

# 第六章 界面优化

## 1. QSS

### 1.1 背景介绍

在网页前端开发领域中, CSS 是一个至关重要的部分. 描述了一个网页的 "样式". 从而起到对网页美化的作用.

所谓样式, 包括但不限于大小, 位置, 颜色, 背景, 间距, 字体等等.



现在的网页很难找到没有 CSS 的. 可以说让 "界面好看" 是一个刚需.



"好看" 这个事情有没有意义呢? 是否一个软件, 能满足核心功能即可, 界面好看无所谓呢?

参考知乎上的这个帖子: <https://www.zhihu.com/question/30918916/answer/49934463>

网页开发作为 GUI 的典型代表, 也对于其他客户端 GUI 开发产生了影响. Qt 也是其中之一.

Qt 仿照 CSS 的模式, 引入了 QSS, 来对 Qt 中的控件做出样式上的设定, 从而允许程序猿写出界面更好看的代码.

同样受到 HTML 的影响, Qt 还引入了 QML 来描述界面, 甚至还可以直接把一个原生的 html 页面加载到界面上. 这部分内容咱们课堂上并不讨论.

当然, 由于 Qt 本身的设计理念和网页前端还是存在一定差异的, 因此 QSS 中只能支持部分 CSS 属性. 整体来说 QSS 要比 CSS 更简单一些.

另外, 大家也不必担心, 没有 CSS 基础的同学并不影响学习 QSS.

### ☀ 注意:

如果通过 QSS 设置的样式和通过 C++ 代码设置的样式冲突, 则 QSS 优先级更高.

## 1.2 基本语法

对于 CSS 来说, 基本的语法结构非常简单.

```
1 选择器 {  
2      属性名: 属性值;  
3 }
```

QSS 沿用了这样的设定.

```
1 选择器 {  
2      属性名: 属性值;  
3 }
```

其中:

- **选择器** 描述了 "哪个 widget 要应用样式规则".
- **属性** 则是一个键值对, 属性名表示要设置哪种样式, 属性值表示了设置的样式的值.

例如:

```
1 QPushButton { color: red; }
```

或者

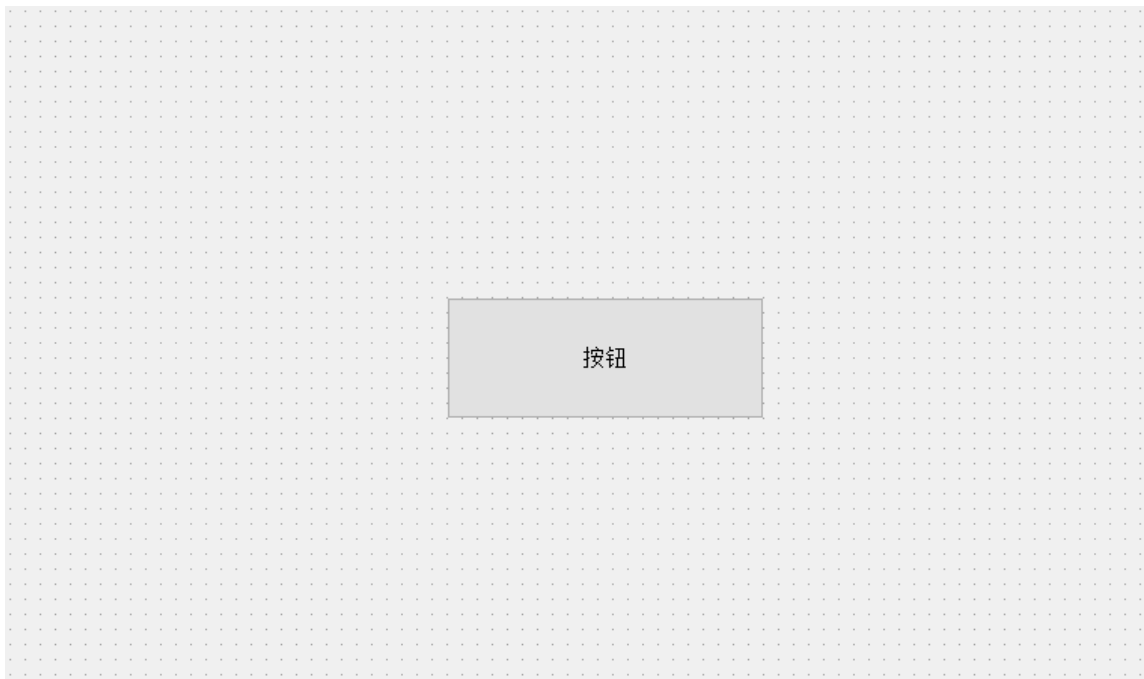
```
1 QPushButton {  
2     color: red;  
3 }
```

上述代码的含义表示, 针对界面上所有的 QPushButton , 都把文本颜色设置为 红色 .

编写 QSS 时使用单行的格式和多行的格式均可.

### 代码示例: QSS 基本使用

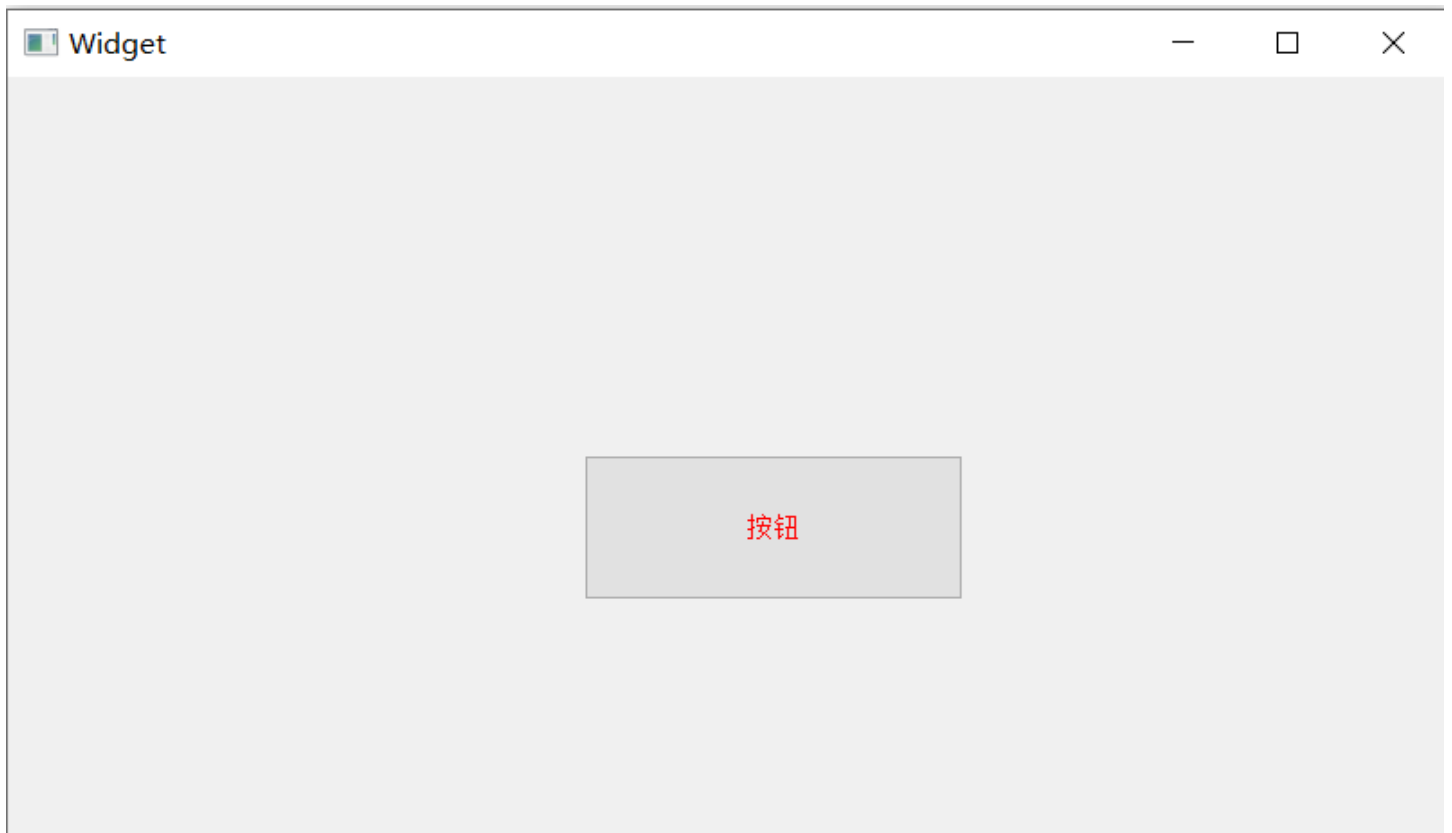
1) 在界面上创建一个按钮.



2) 编写代码, 设置样式

```
1 Widget::Widget(QWidget *parent)  
2     : QWidget(parent)  
3     , ui(new Ui::Widget)  
4 {  
5     ui->setupUi(this);  
6  
7     ui->pushButton->setStyleSheet("QPushButton { color: red; }");  
8 }
```

3) 运行程序, 观察效果. 可以看到文本已经是红色了.



**注意:** 上述代码中, 我们是只针对这一个按钮通过 `setStyleSheet` 方法设置的样式. 此时这个样式仅针对该按钮生效. 如果创建其他按钮, 其他按钮不会受到影响.

## 1.3 QSS 设置方式

### 1.3.1 指定控件样式设置

`QWidget` 中包含了 `setStyleSheet` 方法, 可以直接设置样式.

上述代码我们已经演示了上述设置方式.

另一方面, 给指定控件设置样式之后, 该控件的子元素也会受到影响.

**代码示例:** 子元素受到影响

1) 在界面上创建一个按钮



2) 修改 widget.cpp, 这次我们不再给按钮设置样式, 而是给 Widget 设置样式 (Widget 是 QPushButton 的父控件).

```
1 Widget::Widget(QWidget *parent)
2     : QWidget(parent)
3     , ui(new Ui::Widget)
4 {
5     ui->setupUi(this);
6
7     // 给 widget 本身设置样式.
8     this->setStyleSheet("QPushButton { color: red;} ");
9 }
```

3) 运行程序, 可以看到样式对于子控件按钮同样会生效.



### 1.3.2 全局样式设置

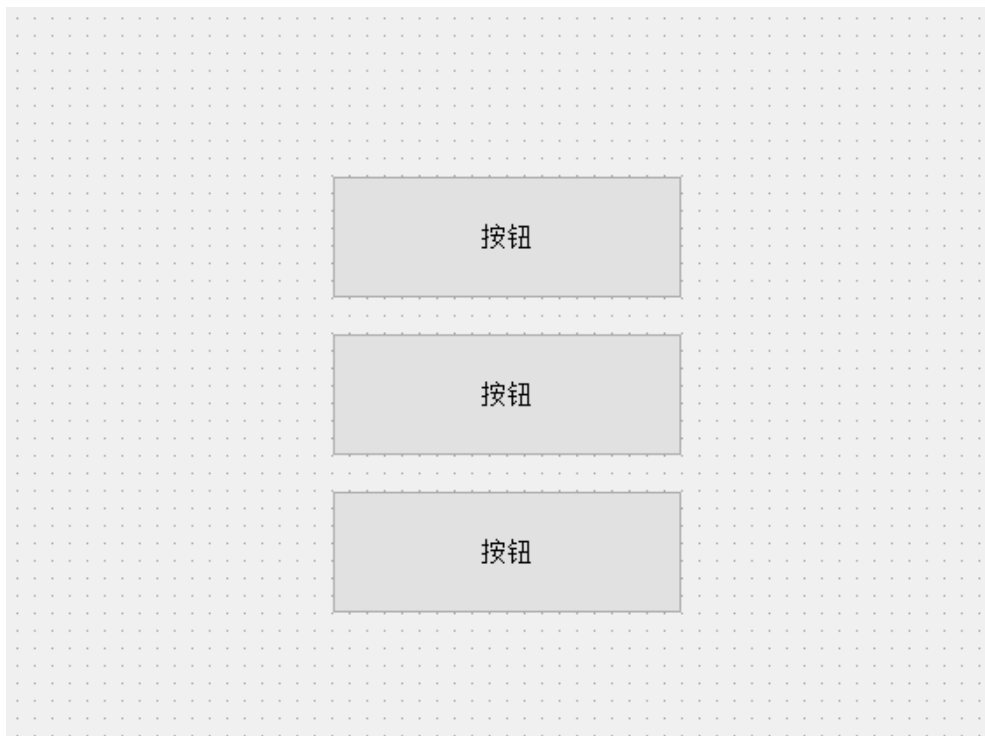
还可以通过 `QApplication` 的 `setStyleSheet` 方法设置整个程序的全局样式。

全局样式优点：

- 使同一个样式针对多个控件生效, 代码更简洁。
- 所有控件样式内聚在一起, 便于维护和问题排查。

**代码示例:** 使用全局样式

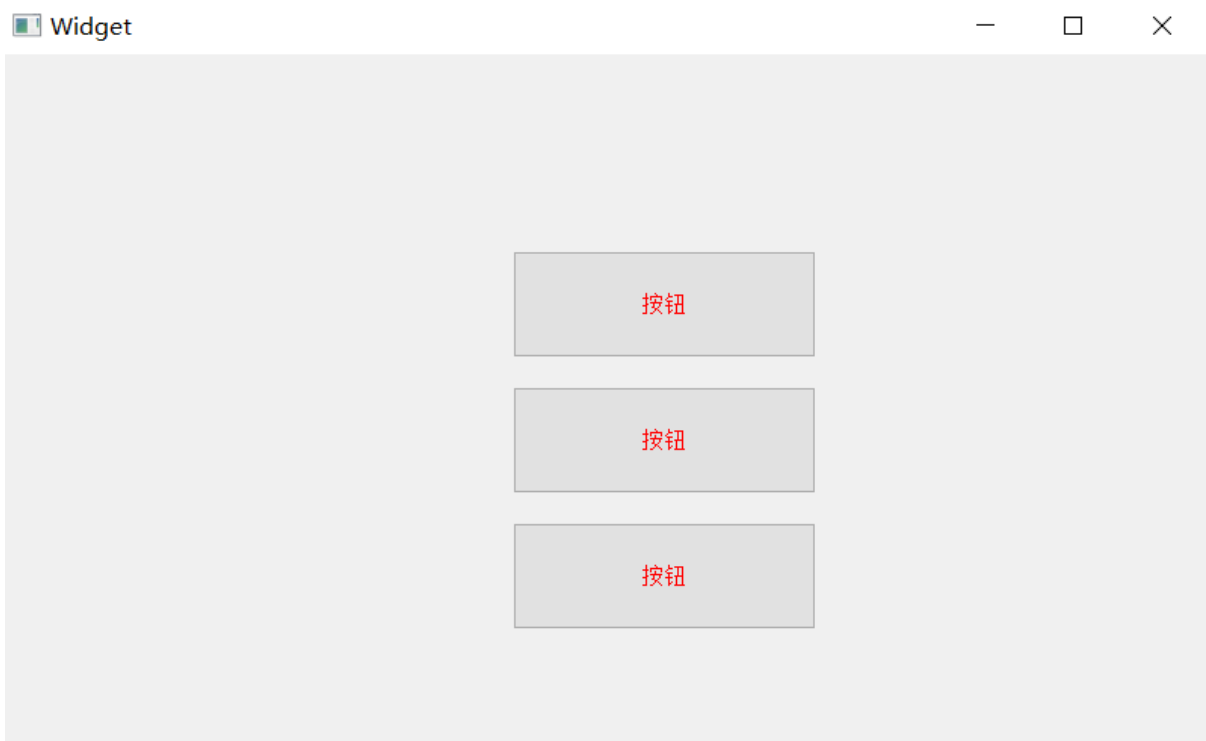
1) 在界面上创建三个按钮。



## 2) 编辑 main.cpp, 设置全局样式

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     // 设置全局样式
6     a.setStyleSheet("QPushButton { color: red; }");
7
8     Widget w;
9     w.show();
10    return a.exec();
11 }
```

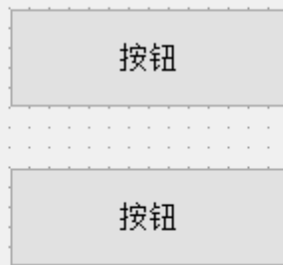
## 3) 运行程序, 可以看到此时三个按钮的颜色都设置为红色了.



### 代码示例: 样式的层叠特性

如果通过全局样式给某个控件设置了属性1, 通过指定控件样式给控件设置属性2, 那么这两个属性都会产生作用.

1) 在界面上创建两个按钮



2) 编写 main.cpp, 设置全局样式, 把按钮文本设置为红色.

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     // 设置全局样式
6     a.setStyleSheet("QPushButton { color: red; }");
7
8     Widget w;
9     w.show();
10    return a.exec();
11 }
```

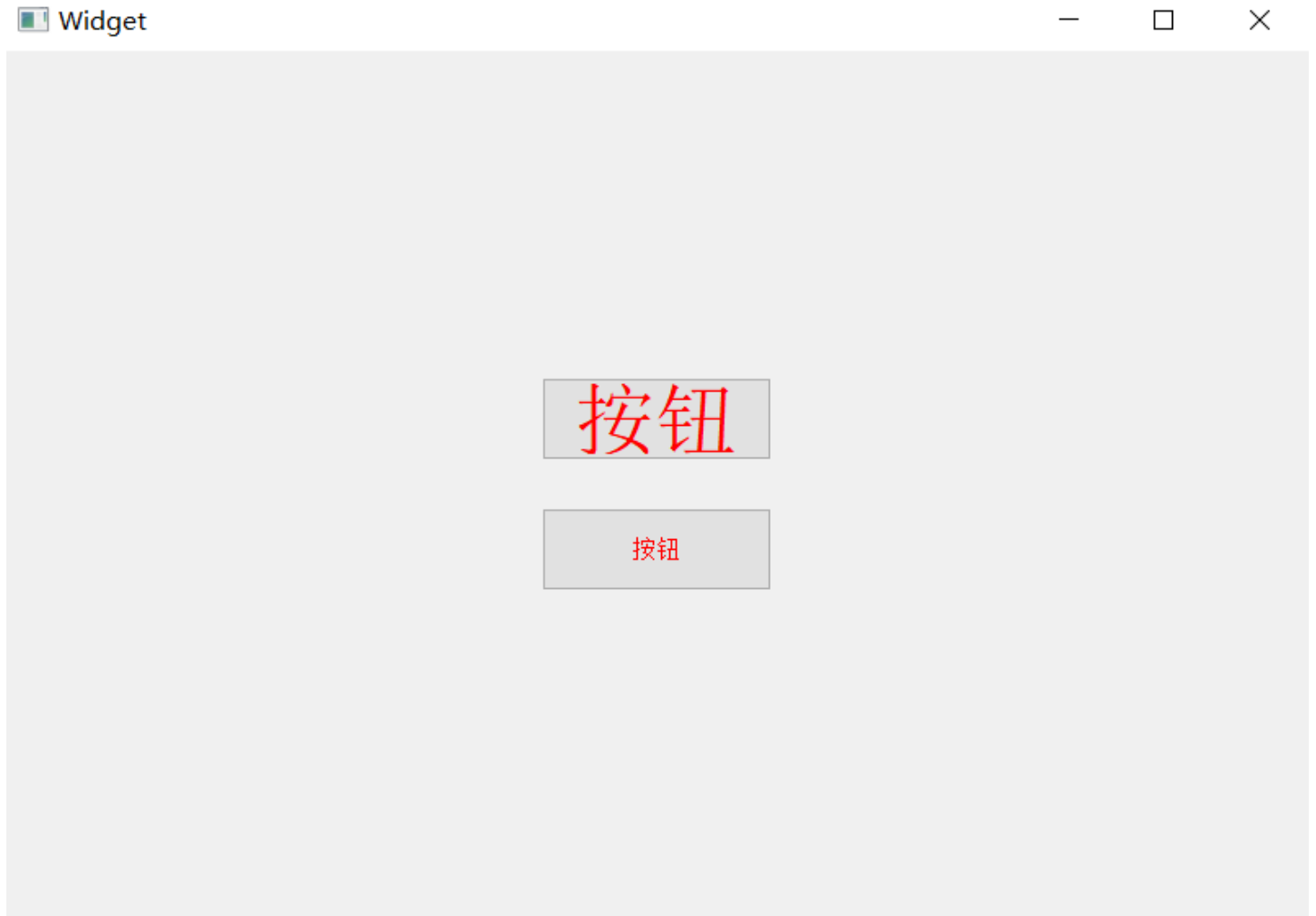
3) 编写 widget.cpp, 给第一个按钮设置字体大小.


```
1 Widget::Widget(QWidget *parent)
2     : QWidget(parent)
3     , ui(new Ui::Widget)
4 {
```

```
5     ui->setupUi(this);
6
7     // 设置指定控件样式
8     ui->pushButton->setStyleSheet("QPushButton { font-size: 50px} ");
9 }
```

4) 运行程序, 可以看到, 对于第一个按钮来说, 同时具备了颜色和字体大小样式. 而第二个按钮只有颜色样式.

说明针对第一个按钮, 两种设置方式设置的样式, 叠加起来了.



 形如上述这种属性叠加的效果, 我们称为 "层叠性".

CSS 全称为 Cascading Style Sheets, 其中 Cascading 就是 "层叠性" 的意思. QSS 也继承了这样的设定.

实际上把 QSS 叫做 QCSS 也许更合适一些~

## 代码示例: 样式的优先级

如果全局样式, 和指定控件样式冲突, 则指定控件样式优先展示.

1) 在界面上创建两个按钮



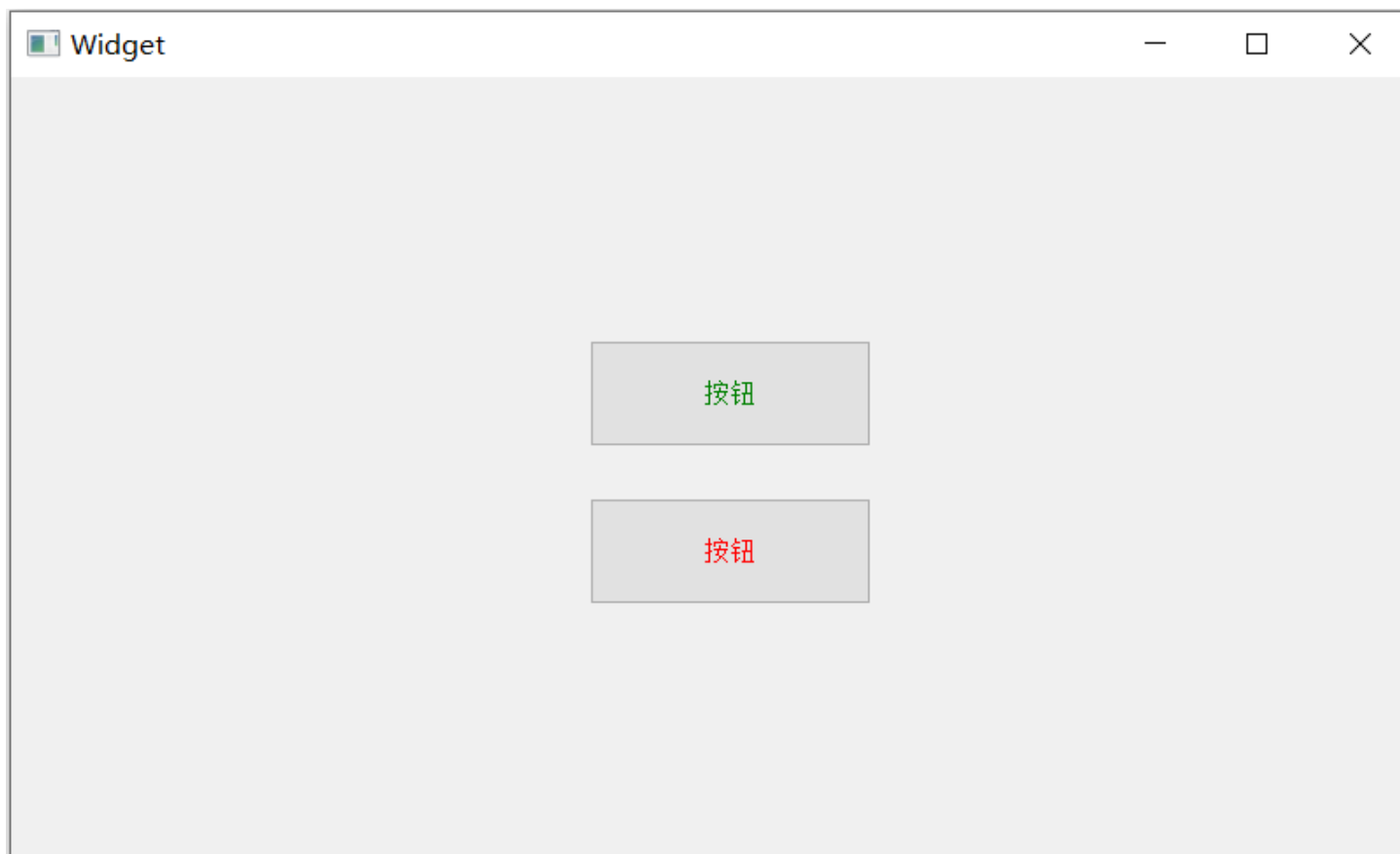
2) 编辑 main.cpp, 把全局样式设置为红色.

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     // 设置全局样式
6     a.setStyleSheet("QPushButton { color: red; }");
7
8     Widget w;
9     w.show();
10    return a.exec();
11 }
```

3) 编辑 widget.cpp, 把第一个按钮样式设为绿色.

```
1 Widget::Widget(QWidget *parent)
2     : QWidget(parent)
3     , ui(new Ui::Widget)
4 {
5     ui->setupUi(this);
6
7     // 设置第一个按钮颜色为绿色
8     ui->pushButton->setStyleSheet("QPushButton { color: green; }");
9 }
```

4) 运行程序, 观察效果. 可以看到第一个按钮已经成为绿色了, 但是第二个按钮仍然是红色.



在 CSS 中也存在类似的优先级规则. 通常来说都是 "局部" 优先级高于 "全局" 优先级. 相当于全局样式先 "奠定基调", 再通过指定控件样式来 "特事特办".

### 1.3.3 从文件加载样式表

上述代码都是把样式通过硬编码的方式设置的. 这样使 QSS 代码和 C++ 代码耦合在一起了, 并不方便代码的维护.

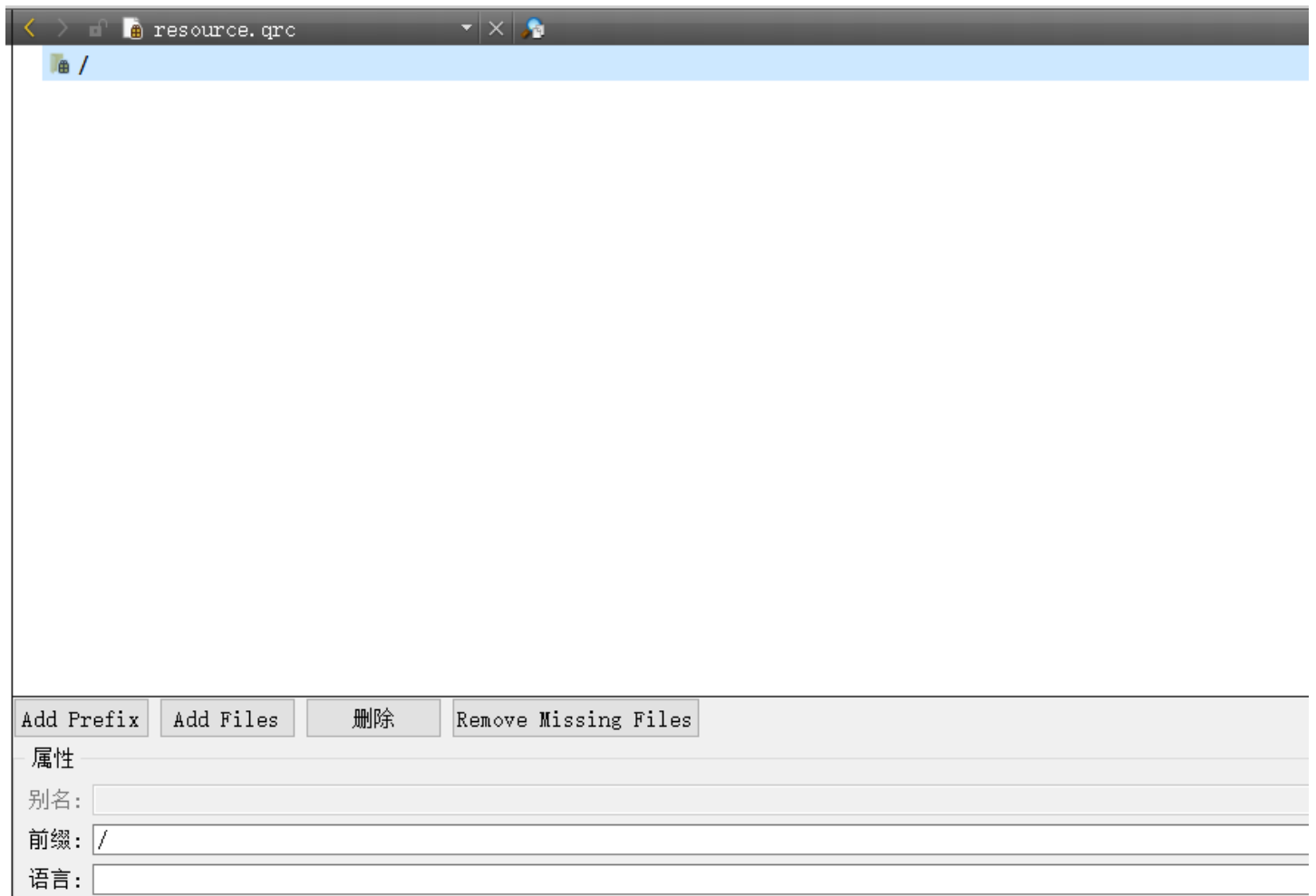
因此更好的做法是把样式放到单独的文件中, 然后通过读取文件的方式来加载样式.

#### 代码示例: 从文件加载全局样式

1) 在界面上创建一个按钮



2) 创建 `resource.qrc` 文件, 并设定前缀为 `/`.



3) 创建 `style.qss` 文件, 并添加到 `resource.qrc` 中.

- `style.qss` 是需要程序运行时加载的. 为了规避绝对路径的问题, 仍然使用 `qrc` 的方式来组织. (即把资源文件内容打包到 `cpp` 代码中).
- Qt Creator 没有提供创建 `qss` 文件的选项. 咱们直接 右键 -> 新建文件 -> 手动设置文件扩展名为 `qss` 即可.

4) 使用 Qt Creator 打开 `style.qss` , 编写内容

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

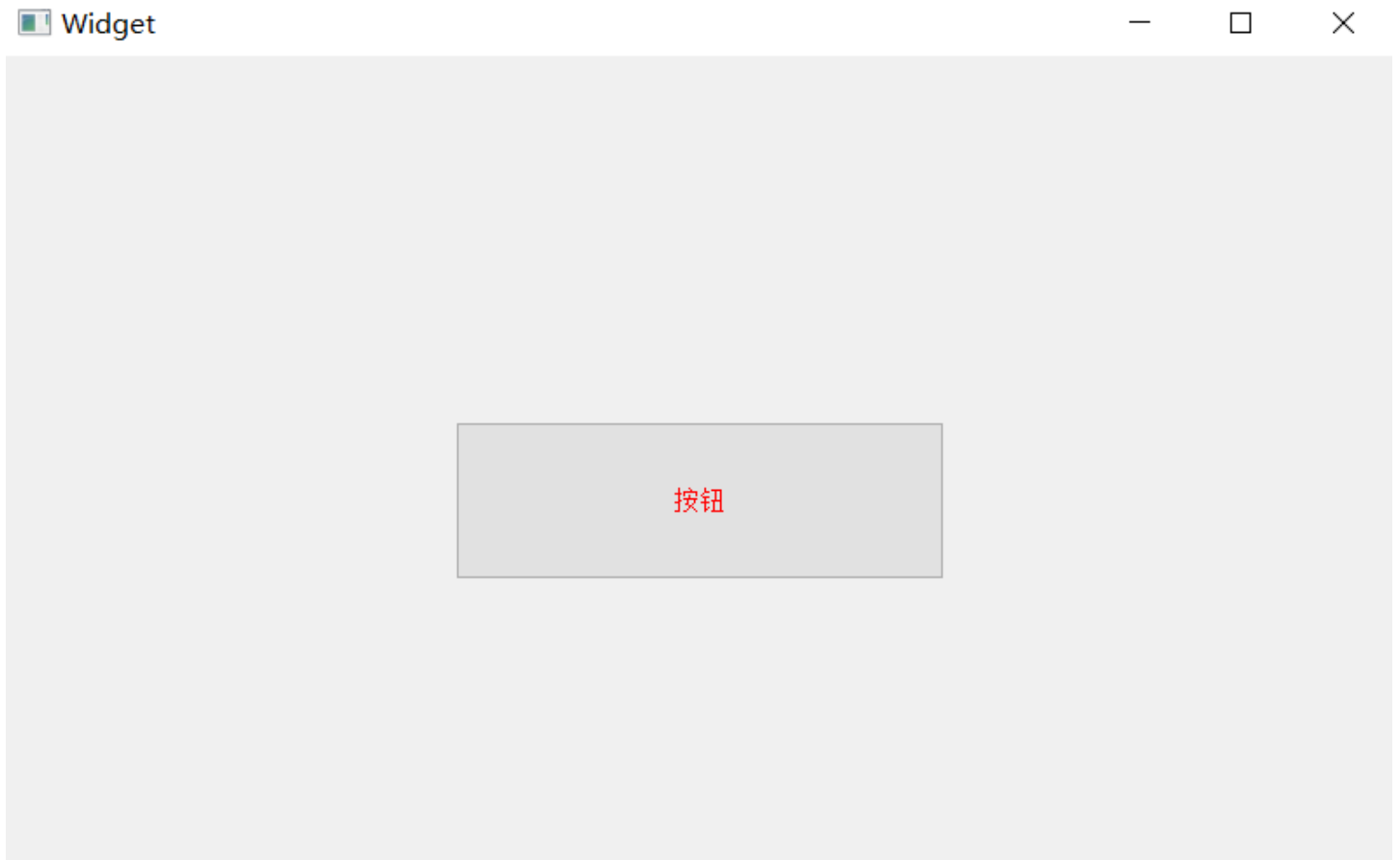
5) 修改 main.cpp, 新增一个函数用来加载样式

```
1 QString loadQSS() {
2     QFile file(":/style.qss");
3     // 打开文件
4     file.open(QFile::ReadOnly);
5     // 读取文件内容。虽然 readAll 返回的是 QByteArray, 但是 QString 提供了
    QByteArray 版本的构造函数。
6     QString style = file.readAll();
7     // 关闭文件
8     file.close();
9     return style;
10 }
```

6) 修改 main.cpp, 在 main 函数中调用上述函数, 并设置样式。

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     // 调用上述函数加载样式
6     const QString& style = loadQSS();
7     a.setStyleSheet(style);
8
9     Widget w;
10    w.show();
11    return a.exec();
12 }
```

7) 运行程序, 可以看到样式已经生效了.



🍌 这里要小小的吐槽一下. 理论上来说 Qt 应该要提供, 直接从文件加载样式表的接口. 类似于 `setStyleSheetFromFile(const QString& path)` 这种, 在内部把读文件操作封装好.

### 1.3.4 使用 Qt Designer 编辑样式

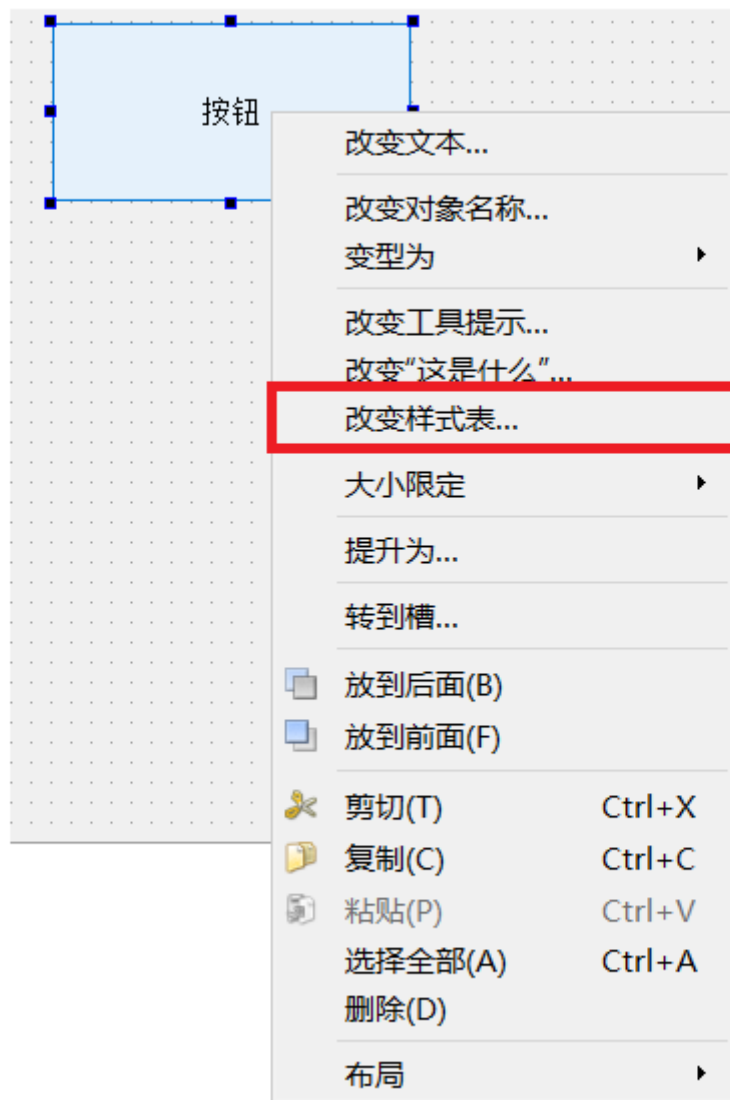
QSS 也可以通过 Qt Designer 直接编辑, 从而起到实时预览的效果. 同时也能避免 C++ 和 QSS 代码的耦合.

**代码示例:** 使用 Qt Designer 编辑样式

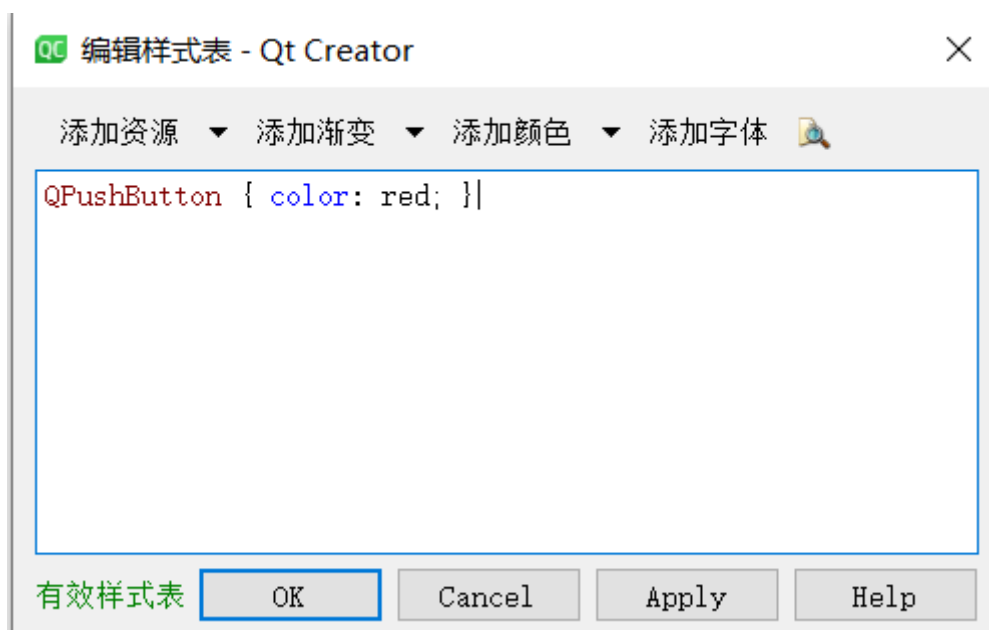
1) 在界面上创建一个按钮



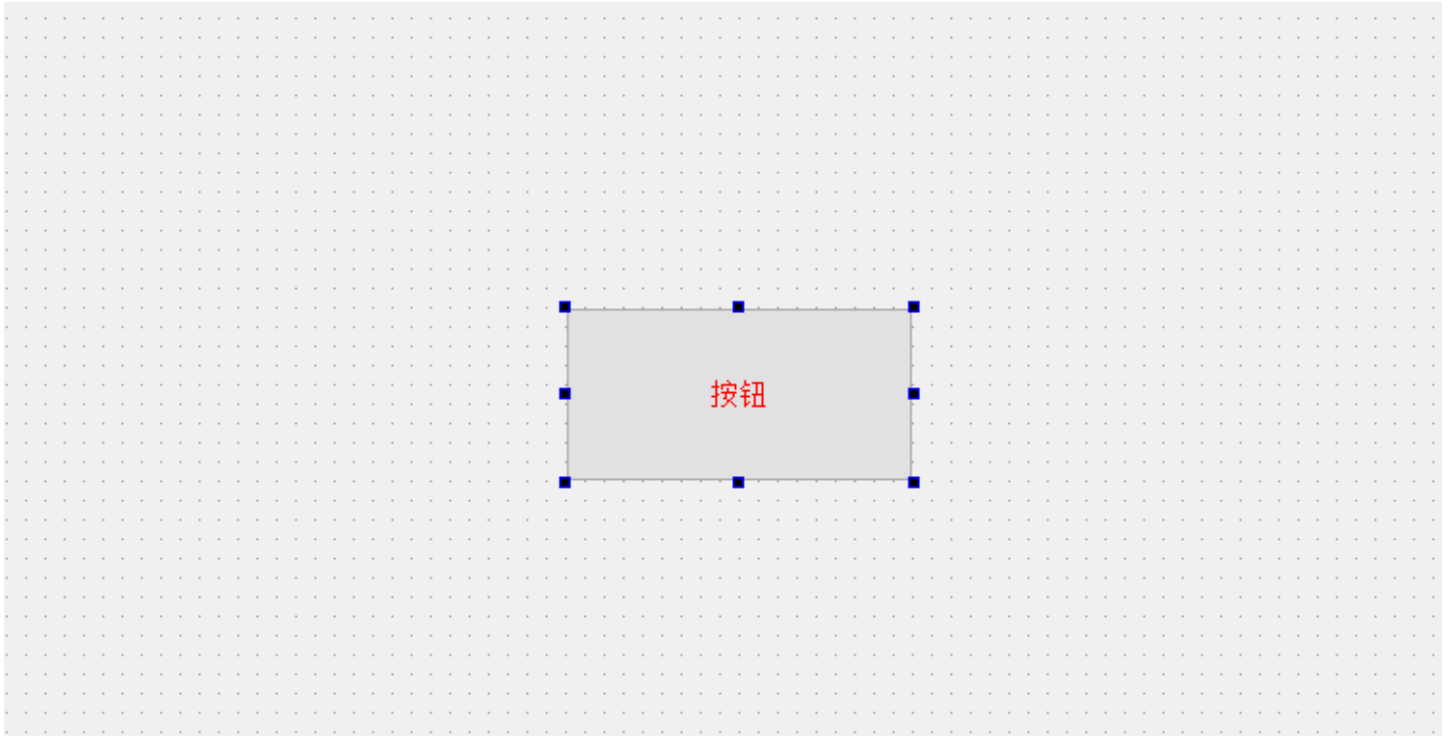
2) 右键按钮, 选择 "改变样式表"



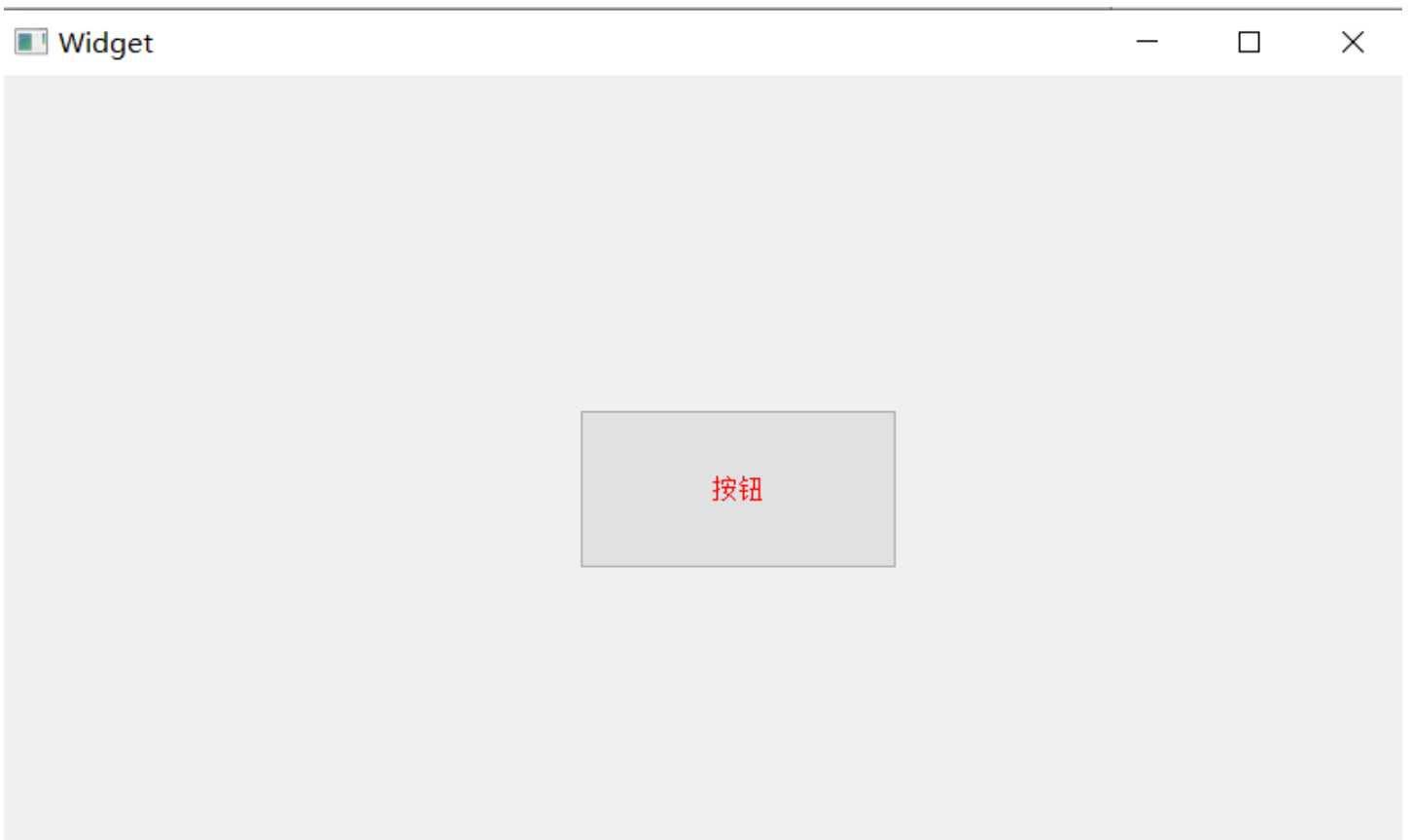
3) 在弹出的样式表编辑器中, 可以直接填写样式. 填写完毕, 点击 OK 即可.



4) 此时 Qt Designer 的预览界面就会实时显示出样式的变化.



5) 运行程序, 可以看到样式确实发生了改变.



这种方式设置样式, 样式内容会被以 xml 格式记录到 ui 文件中.

```
1 <property name="styleSheet">
```

```
2 <string notr="true">QPushButton { color: red; }</string>
3 </property>
```

同时在控件的 `styleSheet` 属性中也会体现.

<code>styleSheet</code>	<code>QPushButton { color: red; }</code>
-------------------------	--



当我们发现一个控件的样式不符合预期的时候, 要记得排查这四个地方:

- 全局样式
- 指定控件样式
- qss 文件中的样式
- ui 文件中的样式

## 1.4 选择器

### 1.4.1 选择器概况

QSS 的选择器支持以下几种:

选择器	示例	说明
全局选择器	*	选择所有的 widget.
类型选择器 (type selector)	QPushButton	选择所有的 QPushButton 和 其子类 的控件.
类选择器 (class selector)	.QPushButton	选择所有的 QPushButton 的控件. 不会选择子类.
ID 选择器	#pushButton_2	选择 <code>objectName</code> 为 <code>pushButton_2</code> 的控件.
后代选择器	QDialog QPushButton	选择 QDialog 的所有后代(子控件, 孙子控件等等)中的 QPushButton.
子选择器	QDialog > QPushButton	选择 QDialog 的所有子控件中的 QPushButton.
并集选择器	QPushButton, QLineEdit, QComboBox	选择 QPushButton, QLineEdit, QComboBox 这三种控件. (即接下来的样式会针对这三种控件都生效).

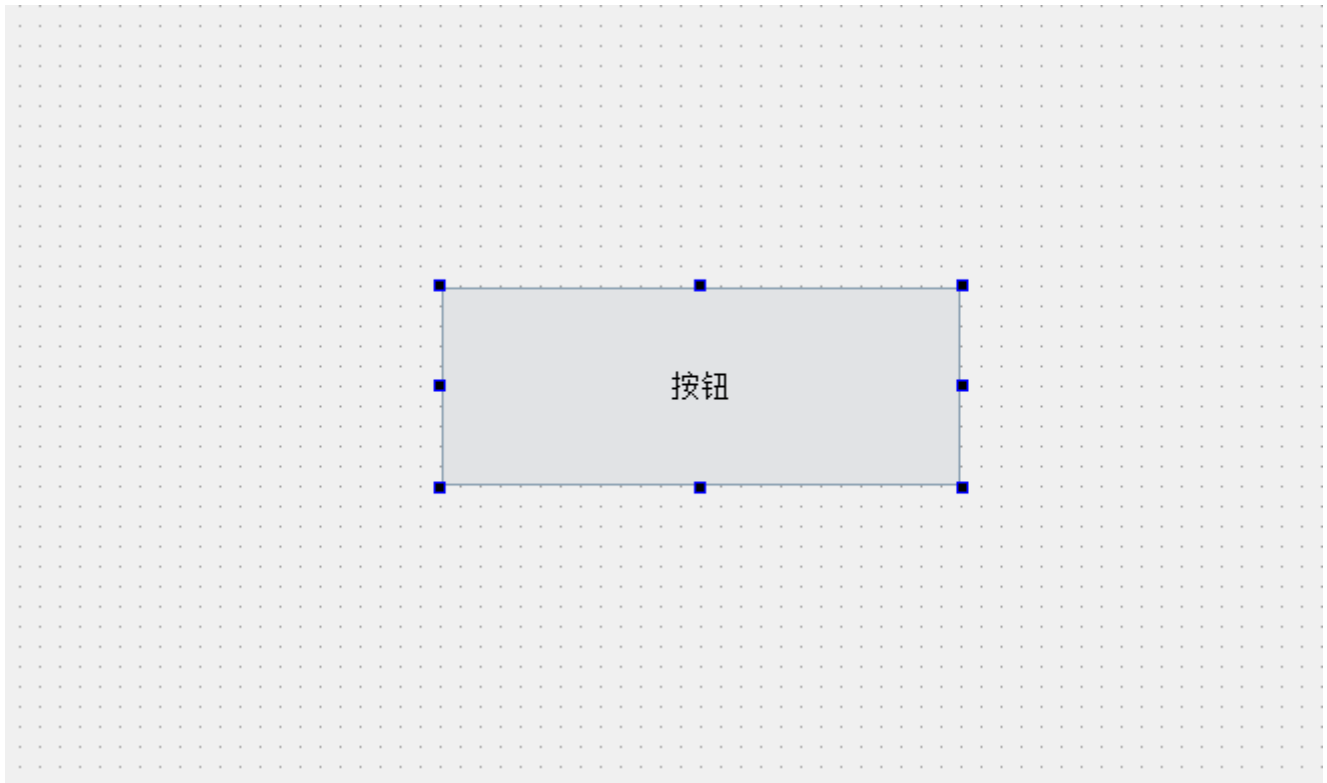
属性选择器	<code>QPushButton[flat="false"]</code>	选择所有 QPushButton 中, flat 属性为 false 的控件.
-------	--	---

总体来说, QSS 选择器的规则和 CSS 选择器基本一致.

上述选择器咱们也不需要全都掌握, 就只熟悉最常用的几个即可(上述加粗的).

### 代码示例: 使用类型选择器选中子类控件

1) 在界面上创建一个按钮.



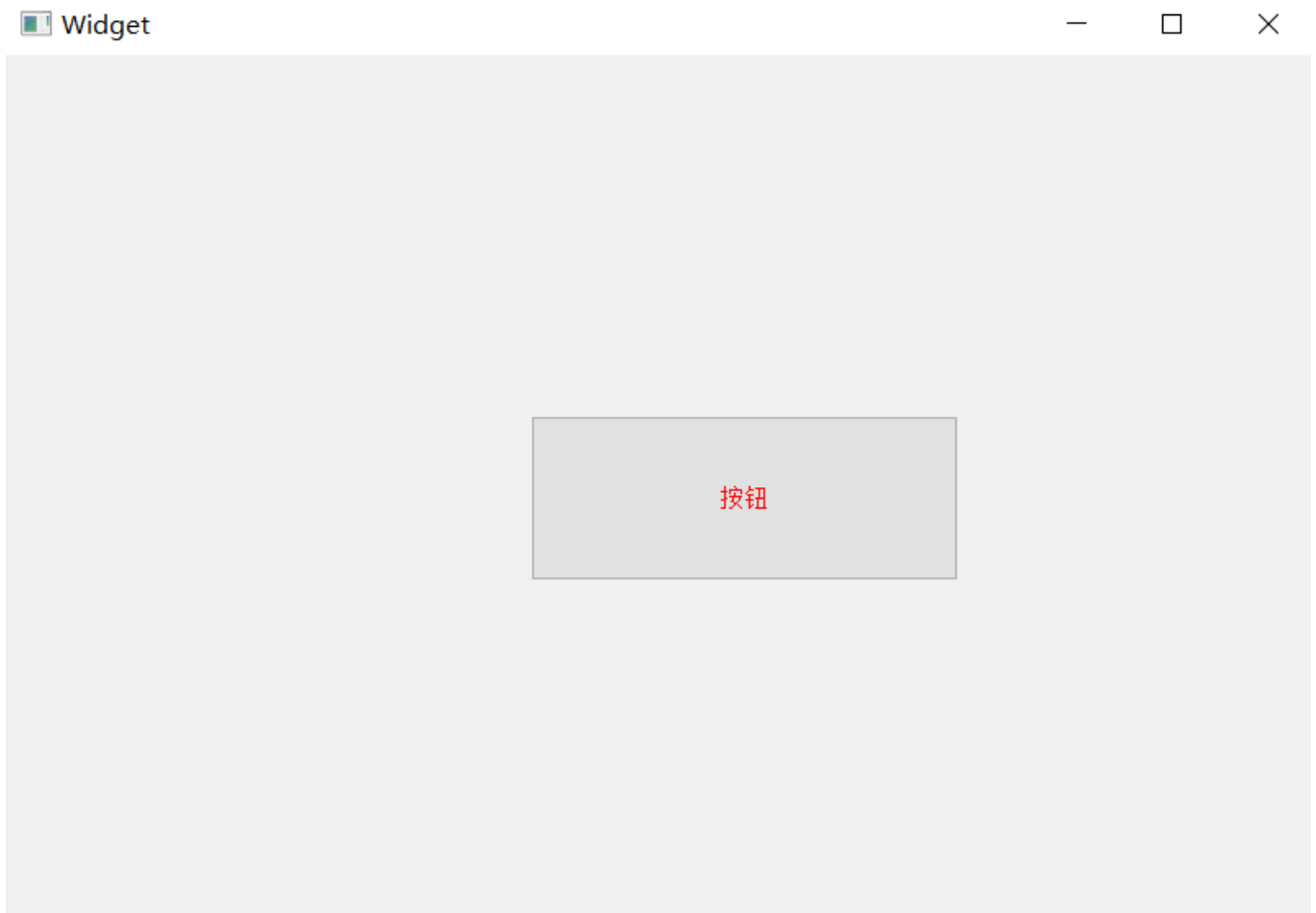
2) 修改 main.cpp, 设置全局样式

注意, 此处选择器使用的是 `QWidget`. `QPushButton` 也是 `QWidget` 的子类, 所以会受到 `QWidget` 选择器的影响.

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     // 设置全局样式
6     a.setStyleSheet("QWidget { color: red; }");
7
8     Widget w;
9     w.show();
```

```
10     return a.exec();  
11 }
```

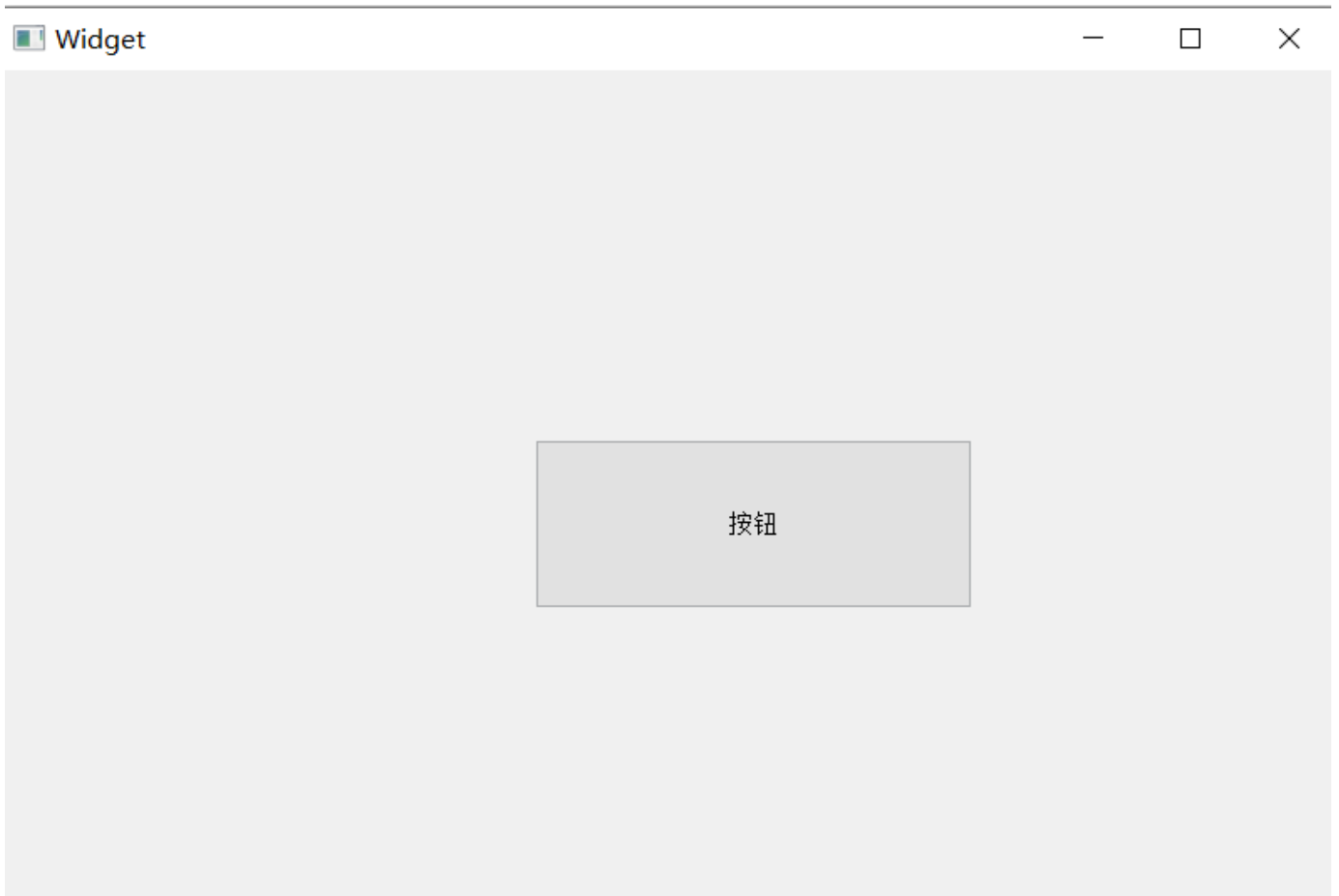
3) 运行程序, 可以看到按钮的文本颜色已经是红色了.



5) 如果把上述样式代码修改为下列代码

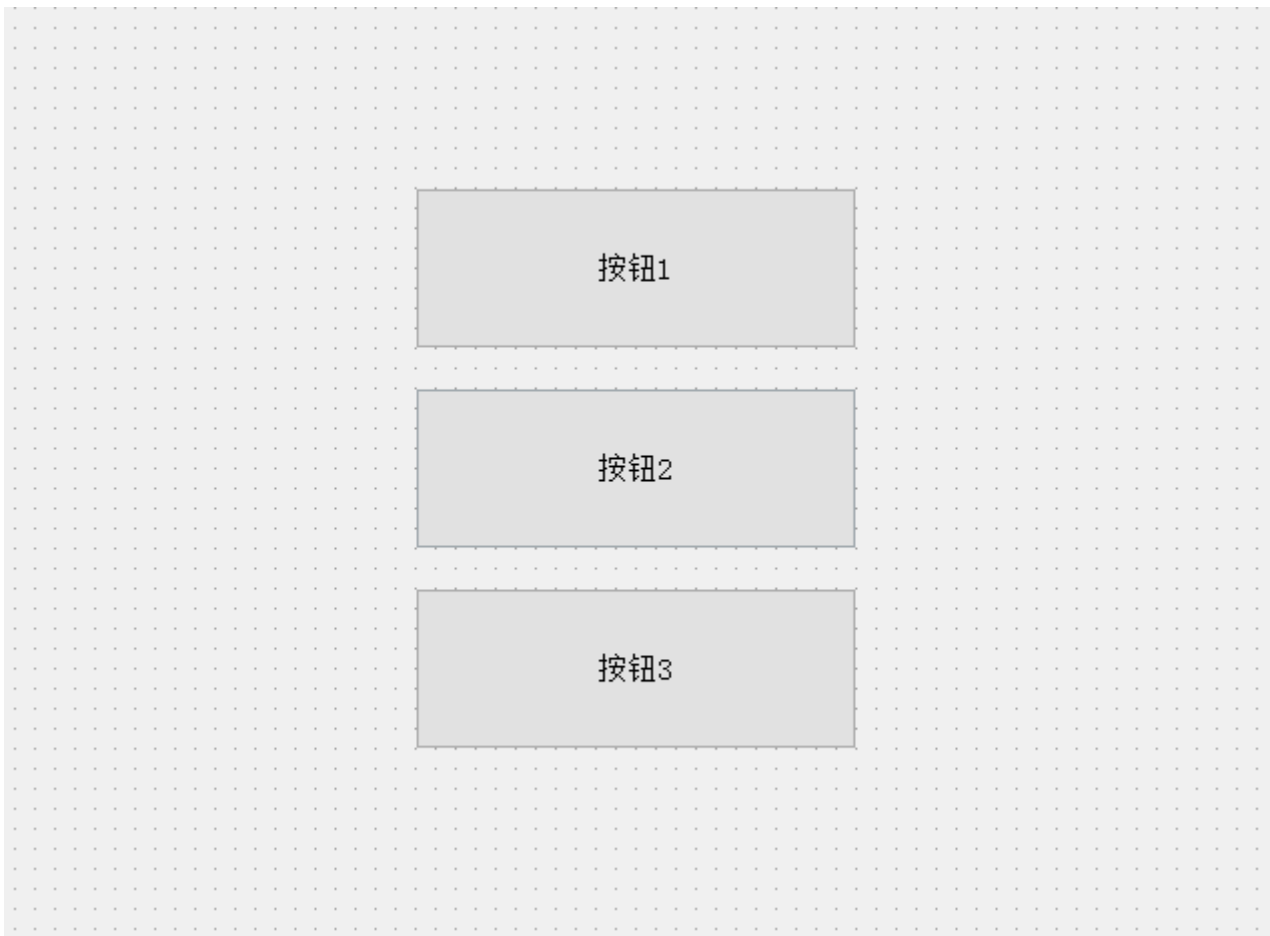
```
1 a.setStyleSheet(".QWidget { color: red; });
```

此时按钮的颜色不会发生改变. 此时只是选择 `QWidget` 类, 而不会选择 `QWidget` 的子类 `QPushButton` 了.



### 代码示例: 使用 id 选择器

1) 在界面上创建 3 个按钮, `objectName` 为 `pushButton`, `pushButton_2`, `pushButton_3`



## 2) 编写 main.cpp, 设置全局样式

- 先通过 QPushButton 设置所有的按钮为黄色.
- 再通过 #pushButton 和 #pushButton\_2 分别设置这两个按钮为红色和绿色.

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     // 设置全局样式
6     QString style = "";
7     style += "QPushButton { color: yellow; }";
8     style += "#pushButton { color: red; }";
9     style += "#pushButton_2 { color: green; }";
10    a.setStyleSheet(style);
11
12    Widget w;
13    w.show();
14    return a.exec();
15 }
```

## 3) 执行程序, 观察效果

按钮1

按钮2

按钮3

当某个控件身上, 通过类型选择器和 ID 选择器设置了冲突的样式时, ID 选择器样式优先级更高.

同理, 如果是其他的多种选择器作用同一个控件时出现冲突的样式, 也会涉及到优先级问题. Qt 文档上有具体的优先级规则介绍 (参见 The Style Sheet Syntax 的 Conflict Resolution 章节).

这里的规则计算起来非常复杂(CSS 中也存在类似的设定), 咱们此处对于优先级不做进一步讨论.

实践中我们可以简单的认为, **选择器描述的范围越精准, 则优先级越高**. 一般来说, ID 选择器优先级是最高的.

## 代码示例: 使用并集选择器

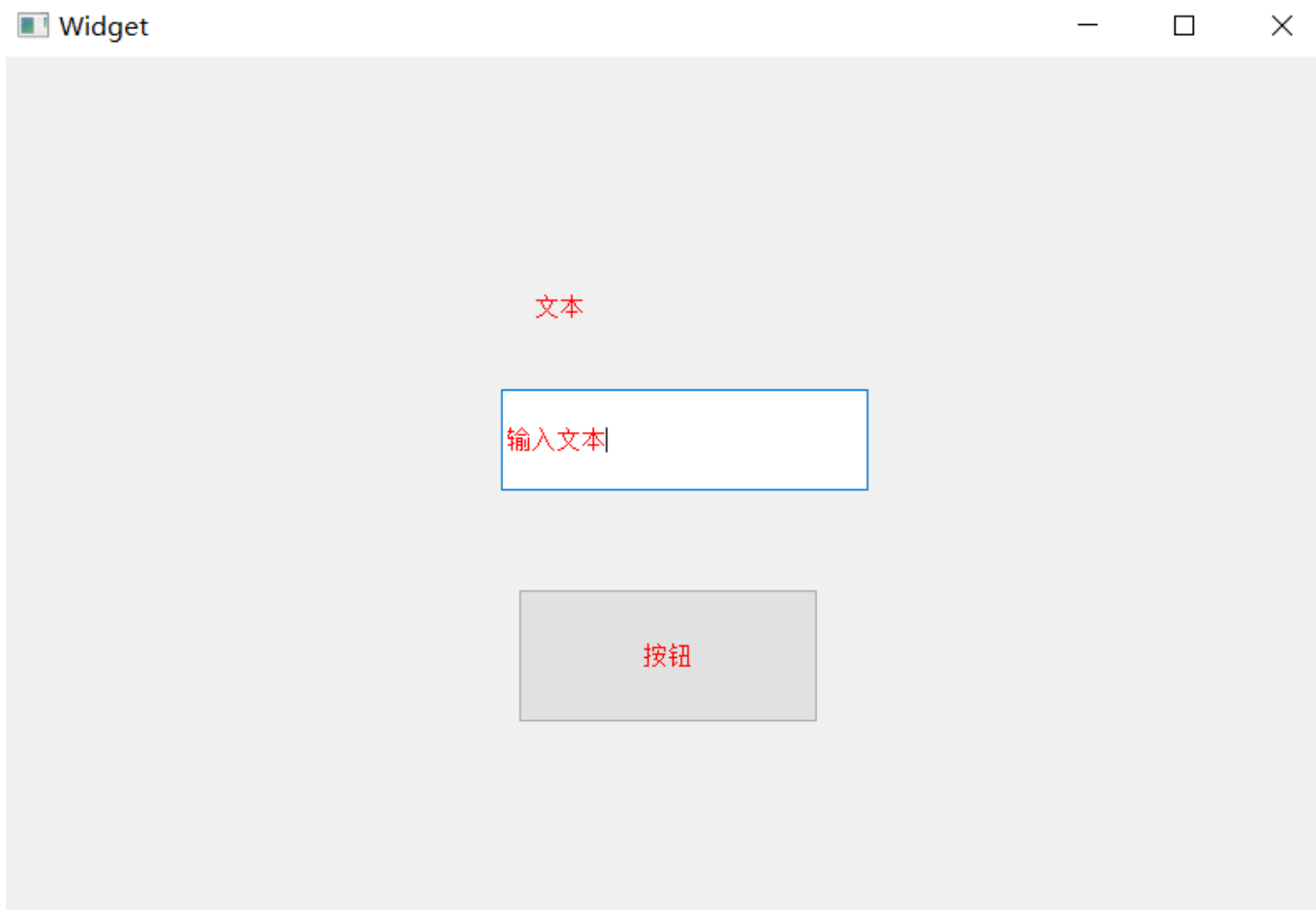
1) 创建按钮, label, 单行输入框



2) 编写 main.cpp, 设置全局样式

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     // 设置全局样式
6     a.setStyleSheet("QPushButton, QLabel, QLineEdit { color: red; } ");
7
8     Widget w;
9     w.show();
10    return a.exec();
11 }
```

3) 运行程序, 可以看到这三种控件的文字颜色都设置为了红色



并集选择器是一种很好的代码复用的方式. 很多时候我们希望界面上的多个元素风格是统一的, 就可以使用并集选择器, 把样式属性同时指定给多种控件.

### 1.4.2 子控件选择器 (Sub-Controls)

有些控件内部包含了多个 "子控件". 比如 QComboBox 的下拉后的面板, 比如 QSpinBox 的上下按钮等.

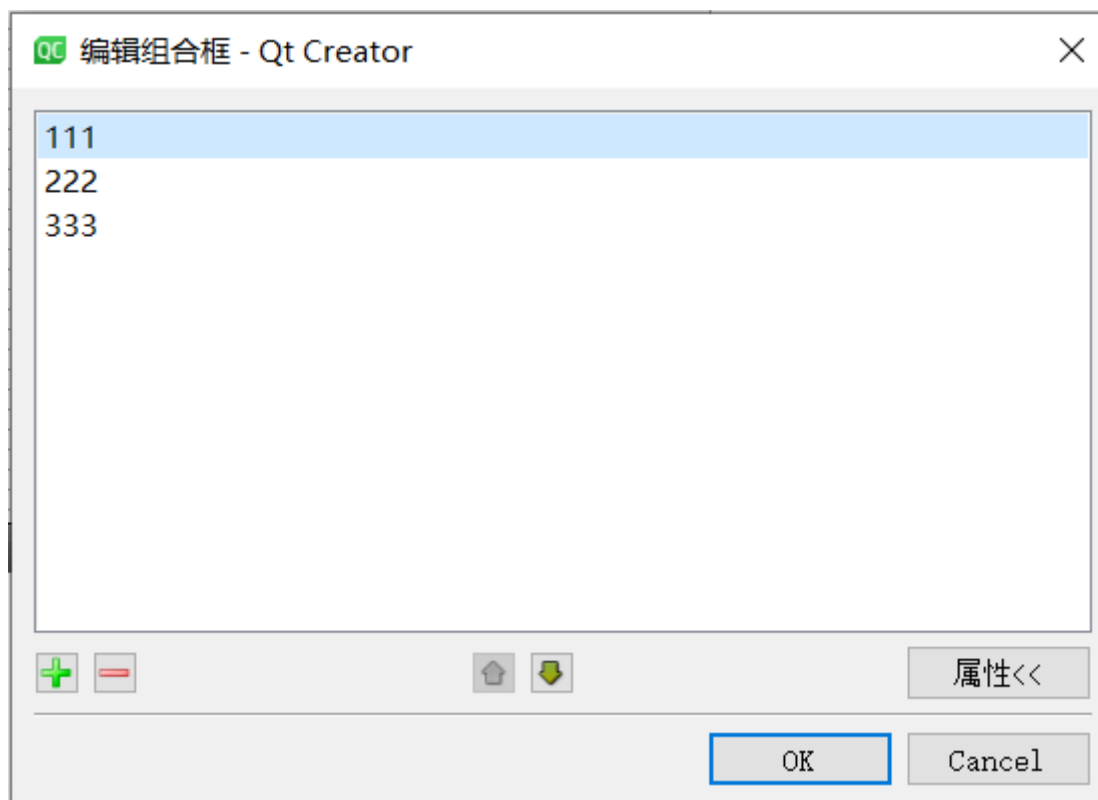
可以通过子控件选择器 `::`, 针对上述子控件进行样式设置.



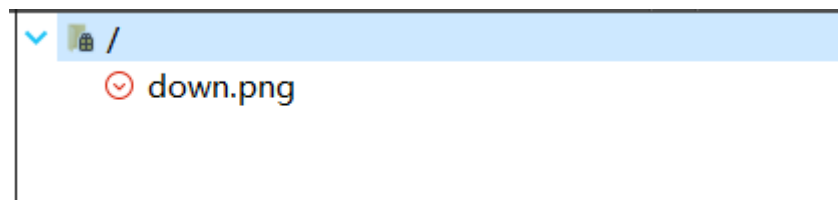
哪些控件拥有哪些子控件, 参考文档 Qt Style Sheets Reference 中 List of Sub-Controls 章节.

**代码示例:** 设置下拉框的下拉按钮样式

1) 在界面上创建一个下拉框, 并创建几个选项



2) 创建 `resource.qrc` , 并导入图片 `down.png`



图标可以从阿里巴巴矢量图标库下载. <https://www.iconfont.cn/search/index?searchType=icon&q=down>

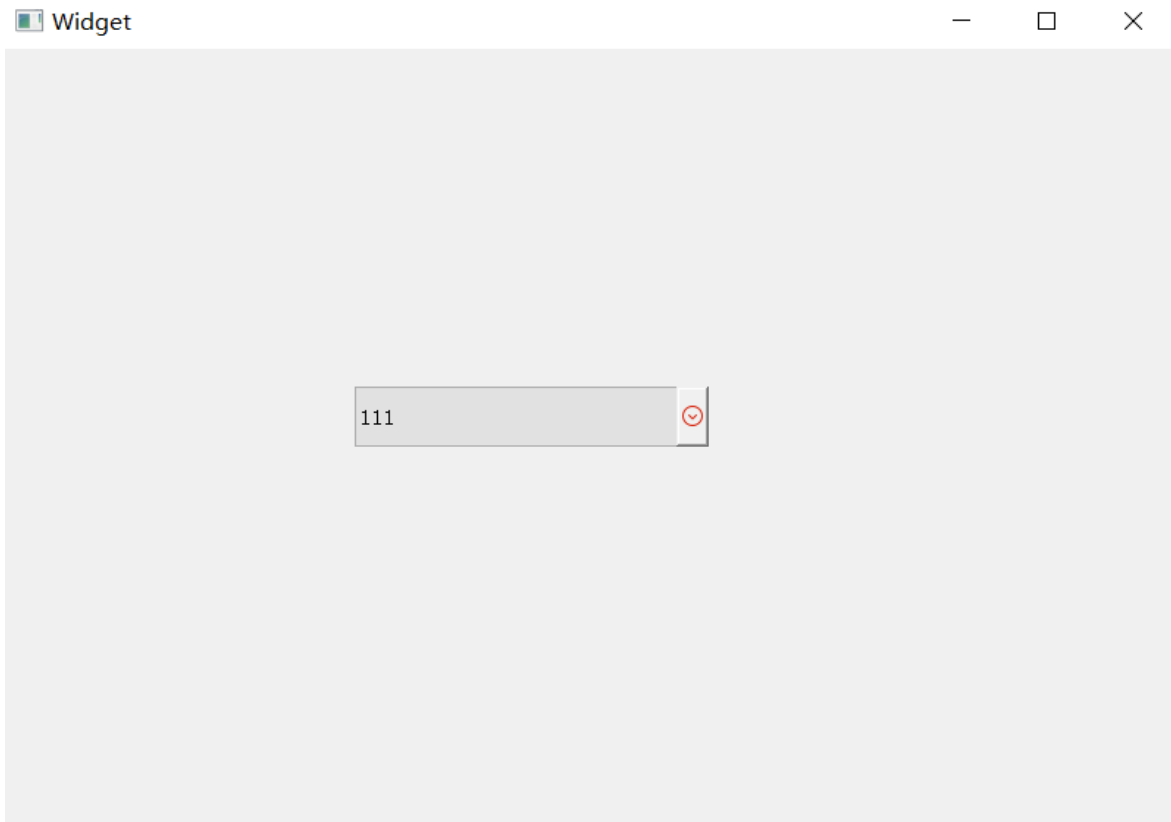
3) 修改 `main.cpp`, 编写全局样式

- 使用子控件选择器 `QComboBox::down-arrow` 选中了 `QComboBox` 的下拉按钮.
- 再通过 `image` 属性设置图片.

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     QString style = "";
6     style += "QComboBox::down-arrow { image: url(:/down.png) }";
7 }
```

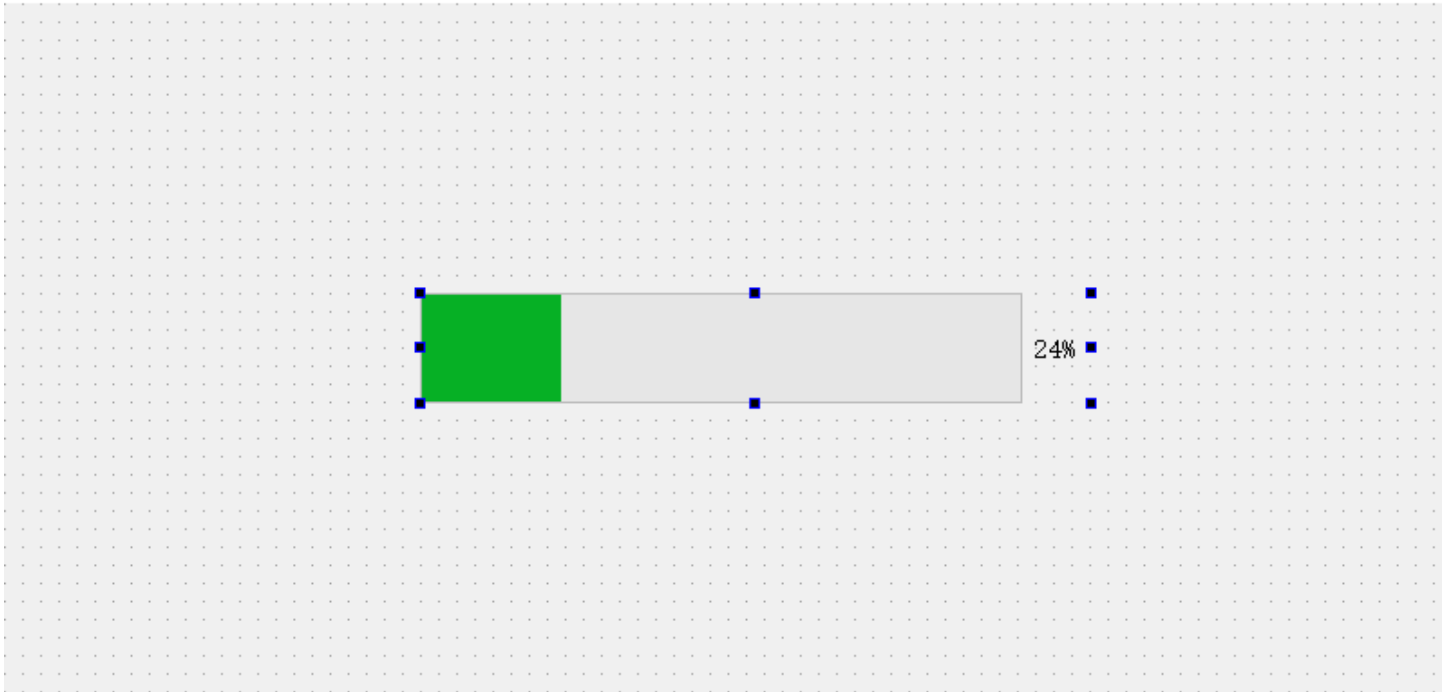
```
8     a.setStyleSheet(style);
9
10    Widget w;
11    w.show();
12    return a.exec();
13 }
```

#### 4) 执行程序, 观察效果



#### 代码示例: 修改进度条的颜色

1) 在界面上创建一个进度条.



2) 在 Qt Designer 右侧的属性编辑器中, 找到 QWidget 的 `styleSheet` 属性.

编辑如下内容:

- 其中的 `chunk` 是选中进度条中的每个 "块". 使用 `QProgressBar::text` 则可以选中文本.

```
1 QProgressBar::chunk {background-color: #FF0000;}
```

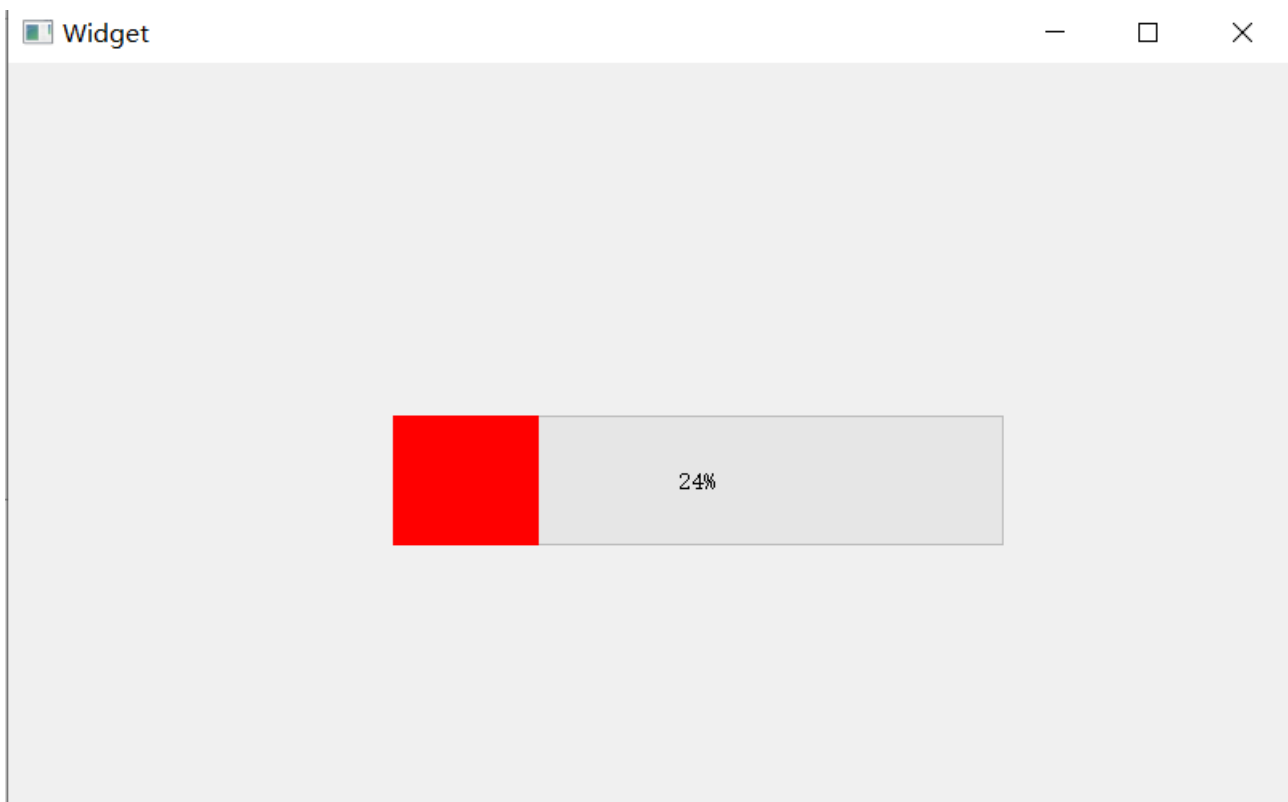
<code>styleSheet</code>	<code>QProgressBar::chunk {background-color: #FF0000;}</code>
-------------------------	---

同时把 `QProcessBar` 的 `alignment` 属性设置为垂直水平居中.

✓ <code>alignment</code>	<code>AlignHCenter, AlignVCenter</code>
水平的	<code>AlignHCenter</code>
垂直的	<code>AlignVCenter</code>

此处如果不设置 `alignment`, 进度条中的数字会跑到左上角. 这个怀疑是 Qt 本身的 bug, 暂时只能先使用 `alignment` 来手动调整下.

3) 执行程序, 可以看到如下效果. 我们就得到了一个红色的进度条.



通过上述方式, 也可以修改文字的颜色, 字体大小等样式.

### 1.4.3 伪类选择器 (Pseudo-States)

伪类选择器, 是根据控件所处的某个状态被选择的. 例如按钮被按下, 输入框获取到焦点, 鼠标移动到某个控件上等.

- 当状态具备时, 控件被选中, 样式生效.
- 当状态不具备时, 控件不被选中, 样式失效.

使用 `:<state>` 的方式定义伪类选择器.

常用的伪类选择器

伪类选择器	说明
<code>:hover</code>	鼠标放到控件上
<code>:pressed</code>	鼠标左键按下时
<code>:focus</code>	获取输入焦点时
<code>:enabled</code>	元素处于可用状态时
<code>:checked</code>	被勾选时
<code>:read-only</code>	元素为只读状态时

这些状态可以使用 `!` 来取反. 比如 `:!hover` 就是鼠标离开控件时, `:!pressed` 就是鼠标松开时, 等等.



更多伪类选择器的详细情况, 参考 Qt Style Sheets Reference 的 Pseudo-States 章节.

**代码示例:** 设置按钮的伪类样式.

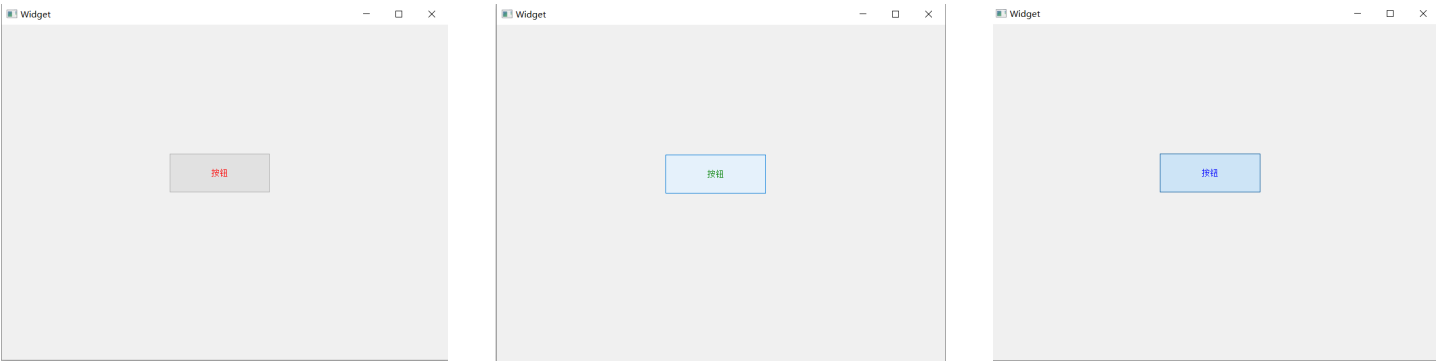
1) 在界面上创建一个按钮



2) 编写 main.cpp, 创建全局样式

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     QString style = "";
6     style += "QPushButton { color: red; }";
7     style += "QPushButton:hover { color: green; }";
8     style += "QPushButton:pressed { color: blue; }";
9     a.setStyleSheet(style);
10
11     Widget w;
12     w.show();
13     return a.exec();
14 }
```

3) 运行程序, 可以看到, 默认情况下按钮文字是红色, 鼠标移动上去是绿色, 鼠标按下按钮是蓝色.



上述代码也可以使用事件的方式来实现.

**代码示例:** 使用事件方式实现同样效果

1) 创建 `MyPushButton` 类, 继承自 `QPushButton`

C++ Class

Details

Summary

Define Class

Class name: `MyPushButton`

Base class: `<Custom>`  
`QPushButton`

☐ Include QObject

☐ Include QWidget

☐ Include QMainWindow

☐ Include QDeclarativeItem - Qt Quick 1

☐ Include QQuickItem - Qt Quick 2

☐ Include QSharedData

☐ Add Q\_OBJECT

Header file: `mypushbutton.h`

Source file: `mypushbutton.cpp`

Path: `D:\project\ke\qt\DemoCode\DemoEvent`

浏览...

下一步(N)

取消

2) 把生成代码中的构造函数改成带参数 `QWidget*` 版本的构造函数. (否则无法和 Qt Designer 生成的代码适配).

`mypushbutton.h`

```
1 #include <QPushButton>
```

```
2
3 class MyPushButton : public QPushButton
4 {
5 public:
6     MyPushButton(QWidget* parent);
7 };
```

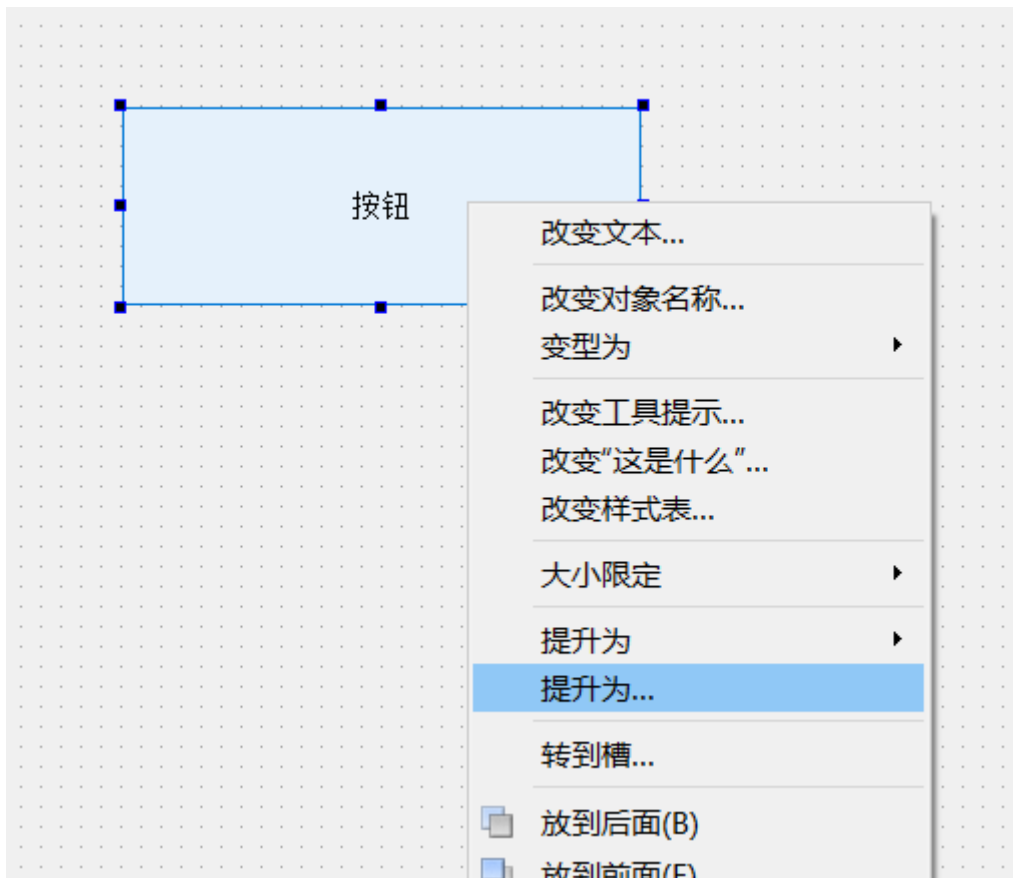
mypushbutton.cpp

```
1 #include "mypushbutton.h"
2
3 MyPushButton::MyPushButton(QWidget* parent) : QPushButton(parent)
4 {
5
6 }
```

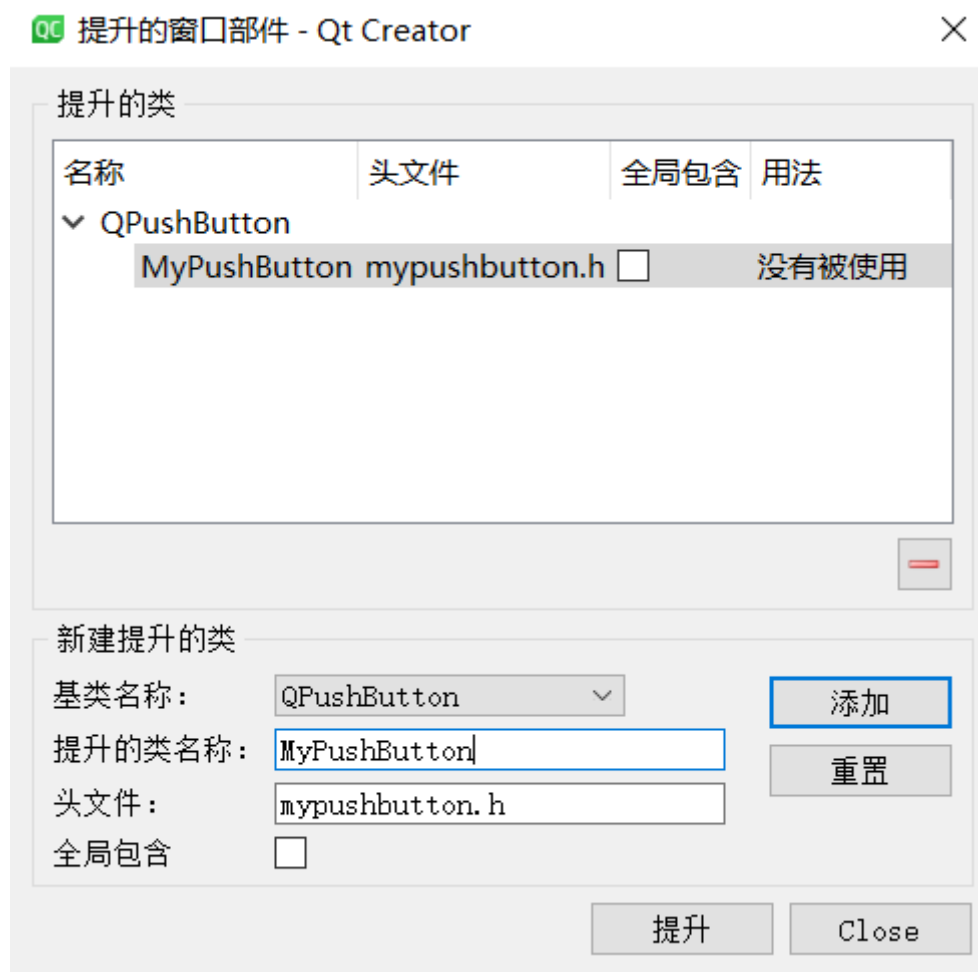
3) 在界面上创建按钮, 并提升为 `MyPushButton` 类型




右键按钮, 选择 "提升为"



填写提升的类名和头文件.



提升完毕后, 在右侧对象树这里, 就可以看到类型的变化.

对象	类
▼  Widget QWidget	
pu...on MyPushButton	

#### 4) 重写 MyPushButton 的四个事件处理函数

修改 mypushbutton.h

```

1 class MyPushButton : public QPushButton
2 {
3 public:
4     MyPushButton(QWidget* parent);
5
6     void mousePressEvent(QMouseEvent* e);
7     void mouseReleaseEvent(QMouseEvent* e);
8     void enterEvent(QEvent* e);
9     void leaveEvent(QEvent* e);
10 };

```

修改 mypushbutton.cpp

- 初始化设为红色
- 鼠标进入时设为绿色, 离开是还原红色.
- 鼠标按下时设为蓝色, 松开时还原绿色(松开时鼠标还是在按钮里).

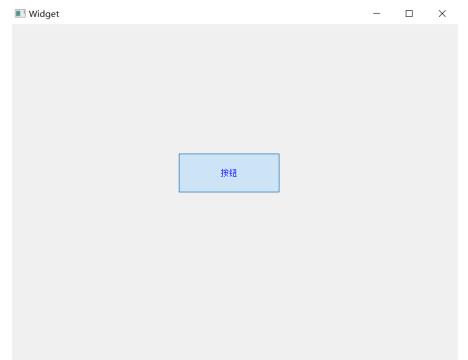
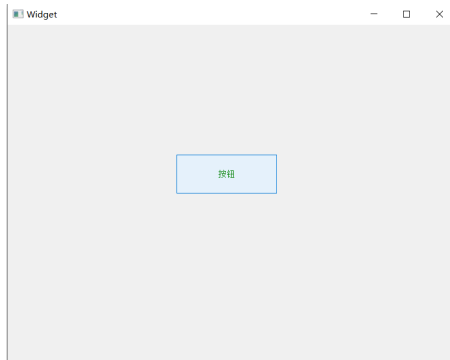
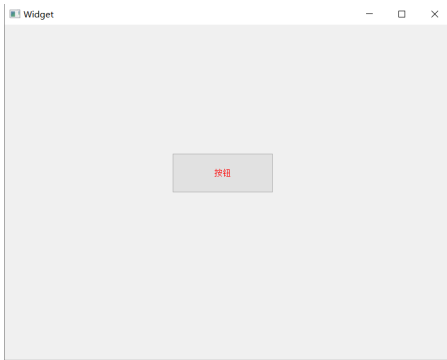
```

1 MyPushButton::MyPushButton(QWidget* parent) : QPushButton(parent)
2 {
3     this->setStyleSheet("QPushButton { color: red; }");
4 }
5
6 void MyPushButton::mousePressEvent(QMouseEvent *e)
7 {
8     this->setStyleSheet("QPushButton { color: blue; }");
9 }
10
11 void MyPushButton::mouseReleaseEvent(QMouseEvent *e)
12 {
13     this->setStyleSheet("QPushButton { color: green; }");
14 }
15
16 void MyPushButton::enterEvent(QEvent *e)
17 {
18     this->setStyleSheet("QPushButton { color: green; }");

```

```
19 }
20
21 void MyPushButton::leaveEvent(QEvent *e)
22 {
23     this->setStyleSheet("QPushButton { color: red; }");
24 }
25
```

5) 运行程序, 可以看到效果和上述案例一致.



很明显, 实现同样的功能, 伪类选择器要比事件的方式简单很多.

但是不能就说事件机制就不好. 事件可以完成的功能很多, 不仅仅是样式的改变, 还可以包含其他业务逻辑. 这一点是伪类选择器无法替代的.

## 1.5 样式属性

QSS 中的样式属性非常多, 不需要都记住. 核心原则还是用到了就去查.

大部分的属性和 CSS 是非常相似的.



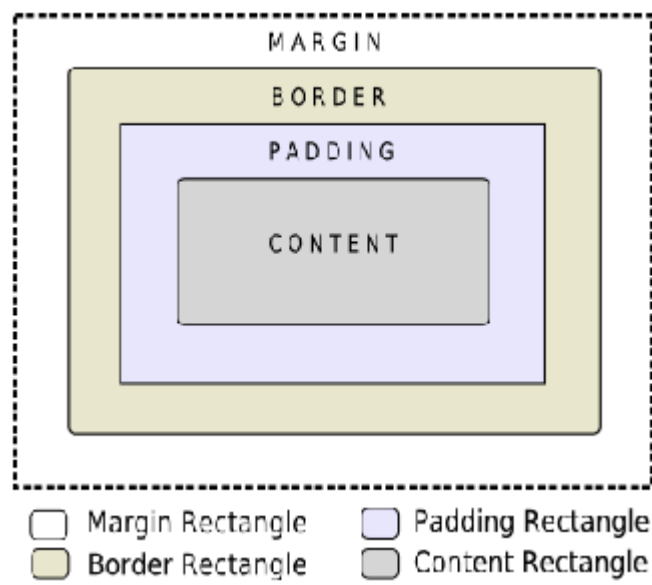
文档的 [Qt Style Sheets Reference](#) 章节详细介绍了哪些控件可以设置属性, 每个控件都能设置哪些属性等.

相关的代码示例, 在后面具体介绍.

在翻阅文档的时候涉及到一个关键术语 "盒模型" (Box Model). 这里我们需要介绍一下.

1.5.1 盒模型 (Box Model)

在文档的 Customizing Qt Widgets Using Style Sheets 的 The Box Model 章节介绍了盒模型.



一个遵守盒模型的控件, 由上述几个部分构成.

- Content 矩形区域: 存放控件内容. 比如包含的文本/图标等.
- Border 矩形区域: 控件的边框.
- Padding 矩形区域: 内边距. 边框和内容之间的距离.
- Margin 矩形区域: 外边距. 边框到控件 geometry 返回的矩形边界的距离

默认情况下, 外边距, 内边距, 边框宽度都是 0.

可以通过一些 QSS 属性来设置上述的边距和边框的样式.

QSS 属性	说明
margin	设置四个方向的外边距. 复合属性.
padding	设置四个方向的内边距. 复合属性.
border-style	设置边框样式
border-width	边框的粗细
border-color	边框的颜色

## 代码示例: 设置边框和内边距

### 1) 在界面上创建一个 label

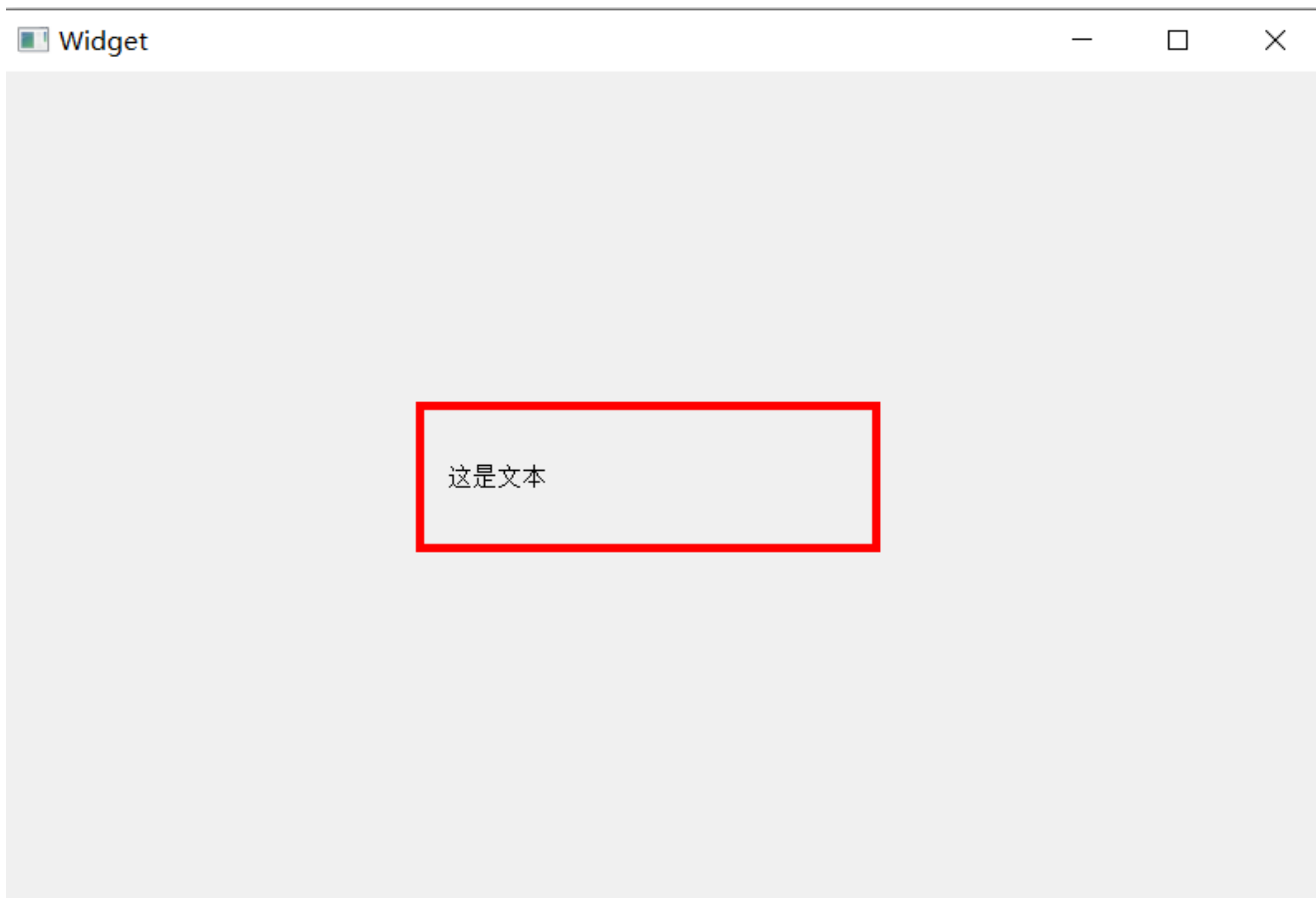


### 2) 修改 main.cpp, 设置全局样式

- `border: 5px solid red` 相当于 `border-style: solid; border-width: 5px; border-color: red;` 三个属性的简写形式.
- `padding-left: 10px;` 是给左侧设置内边距.

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4
5     a.setStyleSheet("QLabel { border: 5px solid red; padding-left: 10px; }");
6
7     Widget w;
8     w.show();
9     return a.exec();
10 }
```

### 3) 运行程序, 可以看到样式发生了变化



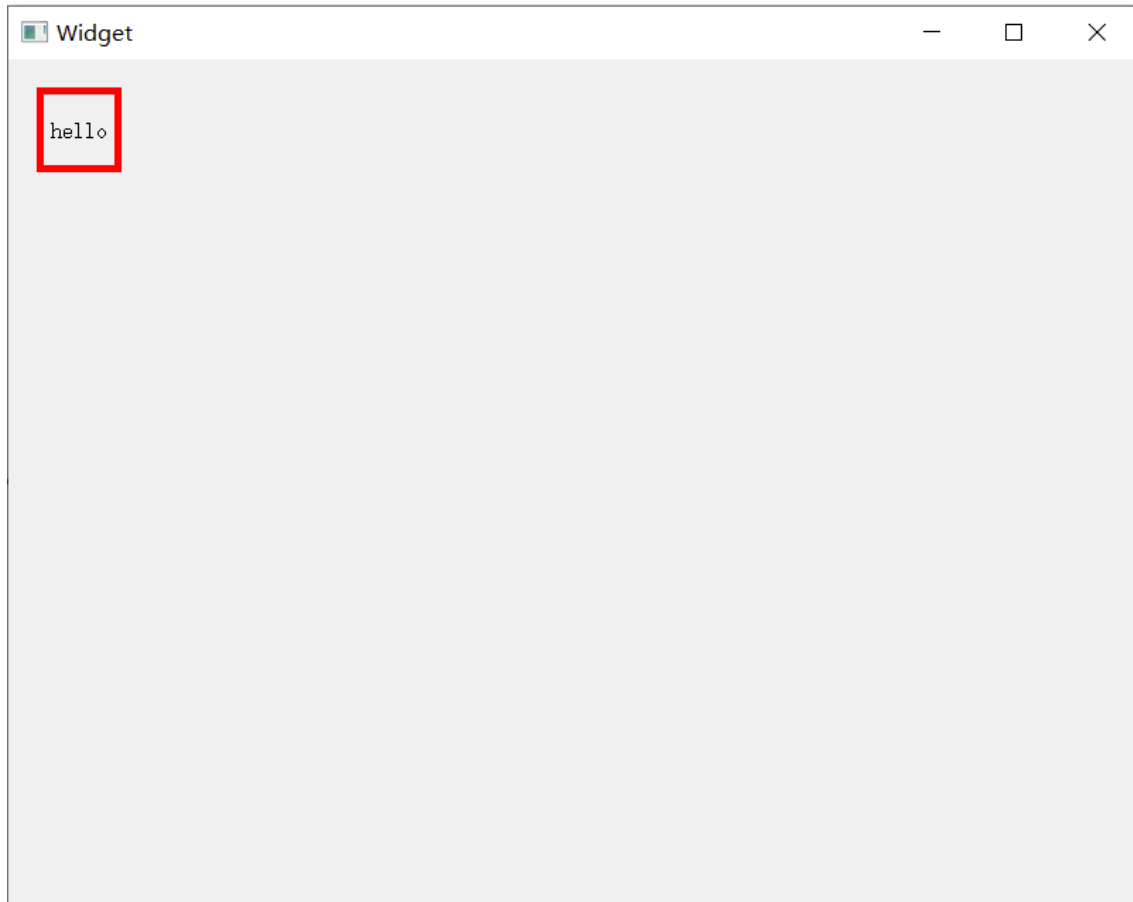
### 代码示例: 设置外边距

为了方便确定控件位置, 演示外边距效果, 我们使用代码创建一个按钮.

1) 修改 widget.cpp, 创建按钮, 并设置样式.

```
1 Widget::Widget(QWidget *parent)
2     : QWidget(parent)
3     , ui(new Ui::Widget)
4 {
5     ui->setupUi(this);
6
7     QPushButton* btn = new QPushButton(this);
8     btn->setGeometry(0, 0, 100, 100);
9     btn->setText("hello");
10    btn->setStyleSheet("QPushButton { border: 5px solid red; margin: 20px; }");
11
12    const QRect& rect = btn->geometry();
13    qDebug() << rect;
14 }
```

2) 运行程序, 可以看到, 当前按钮的边框被外边距挤的缩小了. 但是获取到的按钮的 `Geometry` 是不变的.



```
QRect(0,0 100x100)
```

## 1.6 控件样式示例

下面给出一些常用控件的样式示例.

### 1.6.1 按钮

**代码示例:** 自定义按钮

1) 界面上创建一个按钮



2) 右键 -> 改变样式表, 使用 Qt Designer 设置样式

```
1 QPushButton {  
2     font-size: 20px;  
3     border: 2px solid #8f8f91;  
4     border-radius: 15px;  
5     background-color: #dadbde;  
6 }  
7  
8 QPushButton:pressed {  
9     background-color: #f6f7fa;  
10 }
```

3) 执行程序, 可以看到效果



按钮点击之前



按钮点击之后

属性小结

属性	说明
font-size	设置文字大小.
border-radius	设置圆角矩形. 数值设置的越大, 角就 "越圆".
background-color	设置背景颜色.

🏐 形如 `#daddbde` 是计算机中通过十六进制表示颜色的方式.

这里在 "常用控件" 章节已经介绍过了. 此处不再赘述.

另外, 在实际开发中, 颜色具体使用哪种好看, 是需要一定的 "艺术细菌" 的. 往往公司会有专门的美工/设计人员来负责. 程序猿只需要按照设计稿实现程序即可.

### 1.6.2 复选框

#### 代码示例: 自定义复选框

1) 创建一个 `resource.qrc` 文件, 并导入以下图片.

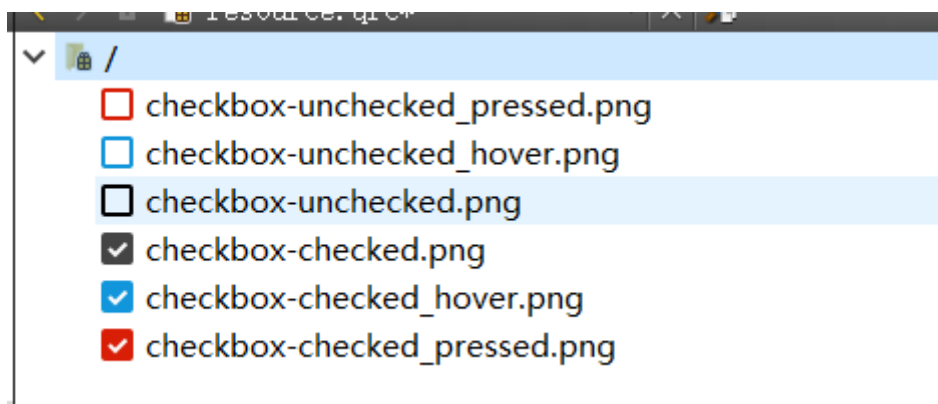


使用黑色作为默认形态.

使用蓝色作为 hover 形态.

使用红色作为 pressed 形态.

注意这里的文件命名.



🏐 使用阿里矢量图标库, 可以下载到上述图片.

下载的时候可以手动选择颜色.

## 2) 创建一个复选框



## 3) 编辑复选框的样式

```
1 QCheckBox {
2     font-size: 20px;
3 }
4
5 QCheckBox::indicator {
6     width: 20px;
7     height: 20px;
8 }
9
10 QCheckBox::indicator:unchecked {
11     image: url(:/checkbox-unchecked.png);
12 }
13
14 QCheckBox::indicator:unchecked:hover {
15     image: url(:/checkbox-unchecked_hover.png);
16 }
```

```
17
18 QCheckBox::indicator:unchecked:pressed {
19     image: url(/checkbox-unchecked_pressed.png);
20 }
21
22 QCheckBox::indicator:checked {
23     image: url(/checkbox-checked.png);
24 }
25
26 QCheckBox::indicator:checked:hover {
27     image: url(/checkbox-checked_hover.png);
28 }
29
30 QCheckBox::indicator:checked:pressed {
31     image: url(/checkbox-checked_pressed.png);
32 }
```

4) 运行程序, 可以看到此时的复选框就变的丰富起来了.



此处截图看不出来效果. 大家可以自行验证.

小结:

要点	说明
::indicator	子控件选择器. 选中 checkbox 中的对钩部分.
:hover	伪类选择器.

	选中鼠标移动上去的状态.
:pressed	伪类选择器. 选中鼠标按下的状态.
:checked	伪类选择器. 选中 checkbox 被选中的状态.
:unchecked	伪类选择器. 选中 checkbox 未被选中的状态.
width	设置子控件宽度. 对于普通控件无效 (普通控件使用 geometry 方式设定尺寸).
height	设置子控件高度. 对于普通控件无效 (普通控件使用 geometry 方式设定尺寸).
image	设置子控件的图片. 像 QSpinBox, QComboBox 等可以使用这个属性来设置子控件的图片.

1.6.3 单选框

代码示例: 自定义单选框

1) 创建 `resource.qrc` 文件, 并导入以下图片.

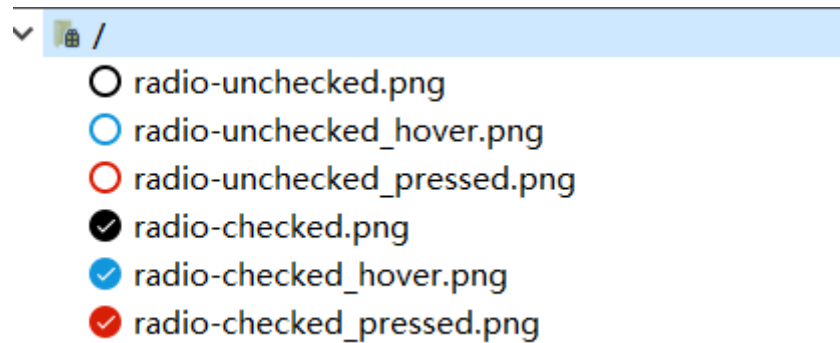


使用黑色作为默认形态.

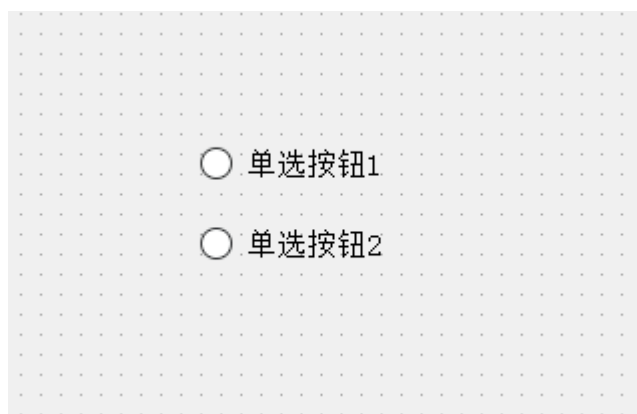
使用蓝色作为 hover 形态.

使用红色作为 pressed 形态.

注意这里的文件命名.



2) 在界面上创建两个单选按钮



3) 在 Qt Designer 中编写样式

此处为了让所有 QRadioButton 都能生效, 把样式设置在 `Widget` 上了. 并且使用后代选择器选中了 `QWidget` 里面的 `QRadioButton`.

#### 注意!

QSS 中有些属性, 子元素能继承父元素(例如 font-size, color 等). 但是也有很多属性是不能继承的.

具体哪些能继承哪些不能继承, 规则比较复杂, 咱们不去具体研究. 实践中我们编写更精准的选择器是上策.

```
1 QWidget QRadioButton {
2     font-size: 20px;
3 }
4
5 QWidget QRadioButton::indicator {
6     width: 20px;
7     height: 20px;
8 }
9
10 QWidget QRadioButton::indicator:unchecked {
```

```

11     image: url(:/radio-unchecked.png);
12 }
13
14 QWidget QRadioButton::indicator:unchecked:hover {
15     image: url(:/radio-unchecked_hover.png);
16 }
17
18 QWidget QRadioButton::indicator:unchecked:pressed {
19     image: url(:/radio-unchecked_pressed.png);
20 }
21
22 QWidget QRadioButton::indicator:checked {
23     image: url(:/radio-checked.png);
24 }
25
26 QWidget QRadioButton::indicator:checked:hover {
27     image: url(:/radio-checked_hover.png);
28 }
29
30 QWidget QRadioButton::indicator:checked:pressed {
31     image: url(:/radio-checked_pressed.png);
32 }

```

#### 4) 运行程序, 观察效果



此处截图看不出来效果. 大家可以自行验证.

要点	说明
::indicator	子控件选择器.

	选中 radioButton 中的对钩部分.
:hover	伪类选择器. 选中鼠标移动上去的状态.
:pressed	伪类选择器. 选中鼠标按下的状态.
:checked	伪类选择器. 选中 radioButton 被选中的状态.
:unchecked	伪类选择器. 选中 radioButton 未被选中的状态.
width	设置子控件宽度. 对于普通控件无效 (普通控件使用 geometry 方式设定尺寸).
height	设置子控件高度. 对于普通控件无效 (普通控件使用 geometry 方式设定尺寸).
image	设置子控件的图片. 像 QSpinBox, QComboBox 等可以使用这个属性来设置子控件的图片.

1.6.4 输入框

代码示例: 自定义单行编辑框

1) 在界面上创建一个单行编辑框



2) 在 Qt Designer 中编写样式.

```
1 QLineEdit {
2     border-width: 1px;
3     border-radius: 10px;
4     border-color: rgb(58, 58, 58);
5     border-style: inset;
6     padding: 0 8px;
7     color: rgb(255, 255, 255);
8     background:rgb(100, 100, 100);
9     selection-background-color: rgb(187, 187, 187);
10    selection-color: rgb(60, 63, 65);
11 }
```

3) 执行程序观察效果.



属性	说明
border-width	设置边框宽度.
border-radius	设置边框圆角.
border-color	设置边框颜色.
border-style	设置边框风格.
padding	设置内边距.
color	设置文字颜色.
background	设置背景颜色.
selection-background-color	设置选中文字的背景颜色.

selection-color	设置选中文字的文本颜色.
-----------------	--------------

## 1.6.5 列表

### 代码示例: 自定义列表框

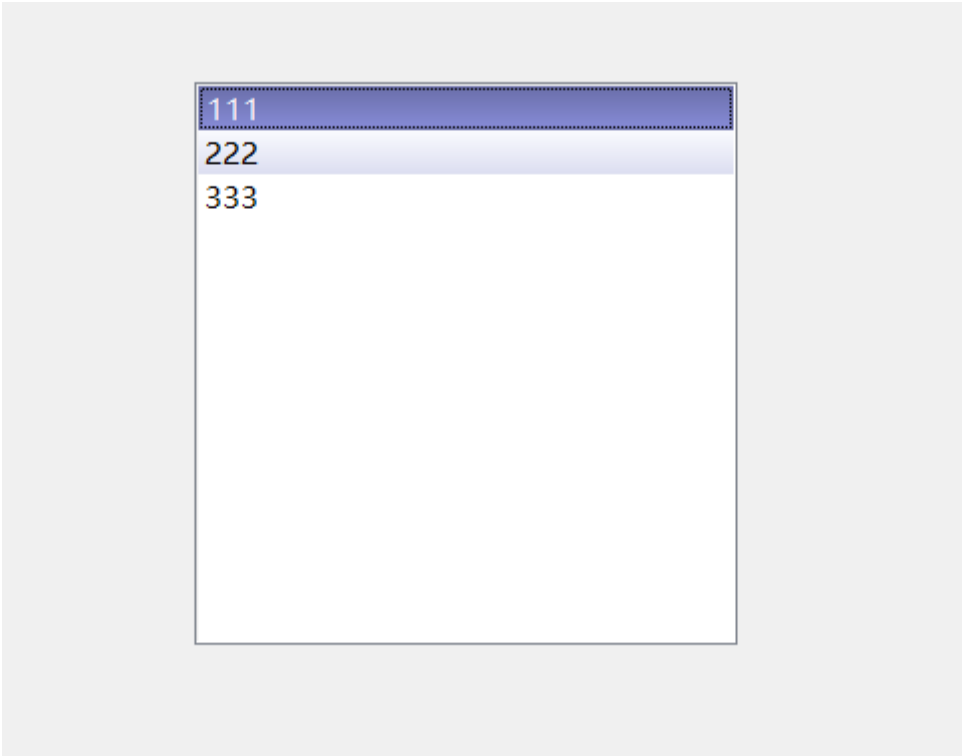
#### 1) 在界面上创建一个 ListView



#### 2) 编写代码

```
1  QListView::item: hover {
2      background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
3                                  stop: 0 #FAFBFE, stop: 1 #DCDEF1);
4  }
5
6
7
8  QListView::item: selected {
9      border: 1px solid #6a6ea9;
10     background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
11                                 stop: 0 #6a6ea9, stop: 1 #888dd9);
12 }
```

3) 执行程序, 观察效果



要点	说明
::item	选中 QListView 中的具体条目.
:hover	选中鼠标悬停的条目
:selected	选中某个被选中的条目.
background	设置背景颜色
border	设置边框.
qlineargradient	设置渐变色.



关于 `qlineargradient`

`qlineargradient` 有 6 个参数.

`x1, y1`: 标注了一个起点.

`x2, y2`: 标注了一个终点.

这两个点描述了一个 "方向".

例如:

- `x1: 0, y1: 0, x2: 0, y2: 1` 就是垂直方向从上向下 进行颜色渐变.

- x1: 0, y1: 0, x2: 1, y2: 0 就是水平方向从左向右 进行颜色渐变.
- x1: 0, y1: 0, x2: 1, y2: 1 就是从左上往右下方向 进行颜色渐变.

stop0 和 stop1 描述了两个颜色. 渐变过程就是从 stop0 往 stop1 进行渐变的.

### 代码例子: 理解渐变色

1) 界面不创建任何控件

2) 编写样式

```
1 QWidget {  
2     background-color: qlineargradient(x1:0, y1:0, x2:0, y2:1, stop: 0 #fff,  
    stop: 1 #000);  
3 }
```

当前按照 **垂直从上往下** 从 白色 过渡到 黑色

执行效果



### 3) 修改代码

```
1 QWidget {  
2     background-color: qlineargradient(x1:0, y1:0, x2:1, y2:0, stop: 0 #fff,  
    stop: 1 #000);  
3 }
```

当前按照 **水平从左往右** 从 白色 过渡到 黑色

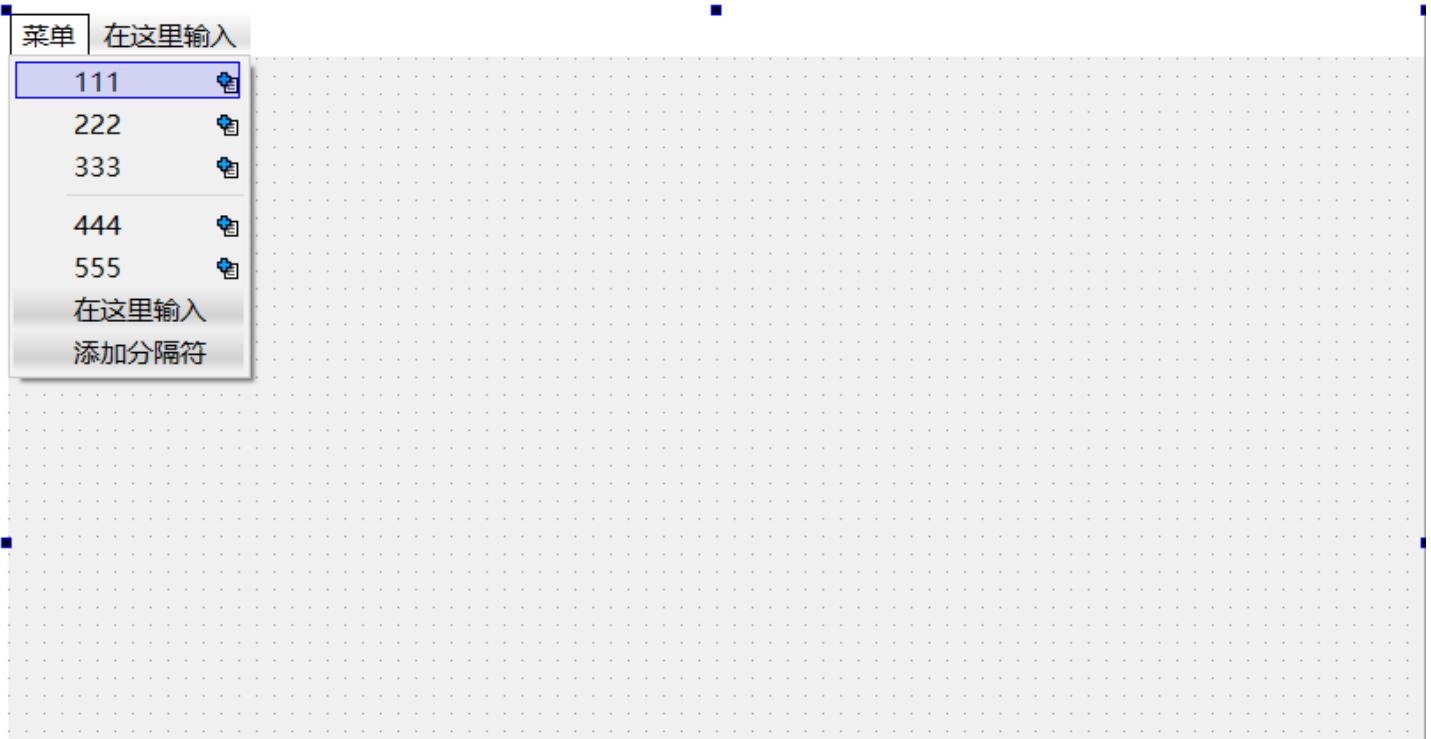
执行效果



### 1.6.6 菜单栏

**代码示例:** 自定义菜单栏

1) 创建菜单栏



创建若干菜单项和一个分隔符.

## 2) 编写样式

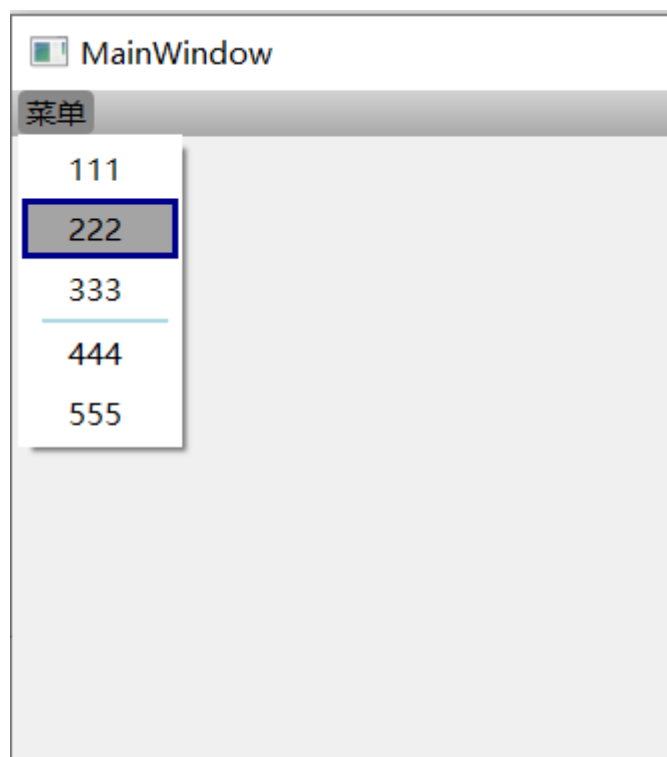
```
1 QMenuBar {
2     background-color: qlineargradient(x1:0, y1:0, x2:0, y2:1,
3                                     stop:0 lightgray, stop:1 darkgray);
4     spacing: 3px; /* spacing between menu bar items */
5 }
6
7 QMenuBar::item {
8     padding: 1px 4px;
9     background: transparent;
10    border-radius: 4px;
11 }
12
13 QMenuBar::item:selected { /* when selected using mouse or keyboard */
14     background: #a8a8a8;
15 }
16
17 QMenuBar::item:pressed {
18     background: #888888;
19 }
20
21 QMenu {
22     background-color: white;
23     margin: 0 2px; /* some spacing around the menu */
```

```

24 }
25
26 QMenu::item {
27     padding: 2px 25px 2px 20px;
28     border: 3px solid transparent; /* reserve space for selection border */
29 }
30
31 QMenu::item:selected {
32     border-color: darkblue;
33     background: rgba(100, 100, 100, 150);
34 }
35
36
37 QMenu::separator {
38     height: 2px;
39     background: lightblue;
40     margin-left: 10px;
41     margin-right: 5px;
42 }

```

### 3) 执行程序, 观察效果



要点	说明
QMenuBar::item	选中菜单栏中的元素.

QMenuBar::item:selected	选中菜单来中的被选中的元素.
QMenuBar::item:pressed	选中菜单栏中的鼠标点击的元素.
QMenu::item	选中菜单中的元素
QMenu::item:selected	选中菜单中的被选中的元素.
QMenu::separator	选中菜单中的分割线.

### 1.6.7 登录界面

基于上述学习过的 QSS 样式, 制作一个美化版本的登录界面.

1) 在界面上创建元素



2) 使用布局管理器, 把上述元素包裹一下.



- 使用 `QVBoxLayout` 来管理上述控件.
- 两个输入框和按钮的 `minimumHeight` 均设置为 50. (元素在布局管理器中无法直接设置 `width` 和 `height`, 使用 `minimumWidth` 和 `minimumHeight` 代替, 此时垂直方向的 `sizePolicy` 要设为 `fixed`).
- 右键 `QCheckBox`, 选择 `Layout Alignment` 可以设置 checkbox 的对齐方式(左对齐, 居中对齐, 右对齐).

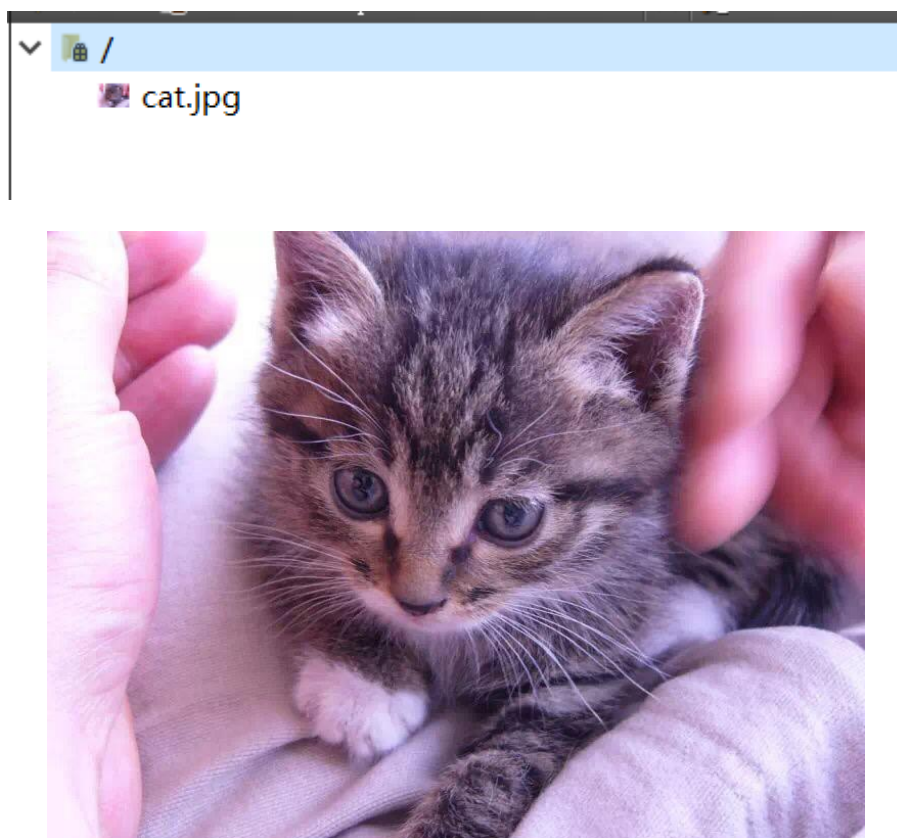
### 3) 设置背景图片.

把上述控件添加一个父元素 `QFrame`, 并设置 `QFrame` 和 窗口一样大.

对象	类
Widget	QWidget
frame	QFrame
verticalLayout	QVBoxLayout
checkbox	QCheckBox
lineEdit	QLineEdit
lineEdit_2	QLineEdit
pushButton	QPushButton

顶层窗口的 `QWidget` 无法设置背景图片. 因此我们需要再套上一层 `QFrame`. 背景图片就设置到 `QFrame` 上即可.

创建 resource.qrc, 并导入图片

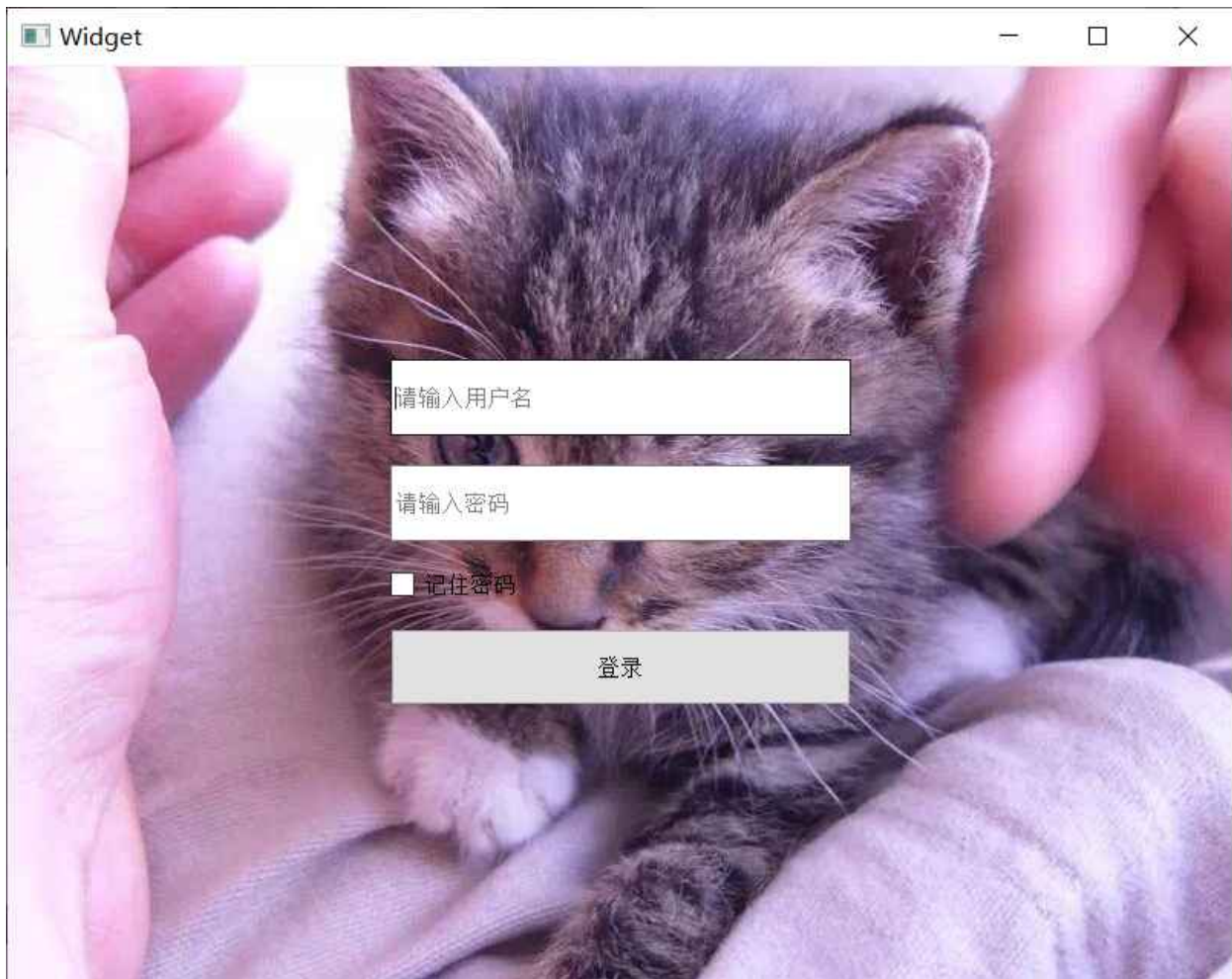


编写 QSS 样式.

- 使用 `border-image` 设置背景图片, 而不是 `background-image`. 主要是因为 `border-image` 是可以自动缩放的. 这一点在窗口大小发生改变时是非常有意义的.

```
1 QFrame {  
2     border-image: url(:/cat.jpg);  
3 }
```

此时效果为



#### 4) 设置输入框样式

编写 QSS 代码

```
1 QLineEdit {  
2     color: #8d98a1;  
3     background-color: #405361;  
4     padding: 0 5px;  
5     font-size: 20px;  
6     border-style: none;  
7     border-radius: 10px;  
8 }
```

运行程序效果:

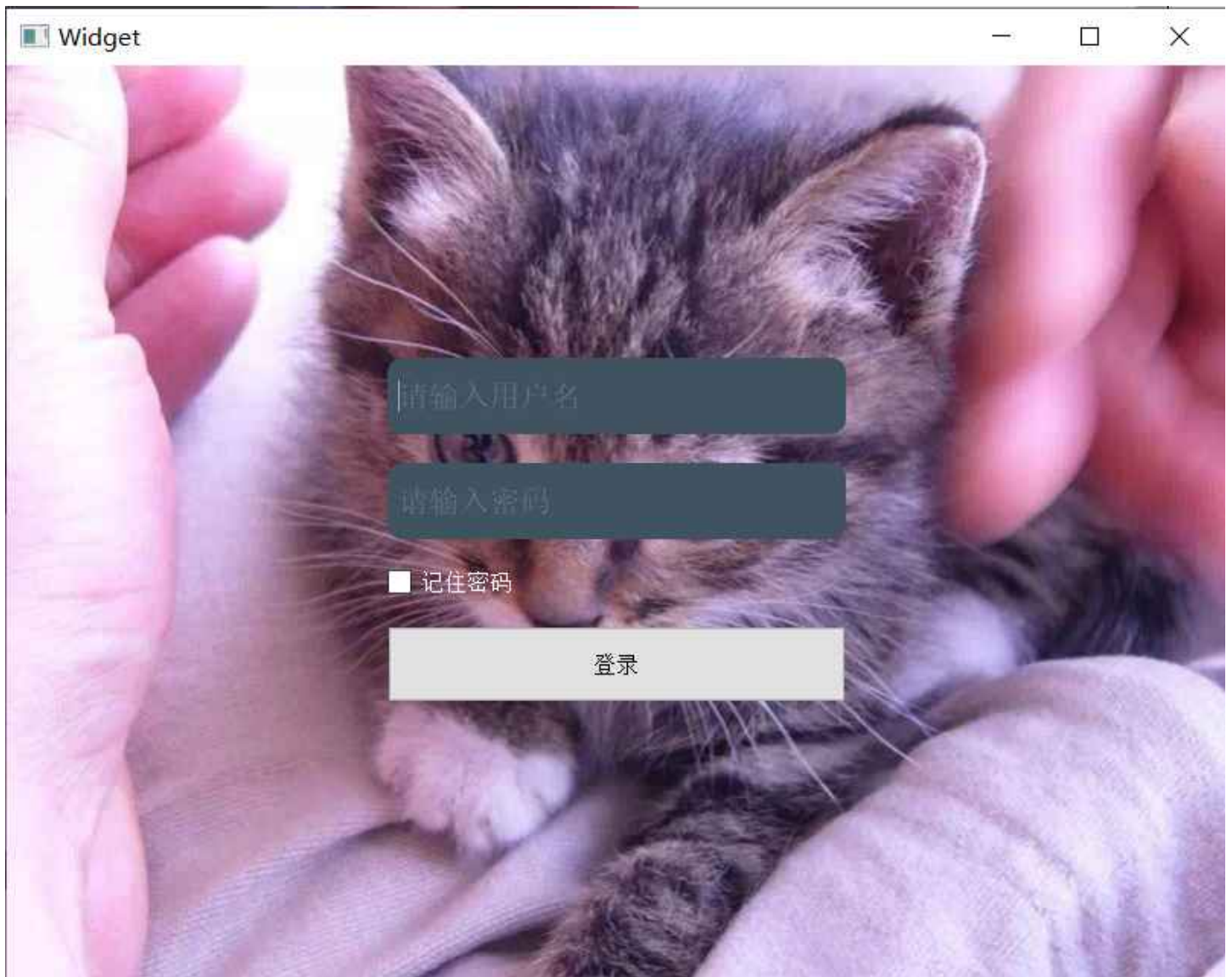


## 5) 设置 checkbox 样式

- 背景色使用 `transparent` 表示完全透明 (应用父元素的背景).

```
1 QCheckBox {  
2     color: white;  
3     background-color: transparent;  
4 }
```

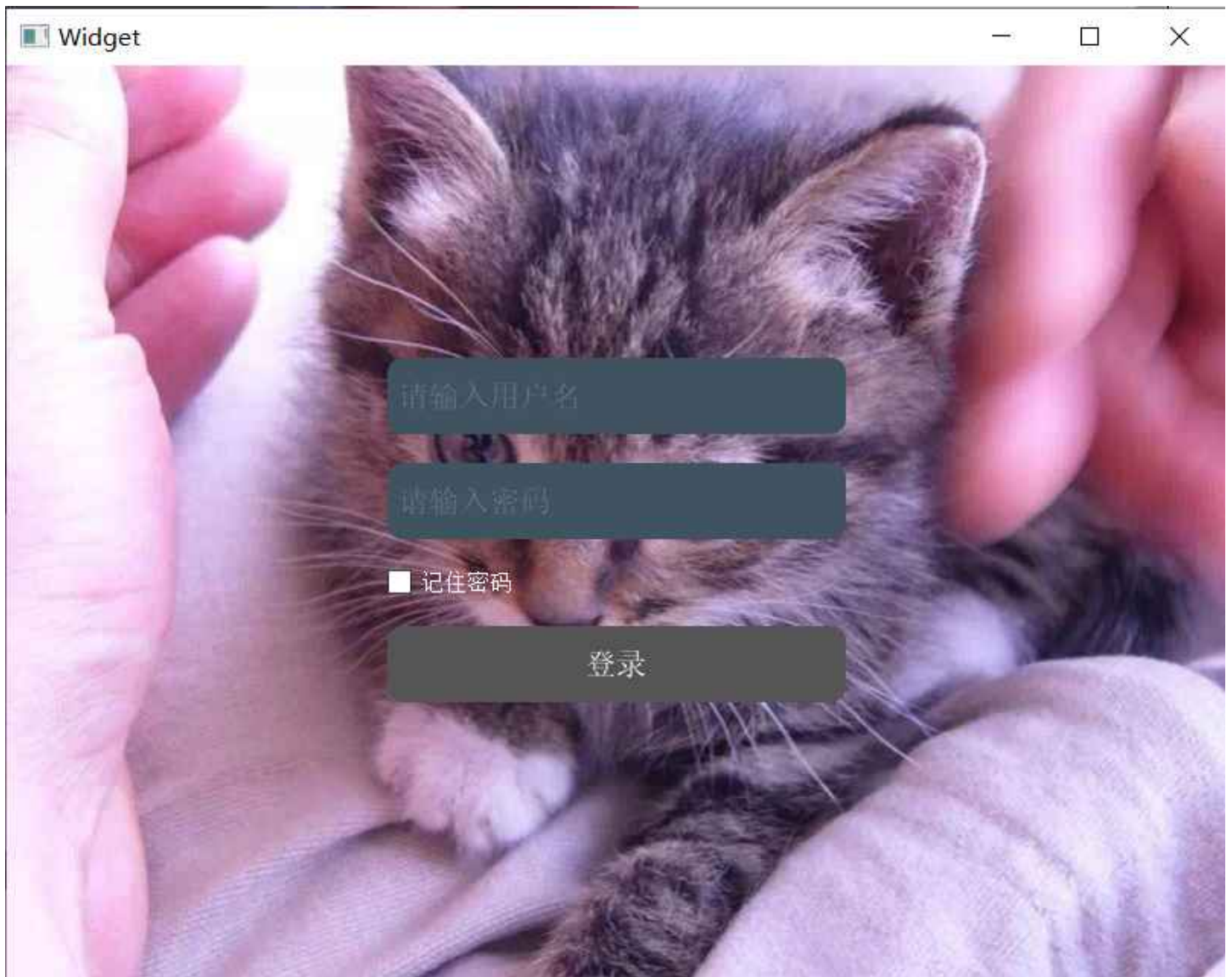
执行效果



## 6) 设置按钮样式

```
1 QPushButton {  
2     font-size: 20px;  
3     color: white;  
4     background-color: #555;  
5     border-style: outset;  
6     border-radius: 10px;  
7 }  
8  
9 QPushButton:pressed {  
10     color: black;  
11     background-color: #ced1db;  
12     border-style: inset;  
13 }
```

执行程序



最终完整样式代码. 这些代码设置到 QFrame 的属性中即可.

通常我们建议把样式代码集中放置, 方便调整和排查.

```
1 QFrame {
2     border-image: url(../cat.jpg);
3 }
4
5 QLineEdit {
6     color: #8d98a1;
7     background-color: #405361;
8     padding: 0 5px;
9     font-size: 20px;
10    border-style: none;
11    border-radius: 10px;
12 }
13
14 QCheckBox {
```

```
15         color: white;
16         background-color: transparent;
17     }
18
19     QPushButton {
20         font-size: 20px;
21         color: white;
22         background-color: #555;
23         border-style: outset;
24         border-radius: 10px;
25     }
26
27     QPushButton:pressed {
28         color: black;
29         background-color: #ced1db;
30         border-style: inset;
31     }
```

## 1.7 小结

QSS 本身给 Qt 提供了更丰富的样式设置的能力, 但是整体来说 QSS 的功能是不如 CSS 的.

在 CSS 中, 整个网页的样式, 都是 CSS 一手负责, CSS 功能更强大, 并且也更可控.

相比之下, Qt 中是以原生 api 为主, 来控制控件之间的尺寸, 位置等, QSS 只是起到辅助的作用.

而且 Qt 中提供一些 "组合控件" (像 QComboBox, QSpinBox 等) 内部的结构是不透明的, 此时进行一些样式设置也会存在一定的局限性.

另外, 做出好看的界面, 光靠 QSS 是不够的. 更重要的是需要专业美工做出设计稿.

因此通过 QSS 的学习, 我们的目的是了解这个技术, 而不要求大家立即就能做出非常好看的界面.

更多参考内容:

- 官方文档中的 `Qt Style Sheets Examples` 章节
- <https://github.com/GTRONICK/QSS>


## 2. 绘图

## 2.1 基本概念

虽然 Qt 已经内置了很多的控件, 但是不能保证现有控件就可以应对所有场景.

很多时候我们需要更强的 "自定义" 能力.

Qt 提供了画图相关的 API, 可以允许我们在窗口上绘制任意的图形形状, 来完成更复杂的界面设计.

 所谓的 "控件", 本质上也是通过画图的方式画上去的.


画图 API 和 控件 之间的关系, 可以类比成机器指令和高级语言之间的关系.

控件是对画图 API 的进一步封装; 画图 API 是控件的底层实现.

### 绘图 API 核心类

类	说明
QPainter	"绘画者" 或者 "画家".  用来绘图的对象, 提供了一系列 drawXXX 方法, 可以允许我们绘制各种图形.
QPaintDevice	"画板".  描述了 QPainter 把图形画到哪个对象上. 像咱们之前用过的 QWidget 也是一种 QPaintDevice (QWidget 是 QPaintDevice 的子类).
QPen	"画笔".  描述了 QPainter 画出来的线是什么样的.
QBrush	"画刷".  描述了 QPainter 填充一个区域是什么样的.

绘图 API 的使用, 一般不会在 QWidget 的构造函数中使用, 而是要放到 `paintEvent` 事件中.

 关于 `paintEvent`

paintEvent 会在以下情况下被触发:

- 控件首次创建.
- 控件被遮挡, 再解除遮挡.
- 窗口最小化, 再恢复.

- 控件大小发生变化时.
- 主动调用 `repaint()` 或者 `update()` 方法. (这两个方法都是 `QWidget` 的方法).
- .....

因此, 如果把绘图 api 放到构造函数中调用, 那么一旦出现上述的情况, 界面的绘制效果就无法确保符合预期了.

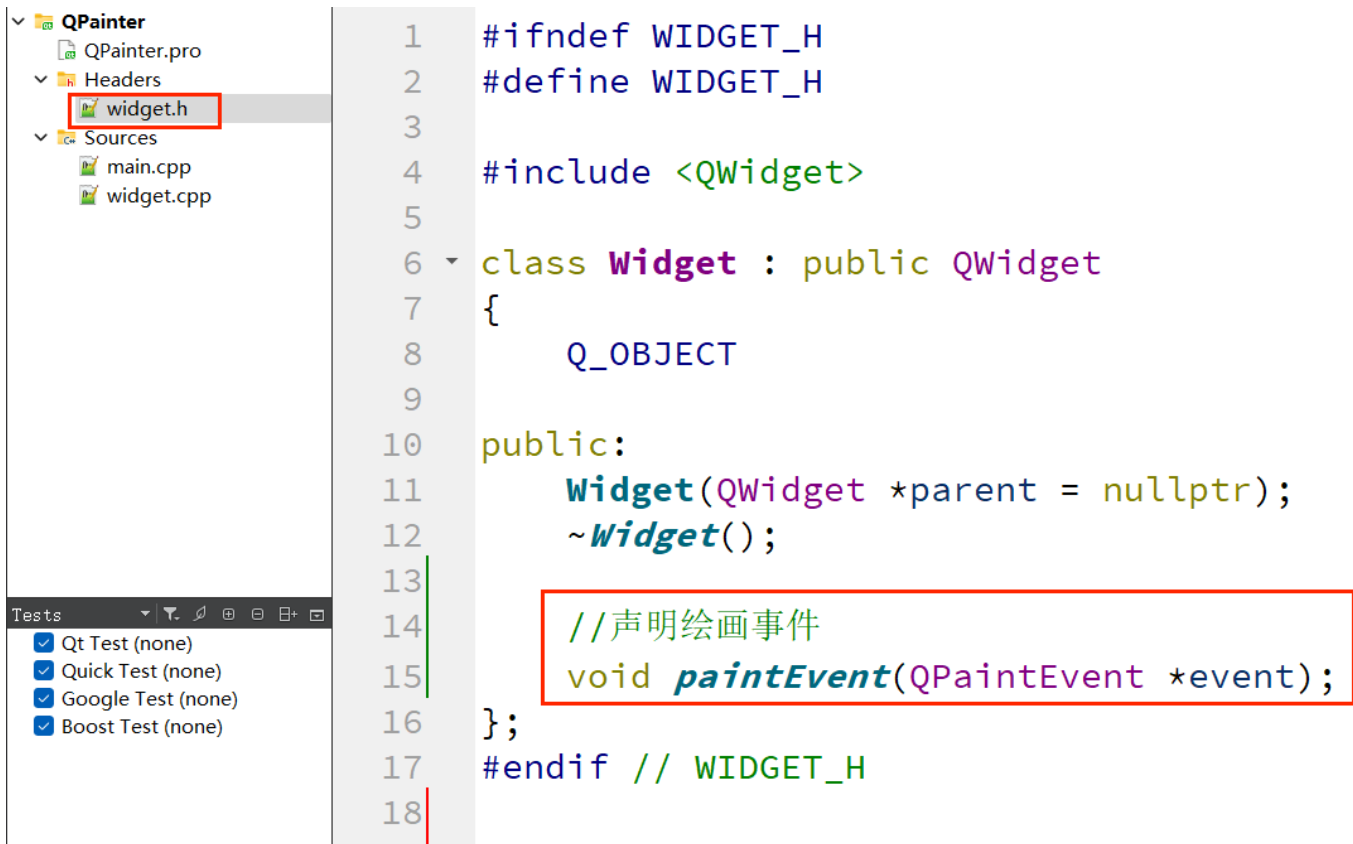
## 2.2 绘制各种形状

### 2.2.1 绘制线段

示例1:

```
1 void drawLine(const QPoint &p1, const QPoint &p2);  
2     参数:  
3     p1: 绘制起点坐标  
4     p2: 绘制终点坐标
```

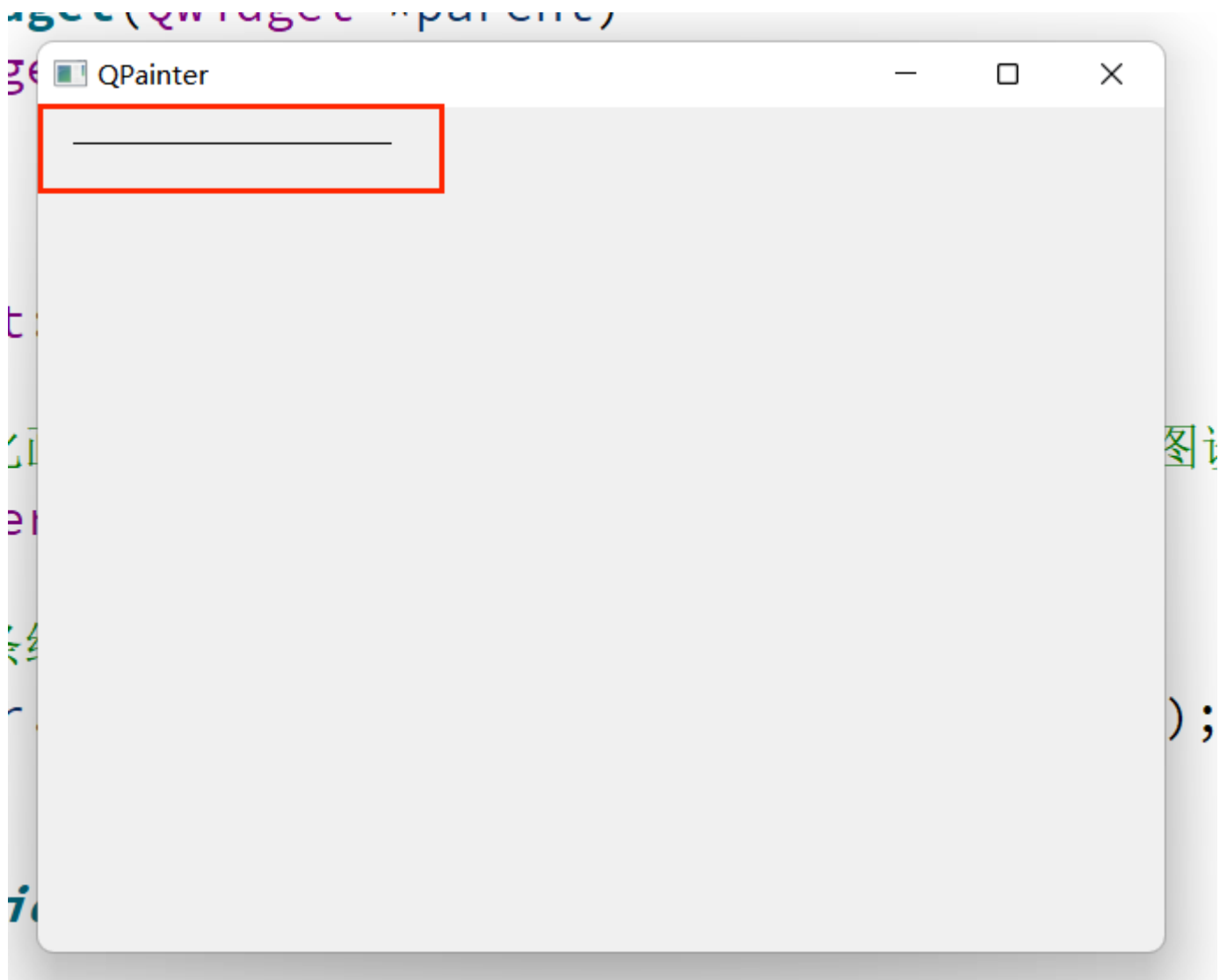
1、在 "widget.h" 头文件中声明绘图事件;



2、在 "widget.cpp" 文件中重写 paintEvent() 方法；



实现效果如下：



## 示例2:

```
1 void drawLine ( int x1, int y1, int x2, int y2 );
2 参数:
3     x1,y1: 绘制起点坐标
4     x2,y2: 绘制终点坐标
```



## 2.2.2 绘制矩形

```
1 void QPainter::drawRect(int x, int y, int width, int height);
2 参数:
3   x: 窗口横坐标;
4   y: 窗口纵坐标;
5   width: 所绘制矩形的宽度;
6   height: 所绘制矩形的高度;
```

示例:



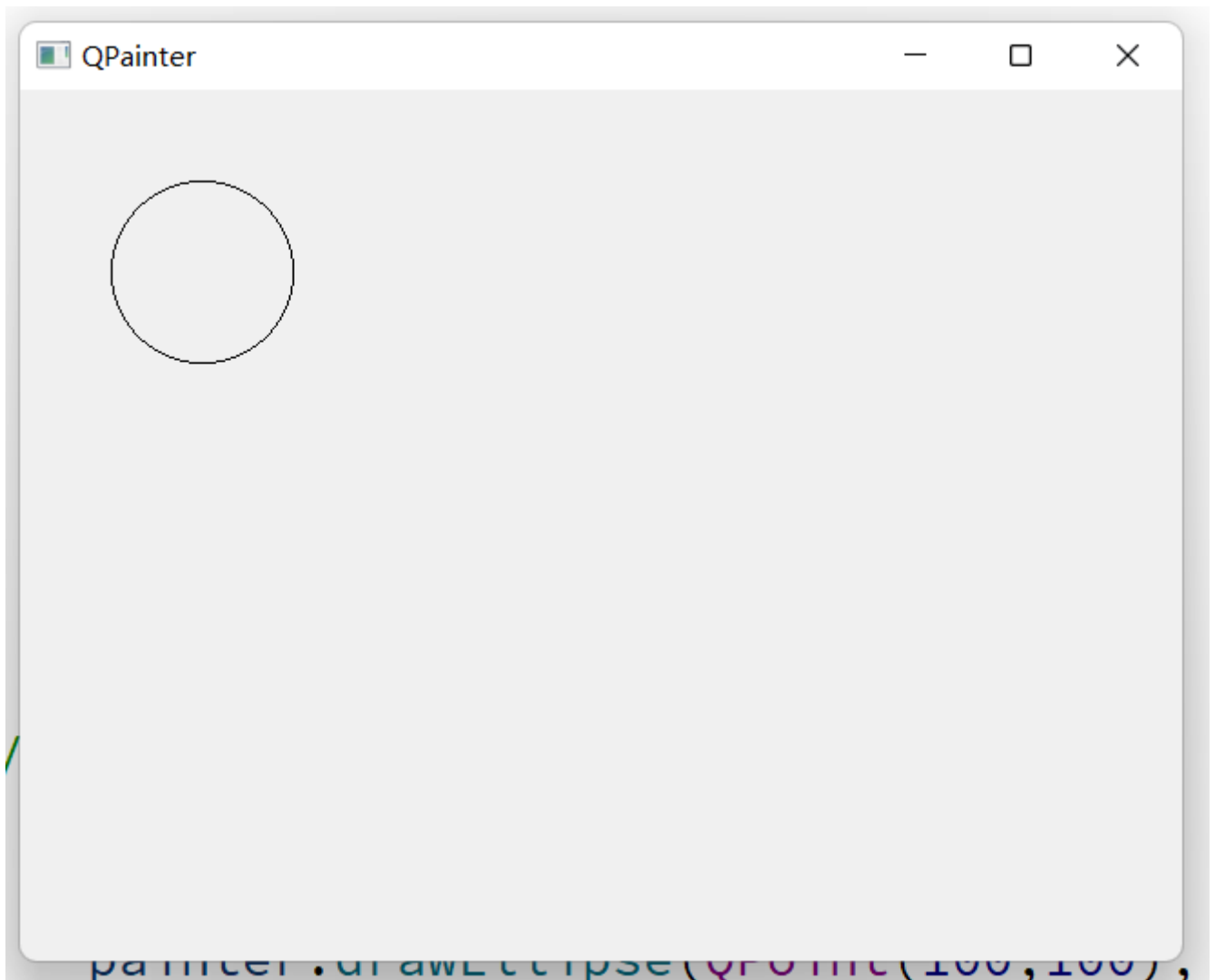
### 2.2.3 绘制圆形

```
1 void QPainter::drawEllipse(const QPoint &center, int rx, int ry)
2 参数:
3     center: 中心点坐标
4     rx: 横坐标
5     ry: 纵坐标
```

示例:



实现效果:



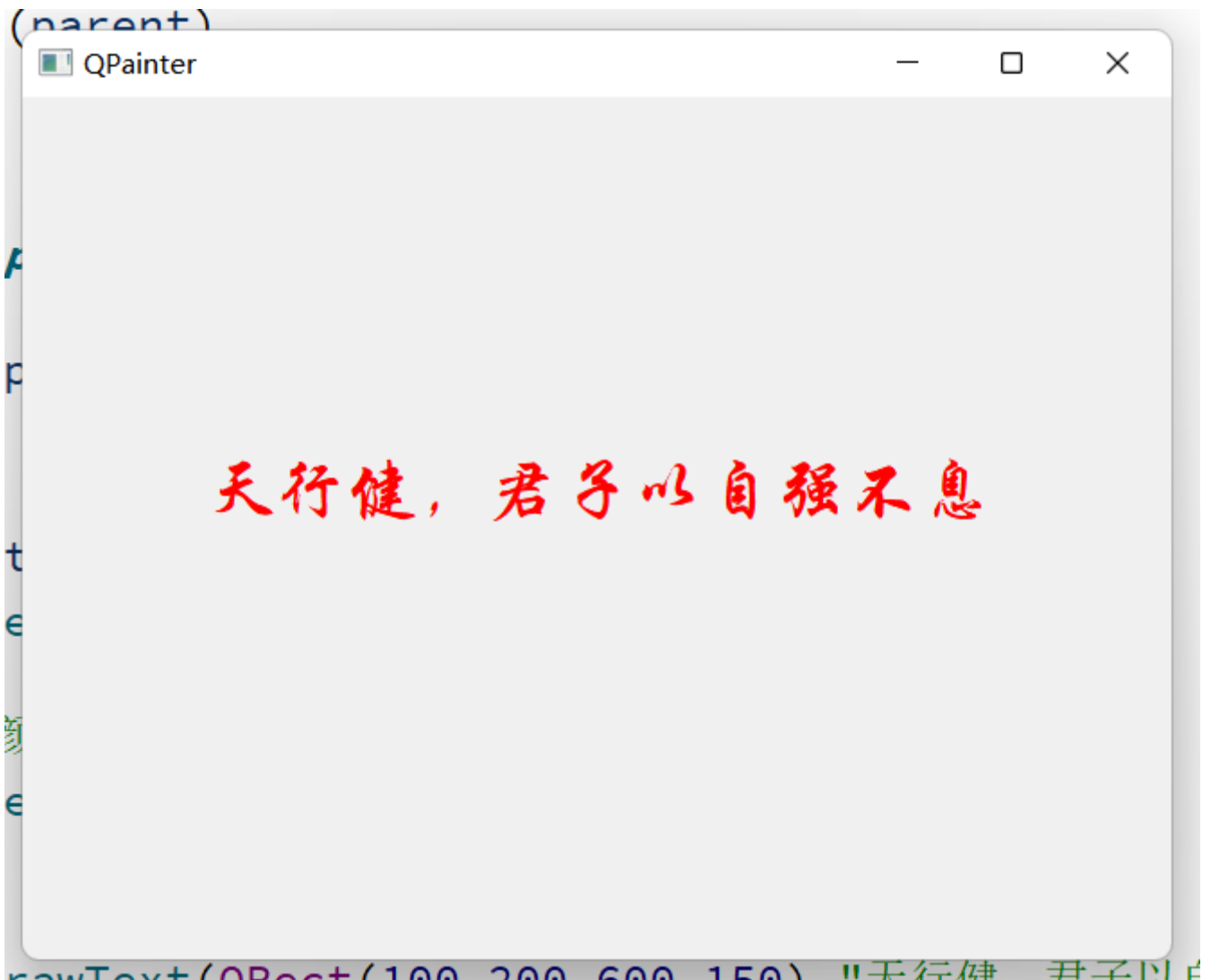
#### 2.2.4 绘制文本

QPainter类 中不仅提供了绘制图形的功能，还可以使用 QPainter::drawText() 函数来绘制文字，也可以使用QPainter::setFont() 设置字体等信息。

示例：



实现效果如下：



2.2.5 设置画笔

QPainter 在绘制时，是有一个默认的画笔的。在使用时也可以自定义画笔。在 Qt 中，**QPen类**中定义了 QPainter 应该如何绘制形状、线条和轮廓。同时通过 **QPen类** 可以设置画笔的**线宽、颜色、样式、画刷**等。


画笔的颜色可以在实例化画笔对象时进行设置，画笔的宽度是通过 **setWidth()** 方法进行设置，画笔的风格是通过**setStyle()** 方法进行设置，设置画刷主要是通过 **setBrush()** 方法。


- 设置画笔颜色：**QPen::QPen(const QColor &color)**    画笔的颜色主要是通过 **QColor** 类设置；
- 设置画笔宽度：**void QPen::setWidth(int width)**
- 设置画笔风格：**void QPen::setStyle(Qt::PenStyle style)**


画笔的风格有：


enum Qt::PenStyle


This enum type defines the pen styles that can be drawn using QPainter. The styles are:


Qt::SolidLine

Qt::DashLine

Qt::DotLine

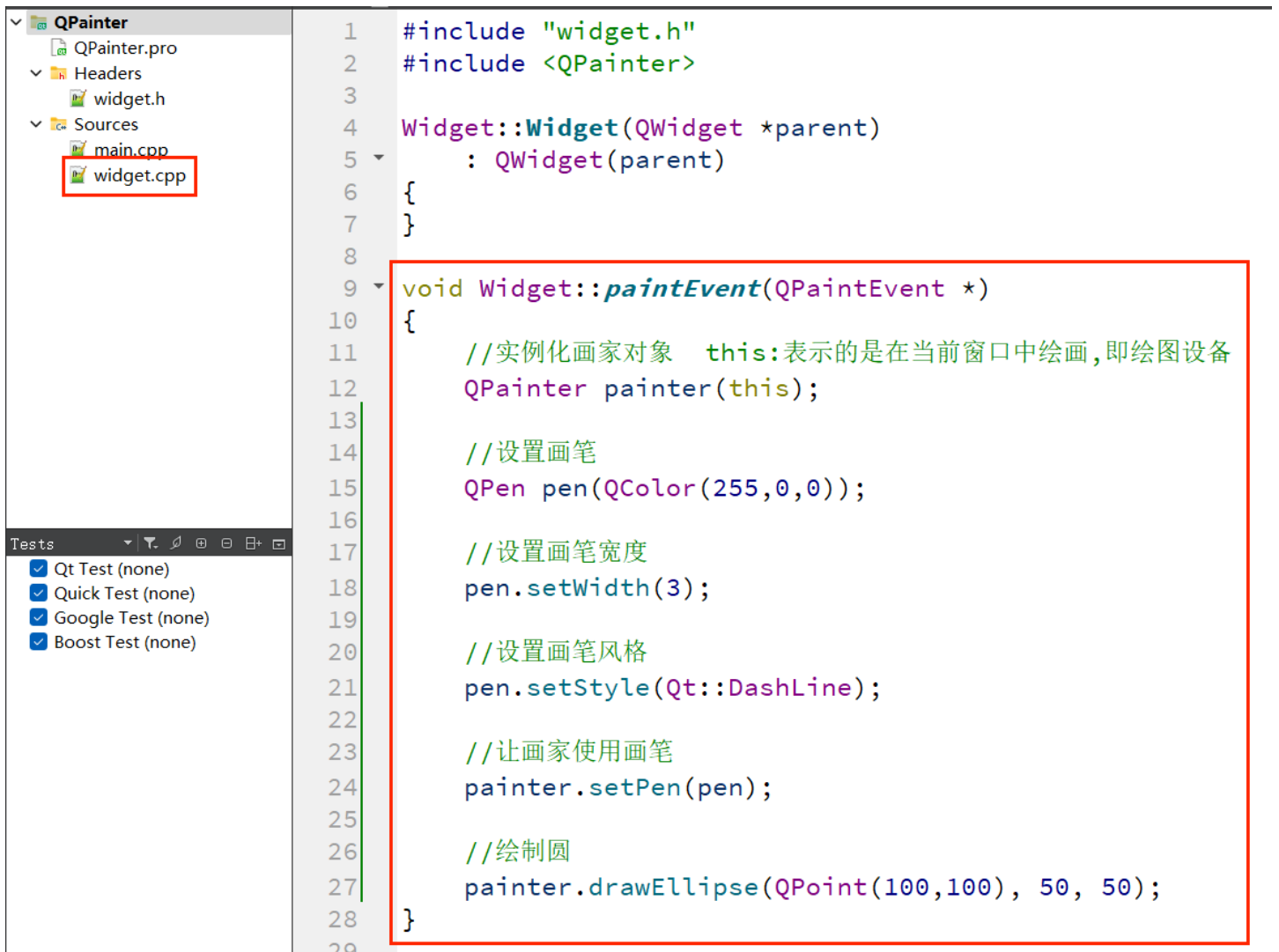
Qt::DashDotLine

Qt::DashDotDotLine

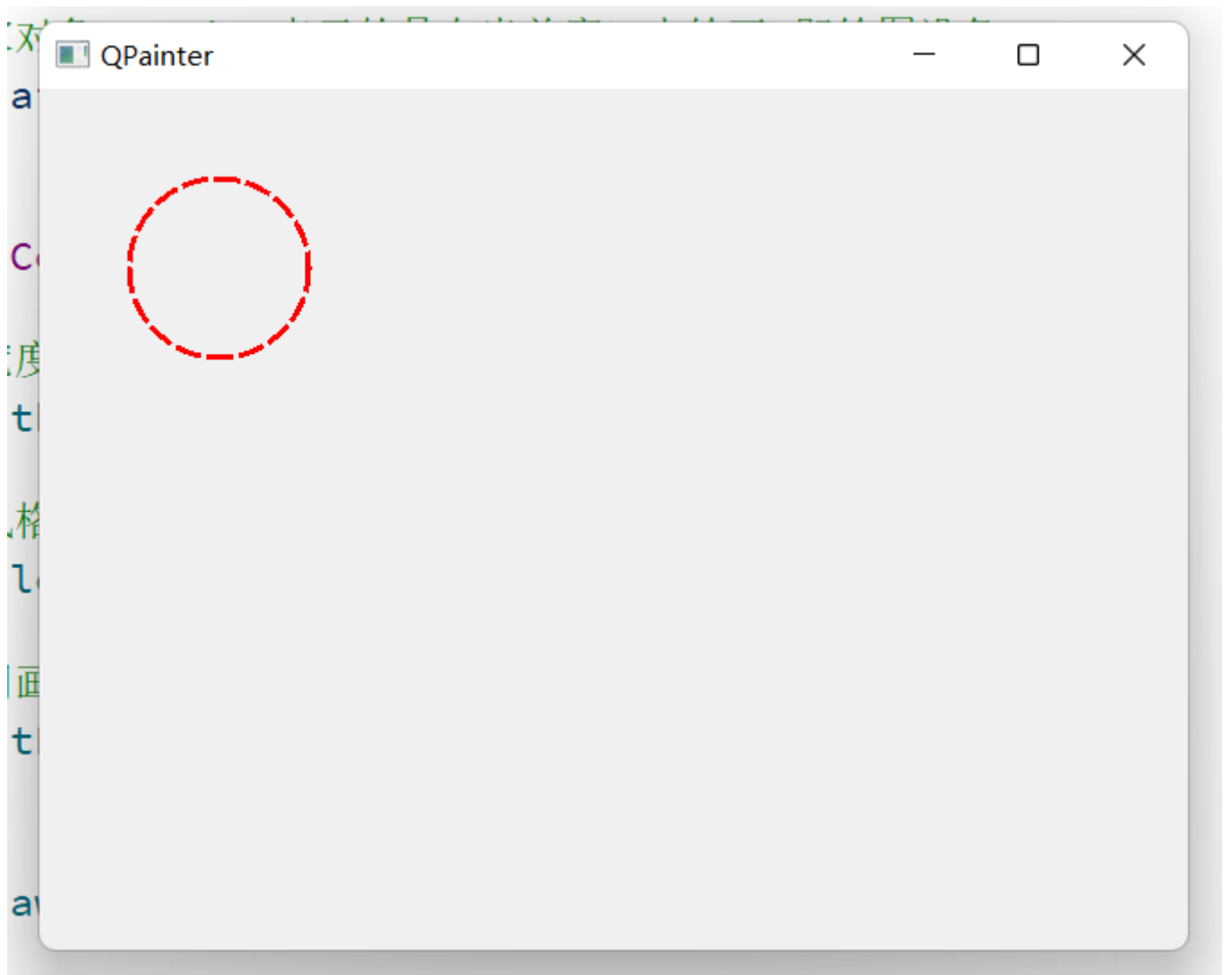
Qt::CustomDashLine

Constant	Value	Description
Qt::NoPen	0	no line at all. For example, QPainter::drawRect() fills but does not draw any boundary line.
Qt::SolidLine	1	A plain line.
Qt::DashLine	2	Dashes separated by a few pixels.
Qt::DotLine	3	Dots separated by a few pixels.
Qt::DashDotLine	4	Alternate dots and dashes.
Qt::DashDotDotLine	5	One dash, two dots, one dash, two dots.
Qt::CustomDashLine	6	A custom pattern defined using QPainterPathStroker::setDashPattern().

示例：画笔的使用



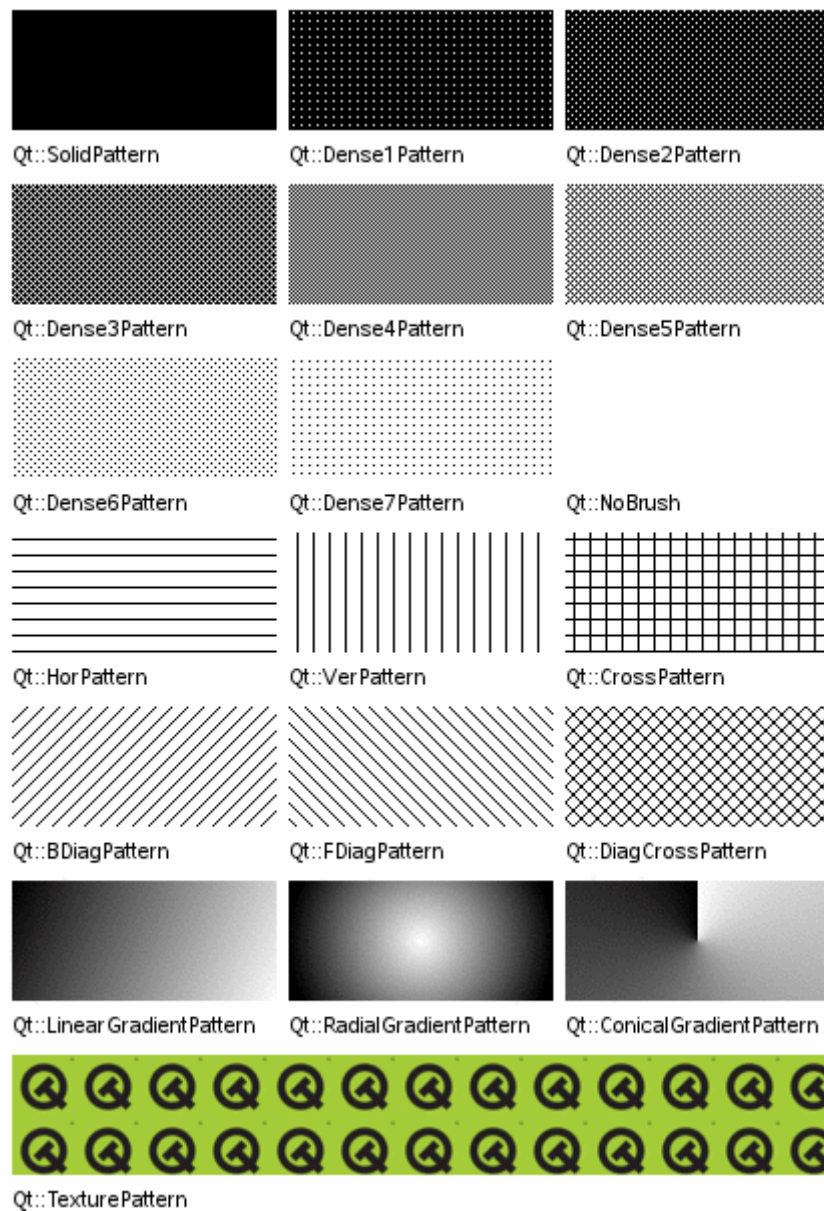
实现效果如下：



### 2.2.6 设置画刷

在 Qt 中，画刷是使用 `QBrush` 类来描述，画刷大多用于填充。`QBrush` 定义了 `QPainter` 的填充模式，具有样式、颜色、渐变以及纹理等属性。

画刷的格式中定义了填充的样式，使用 `Qt::BrushStyle` 枚举，默认值是 `Qt::NoBrush`，也就是不进行任何填充。可以通过 Qt 助手查找画刷的格式。如下图示：



设置画刷主要通过 `void QPen::setBrush(const QBrush &brush)` 方法，其参数为画刷的格式。

示例:

QPainter

QPainter.pro

Headers

widget.h

Sources

main.cpp

widget.cpp

Tests

Qt Test (none)

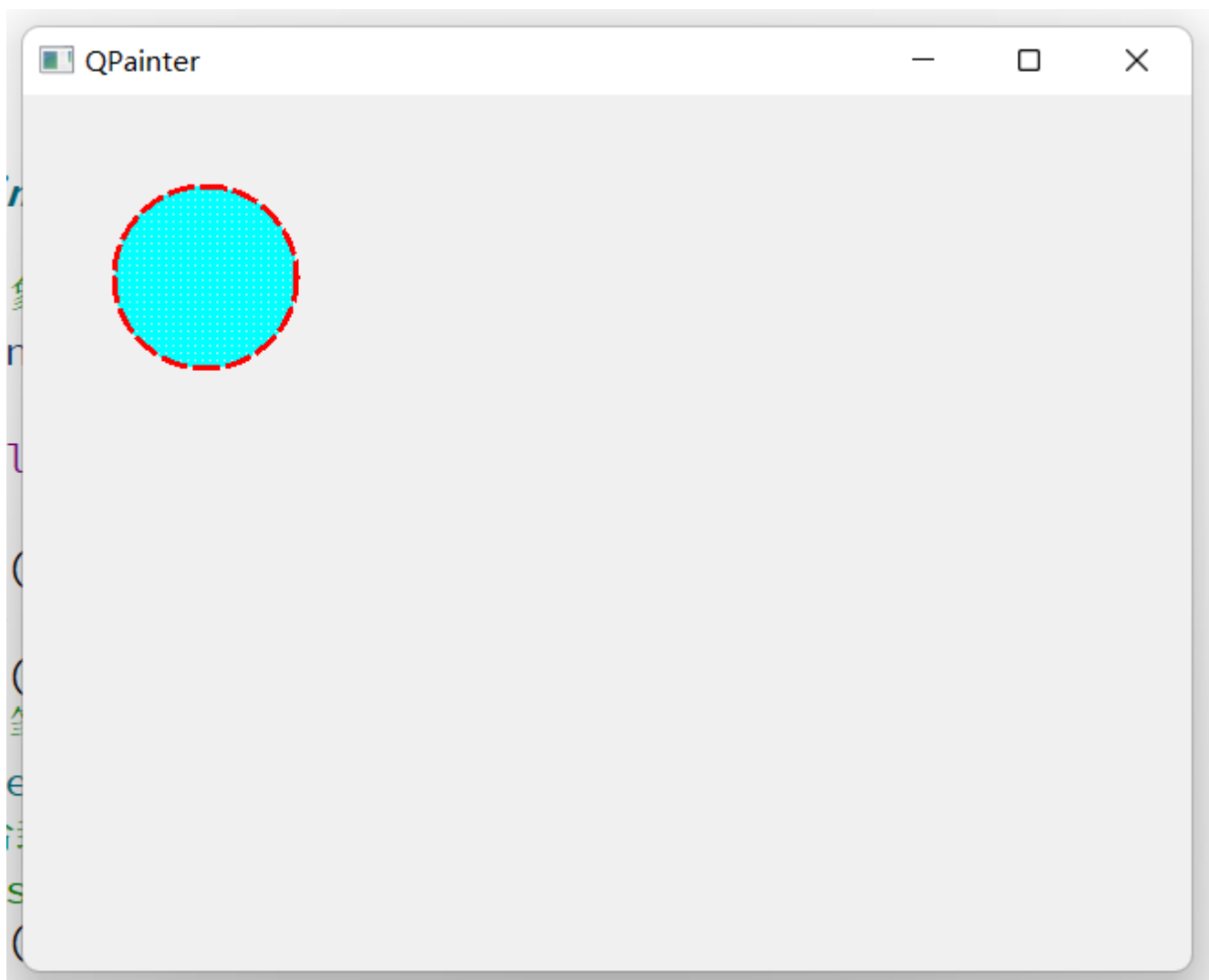
Quick Test (none)

Google Test (none)

Boost Test (none)

```
1  #include "widget.h"
2  #include <QPainter>
3
4  Widget::Widget(QWidget *parent)
5      : QWidget(parent)
6  {
7  }
8
9  void Widget::paintEvent(QPaintEvent *)
10 {
11     //实例化画家对象 this:表示的是在当前窗口中绘画,即绘图设备
12     QPainter painter(this);
13     //设置画笔
14     QPen pen(QColor(255,0,0));
15     //设置画笔宽度
16     pen.setWidth(3);
17     //设置画笔风格
18     pen.setStyle(Qt::DashLine);
19     //让画家使用画笔
20     painter.setPen(pen);
21     //设置画刷 给封闭图形填充颜色
22     // QBrush brush(QColor(0,255,0));
23     QBrush brush(Qt::cyan);
24     //设置画刷风格
25     brush.setStyle(Qt::Dense1Pattern);
26     //让画家使用画刷
27     painter.setBrush(brush);
28     //绘制圆
29     painter.drawEllipse(QPoint(100,100), 50, 50);
30 }
```

实现效果：

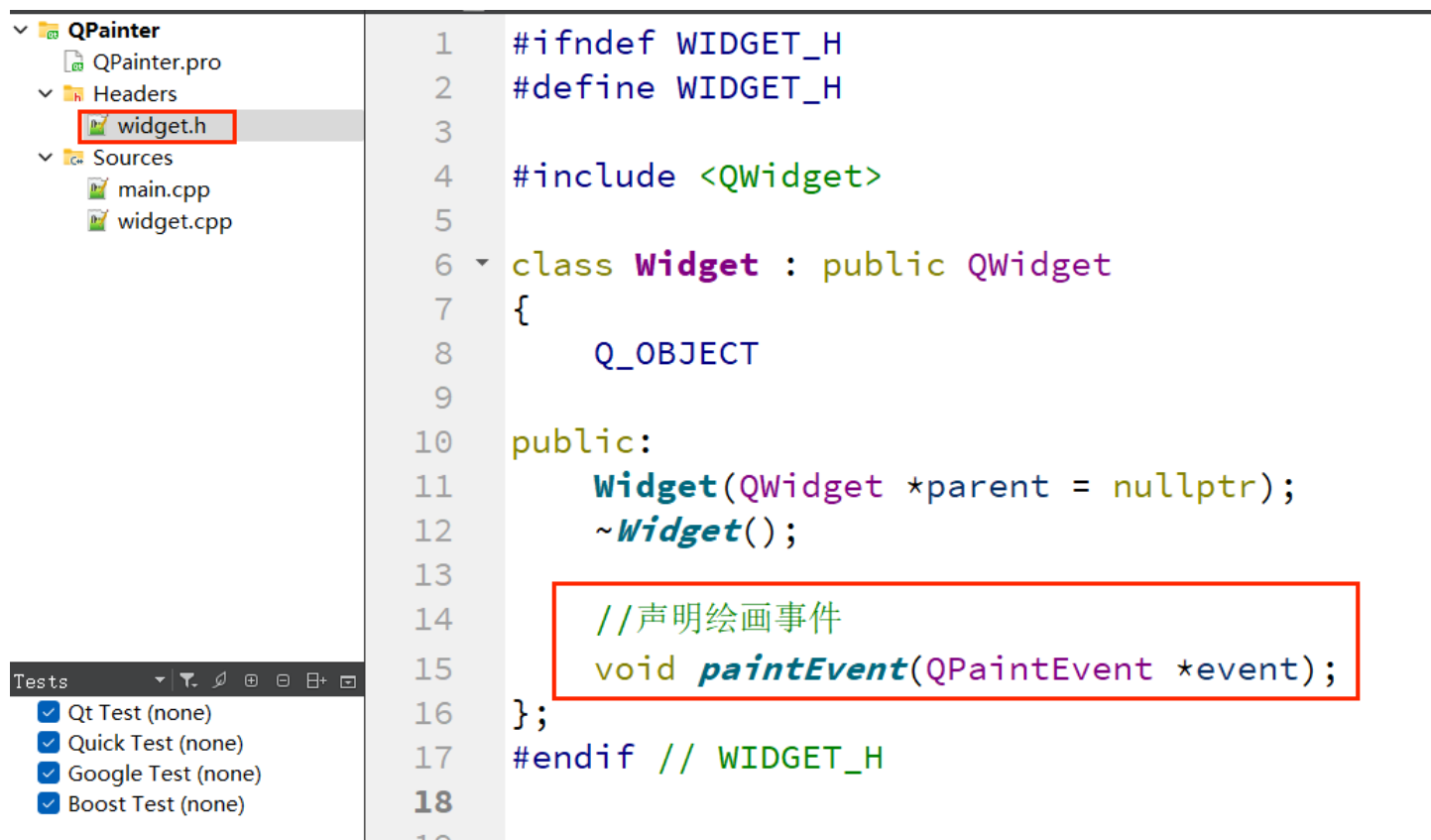


## 2.3 绘制图片

Qt 提供了四个类来处理图像数据：QImage、QPixmap、QBitmap 和 QPicture，它们都是常用的绘图设备。其中QImage主要用来进行 I/O 处理，它对 I/O 处理操作进行了优化，而且可以用来直接访问和操作像素；QPixmap 主要用来在屏幕上显示图像，它对在屏幕上显示图像进行了优化；QBitmap 是 QPixmap 的子类，用来处理颜色深度为1的图像，即只能显示黑白两种颜色；QPicture 用来记录并重演 QPainter 命令。这一节只讲解 QPixmap。

### 2.3.1 绘制简单图片

1. 新建 Qt 项目，基类选择 QWidget，项目名称为 QPainter。在 "widget.h" 头文件中声明绘画事件；如下图示：

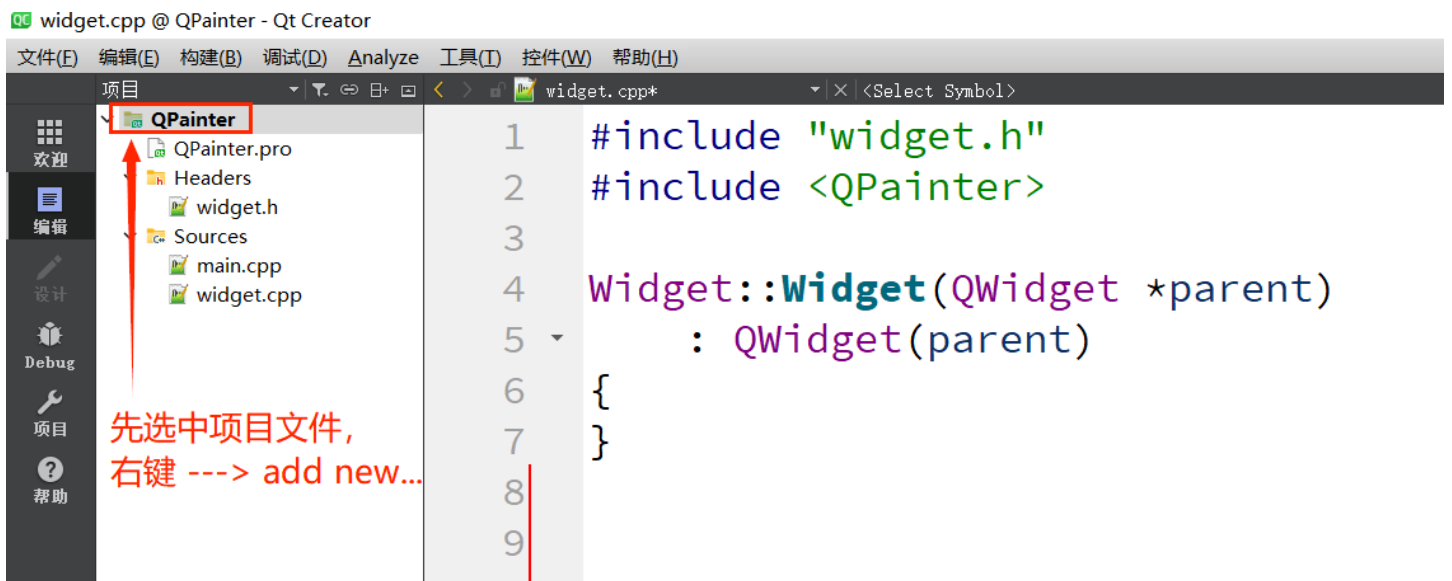


## 2. 添加资源文件；

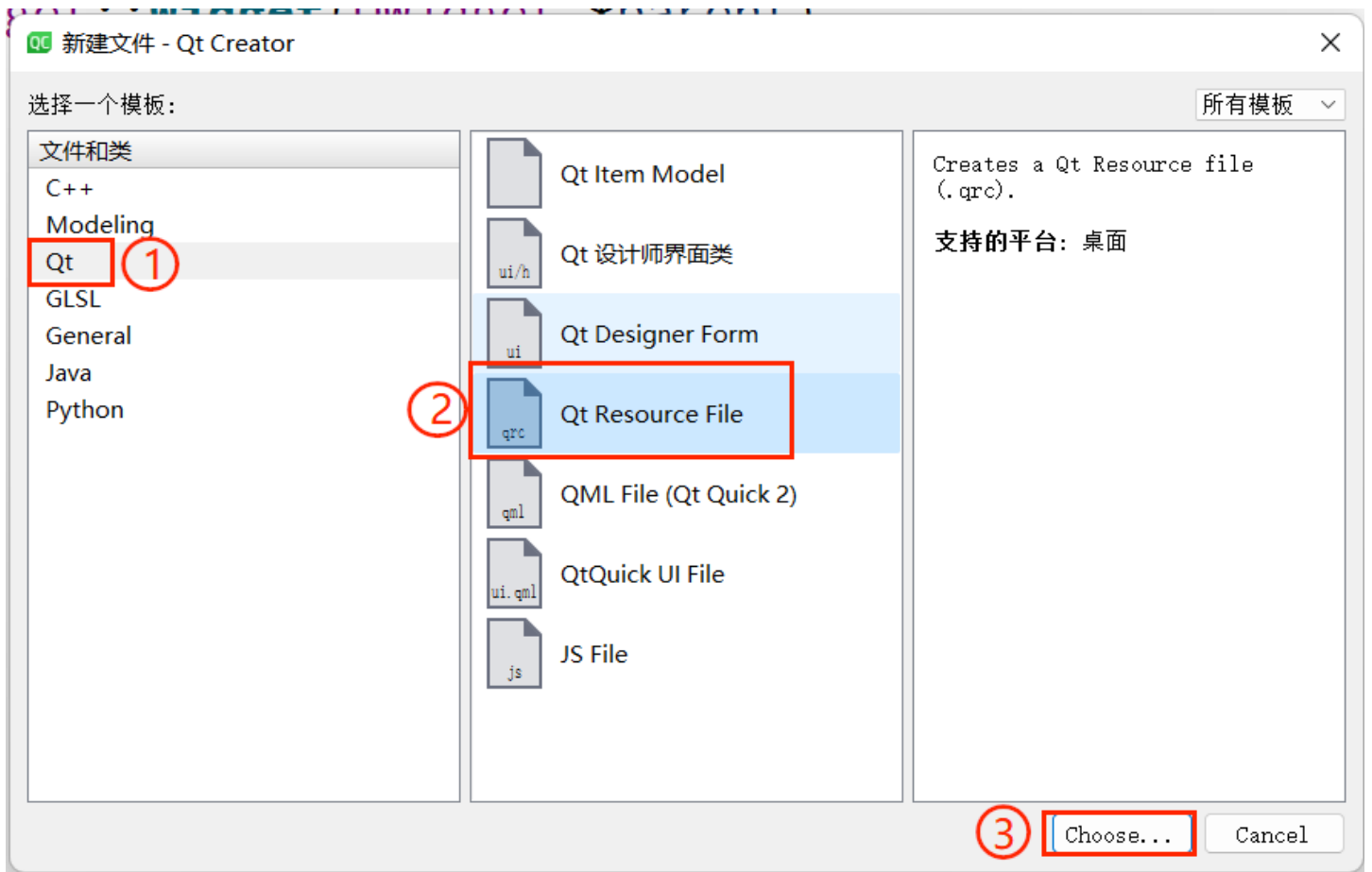
首先准备一些图片资源文件，并将这些图片资源文件放在同一个文件夹中，将该文件夹复制到本项目中；



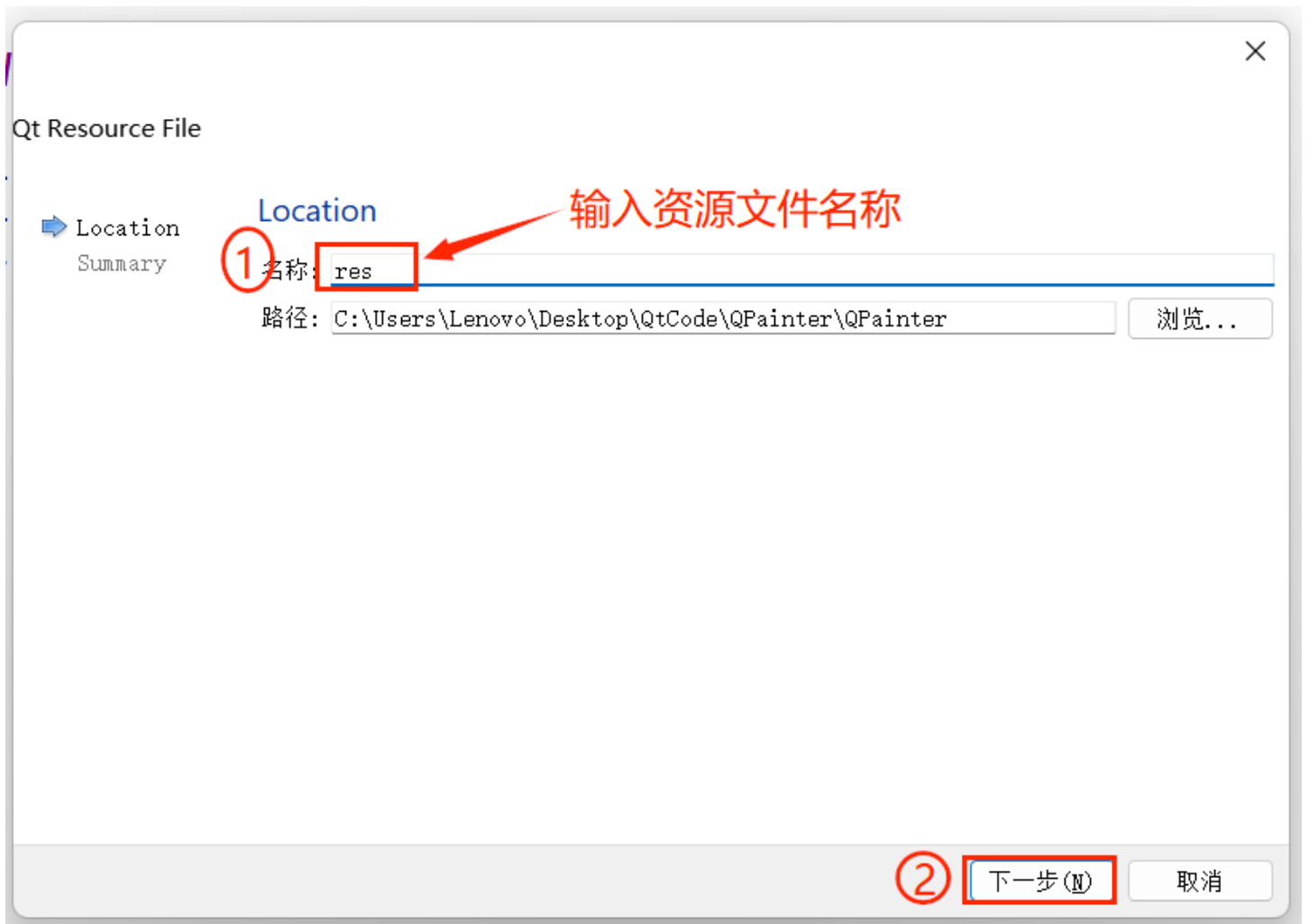
## 3. 选中项目文件，鼠标右键 -----> add new...



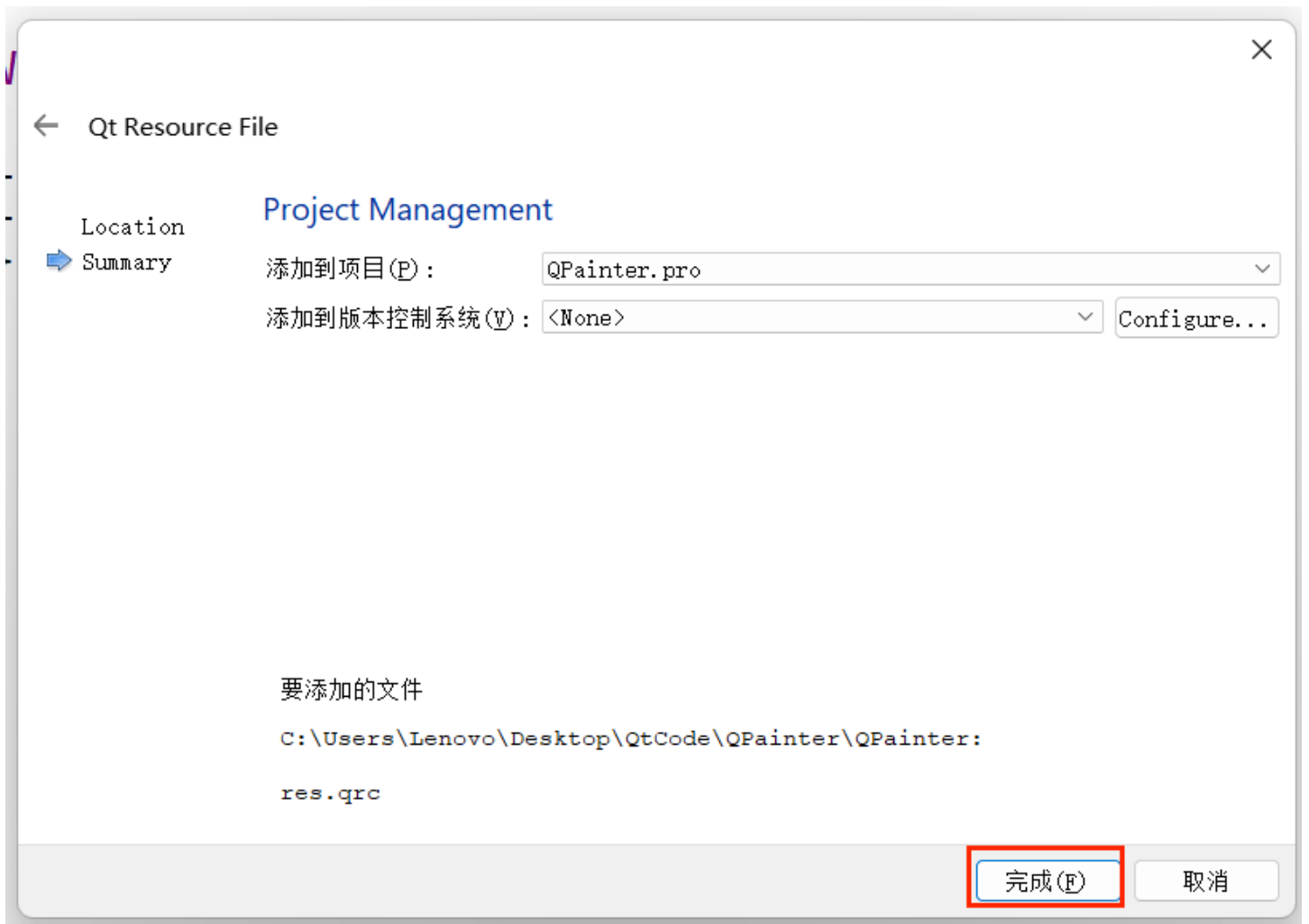
4. 点击 "add new..." 之后，出现如下界面：



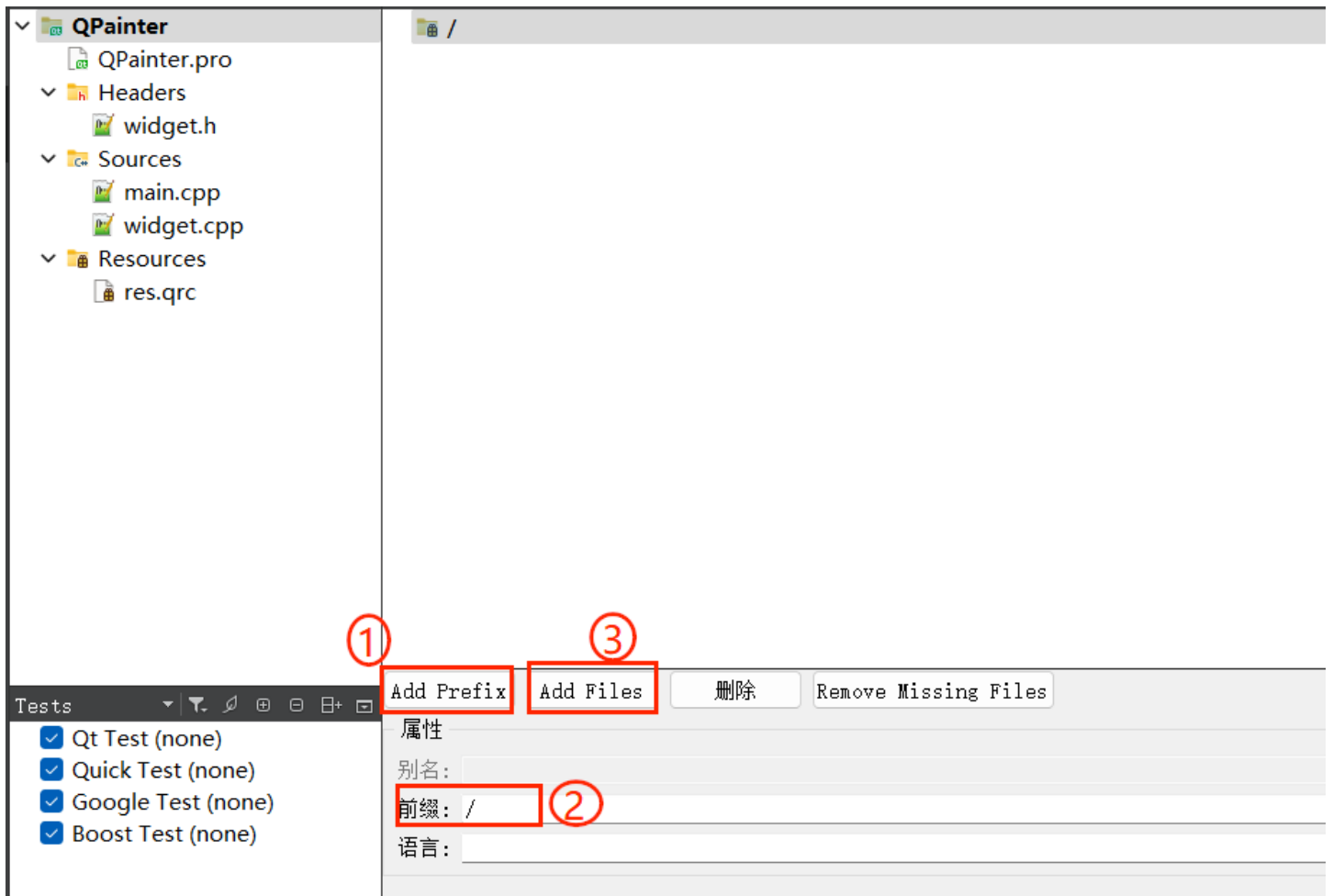
5. 选择 "Choose..." 之后，给资源文件命名；



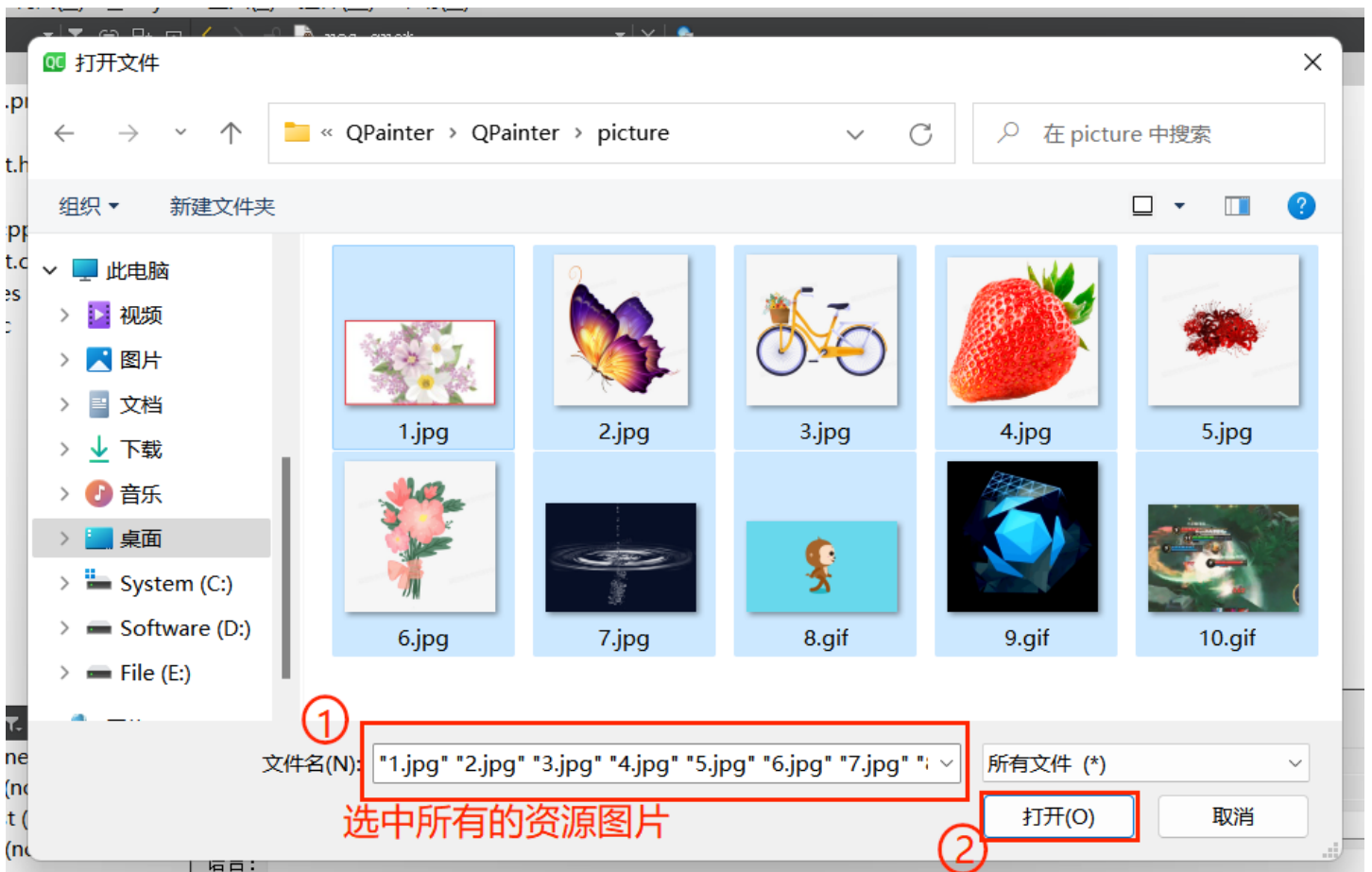
6. 点击 "下一步", 出现如下界面, 点击 "完成";



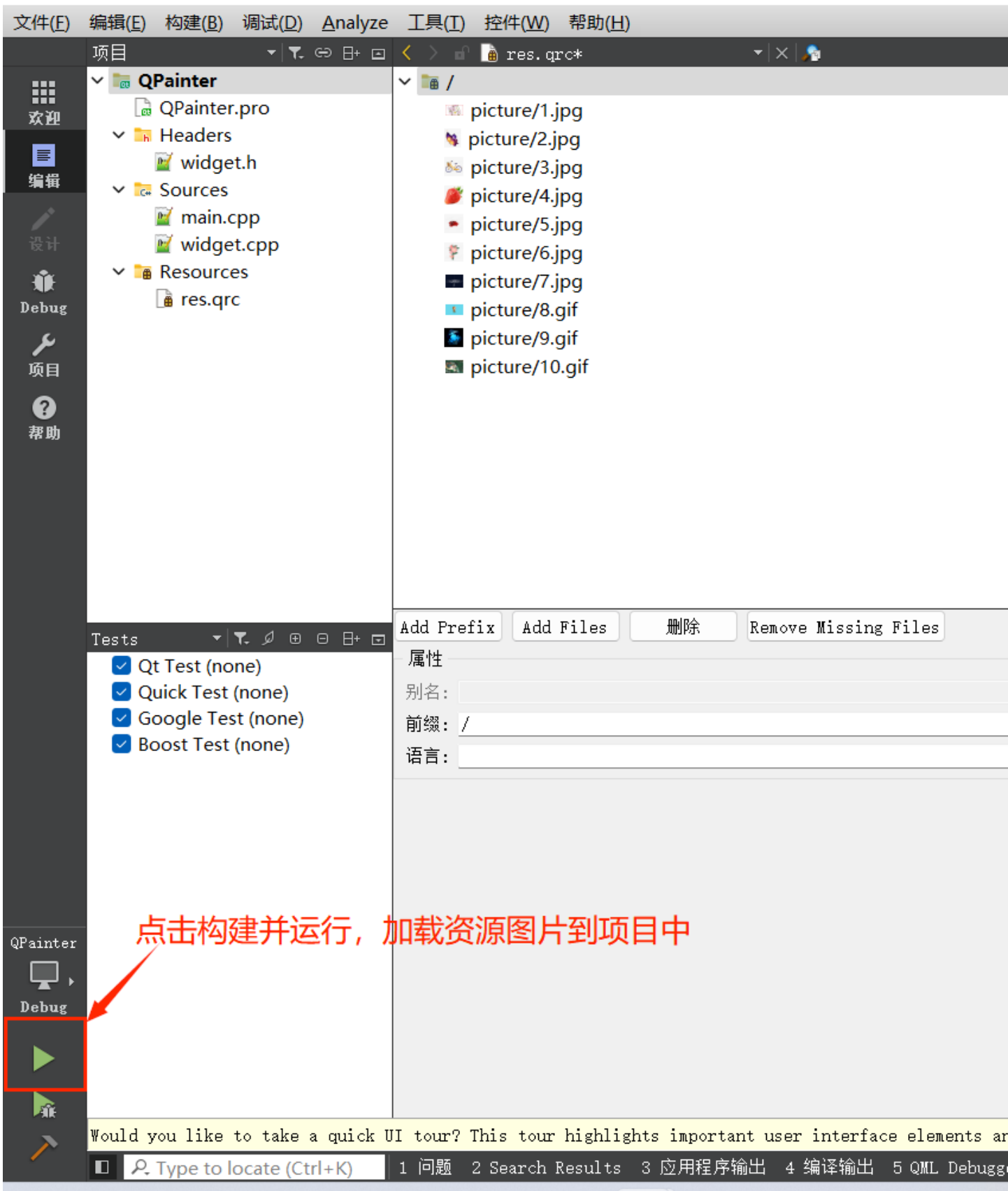
7. 给资源文件添加前缀，并将资源文件添加至项目中；



8. 将所有的资源文件添加到项目中，方便后续使用；



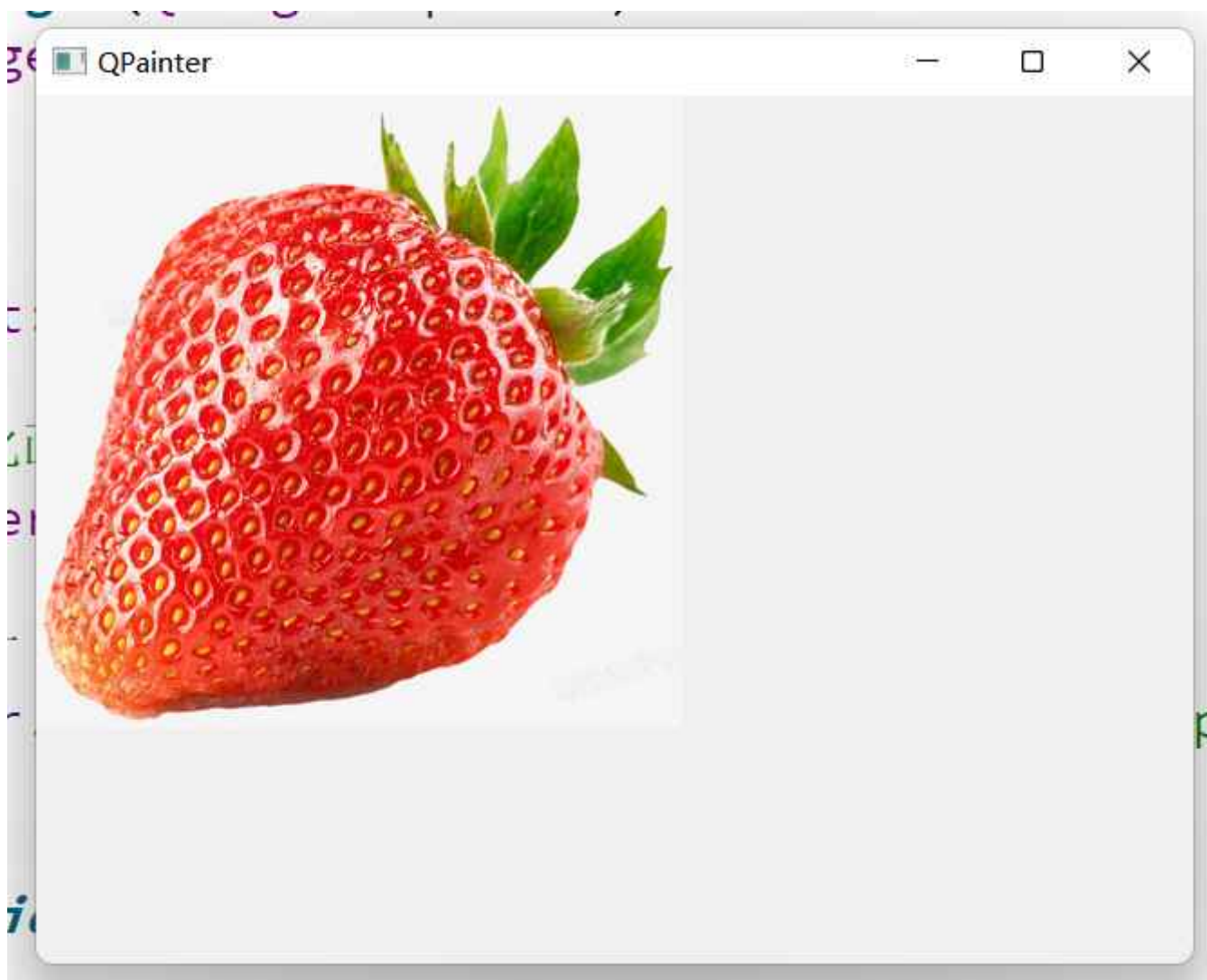
9. 点击 "构建并运行" 按钮，将资源文件添加到项目中；



10. 在 "widget.cpp" 文件中实现画图片功能；



实现效果如下：



### 2.3.2 平移图片

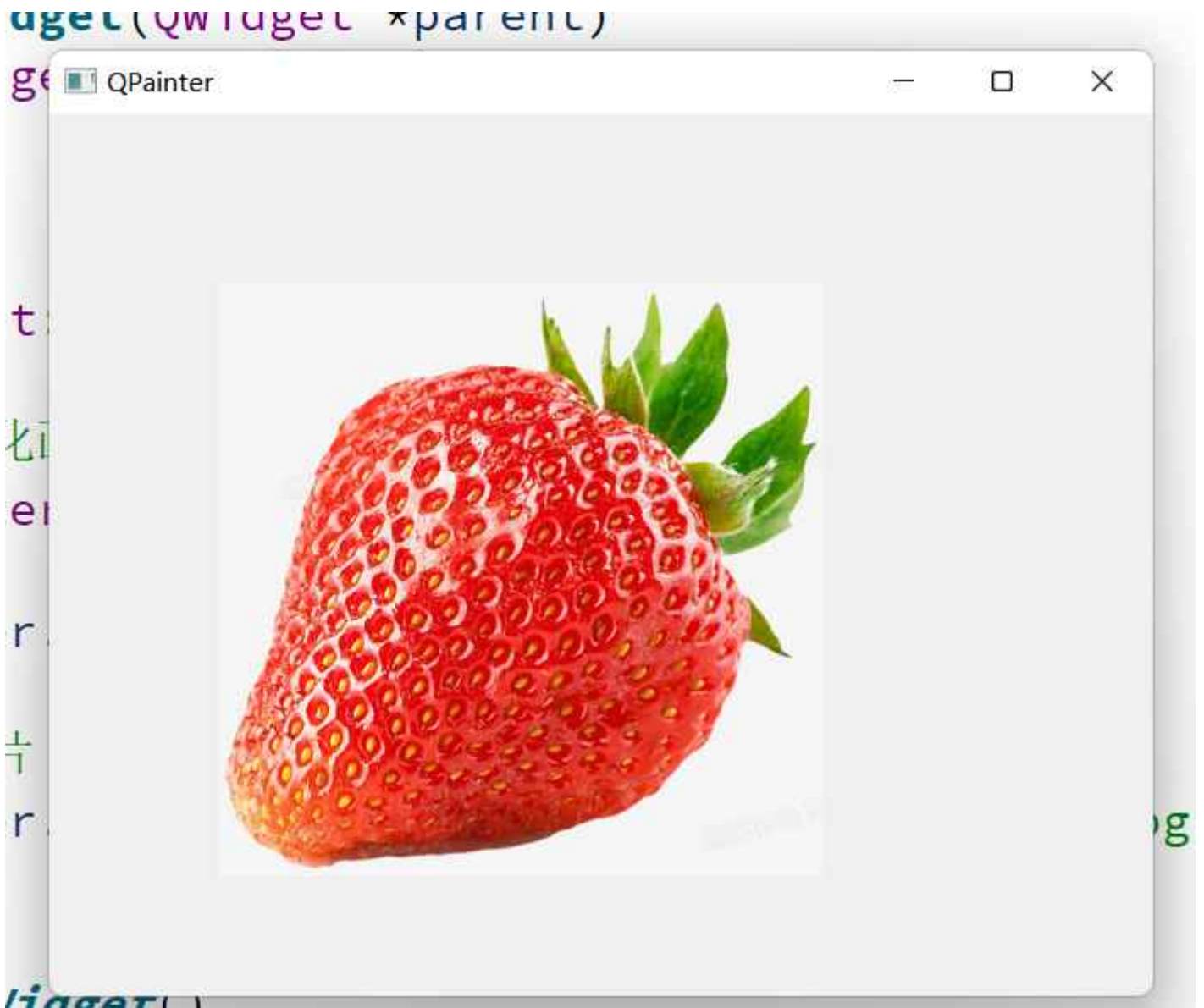
平移图片实际是通过改变坐标来实现。QPainter类中提供了 `translate()` 函数 来实现坐标原点的改变。

如下示例：



```
1  #include "widget.h"
2  #include <QPainter>
3
4  Widget::Widget(QWidget *parent)
5      : QWidget(parent)
6  {
7  }
8
9  void Widget::paintEvent(QPaintEvent *)
10 {
11     //实例化画家对象
12     QPainter painter(this);
13
14     painter.translate(100,100);
15
16     //画图片
17     painter.drawPixmap(0,0,QPixmap(":/picture/4.jpg"));
18 }
19
```

实现效果如下：



### 2.3.3 缩放图片

在 Qt 中，图片的放大和缩小可以使用 **QPainter**类 中的 **drawPixmap()**函数 来实现。

示例：

QPainter

QPainter.pro

Headers

widget.h

Sources

main.cpp

widget.cpp

Resources

res.qrc

/

picture

1.jpg

10.gif

2.jpg

3.jpg

4.jpg

5.jpg

6.jpg

7.jpg

8.gif

9.gif

Tests

Qt Test (none)

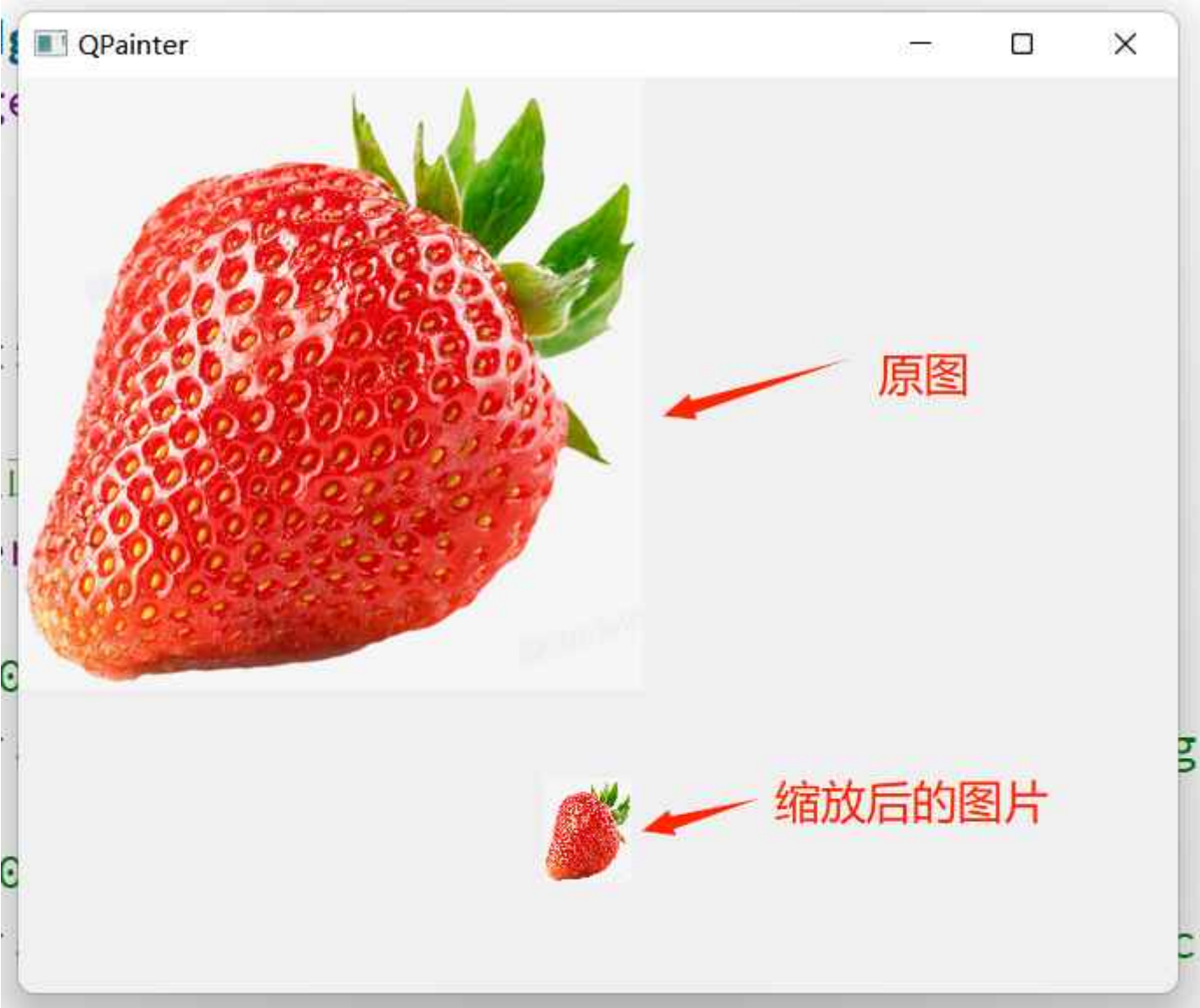
Quick Test (none)

Google Test (none)

Boost Test (none)

```
1 #include "widget.h"
2 #include <QPainter>
3 #include <QPixmap>
4
5 Widget::Widget(QWidget *parent)
6     : QWidget(parent)
7 {
8 }
9
10 void Widget::paintEvent(QPaintEvent *)
11 {
12     //实例化画家对象
13     QPainter painter(this);
14
15     //以(0,0)点开始画图，图片的尺寸跟原图保持一致
16     painter.drawPixmap(0,0,QPixmap(":/picture/4.jpg"));
17
18     //以(300,400)点开始画图，图片尺寸变为: 50*60
19     painter.drawPixmap(300,400,50,60,QPixmap(":/picture/4.jpg"));
20 }
21
22
```

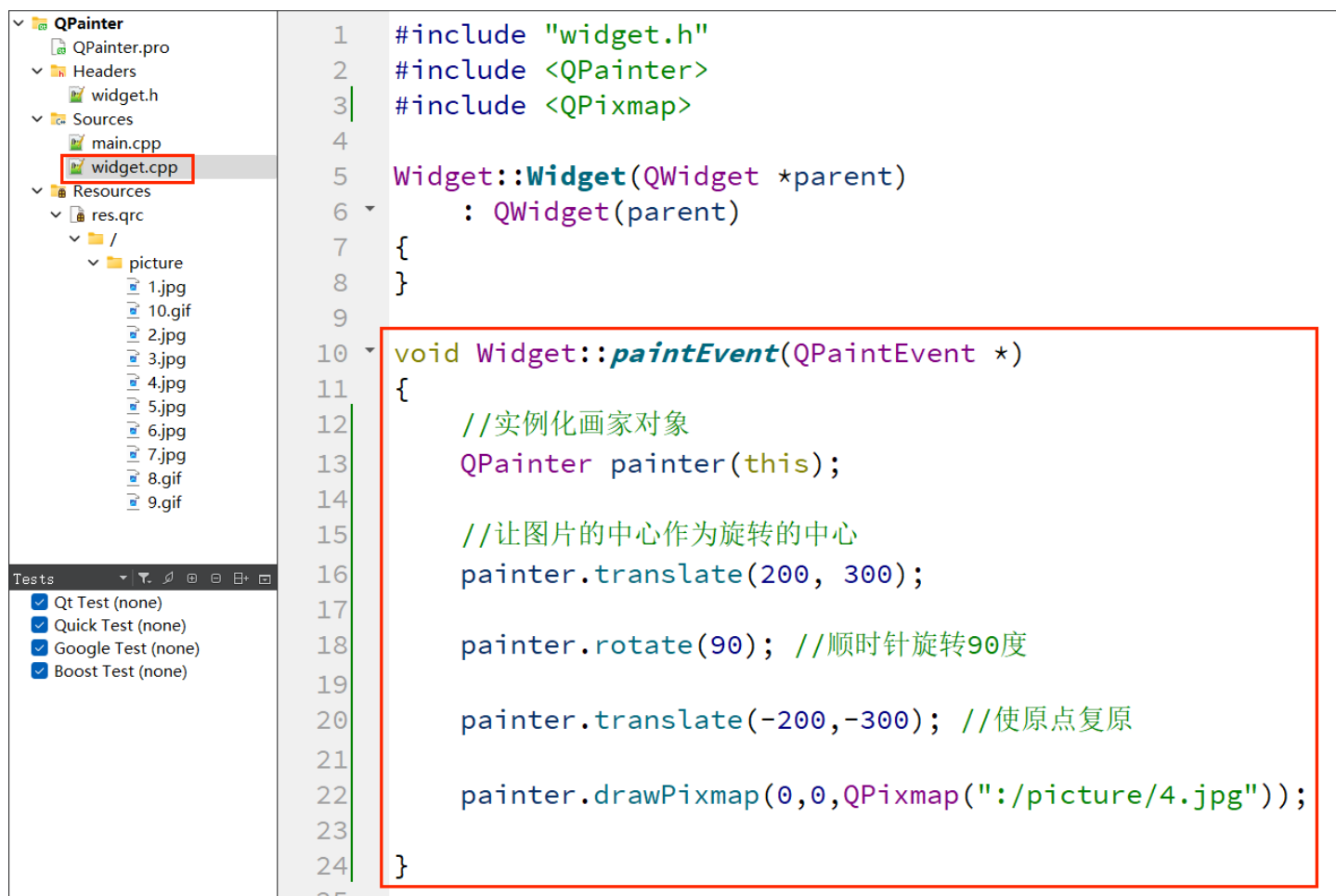
实现效果如下：



### 2.3.4 旋转图片

图片的旋转使用的是 **QPainter**类 中的 **rotate()**函数，它默认是以原点为中心进行旋转的。如果要改变旋转的中心，可以使用 **translate()**函数 完成。

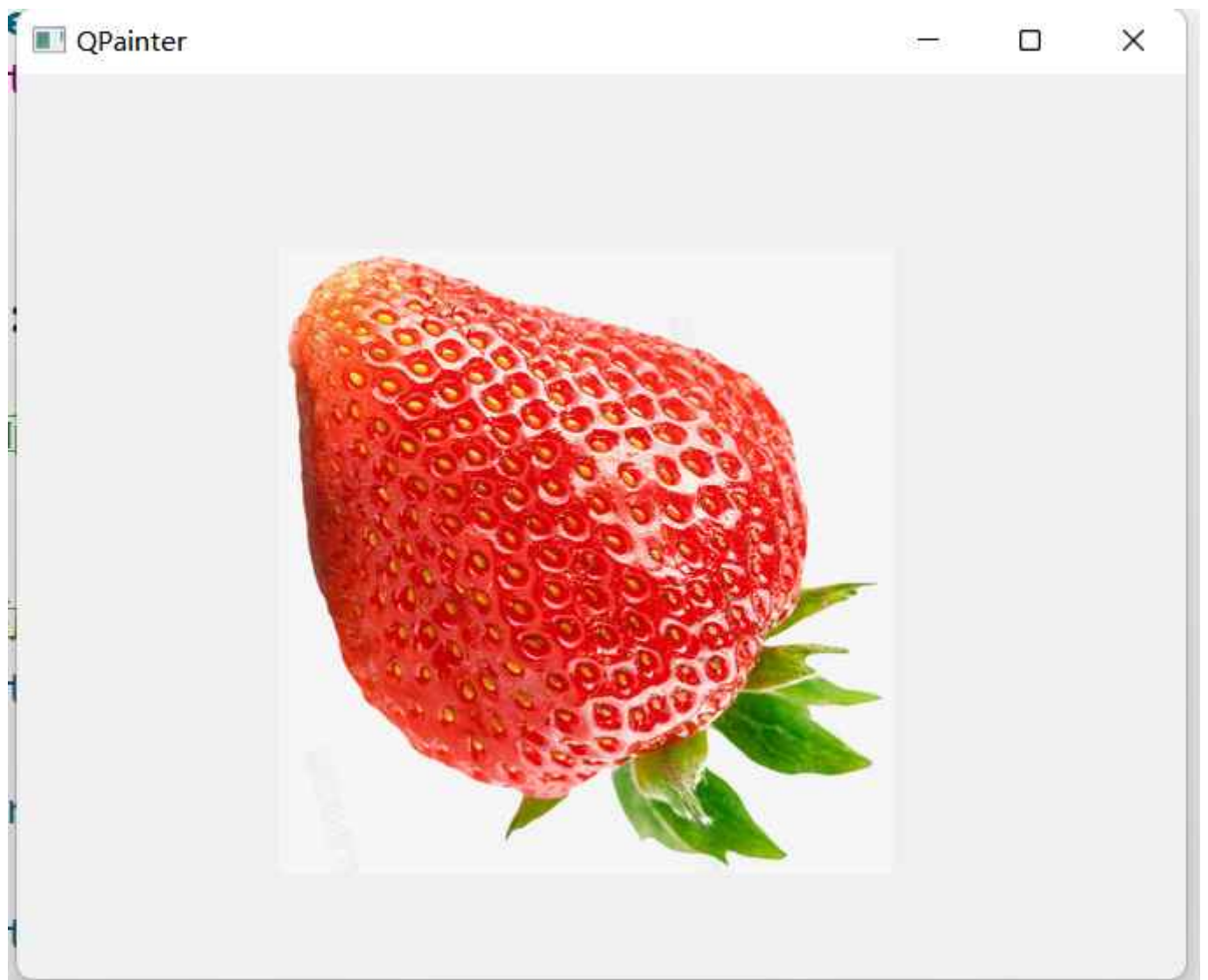
示例：



The screenshot displays the Qt Creator IDE interface. On the left, the 'Project Explorer' shows the project structure for 'QPainter'. The 'Sources' folder contains 'main.cpp' and 'widget.cpp', with 'widget.cpp' highlighted. Below it, the 'Resources' folder contains 'res.qrc', which lists various image files in the 'picture' directory, including '4.jpg'. The 'Tests' panel at the bottom shows several test suites like 'Qt Test (none)', 'Quick Test (none)', 'Google Test (none)', and 'Boost Test (none)'. The main editor window shows the C++ code for 'widget.cpp'. The code includes necessary headers and defines a 'Widget' class. The 'paintEvent' method is highlighted with a red box and contains the logic for rotating an image: it creates a 'QPainter' object, translates the origin to (200, 300), rotates 90 degrees clockwise, translates the origin back to (0, 0), and finally draws the image '4.jpg'.

```
1  #include "widget.h"
2  #include <QPainter>
3  #include <QPixmap>
4
5  Widget::Widget(QWidget *parent)
6      : QWidget(parent)
7  {
8  }
9
10 void Widget::paintEvent(QPaintEvent *)
11 {
12     //实例化画家对象
13     QPainter painter(this);
14
15     //让图片的中心作为旋转的中心
16     painter.translate(200, 300);
17
18     painter.rotate(90); //顺时针旋转90度
19
20     painter.translate(-200,-300); //使原点复原
21
22     painter.drawPixmap(0,0,QPixmap(":/picture/4.jpg"));
23
24 }
```

实现效果如下：



## 2.4 其他设置

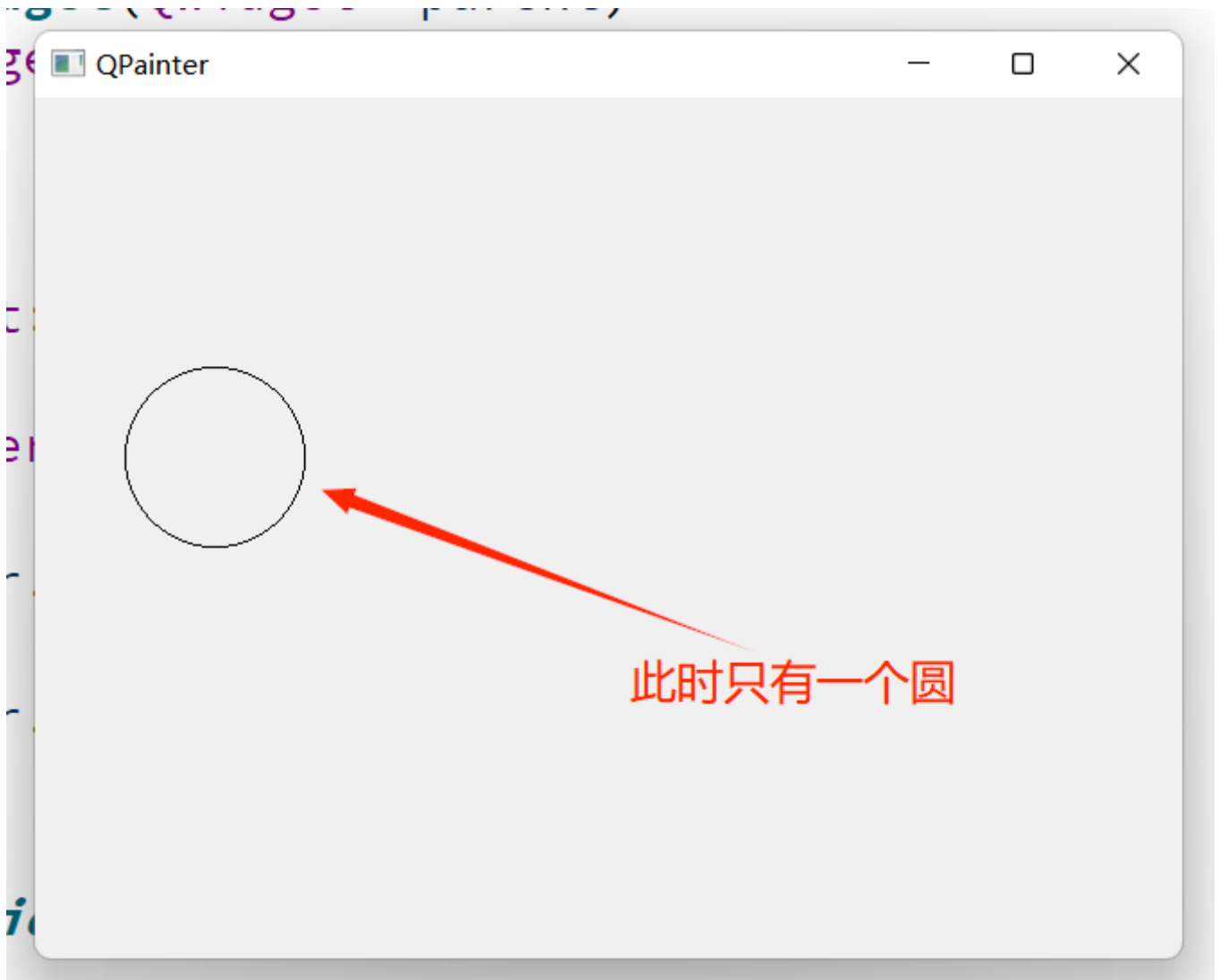
### 2.4.1 移动画家位置

有时候在绘制多个图形时，想使用同一坐标位置，那么绘制出来的图形肯定会重合，此时，可以通过移动画家的位置来使图形不发生重合。

#### 示例1：未移动画家位置



实现效果如下：

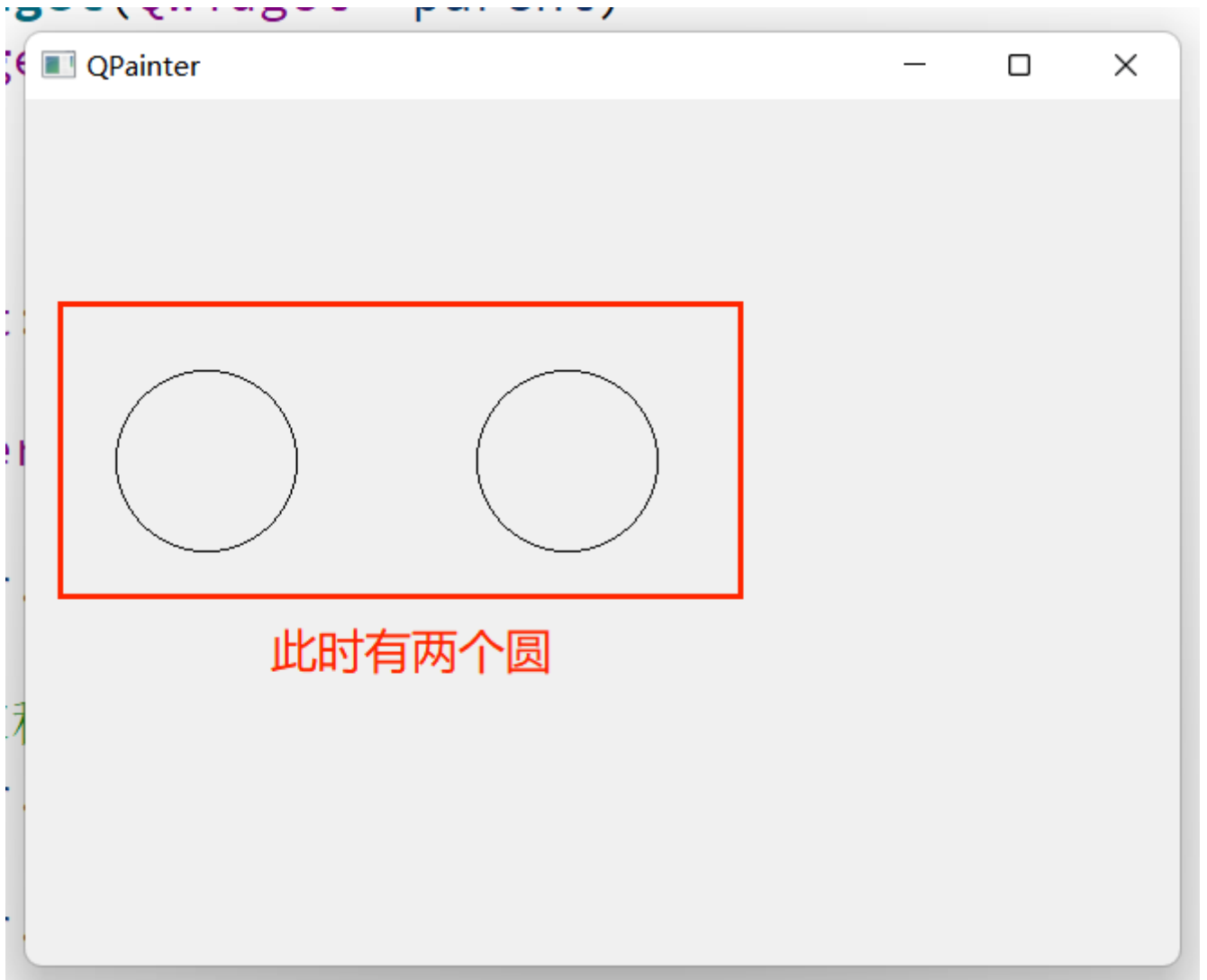


## 示例2：移动画家位置

- 使用 `translate` 移动画家所在位置.



实现效果如下：



## 2.4.2 保存/加载画家的状态

在绘制图形的过程中，可以通过 `save()` 函数来保存画家的状态，使用 `restore()` 函数还原画家状态。

`save()` 函数原型如下：

```
void QPainter::save();
```

```
void QPainter::save()
```

Saves the current painter state (pushes the state onto a stack). A `save()` must be followed by a corresponding `restore()`; the `end()` function unwinds the stack.

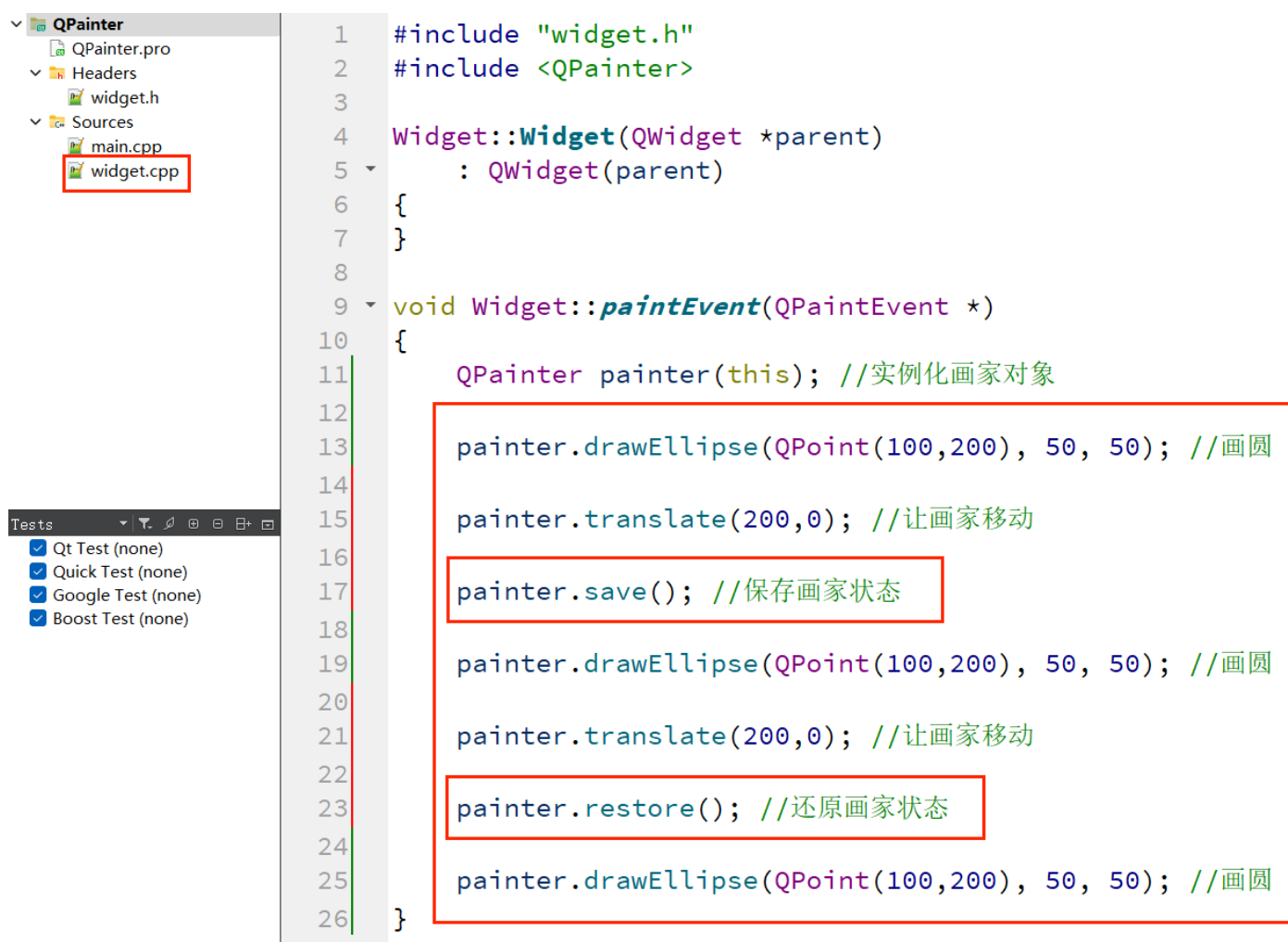
`restore()` 函数原型如下：

```
void QPainter::restore();
```

```
void QPainter::restore()
```

Restores the current painter state (pops a saved state off the stack).

示例：



```
1  #include "widget.h"
2  #include <QPainter>
3
4  Widget::Widget(QWidget *parent)
5      : QWidget(parent)
6  {
7  }
8
9  void Widget::paintEvent(QPaintEvent *)
10 {
11     QPainter painter(this); //实例化画家对象
12
13     painter.drawEllipse(QPoint(100,200), 50, 50); //画圆
14
15     painter.translate(200,0); //让画家移动
16
17     painter.save(); //保存画家状态
18
19     painter.drawEllipse(QPoint(100,200), 50, 50); //画圆
20
21     painter.translate(200,0); //让画家移动
22
23     painter.restore(); //还原画家状态
24
25     painter.drawEllipse(QPoint(100,200), 50, 50); //画圆
26 }
```

实现效果如下：



说明：

上述示例中，在画第三个圆之前，由于还原了画家的状态，所以此时画家的位置坐标会移动到画家状态保存的地方，所以在绘制第三个圆的位置时实际是和第二个圆发生了重叠。

## 2.5 特殊的绘图设备

前面的代码中我们是使用 `QWidget` 作为绘图设备. 在 Qt 中还存在下列三个比较特殊的绘图设备. 此处我们也简要介绍.

- `QPixmap` 用于在显示器上显示图片.
- `QImage` 用于对图片进行像素级修改.
- `QPicture` 用于对 `QPainter` 的一系列操作进行存档.

### 2.5.1 QPixmap

`QPixmap` 核心特性:

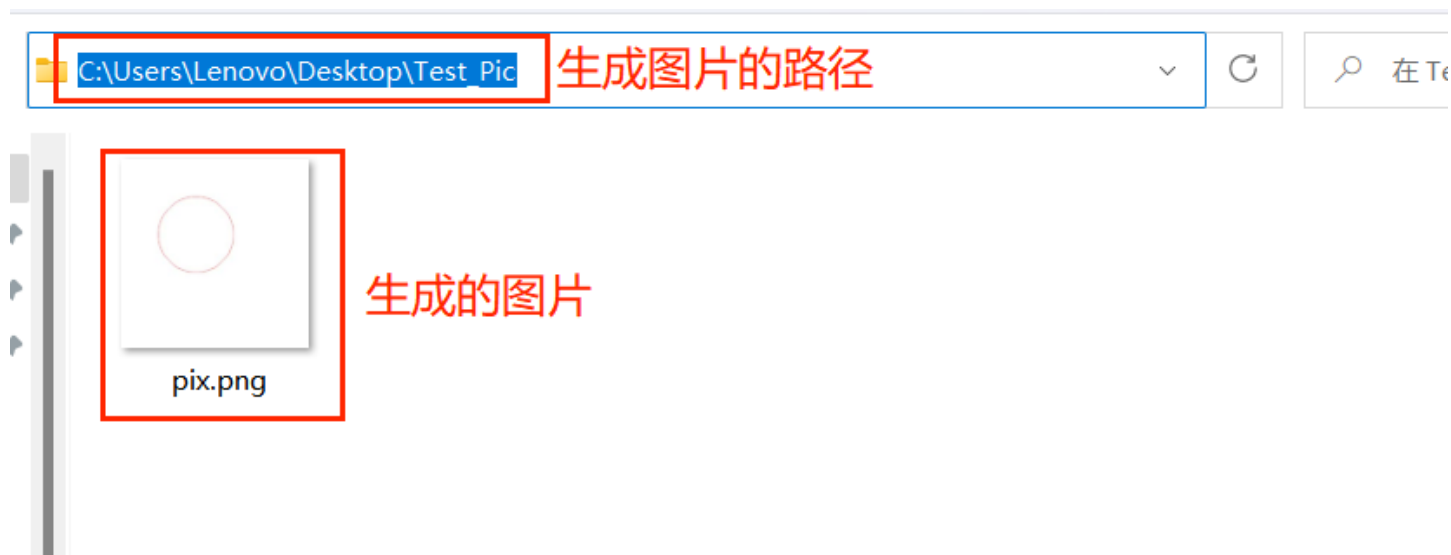
- 使用 `QPainter` 直接在上面进行绘制图形.

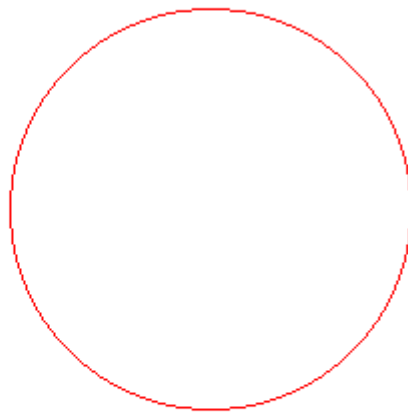
- 通过文件路径加载并显示图片.
- 搭配 QPainter 的 drawPixmap()函数, 可以把这个图片绘制到一个 QLabel、QPushButton 等控件上.
- 和系统/显示设备强相关, 不同系统/显示设备下, QPixmap 的显示可能会有所差别.

## 示例:



## 实现效果:





## 2.5.2 QImage

QImage 的核心特性:

- 使用 QPainter 直接在上面进行绘制图形.
- 通过文件路径加载并显示图片.
- 能够针对图片进行像素级别的操作(操作某个指定的像素).
- 独立于硬件的绘制系统, 能够在不同系统之上提供一致的显示.

**代码示例:** QImage 作为绘图设备的使用

QPainter

QPainter.pro

Headers

widget.h

Sources

main.cpp

widget.cpp

Resources

res.qrc

/

picture

1.jpg

10.gif

2.jpg

3.jpg

4.jpg

5.jpg

6.jpg

7.jpg

8.gif

9.gif

Tests

Qt Test (none)

Quick Test (none)

Google Test (none)

Boost Test (none)

```
1 #include "widget.h"
2 #include <QPainter>
3
4 Widget::Widget(QWidget *parent)
5     : QWidget(parent)
6 {
7     //绘图设备的大小为: 500*500 绘图格式为: QImage::Format_RGB32
8     //绘图格式可通过 Qt助手 查看
9     QImage img(500,500,QImage::Format_RGB32);
10
11     img.fill(Qt::white); //填充色为白色, 默认背景色为黑色
12
13     QPainter painter(&img); //声明画家, 画图设备为img
14
15     painter.setPen(QPen(Qt::cyan)); //设置画笔颜色为蓝绿色(青色)
16
17     painter.drawEllipse(QPoint(200,200),100,100); //画圆
18
19     img.save("C:\\Users\\Lenovo\\Desktop\\Test_Pic\\img.jpg"); //保存画好的图片
20 }
21
```

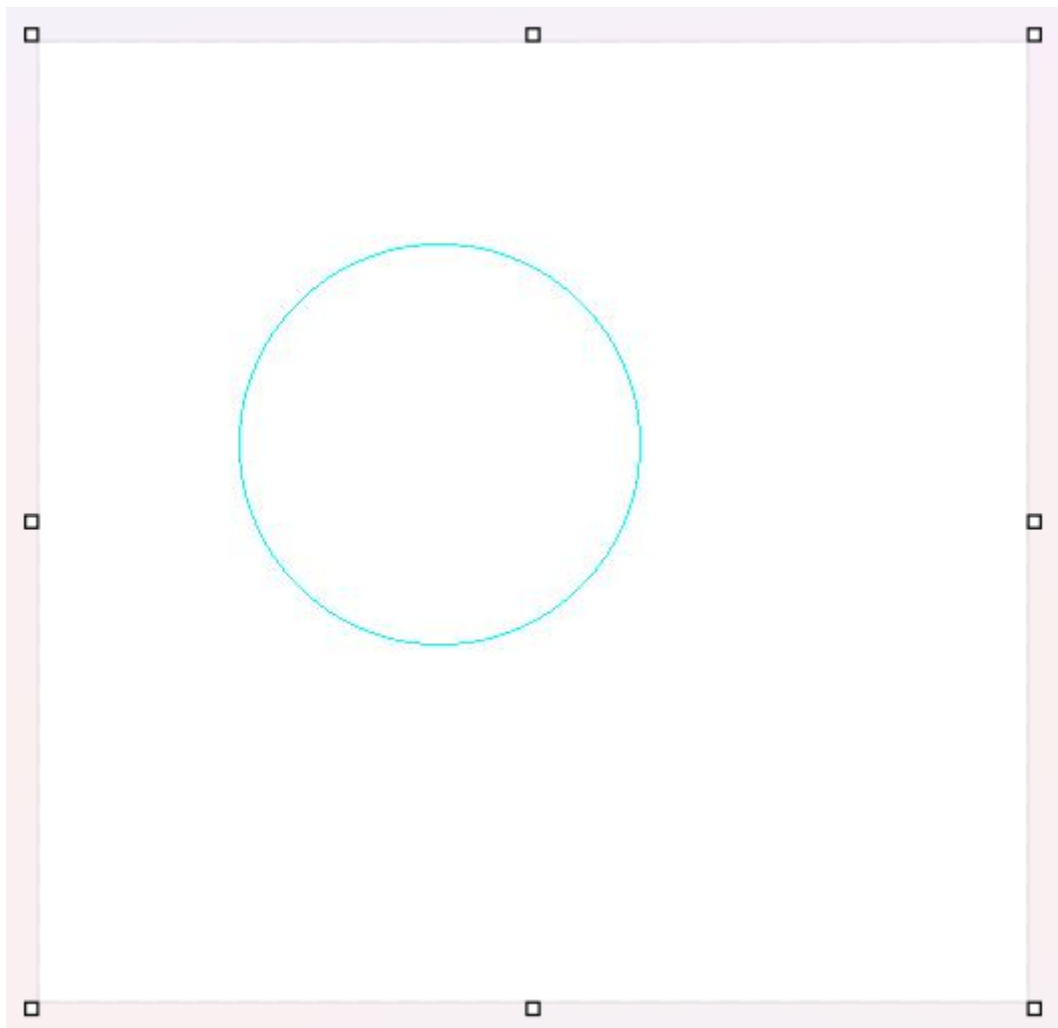
C:\Users\Lenovo\Desktop\Test\_Pic

生成图片的路径

⌵

↺





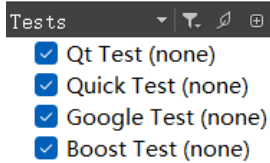
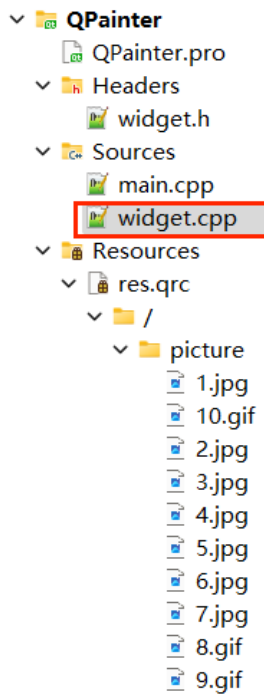
**代码示例: QImage 绘图时对像素的修改**

- 1) 新建 Qt 项目，添加图片资源文件到项目中；如下图示：
- 2) 在 widget.h 头文件中声明绘图事件；



3) 在 widget.cpp 文件中重写绘图事件，使用 QImage 对图片像素进行修改；

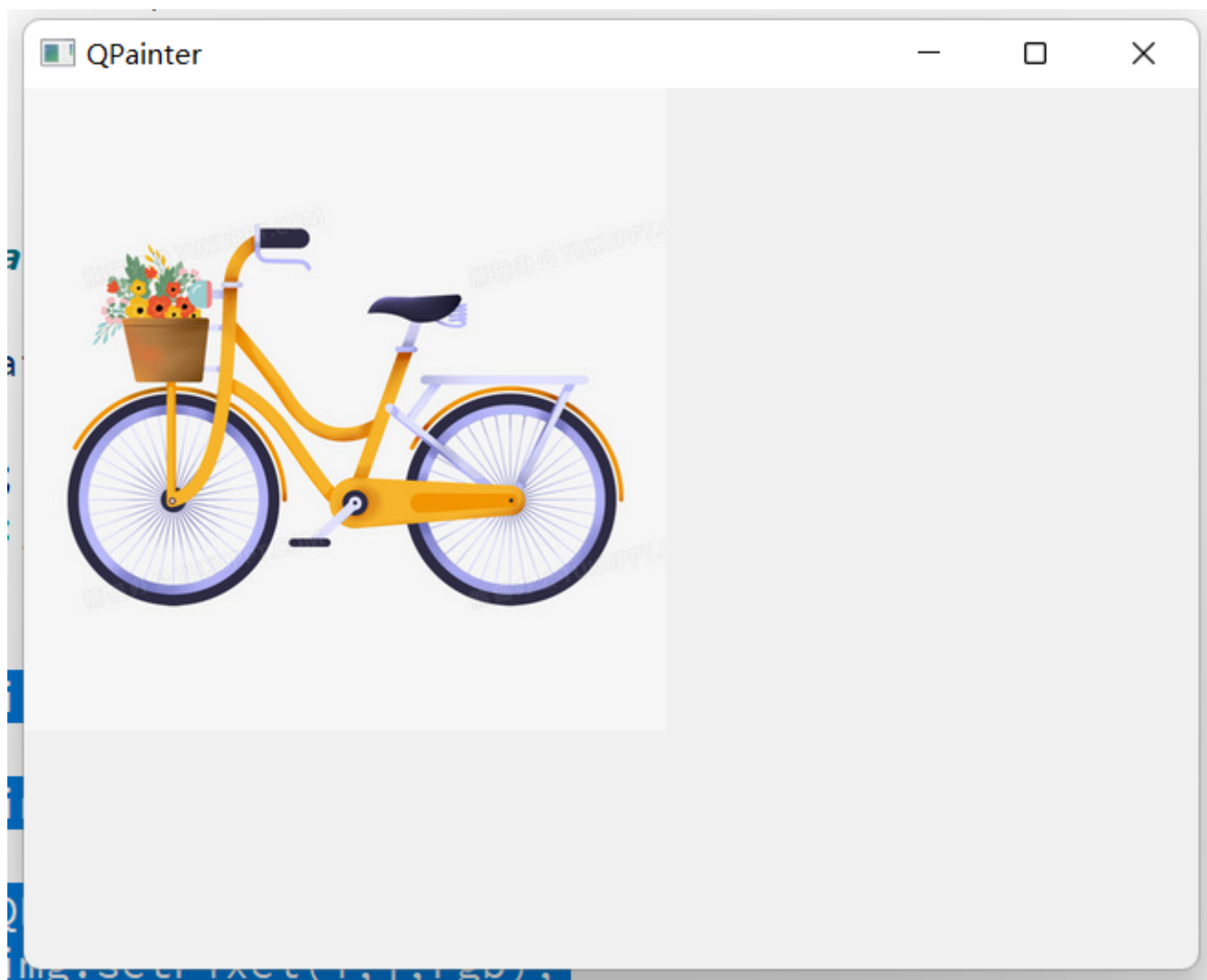
- 通过 `setPixel` 设置某个像素的颜色值.
- 使用 `qRgb` 表示一个具体的颜色.



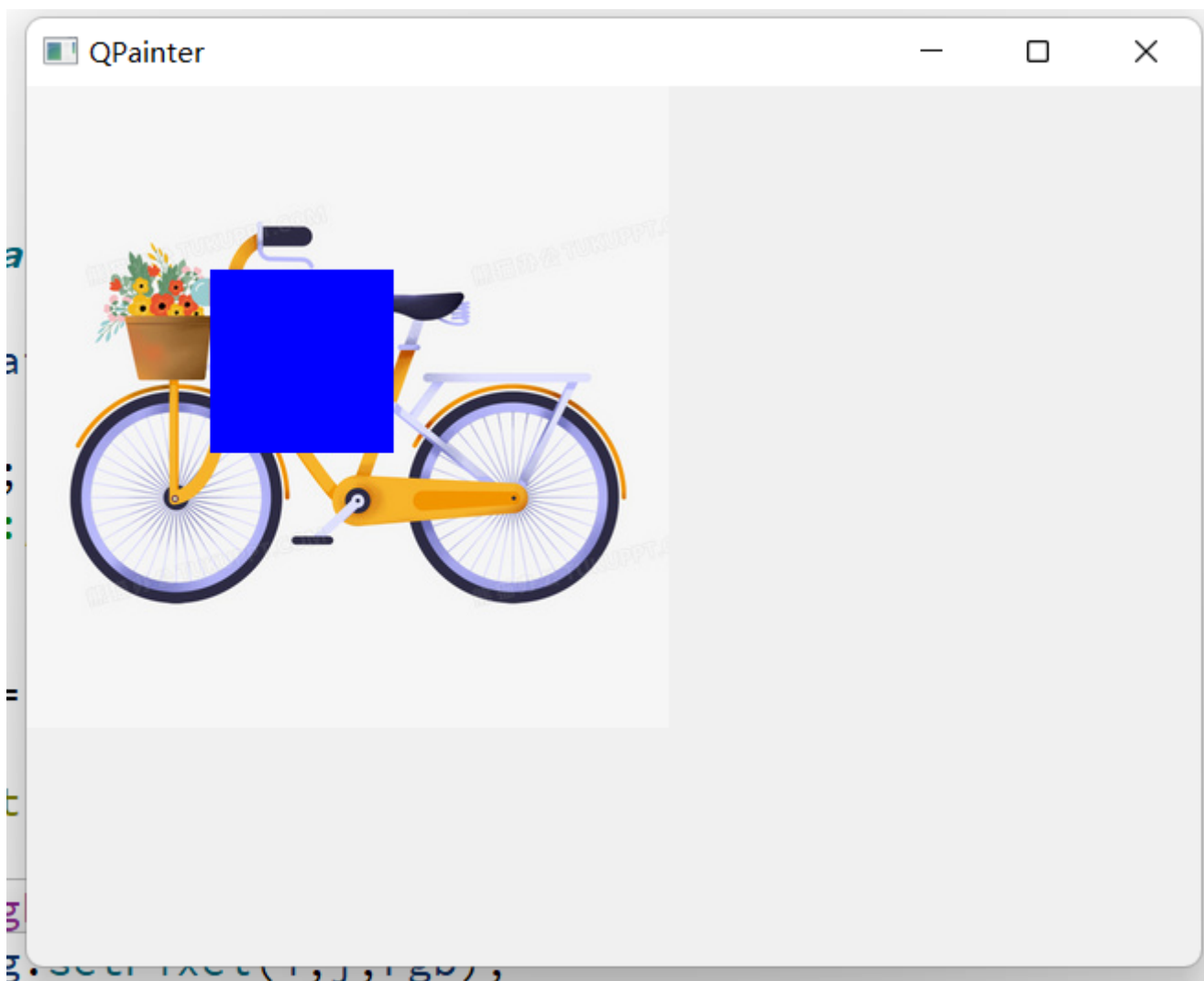
```
1  #include "widget.h"
2  #include <QPainter>
3
4  Widget::Widget(QWidget *parent)
5      : QWidget(parent)
6  {
7
8  }
9
10 void Widget::paintEvent(QPaintEvent *)
11 {
12     QPainter painter(this); //实例化画家对象
13
14     QImage img;
15     img.load(":/picture/3.jpg"); //加载图片
16
17     //修改像素点
18     for(int i = 100; i < 200; i++)
19     {
20         for(int j = 100; j < 200; j++)
21         {
22             QRgb rgb = qRgb(0,0,255);
23             img.setPixel(i,j,rgb);
24         }
25     }
26
27     painter.drawImage(0,0,img);
28 }
```

4) 执行效果如下：

没有修改像素之前：



修改像素之后：



### 2.5.3 QPicture

QPicture 核心特性:

- 使用 QPainter 直接在上面进行绘制图形.
- 通过文件路径加载并显示图片.
- 能够记录 QPainter 的操作步骤.
- 独立于硬件的绘制系统, 能够在不同系统之上提供一致的显示.

**注意:**

QPicture 加载的必须是自身的存档文件, 而不能是任意的 png, jpg 等图片文件.



QPicture 类似于很多游戏的 Replay 功能.

例如像 war3 这样的经典游戏, 即使是一场 60 分钟的膀胱局, 生成的 replay 文件, 也不过几百个 KB.

此处的 Replay 功能并非是把整个游戏画面都录制保存下来, 而是记录了地图中发生的所有事件(地图元素, 玩家单位操作, 中立生物行为等...).

当回放 Replay 的时候其实就是把上述记录的事件再一条一条的执行一遍即可还原之前的游戏场景了。

不了解游戏的同学, 也可以理解成警察蜀黍录笔录, 并通过笔录还原案发现场。

如果要记录下 QPainter 的命令, 首先要使用 QPainter::begin() 函数, 将 QPicture 实例作为参数传递进去, 以便告诉系统开始记录, 记录完毕后使用 QPainter::end() 命令终止。如下示例:

示例:

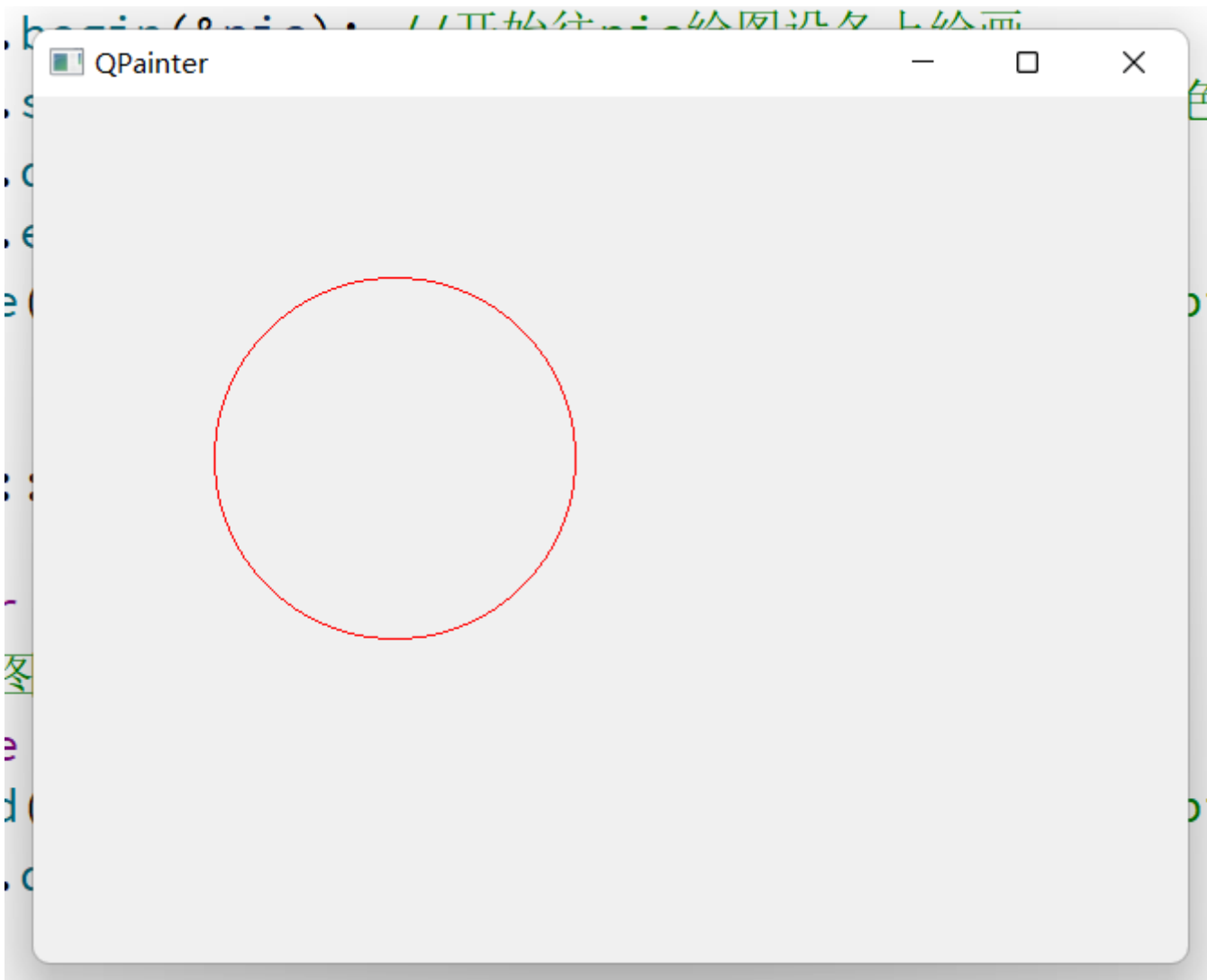


```
1  #include "widget.h"
2  #include <QPainter>
3  #include <QPicture>
4
5  Widget::Widget(QWidget *parent)
6      : QWidget(parent)
7  {
8      QPicture pic;
9      QPainter painter;
10     painter.begin(&pic); //开始往pic绘图设备上绘画
11     painter.setPen(QPen(Qt::red)); //设置画笔颜色为蓝绿色(青色)
12     painter.drawEllipse(QPoint(200,200),100,100);
13     painter.end(); //结束绘画
14     pic.save("C:\\Users\\Lenovo\\Desktop\\Test_Pic\\pic.pic");
15 }
16
17 void Widget::paintEvent(QPaintEvent *)
18 {
19     QPainter painter(this); //实例化画家对象
20     //重现绘图指令
21     QPicture pic;
22     pic.load("C:\\Users\\Lenovo\\Desktop\\Test_Pic\\pic.pic"); //加载图片
23     painter.drawPicture(0,0,pic);
24 }
25
```

实现效果:



通过 QPicture 重现绘图指令后，实现的效果如下：



### 3. 其他话题

Qt 中对于界面的优化美化, 还涉及到很多其他的话题. 大家未来在工作中如果涉及到了, 再针对性学习即可.

- Qt 动画
- Qt 3D 图形
- QQuick
- 使用第三方控件库
- Qt Design Studio
- .....

这些内容我们就不再一一介绍了.

这部分内容更偏向 "设计美工" 方面的内容, 作为普通程序猿也不需要过多关注.