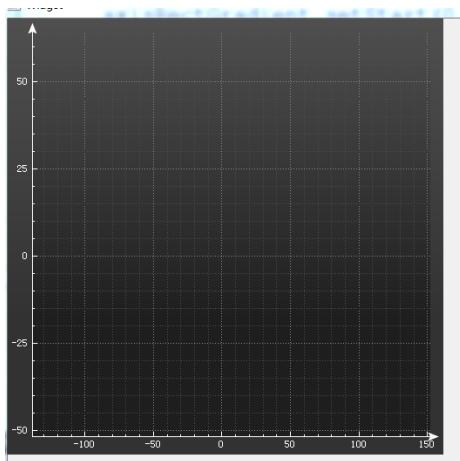


你看问题得到解决

设置坐标图的背景颜色和坐标 xy 轴的刻度线风格

```
customPlot = new QCustomPlot(this); // 创建曲线框类
customPlot->resize(500, 500);
// set some pens, brushes and backgrounds:
customPlot->xAxis->setBasePen(QPen(Qt::white, 1));
customPlot->yAxis->setBasePen(QPen(Qt::white, 1));
customPlot->xAxis->setTickPen(QPen(Qt::white, 1));
customPlot->yAxis->setTickPen(QPen(Qt::white, 1));
customPlot->xAxis->setSubTickPen(QPen(Qt::white, 1));
customPlot->yAxis->setSubTickPen(QPen(Qt::white, 1));
customPlot->xAxis->setTickLabelColor(Qt::white);
customPlot->yAxis->setTickLabelColor(Qt::white);
customPlot->xAxis->grid()->setPen(QPen(QColor(140, 140, 140), 1, Qt::DotLine));
customPlot->yAxis->grid()->setPen(QPen(QColor(140, 140, 140), 1, Qt::DotLine));
customPlot->xAxis->grid()->setSubGridPen(QPen(QColor(80, 80, 80), 1, Qt::DotLine));
customPlot->yAxis->grid()->setSubGridPen(QPen(QColor(80, 80, 80), 1, Qt::DotLine));
customPlot->xAxis->grid()->setSubGridVisible(true);
customPlot->yAxis->grid()->setSubGridVisible(true);
customPlot->xAxis->grid()->setZeroLinePen(Qt::NoPen);
customPlot->yAxis->grid()->setZeroLinePen(Qt::NoPen);
customPlot->xAxis->setUpperEnding(QCPLineEnding::esSpikeArrow);
customPlot->yAxis->setUpperEnding(QCPLineEnding::esSpikeArrow);
QLinearGradient plotGradient;
plotGradient.setStart(0, 0);
plotGradient.setFinalStop(0, 350);
plotGradient.setColorAt(0, QColor(80, 80, 80));
plotGradient.setColorAt(1, QColor(50, 50, 50));
customPlot->setBackground(plotGradient);
QLinearGradient axisRectGradient;
axisRectGradient.setStart(0, 0);
axisRectGradient.setFinalStop(0, 350);
axisRectGradient.setColorAt(0, QColor(80, 80, 80));
axisRectGradient.setColorAt(1, QColor(30, 30, 30));
customPlot->axisRect()->setBackground(axisRectGradient);
customPlot->rescaleAxes();
customPlot->yAxis->setRange(0, 2);
customPlot->setInteractions(QCP::iRangeDrag | QCP::iRangeZoom | QCP::iSelectPlottables);
```

以上就是设置坐标图刻度线和背景颜色，风格的参数自己思考每个函数的意思



有时候曲线的某个点的值在坐标图上看起来不太舒服，改成鼠标指到曲线某个点就输出某个点坐标

```
explicit Widget(QWidget *parent = 0);
~Widget();

private:
    Ui::Widget *ui;
    QCustomPlot *customplot;

private slots:
    void myMouseMove(QMouseEvent* event);

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

    customplot = new QCustomPlot(this);
    customplot->resize(500,500);

    QVector<double> x1(20),y1(20);
    for (int i=0; i<x1.size(); ++i)
    {
        x1[i] = i/(double)(x1.size()-1)*10;
        y1[i] = qCos(x1[i]*0.8+qSin(x1[i]*0.16+1.0))*qSin(x1[i]*0.54)+1.4;
    }

    QCOPGraph *graph1 = customplot->addGraph();
    graph1->setData(x1, y1);
    graph1->setScatterStyle(QCOPScatterStyle::ssCircle, QPen(Qt::black, 1.5), QBrush(QColor(120, 120, 120), 2));
}

connect(customplot, SIGNAL(mouseMove(QMouseEvent*)), this, SLOT(myMouseMove(QMouseEvent*)));
,
```

建立曲线图对象

定义一个槽函数，这个槽函数就是获取鼠标时时刻刻的坐标值

创建曲线这个应该都会了

重点在这里，`SIGNAL(mouseMove(QMouseEvent*))`这个信号意思是：鼠标在曲线窗口移动就会不停的发送信号，让槽函数不停的执行

槽函数从 `event` 回调事件获取鼠标在窗口上的坐标值

因为 `widget` 窗口和曲线窗口坐标值单位是不一样的，所以要转换成曲线窗口坐标值

鼠标移动到这些地方就会打印，我鼠标在曲线的 $x=5.12, y=1.8$ 的位置

widget x = 497
widget y = 309
qcustomplot x = 5.12793
qcustomplot y = 1.82796

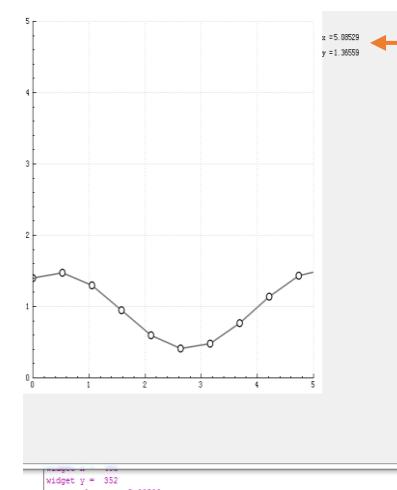
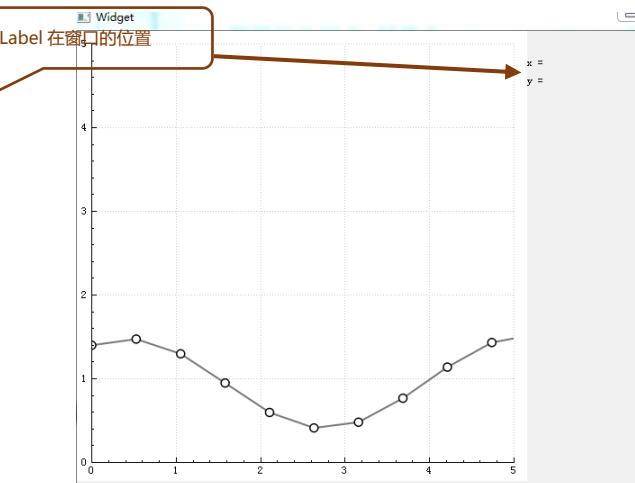
我们用 QLabel 标题框来显示鼠标位置，方便观察。

```
private:  
    Ui::Widget *ui;  
  
    QCustomPlot *customplot;  
    QLabel *labelxstr; //用来显示不动的字符"x ="  
    QLabel *labelxnum; //用来显示实时变化的数字x的值  
    QLabel *labelystr; //用来显示不动的字符"y ="  
    QLabel *labelynum; //用来显示实时变化的数字y的值  
  
private slots:  
    void myMouseMove(QMouseEvent* event);  
};
```

头文件定义 2 个显示字符串和 2 个显示变化数据的 QLabel 这个自己理解

```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
    labelxstr = new QLabel(this);  
    labelxstr->setText("x =");  
    labelxstr->resize(100, 50);  
    labelxstr->move(500, 10);  
  
    labelxnum = new QLabel(this);  
    labelxnum->resize(100, 50);  
    labelxnum->move(520, 10);  
  
    labelystr = new QLabel(this);  
    labelystr->setText("y =");  
    labelystr->resize(100, 50);  
    labelystr->move(500, 30);  
  
    labelynum = new QLabel(this);  
    labelynum->resize(100, 50);  
    labelynum->move(520, 30);  
  
void Widget::myMouseMove(QMouseEvent* event)  
{  
    // 获取鼠标坐标点  
    int x_pos = event->pos().x();  
    int y_pos = event->pos().y();  
  
    // 把鼠标坐标点 转换为 QCustomPlot 内部坐标值 (pixelToCoord 函数)  
    // coordToPixel 函数与之相反 是把内部坐标值 转换为外部坐标点  
    float x_val = customplot->xAxis->pixelToCoord(x_pos);  
    float y_val = customplot->yAxis->pixelToCoord(y_pos);  
  
    // 然后打印在界面上  
    qDebug() << "widget x = " << x_pos; // 打印 widget 窗口鼠标的 x 坐标值  
    qDebug() << "widget y = " << y_pos; // 打印 widget 窗口鼠标的 y 坐标值  
    qDebug() << "qcustomplot x = " << x_val; // 打印 qcustomplot 窗口鼠标的 x 坐标值  
    qDebug() << "qcustomplot y = " << y_val; // 打印 qcustomplot 窗口鼠标的 y 坐标值  
  
    labelxnum->setNum(x_val);  
    labelynum->setNum(y_val);  
}
```

在鼠标移动槽函数里面使用上面定义了的 labelxnum 和 labelynum 对象的 setNum 函数实时更新 x 和 y 的坐标，QLabel::text 是显示不动的字符串，QLabel::setNum 是显示实时动的数值



我们其实可以将鼠标 x 轴和 y 轴的数字放在鼠标箭头上跟着鼠标移动显示

1. 箭头设计用 QCPItemLine 类，这是画线类

```
QCPItemLine *arrow; // 箭头 定义线对象
```

```
QCPGraph *graph1 = customplot->addGraph();
graph1->setData(x1, y1);
graph1->setScatterStyle(QCPSscatterStyle(QCPSscatterStyle));
graph1->setPen(QPen(QColor(120, 120, 120), 2));

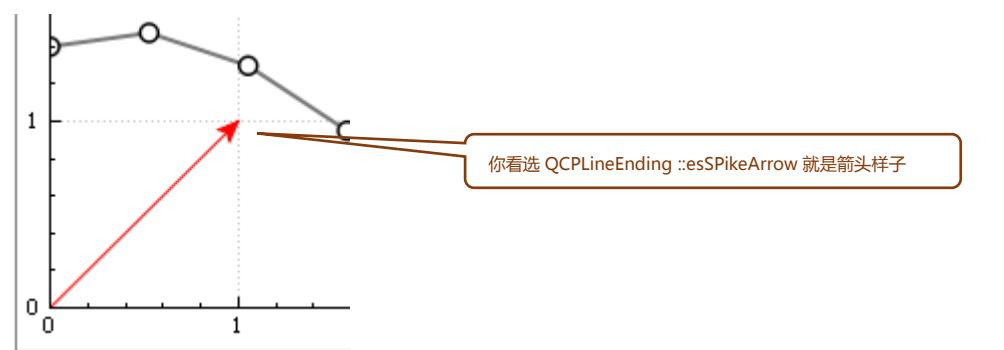
arrow = new QCPItemLine(customplot);
arrow->setPen(QColor(Qt::red)); // 设置箭头的颜色
```



```
arrow = new QCPItemLine(customplot);
arrow->setPen(QColor(Qt::red)); // 设置箭头的颜色
arrow->setHead(QCPLineEnding::esSpikeArrow);
```

添加 QCPItemLine::setHead 函数才是添加箭头

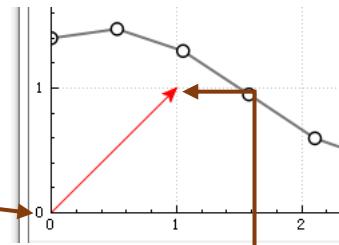
QCPLineEnding.....是选择箭头样子，有箭头型，圆形



```
arrow = new QCPItemLine(customplot);
arrow->setPen(QColor(Qt::red)); // 设置箭头的颜色
arrow->setHead(QCPLineEnding::esSpikeArrow);
arrow->start->setCoords(0, 0);
arrow->end->setCoords(1, 1);
```

start->setCoords 是选择箭头线 xy 的开始坐标

end->setCoords 是选择箭头线 xy 的结束坐标



下面我们让箭头跟着鼠标移动起来

```
arrow = new QCPIItemLine(customplot);
arrow->setPen(QColor(Qt::red)); //设置箭头的颜色
arrow->setHead(QCPLineEnding::esSpikeArrow);
```

箭头初始化函数不变

```
void Widget::myMouseMove(QMouseEvent* event)
{
    //获取鼠标坐标点
    int x_pos = event->pos().x();
    int y_pos = event->pos().y();

    // 把鼠标坐标点 转换为 QCustomPlot 内部坐标值 (pixelToCoord 函数)
    // coordToPixel 函数与之相反 是把内部坐标值 转换为外部坐标点
    float x_val = customplot->xAxis->pixelToCoord(x_pos);
    float y_val = customplot->yAxis->pixelToCoord(y_pos);

    // 然后打印在界面上
    qDebug() << "widget x = " << x_pos; //打印widget窗口鼠标的x坐标值
    qDebug() << "widget y = " << y_pos; //打印widget窗口鼠标的y坐标值
    qDebug() << "qcustomplot x = " << x_val; //打印曲线窗口鼠标的x坐标值
    qDebug() << "qcustomplot y = " << y_val; //打印曲线窗口鼠标的y坐标值

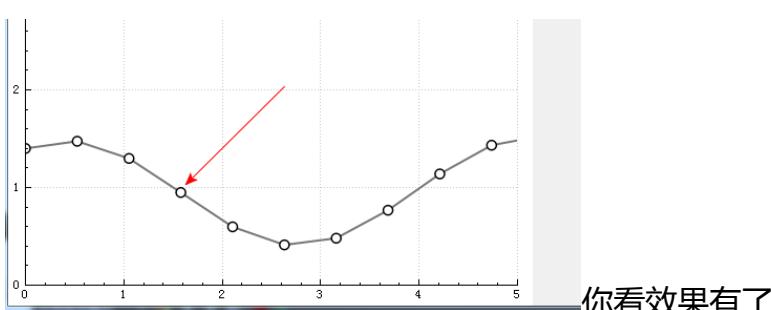
    // labelxnum->setNum(x_val);
    // labelynum->setNum(y_val);

    arrow->start->setCoords(x_val+1, y_val+1);
    arrow->end->setCoords(x_val, y_val);

    customplot->replot();
```

我们只需要在槽函数
里面不停的赋值鼠标
数据给箭头的
setCoords 更新函数，
箭头不停修改开始绘
制和结束绘制的坐标
值就行了

这句必须加，因为这是不停刷新曲线窗
口的函数，如果不加箭头不会移动



你看效果有了

下面我们让箭头只随着曲线数据点移动

```

double dx1[10];
double dy1[10];

private slots:
    void myMouseMove(QMouseEvent* event);
};

customplot = new QCustomPlot(this);
customplot->resize(500,500);

QVector<double>x1(10),y1(10);

for (int i=0; i<10; ++i)
{
    dx1[i] = i/(double)(10-1)*10;
    dy1[i] = qCos(dx1[i]*0.8+qSin(dx1[i]*0.16+1.0))*qSin(dx1[i]*0.54)
}

for(int i=0;i<10;i++)
{
    x1[i] = dx1[i];
    y1[i] = dy1[i];
}

QCPGraph *graph1 = customplot->addGraph();
graph1->setData(x1, y1);
graph1->setScatterStyle(QCPScatterStyle::ssCircle,
graph1->setPen(QPen(QColor(120, 120, 120), 2));

arrow = new QCPIItemLine(customplot);
arrow->setPen(QColor(Qt::red)); //设置箭头的颜色
arrow->setHead(QCPLineEnding::esSpikeArrow);

```

因为槽函数里面 QCPIItemLine::end->setCoords 函数要使用曲线的数据点，但是 QVector 头文件定义无法使用，我就用两个数组做中转

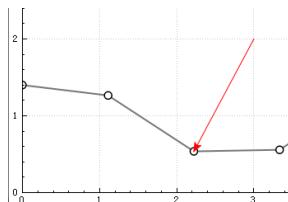
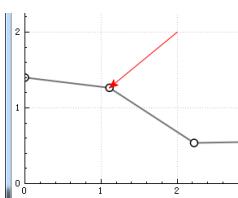
主函数 CPP 文件用中转数组 dx1 , dy1 生成正弦曲线

因为曲线要用 QVector 类型的变量显示，所以把中转数组赋值给 QVector 数组变量显示曲线

这些是老套路不变

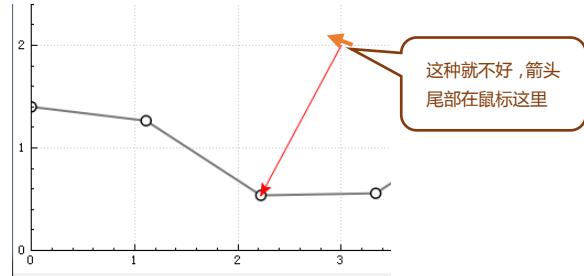
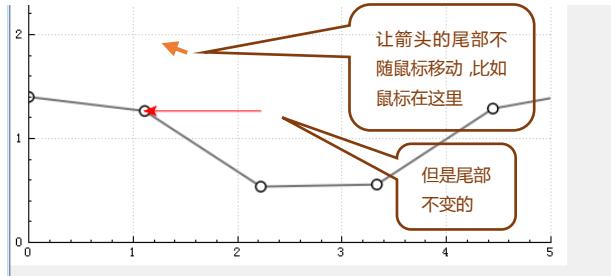
把曲线图的鼠标坐标变成 int 型，因为下面 setCoords 用的是 int 数组来寻找结束坐标点

中转数组就是因为有类成员全局访问的功能，所以赋值给槽函数的箭头结束坐标函数



你看箭头结束坐标就根据鼠标
的移动对准不同的曲线点

如何让箭头的尾巴不随着鼠标移动



其实很简单，只需要修改鼠标事件槽函数

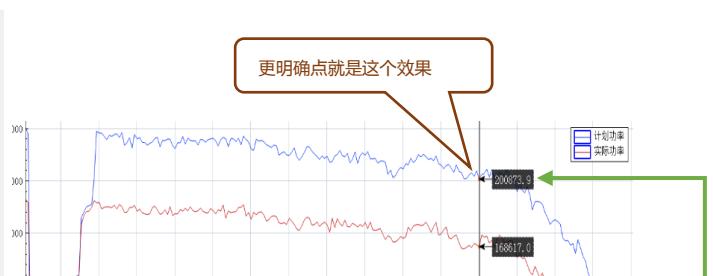
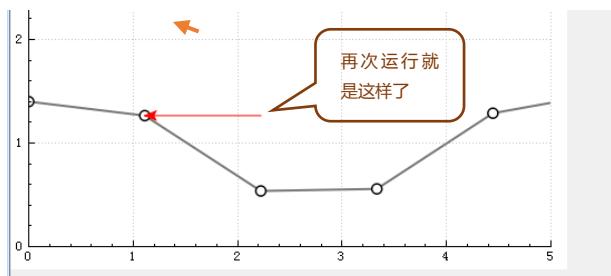
```
void Widget::myMouseMove(QMouseEvent* event)
{
    // 获取鼠标坐标点
    int x_pos = event->pos().x();
    int y_pos = event->pos().y();

    // 把鼠标坐标点 转换为 QCustomPlot 内部坐标值 (pixelToCoord 函数)
    // coordToPixel 函数与之相反 是把内部坐标值 转换为外部坐标点
    int x_val = customplot->xAxis->pixelToCoord(x_pos);
    int y_val = customplot->yAxis->pixelToCoord(y_pos);

    // 然后打印在界面上
    qDebug() << "widget x = " << x_pos; // 打印 widget 窗口鼠标的 x 坐标值
    qDebug() << "widget y = " << y_pos; // 打印 widget 窗口鼠标的 y 坐标值
    qDebug() << "qcustomplot x = " << x_val; // 打印曲线窗口鼠标的 x 坐标值
    qDebug() << "qcustomplot y = " << y_val; // 打印曲线窗口鼠标的 y 坐标值

    //     labelxnum->setNum(x_val);
    //     labelynum->setNum(y_val);
    arrow->start->setCoords(dx1[x_val+1], dy1[x_val]);
    arrow->end->setCoords(dx1[x_val], dy1[x_val]);

    customplot->replot();
```



然后箭头后面的数据点要用 QCPIItemText，然后修改 QCPIItemText 的显示坐标，不能用 QLabel 因为这是在曲线图里面显示字符，所以必须用和 QCustomplot 的相关类

QCPIItemText 类使用方法

这里用 QCPIItemText 是为了完善上一页的内容

```
private:  
    Ui::Widget *ui;  
  
    QCustomPlot *customplot;  
  
    QCPIItemLine *arrow;      // 箭头  
    QCPIItemTracer *tracer;  
  
    QCPIItemText *CPIlabel;   // 显示的数值或字符
```

在头文件定义个曲线框里面显示的字符类

```
for(int i=0;i<10;i++)  
{  
    x1[i] = dx1[i];  
    y1[i] = dy1[i];  
}  
QCPGraph *graph1 = customplot->addGraph();  
graph1->setData(x1, y1);  
graph1->setScatterStyle(QCPScatterStyle::ssCircle);  
graph1->setPen(QPen(QColor(120, 120, 120), 2));  
  
arrow = new QCPIItemLine(customplot);  
arrow->setPen(QColor(Qt::red)); // 设置箭头的颜色  
arrow->setHead(QCPLineEnding::esSpikeArrow);
```

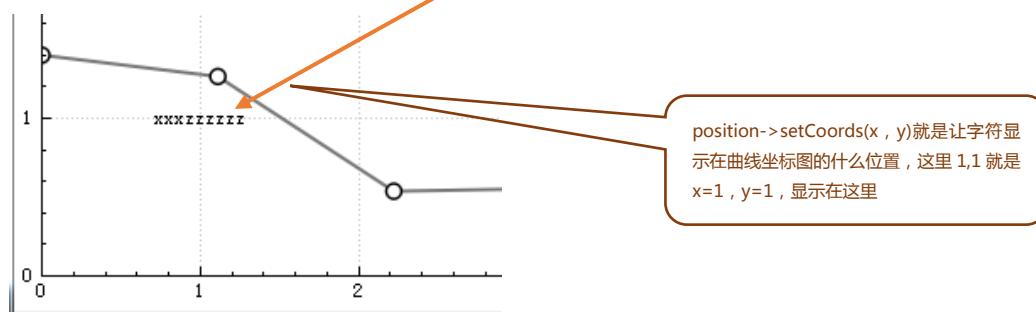
曲线和箭头上一页已经做好了

```
CPIlabel = new QCPIItemText(customplot);  
CPIlabel->setText("xxxxzzzz");  
CPIlabel->position->setCoords(1, 1);
```

这里主要添加字符显示，把字符嵌入曲线图对象

```
connect(customplot, SIGNAL(mouseMove(QMouseEvent*)), this, SLOT(mouseMove(QMouseEvent*)))
```

字符显示的位置



下面我们要让字符显示曲线点实时数据，还要让字符显示在箭头后面

```

QCPGraph *graph1 = customplot->addGraph();
graph1->setData(x1, y1);
graph1->setScatterStyle(QCPS scatte rStyle(QCPS scatterSt
graph1->setPen(QPen(QColor(120, 120, 120), 2));

arrow = new QCPItemLine(customplot);
arrow->setPen(QPen(Qt::red)); //设置箭头的颜色
arrow->setHead(QCPLineEnding::esSpikeArrow);

```

```

CPIlabel = new QCPItemText(customplot);
CPIlabel->setPen(QPen(Qt::red));

```

把 setText 和 setCoords 这两个东西放到槽函数里面去 ,因为字符显示和字符坐标是实时变化的 ,初始化这里加个字符框颜色

```
connect(customplot, SIGNAL(mouseMove(QMouseEvent*)),
```

```

void Widget::myMouseMove(QMouseEvent* event)
{
    //获取鼠标坐标点
    int x_pos = event->pos().x();
    int y_pos = event->pos().y();

    // 把鼠标坐标点 转换为 QCustomPlot 内部坐标值 (pixelToCoord 函数)
    // coordToPixel 函数与之相反 是把内部坐标值 转换为外部坐标点
    int x_val = customplot->xAxis->pixelToCoord(x_pos);
    int y_val = customplot->yAxis->pixelToCoord(y_pos);

    // 然后打印在界面上
    qDebug() << "widget x = " << x_pos; //打印widget窗口鼠标的x坐标值
    qDebug() << "widget y = " << y_pos; //打印widget窗口鼠标的y坐标值
    qDebug() << "qcustomplot x = " << x_val; //打印曲线窗口鼠标的x坐标值
    qDebug() << "qcustomplot y = " << y_val; //打印曲线窗口鼠标的y坐标值

    arrow->start->setCoords(dx1[x_val+1], dy1[x_val]);
    arrow->end->setCoords(dx1[x_val], dy1[x_val]);

    CPIlabel->position->setCoords(dx1[x_val+2], dy1[x_val]);
    QString num = QString::number(x_val);
    CPIlabel->setText(num);
}

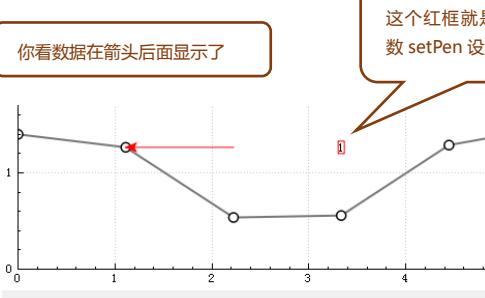
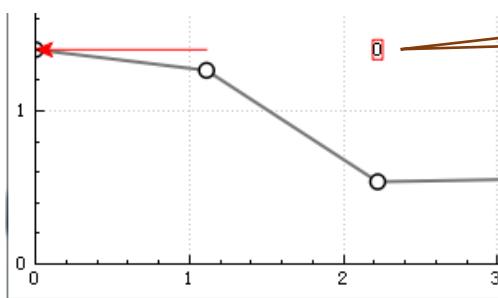
customplot->replot();
}

```

实时显示

因为 QCPItemText 只能显示 QString 类型字符 所以这里的曲线点值要从 int 转换成 QString

这就是字符显示的坐标 , 和箭头一样的位置 , 只是放在箭头后边 , 所以这里值比箭头开始坐标要大一点



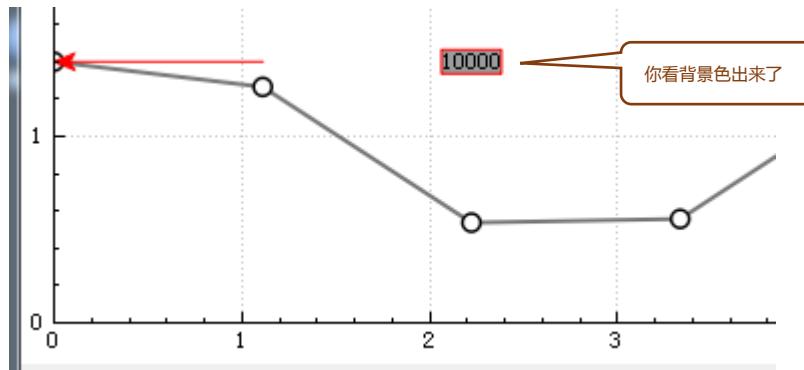
我还可以给红框加深一点的背景色

```
CPIlabel = new QCPIItemText(customplot);
CPIlabel->setPen(QPen(Qt::red));
CPIlabel->setBrush(QBrush(QColor(0, 0, 0, 100)));
```

在初始化函数中加入 setBrush 背景色填充，颜色是 0, 0, 0 就是黑色，透明度 100，就是不怎么透明

```
CPIlabel->position->setCoords(dx1[x_val+2], dy1[x_val]);
QString num = QString::number(10000);
CPIlabel->setText(num);
```

为了看清楚效果，我在槽函数写了个固定值



在曲线图中实现移动十字线功能

QCPIItemStraightLine 使用直线类

```
private:
    Ui::Widget *ui;
    QCustomPlot *customplot;
    QCPIItemLine *arrow; // 箭头
    QCPIItemTracer *tracer;
    QCPIItemText *CPIlabel; // 显示的数值或字符
    QCPIItemStraightLine *m_lineV; // 垂直线
    QCPIItemStraightLine *m_lineH; // 水平线
```

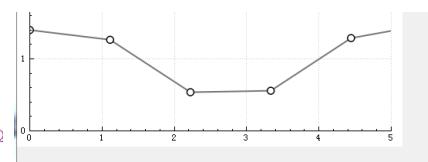
在头文件定义
垂直线和水平
线对象

```
customplot = new QCustomPlot(this);
customplot->resize(500,500);
QVector<double>x1(10),y1(10);

for (int i=0; i<10; ++i)
{
    dx1[i] = i/(double)(10-1)*10;
    dy1[i] = qCos(dx1[i]*0.8+qSin(dx1[i]*0.16+1.0))*qSin(dx1[i]*0.54)+1.4;
}

for(int i=0;i<10;i++)
{
    x1[i] = dx1[i];
    y1[i] = dy1[i];
}
QCPGraph *graph1 = customplot->addGraph();
graph1->setData(x1, y1);
graph1->setScatterStyle(QCPScatterStyle::ssCircle, QPen(QColor(120, 120, 120), 2));
```

构造函数初始
化生成曲线



下面给这个曲线图加垂直直线

```
m_lineV = new QCPIItemStraightLine(customplot); //垂直线  
m_lineV->setLayer("overlay");  
m_lineV->setPen(QPen(Qt::blue)); //垂直线颜色  
m_lineV->setClipToAxisRect(true);  
m_lineV->point1->setCoords(1, 1); //这是第1个点  
m_lineV->point2->setCoords(2, 1); //这是第2个点
```

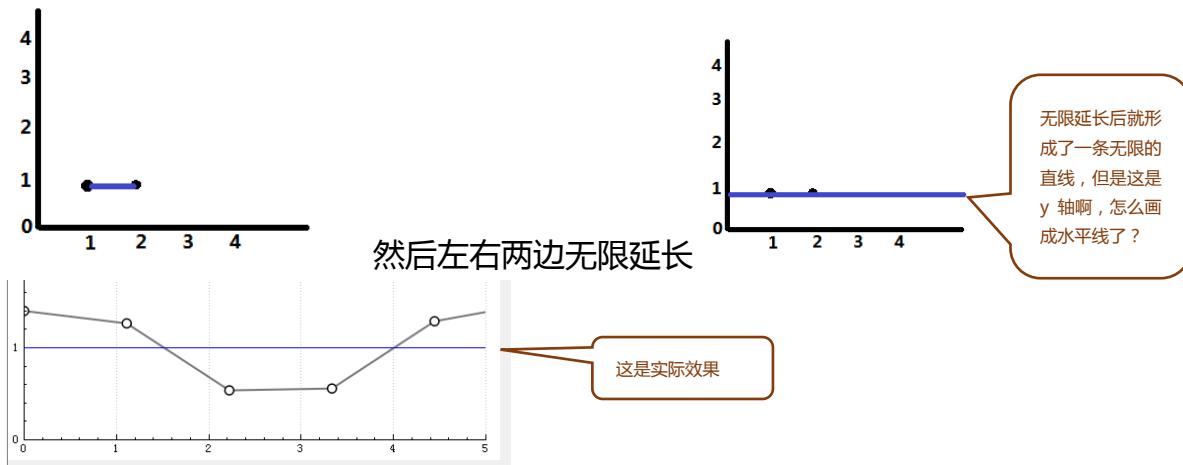
这两个点是什么意思？

记住：两点决定一条直线前后方向

setCoords (x,y) //x横坐标，y纵坐标，一个x和y决定一个点



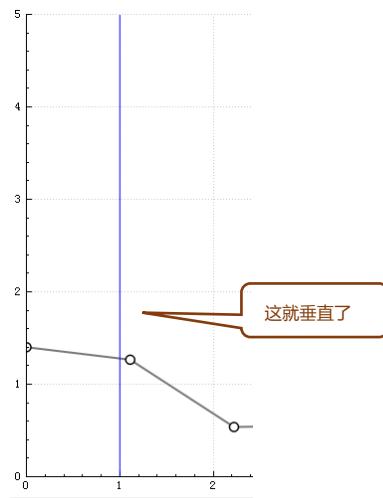
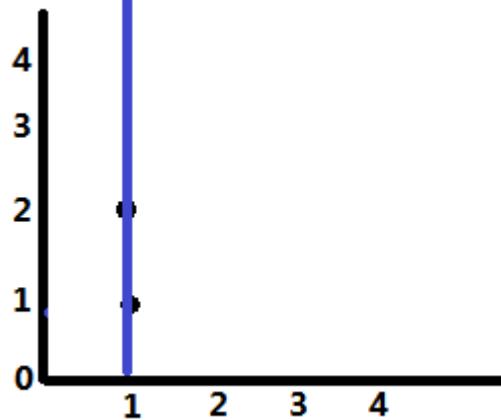
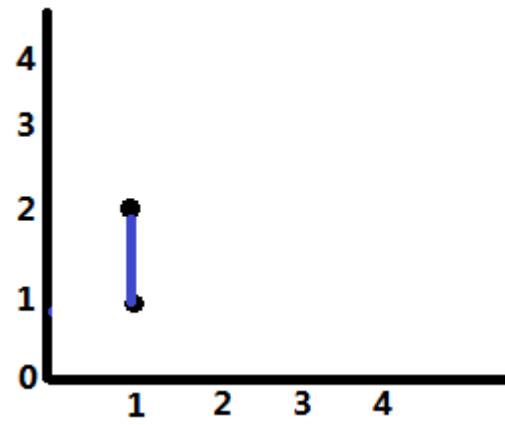
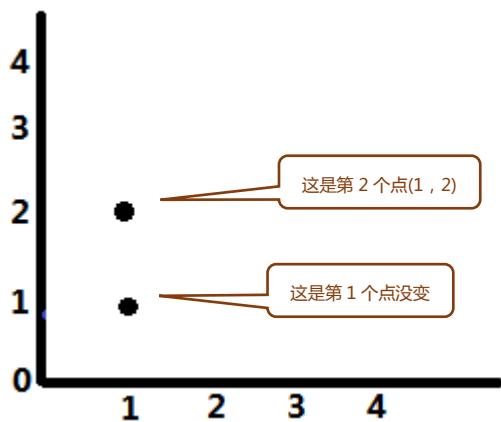
系统会自动连接着两个点



下面我们来修改下

```
m_lineV = new QCPIItemStraightLine(customplot); //垂直线  
m_lineV->setLayer("overlay");  
m_lineV->setPen(QPen(Qt::blue)); //垂直线颜色  
m_lineV->setClipToAxisRect(true);  
m_lineV->point1->setCoords(1, 1); //这是第1个点  
m_lineV->point2->setCoords(1, 2); //这是第2个点
```

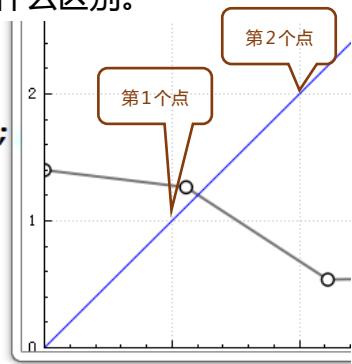
我把(2, 1)改成了(1, 2)



```
QCPIItemStraightLine *m_lineV; //垂直线  
QCPIItemStraightLine *m_lineH; //水平线
```

所以头文件的水平线和垂直线都是一个类，因为都是直线，所以没什么区别。

```
m_lineV = new QCPIItemStraightLine(customplot);  
m_lineV->setLayer("overlay");  
m_lineV->setPen(QPen(Qt::blue)); //垂直线颜色  
m_lineV->setClipToAxisRect(true);  
m_lineV->point1->setCoords(1, 1); //这是第1个点  
m_lineV->point2->setCoords(2, 2); //这是第2个点
```



这样就成了斜线，所以看自己怎么画。

下面实现水平和垂直的十字线实时移动

```

m_lineV = new QCPIItemStraightLine(customplot); //垂直线
m_lineV->setLayer("overlay");
m_lineV->setPen(QPen(Qt::blue)); //垂直线颜色
m_lineV->setClipToAxisRect(true);

m_lineH = new QCPIItemStraightLine(customplot); //水平线
m_lineH->setLayer("overlay");
m_lineH->setPen(QPen(Qt::blue)); //垂直线颜色
m_lineH->setClipToAxisRect(true);

connect(customplot, SIGNAL(mouseMove(QMouseEvent*)), this, SLOT(myMouseMove(QMouseEvent*)));
customplot->setInteractions(QCP::iRangeDrag|QCP::iRangeZoom);
}

void Widget::myMouseMove(QMouseEvent* event)
{
    //获取鼠标坐标点
    int x_pos = event->pos().x();
    int y_pos = event->pos().y();
    // 把鼠标坐标点 转换为 QCustomPlot 内部坐标值 (pixelToCoord 函数)
    // coordToPixel 函数与之相反 是把内部坐标值 转换为外部坐标点
    int x_val = customplot->xAxis->pixelToCoord(x_pos);
    int y_val = customplot->yAxis->pixelToCoord(y_pos);

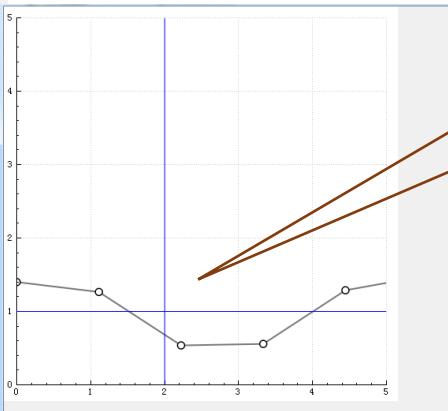
    // 然后打印在界面上
    qDebug() << "widget x = " << x_pos; //打印widget窗口鼠标的x坐标值
    qDebug() << "widget y = " << y_pos; //打印widget窗口鼠标的y坐标值
    qDebug() << "qcustomplot x = " << x_val; //打印曲线窗口鼠标的x坐标值
    qDebug() << "qcustomplot y = " << y_val; //打印曲线窗口鼠标的y坐标值

    m_lineV->point1->setCoords(x_val, y_val); //这是第1个点
    m_lineV->point2->setCoords(x_val, y_val+1); //这是第2个点

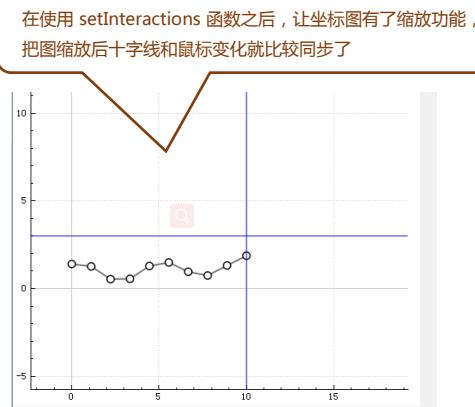
    m_lineH->point1->setCoords(x_val, y_val); //这是第1个点
    m_lineH->point2->setCoords(x_val+1, y_val); //这是第2个点

    customplot->replot();
}

```

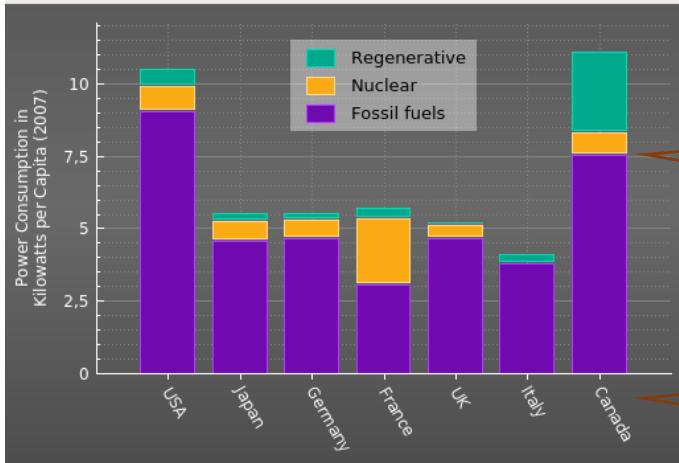


在窗口很大的情况下，坐标移动速度要慢与鼠标，所以看起来十字线要反应迟钝些



在使用 setInteractions 函数之后，让坐标图有了缩放功能，把图缩放后十字线和鼠标变化就比较同步了

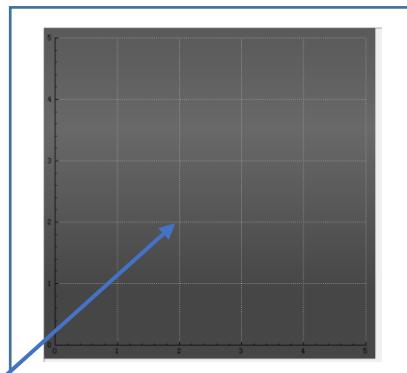
柱状图叠加图形设计



这就是叠加图形实现，
一条柱状上面加另一条

X 坐标用来显示每个数
据柱状图的名字

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



```
// set dark background gradient:  
QLinearGradient gradient(0, 0, 0, 400);  
gradient.setColorAt(0, QColor(90, 90, 90));  
gradient.setColorAt(0.38, QColor(105, 105, 105));  
gradient.setColorAt(1, QColor(70, 70, 70));  
customplot->setBackground(QBrush(gradient));
```

设置坐标图背景颜色

```
// create empty bar chart objects:  
QCPBars *fossil = new QCPBars(customplot->xAxis, customplot->yAxis);
```

创建一个图表对象

```
// set names and colors:  
fossil->setName("Fossil fuels");  
fossil->setPen(QPen(QColor(111, 9, 176).lighter(170)));  
fossil->setBrush(QColor(111, 9, 176));
```

设置柱状条为紫色，写上该柱状条名称

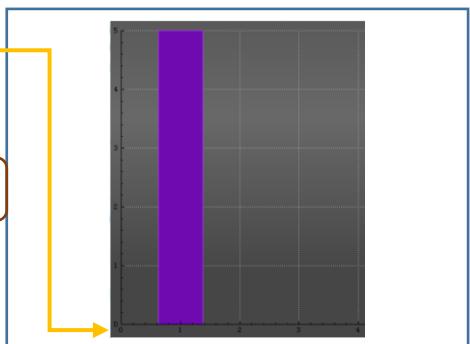
```
 QVector<double> ticks;  
ticks << 1 << 2 << 3 << 4 << 5 << 6 << 7;
```

设置柱状条 x 坐标的位置

```
 QVector<double> fossilData;  
fossilData << 5;  
fossil->setData(ticks, fossilData);
```

给 double 数组放数据 5

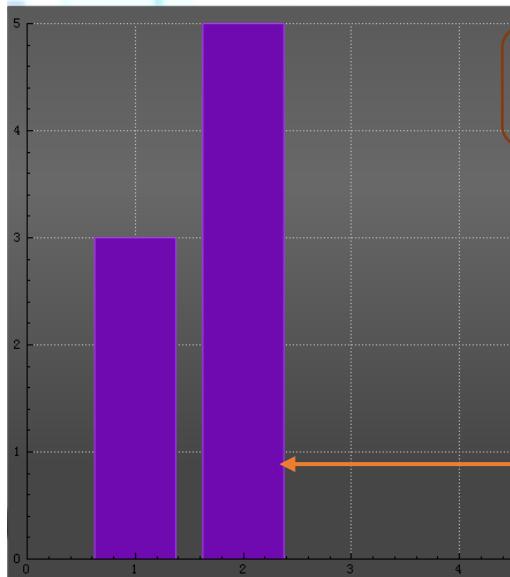
把 double 数据显示在 x 坐标第 1 个位置



为什么数据 5 是现在在第 1 个 x 坐标的位置呢？

```
QVector<double> ticks;  
ticks << 1 << 2 << 3 << 4 << 5 << 6 << 7;
```

```
QVector<double> fossilData;  
fossilData << 3 << 5;  
fossil->setData(ticks, fossilData);
```



我们图表对象用 setData 将 ticks 变量加载进来，那么 x 的位置就有了 7 个，也就是 ticks 定义的 <<1<<2...，这种<<就是定义 x 有多少个坐标位置

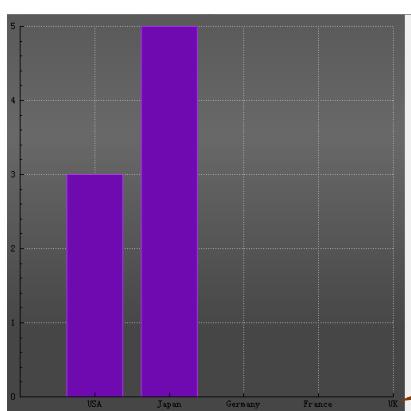
这种好处数我数据也只需要用“<<”符号就可以增加一个柱状条

```
// prepare x axis with country labels:
```

```
QVector<QString> labels;  
labels << "USA" << "Japan" << "Germany" << "France" << "UK" << "Italy" << "Canada";  
  
QSharedPointer<QCPAxisTickerText> textTicker(new QCPAxisTickerText);  
textTicker->addTicks(ticks, labels);  
customplot->xAxis->setTicker(textTicker);
```

然后将 ticks 的数字与 labels 的名字排列一一对应起来

将排列好的 x 坐标名字写入 x 轴

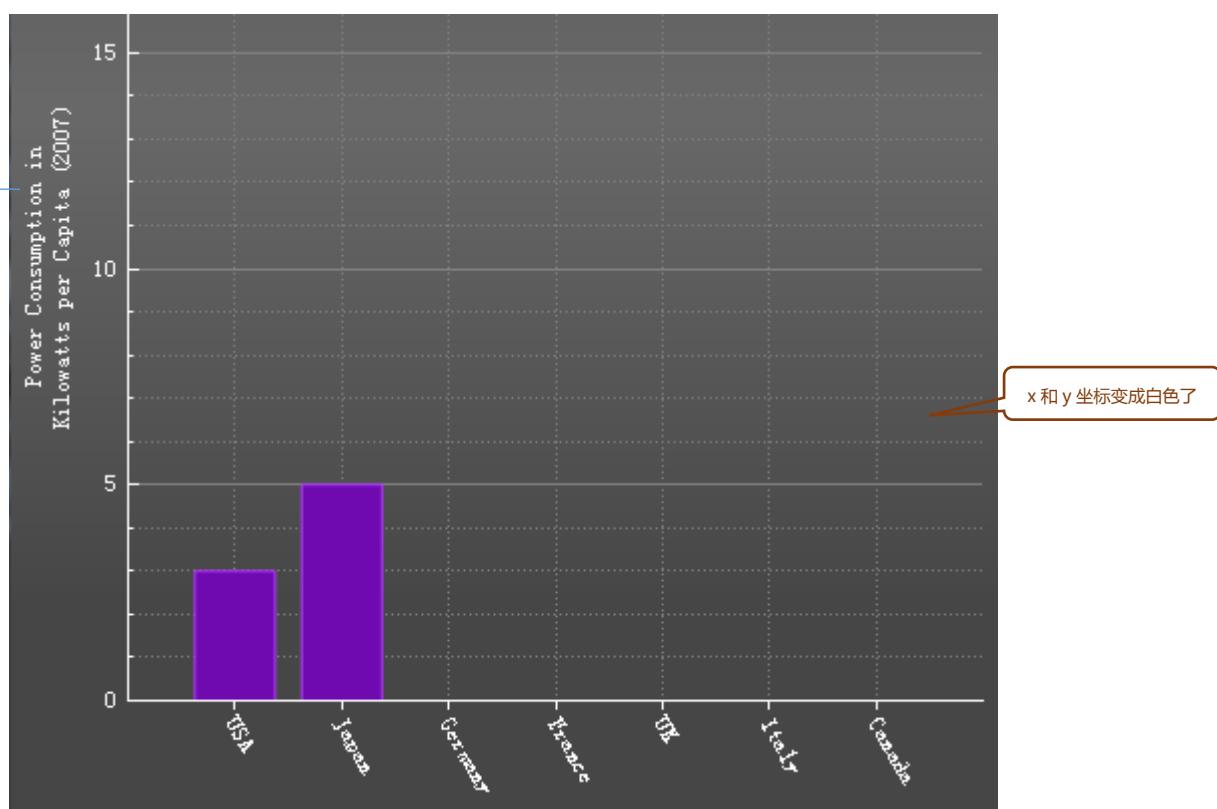


名字有了，用名字代替了 x 轴点的数据

我感觉 x 轴和 y 轴颜色太黑了，下面修改 x 轴和 y 轴颜色

```
customplot->xAxis->setTickLabelRotation(60);
customplot->xAxis->setSubTicks(false);
customplot->xAxis->setTickLength(0, 4);
customplot->xAxis->setRange(0, 8);
customplot->xAxis->setBasePen(QPen(Qt::white));
customplot->xAxis->setTickPen(QPen(Qt::white));
customplot->xAxis->grid()->setVisible(true);
customplot->xAxis->grid()->setPen(QPen(QColor(130, 130, 130), 0, Qt::DotLine));
customplot->xAxis->setTickLabelColor(Qt::white);
customplot->xAxis->setLabelColor(Qt::white);

// prepare y axis:
customplot->yAxis->setRange(0, 20);
customplot->yAxis->setPadding(5); // a bit more space to the left border
customplot->yAxis->setLabel("Power Consumption in\nKilowatts per Capita (2007)");
customplot->yAxis->setBasePen(QPen(Qt::white));
customplot->yAxis->setTickPen(QPen(Qt::white));
customplot->yAxis->setSubTickPen(QPen(Qt::white));
customplot->yAxis->grid()->setSubGridVisible(true);
customplot->yAxis->setTickLabelColor(Qt::white);
customplot->yAxis->setLabelColor(Qt::white);
customplot->yAxis->grid()->setPen(QPen(QColor(130, 130, 130), 0, Qt::SolidLine));
customplot->yAxis->grid()->setSubGridPen(QPen(QColor(130, 130, 130), 0, Qt::DotLine));
```



下面如何给柱状图头上叠加新的柱状图

```
QCPBars *fossil = new QCPBars(customplot->xAxis, customplot->yAxis);
QCPBars *nuclear = new QCPBars(customplot->xAxis, customplot->yAxis);

nuclear->setAntialiased(false);
fossil->setAntialiased(false);

nuclear->setStackingGap(1);
fossil->setStackingGap(1);
```

照着填这些属性

图标创建，我增加了一个 nuclear 图标，
这个图标会叠加到 fossil 头上

```

// set names and colors:
fossil->setName("Fossil fuels");
fossil->setPen(QPen(QColor(111, 9, 176).lighter(170)));
fossil->setBrush(QColor(111, 9, 176));

nuclear->setName("Nuclear");
nuclear->setPen(QPen(QColor(250, 170, 20).lighter(150)));
nuclear->setBrush(QColor(250, 170, 20));
```

设置 nuclear 柱状图为橙色

```

// stack bars on top of each other:
nuclear->moveAbove(fossil);
```

将 nuclear 柱状图叠加到 fossil 头上

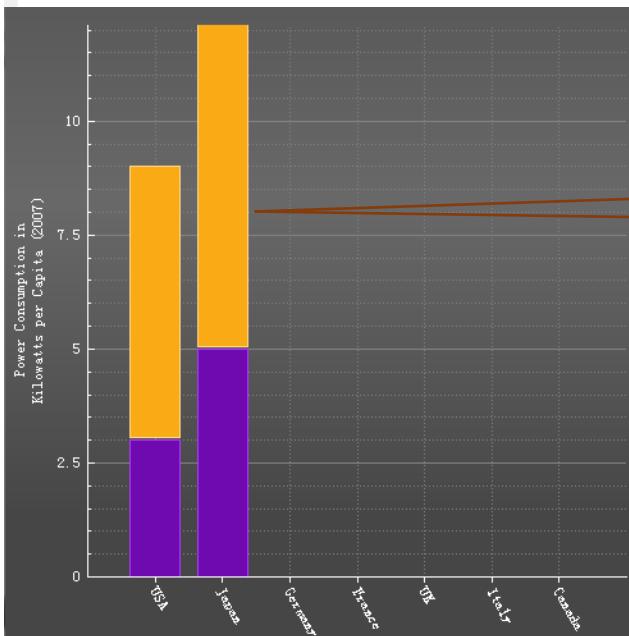
```

// Add data:
QVector<double> fossilData, nuclearData;
fossilData << 3 << 5;
nuclearData << 6 << 10;
```

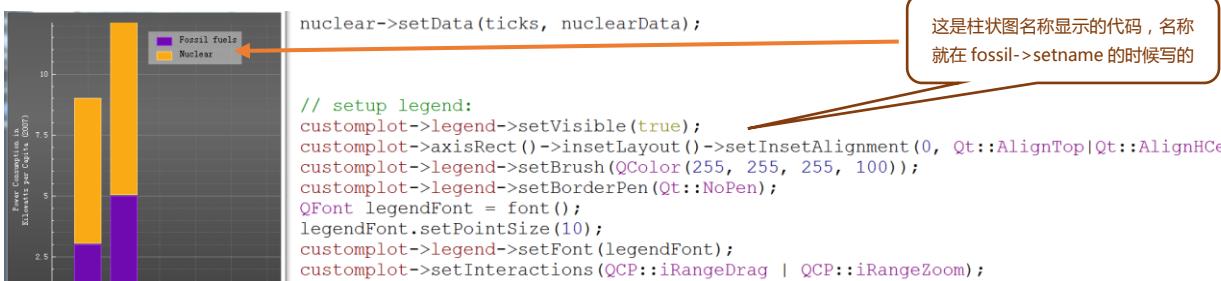
nuclear 的值会在 fossil 柱状图上增加

```

fossil->setData(ticks, fossilData);
nuclear->setData(ticks, nuclearData);
```



这就是叠加的柱状图 nuclear



这是柱状图名称显示的代码，名称就在 fossil->setname 的时候写的

下面为代码展示片段

```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
    customplot = new QCustomPlot(this);  
    customplot->resize(500, 500);  
    // set dark background gradient:  
    QLinearGradient gradient(0, 0, 0, 400);  
    gradient.setColorAt(0, QColor(90, 90, 90));  
    gradient.setColorAt(0.38, QColor(105, 105, 105));  
    gradient.setColorAt(1, QColor(70, 70, 70));  
    customplot->setBackground(QBrush(gradient));  
    // create empty bar chart objects:  
    QCPBars *fossil = new QCPBars(customplot->xAxis, customplot->yAxis);  
    QCPBars *nuclear = new QCPBars(customplot->xAxis, customplot->yAxis);  
    nuclear->setAntialiased(false);  
    fossil->setAntialiased(false);  
    nuclear->setStackingGap(1);  
    fossil->setStackingGap(1);  
    // set names and colors:  
    fossil->setName("Fossil fuels");  
    fossil->setPen(QPen(QColor(111, 9, 176).lighter(170)));  
    fossil->setBrush(QColor(111, 9, 176));  
    nuclear->setName("Nuclear");  
    nuclear->setPen(QPen(QColor(250, 170, 20).lighter(150)));  
    nuclear->setBrush(QColor(250, 170, 20));  
    // stack bars on top of each other:  
    nuclear->moveAbove(fossil);  
    // prepare x axis with country labels:  
    QVector<double> ticks;  
    QVector<QString> labels;  
    ticks << 1 << 2 << 3 << 4 << 5 << 6 << 7;  
    labels << "USA" << "Japan" << "Germany" << "France" << "UK" << "Italy" << "Canada";  
    QSharedPointer<QCPAxisTickerText> textTicker(new QCPAxisTickerText);  
    textTicker->addTicks(ticks, labels);  
    customplot->xAxis->setTicker(textTicker);  
    customplot->xAxis->setTickLabelRotation(60);  
    customplot->xAxis->setSubTicks(false);  
    customplot->xAxis->setTickLength(0, 4);  
    customplot->xAxis->setRange(0, 8);  
    customplot->xAxis->setBasePen(QPen(Qt::white));  
    customplot->xAxis->setTickPen(QPen(Qt::white));
```

```

customplot->xAxis->grid()->setVisible(true);
customplot->xAxis->grid()->setPen(QPen(QColor(130, 130, 130), 0, Qt::DotLine));
customplot->xAxis->setTickLabelColor(Qt::white);
customplot->xAxis->setLabelColor(Qt::white);
// prepare y axis:
customplot->yAxis->setRange(0, 12.1);
customplot->yAxis->setPadding(5); // a bit more space to the left border
customplot->yAxis->setLabel("Power Consumption in\nKilowatts per Capita (2007)");
customplot->yAxis->setBasePen(QPen(Qt::white));
customplot->yAxis->setTickPen(QPen(Qt::white));
customplot->yAxis->setSubTickPen(QPen(Qt::white));
customplot->yAxis->grid()->setSubGridVisible(true);
customplot->yAxis->setTickLabelColor(Qt::white);
customplot->yAxis->setLabelColor(Qt::white);
customplot->yAxis->grid()->setPen(QPen(QColor(130, 130, 130), 0, Qt::SolidLine));
customplot->yAxis->grid()->setSubGridPen(QPen(QColor(130, 130, 130), 0, Qt::DotLine));
// Add data:
QVector<double> fossilData, nuclearData;
fossilData << 3 << 5;
nuclearData << 6 << 10;
fossil->setData(ticks, fossilData);
nuclear->setData(ticks, nuclearData);
// setup legend:
customplot->legend->setVisible(true);
customplot->axisRect()->insetLayout()->setInsetAlignment(0,
Qt::AlignTop|Qt::AlignHCenter);
customplot->legend->setBrush(QColor(255, 255, 255, 100));
customplot->legend->setBorderPen(Qt::NoPen);
QFont legendFont = font();
legendFont.setPointSize(10);
customplot->legend->setFont(legendFont);
customplot->setInteractions(QCP::iRangeDrag | QCP::iRangeZoom);
}

```

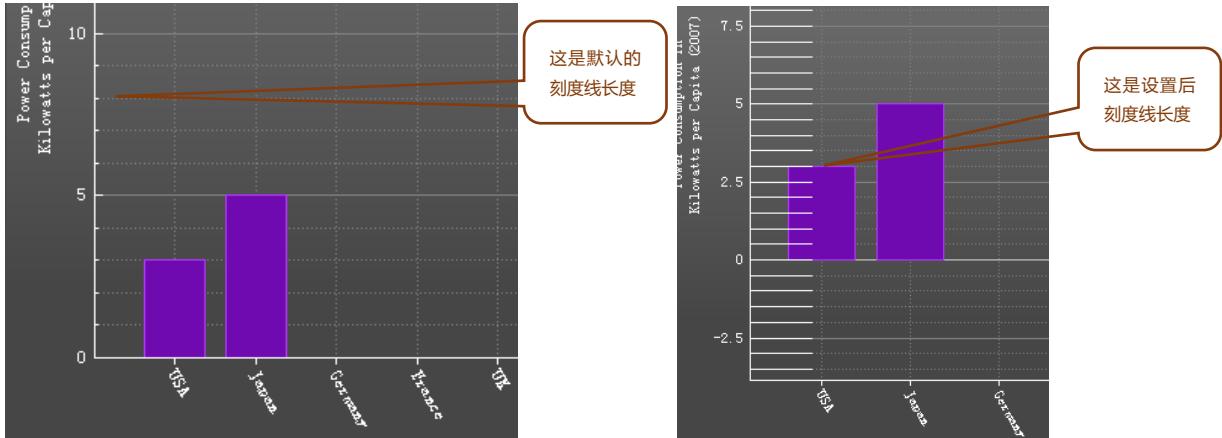
设置坐标轴刻度线，和坐标轴的其它一些属性

```
QCustomPlot *customplot; //定义曲线坐标图对象
```

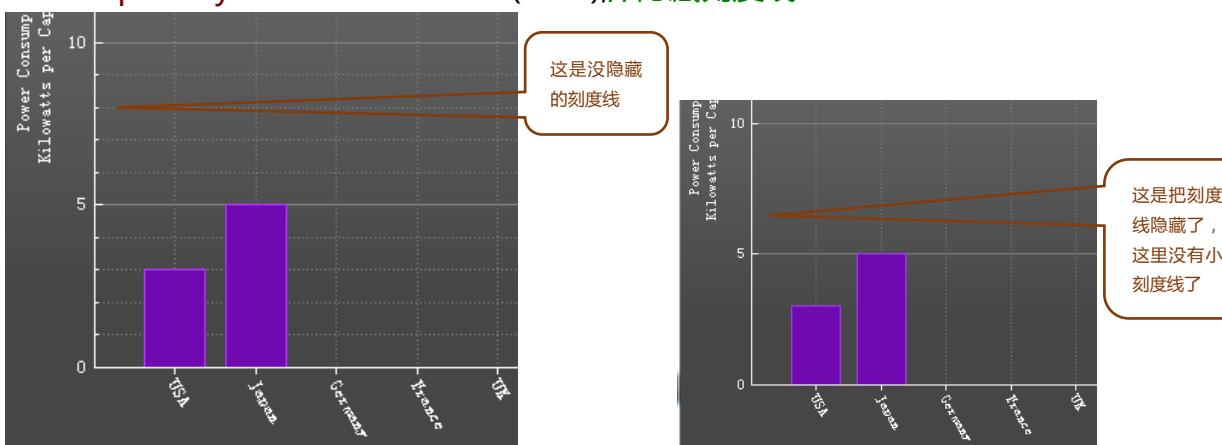
```
customplot = new QCustomPlot(this);
```

```
customplot->yAxis->setTickLength(50); //设置刻度线高度
```

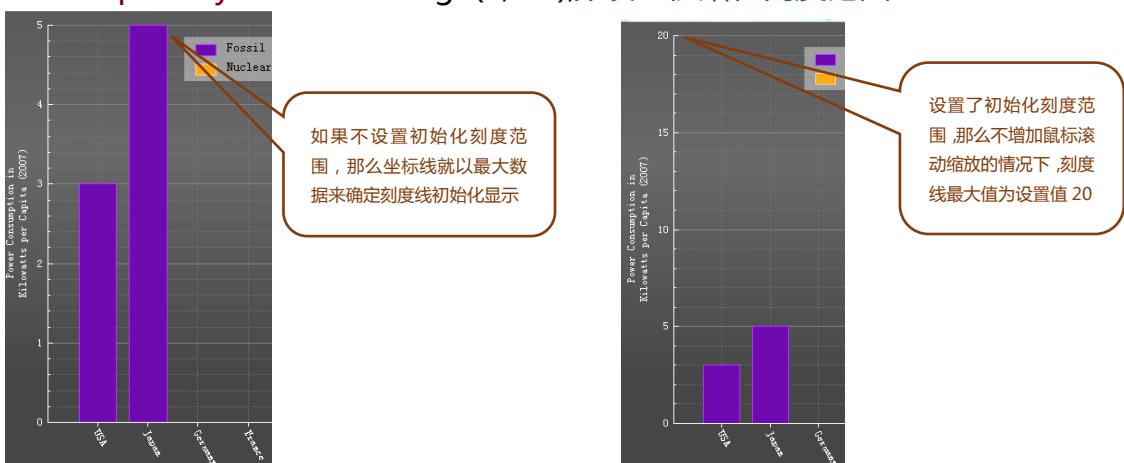
```
customplot->yAxis->setSubTickLengthIn(50); //设置刻度线的长度
```



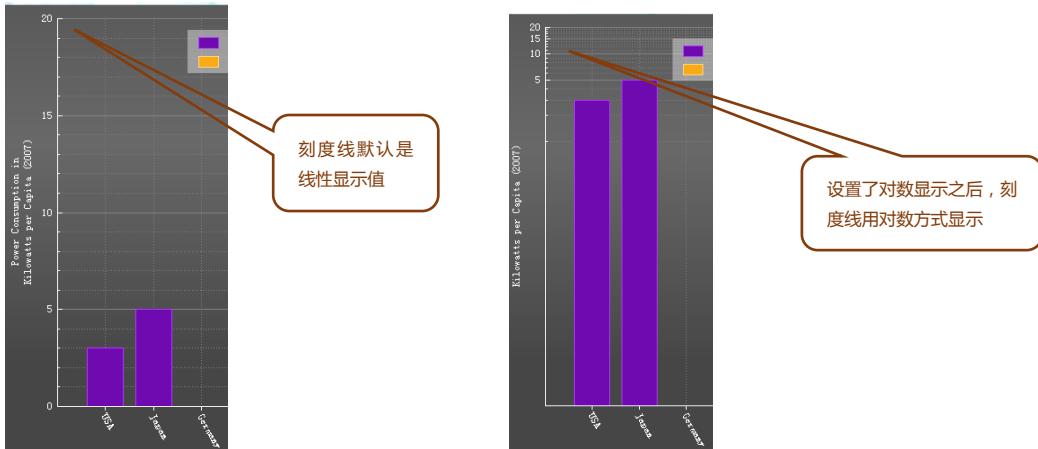
```
customplot->yAxis->setSubTicks(false); //隐藏刻度线
```



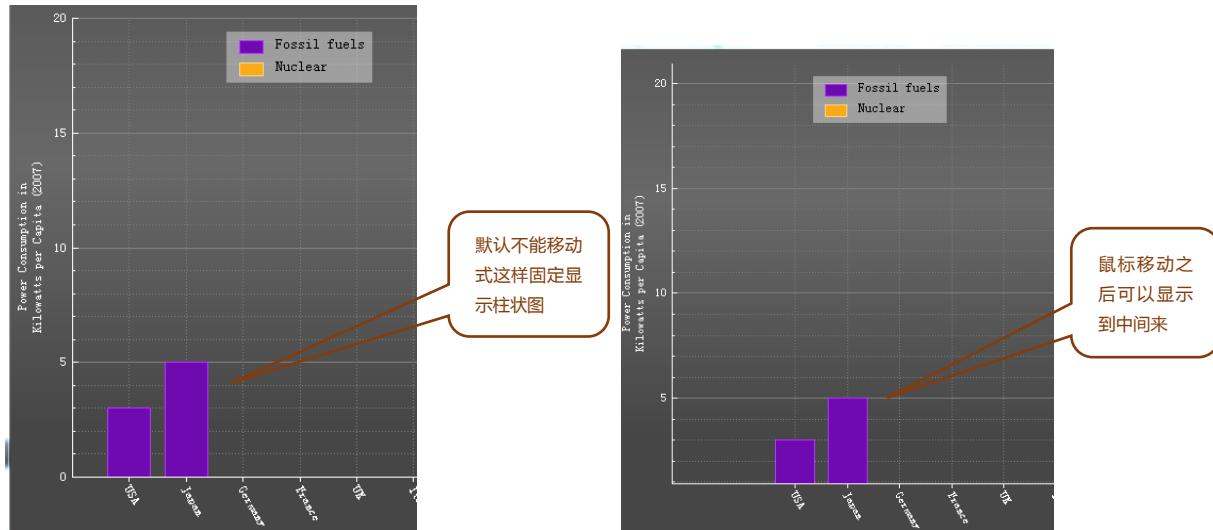
```
customplot->yAxis->setRange(0, 20); //设置初始化刻度范围
```



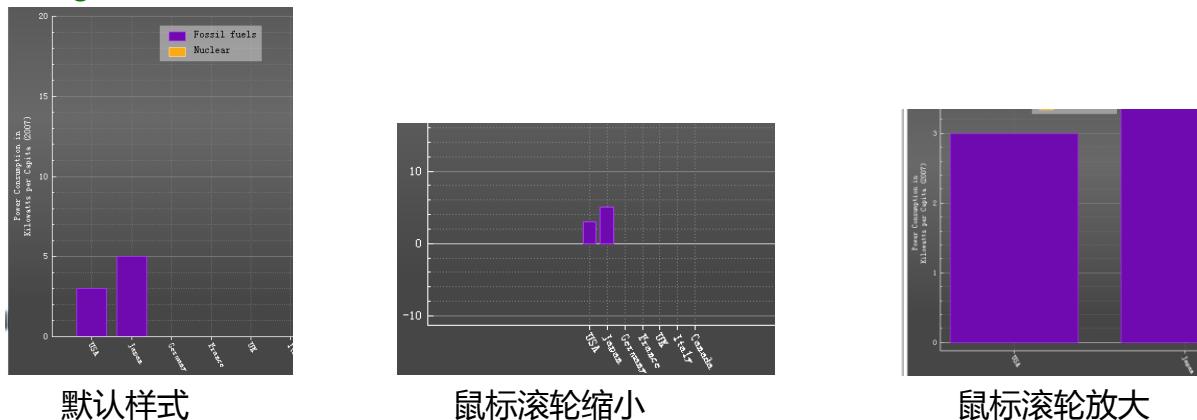
customplot->yAxis->setScaleType(QCPAxis::stLogarithmic); //设置刻度线对数显示



customplot->setInteractions(QCP::iRangeDrag); //只能左右移动坐标轴，不能缩放坐标轴



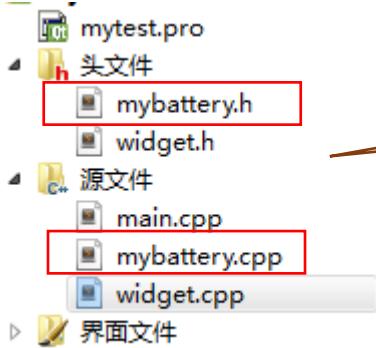
customplot->setInteractions(QCP::iRangeDrag | QCP::iRangeZoom); // 增加 iRangeZoom 可以缩放



电池库使用

myValueControl

解压这个压缩包的文件



加入这两个库文件

```
#include <QWidget>
#include "mybattery.h"
```

包含电池头文件

```
namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

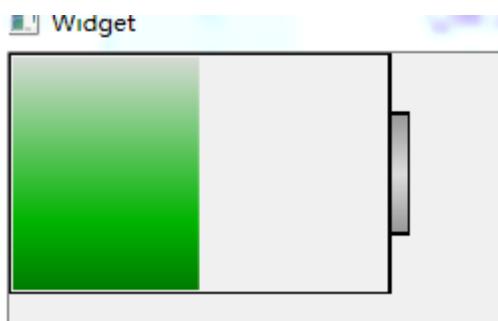
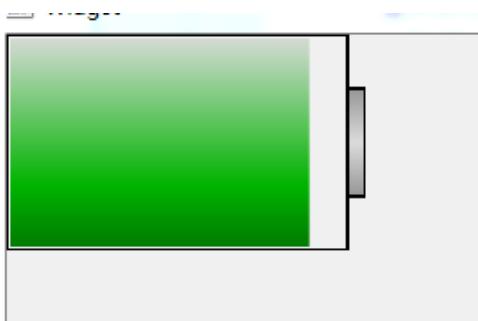
public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
    myBattery *mybattery1;
```

创建电池对象

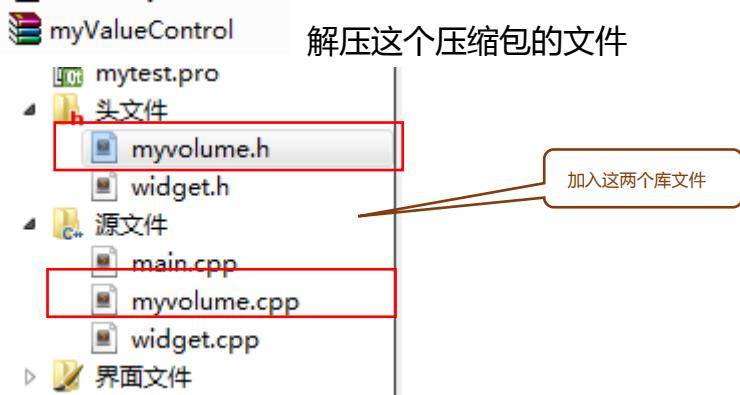
```
Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
    mybattery1 = new myBattery(this);
    mybattery1->setValue(50); //设置电池当前值，记住是int类型值范围0~100
    mybattery1->setVisible(true); //false电池图形隐藏，true电池图形显示
}
```

设置电池值和电池显示



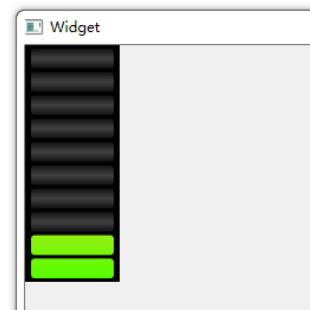
运行之后电池容量会从满格降到 50%，有没有直接运行就显示 50% 的方法呢？那要自己去修改库

音量条显示库使用



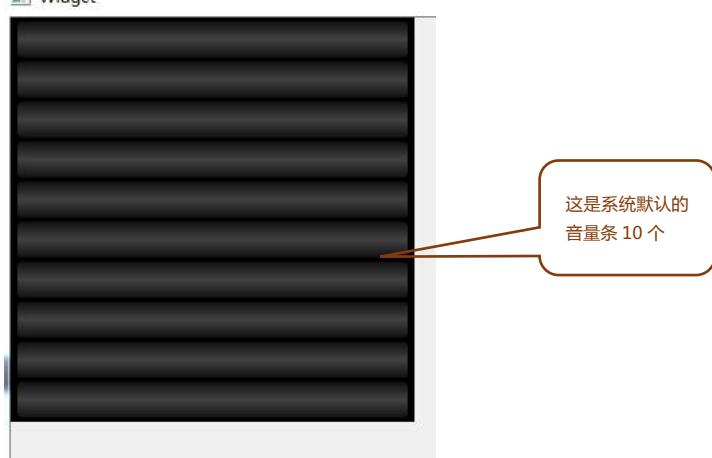
```
#include "myvolume.h"  
包含音量条头文件  
  
namespace Ui {  
class Widget;  
}  
  
class Widget : public QWidget  
{  
    Q_OBJECT  
  
public:  
    explicit Widget(QWidget *parent = 0);  
    ~Widget();  
  
private:  
    Ui::Widget *ui;  
    myVolume *myvolume1; 创建音量条对象
```

```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    myvolume1 = new myVolume(this);  
    myvolume1->setVisible(true); // true 显示, false 不显示  
    myvolume1->setValue(2); // 显示两格颜色  
}
```



至于如何修改音量条大小，和增加音量条格数，去修改库源文件

增加音量条条数



现在我将音量条增加到 20 个

mytest.pro
头文件
myvolume.h
源文件
main.cpp
myvolume.cpp
widget.cpp
界面文件

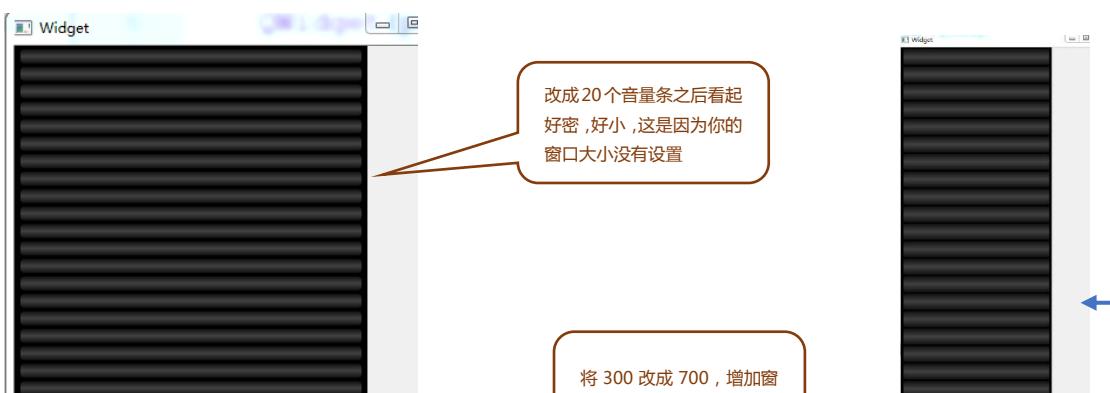
```
1 //  
2 #define MYVOLUME_H  
3  
4 #define PYV_LEFT_SPACE 5  
5 #define PYV_TOP_SPACE 3  
6 #define PYV_TIMER_INTERVAL 500  
7 #define PYV_BAR_COUNT 10  
8 #define PYV_RECT_RADIUS 3  
9 #define PYV_OPACITY 0.4  
10
```

只需要将 myvolume.h 头文件里面的 PYV_BAR_COUNT 10 改成 20 即可

mytest.pro
头文件
myvolume.h
源文件
main.cpp
myvolume.cpp
widget.cpp
界面文件

```
1 //  
2 #define MYVOLUME_H  
3  
4 #define PYV_LEFT_SPACE 5  
5 #define PYV_TOP_SPACE 3  
6 #define PYV_TIMER_INTERVAL 500  
7 #define PYV_BAR_COUNT 20  
8 #define PYV_RECT_RADIUS 3  
9 #define PYV_OPACITY 0.4
```

改成 20 了



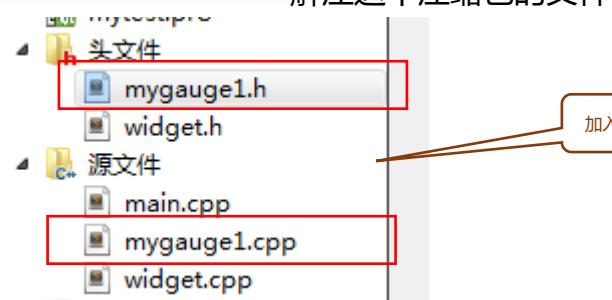
```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
    myvolume1 = new myVolume(this);  
    myvolume1->resize(300,300);
```

将 300 改成 700, 增加窗口高度就解决了

```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
    myvolume1 = new myVolume(this);  
    myvolume1->resize(300,700);
```

圆形计量计库使用

myValueControl 解压这个压缩包的文件



```
#include "mygauge1.h"  
namespace Ui {  
    class Widget;  
}  
  
class Widget : public QWidget  
{  
    Q_OBJECT  
  
public:  
    explicit Widget(QWidget *parent = 0);  
    ~Widget();  
  
private:  
    Ui::Widget *ui;  
    myGauge1 *gauge1;
```

包含 mygauge1 就是计量计头文件

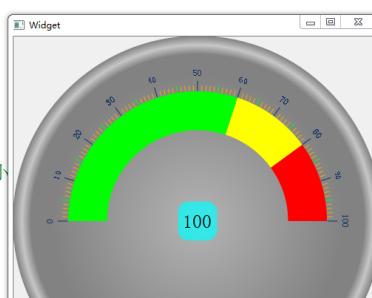
创建计量计对象

```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    gauge1 = new myGauge1(this);  
    gauge1->resize(500,500); //这种圆形计量器大小  
    gauge1->setValue(50); //设置计量数字0~100  
}
```



也可以加到最大 100

```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    gauge1 = new myGauge1(this);  
    gauge1->resize(500,500); //这种圆形计量器大小  
    gauge1->setValue(100); //设置计量数字0~100  
}
```

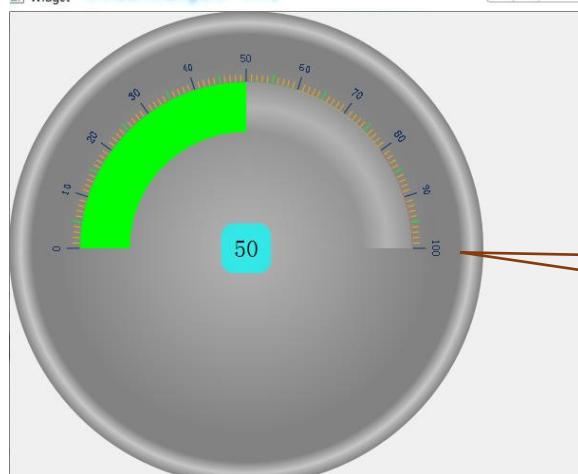


如果你想刻度超过 100，使用更大的值，你得修改 mygauge1 源码

我要修改计量计刻度数

mygauge1.c 文件修改

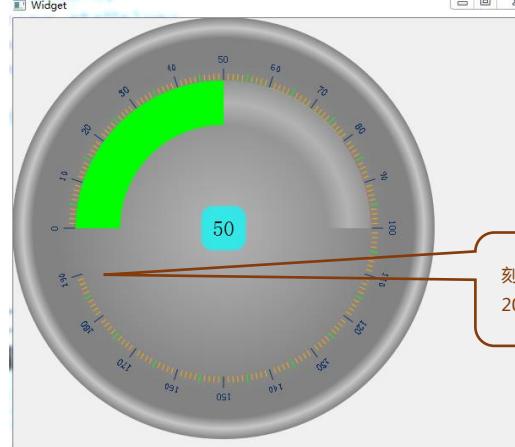
```
156 for (int i=0;i<=100;i++) //++
157 {
158     painter->save();
159     painter->rotate(startAngle);
160     if(i%10==0)
161     {
162         painter->setPen(QColor(16,49,98));
163         QPointF bottomPot(0,m_colorPieRadius+m_spa
```



所以默认刻度是 0~100

原本这行是 100

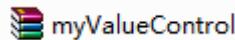
```
156 for (int i=0;i<=190;i++) //++
157 {
158     painter->save();
159     painter->rotate(startAngle);
160     if(i%10==0)
161     {
162         painter->setPen(QColor(16,49,98));
163         QPointF bottomPot(0,m_colorPieRadius+m_spa
```



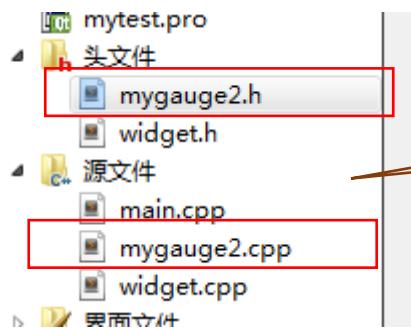
刻度就显示到 190 了，不要写
200，否则 0 刻度会被覆盖

如果将 156 行改成 190

仪表盘库使用



解压这个压缩包的文件



加入这两个库文件

```
#include "mygauge2.h"

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
    myGauge2 *gauge2;
```

包含仪表盘库头文件

创建仪表盘对象

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

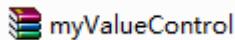
    gauge2=new myGauge2(this);
    gauge2->resize(300,300);
    gauge2->setAnimating(true);
    double value = 50;//设置指针值，因为setValue是double类型所以要这样定义变量
    gauge2->setValue(value);//写入值
```

CPP 文件使用仪表盘

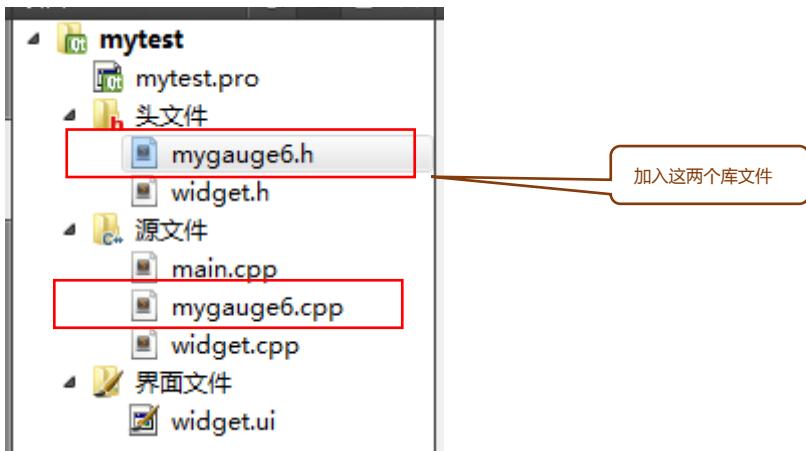


仪表盘指针指到 50 了，如果要修改刻度，请修改库文件源码。

垂直液位表库使用



解压这个压缩包的文件



```
#include "mygauge6.h"

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = 0);
    ~Widget();

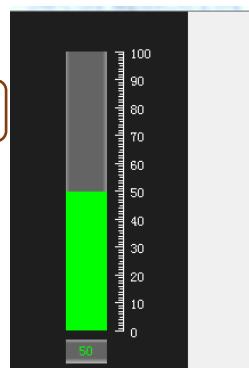
private:
    Ui::Widget *ui;
    myGauge6 *gauge6;
```

创建液位条对象

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

    gauge6 = new myGauge6(this);
    gauge6->SetGraphColor(Qt::green);
    gauge6->setValue(50);
}
```

设置液位条颜色和当前值



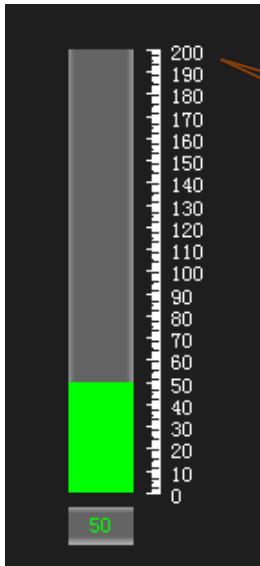
下面我们来修改液位条最大值的范围

```

gauge6 = new myGauge6(this);
gauge6->SetGraphColor(Qt::green);
gauge6->setRange(0, 200);
gauge6->setValue(50);

```

只需要增加 setRange 函数，液位条范围就从 0 到 200 了



16 |

updateTimer->setInterval(30);

我们发现 200 的液位刻度线太密集了，我们将其把刻度线拉大

```

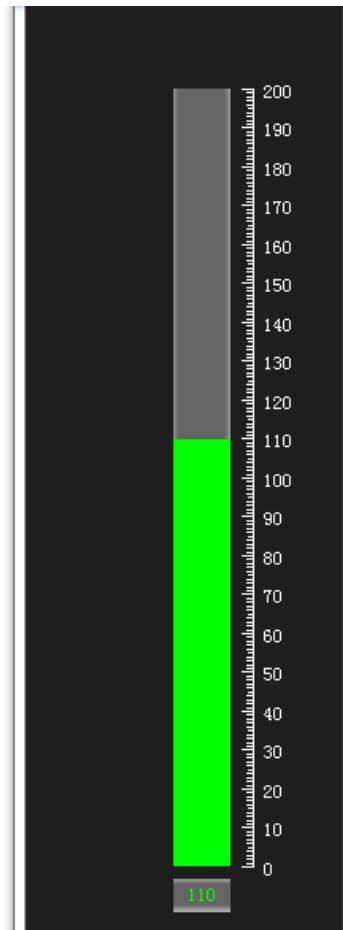
#include "widget.h"
#include "ui_widget.h"
#include <QDebug>

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

    gauge6 = new myGauge6(this);
    gauge6->SetGraphColor(Qt::green);
    gauge6->setRange(0, 200);
    gauge6->setFixedHeight(700);
    gauge6->setValue(110);
}

```

只要用 setFixedHeight
函数将黑色背景区域拉高就能让刻度线变宽



3D 液位条库使用

myValueControl

解压这个压缩包的文件

The screenshot shows the Qt Creator interface with a project named "mytest". The project structure is as follows:

- 头文件:
 - mygauge7.h (highlighted)
 - widget.h
- 源文件:
 - main.cpp
 - mygauge7.cpp (highlighted)
 - widget.cpp
- 界面文件:
 - widget.ui

The code editor displays the following C++ code:

```
namespace Ui {  
    class Widget;  
}  
  
class Widget : public QWidget  
{  
    Q_OBJECT  
  
public:  
    explicit Widget(QWidget *parent = 0);  
    ~Widget();  
  
private:  
    Ui::Widget *ui;  
    myGauge7 *gauge7;
```

A callout bubble points to the line "myGauge7 *gauge7;" with the text "创建液位条对象" (Create liquid level gauge object).

```
Widget::Widget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    gauge7=new myGauge7(this);  
    gauge7->SetGraphColor(Qt::red);  
    gauge7->setValue(50);  
}
```

设置液位条颜色和值

