

Qt Style Sheet 开发总结

第一版

作者： 张涛

联系 QQ: 359127232

目录

简介	4
语法	4
基本语法.....	4
选择器	5
通用选择器.....	5
格式.....	5
注意点.....	5
一般用法.....	5
类型选择器.....	5
格式.....	5
注意点.....	6
一般用法.....	6
类选择器.....	7
格式.....	7
一般用法.....	7
ID 选择器	7
格式.....	7
注意点:.....	8
一般用法.....	8
后代选择器.....	9
格式.....	9
注意点:.....	9
一般用法.....	9
子元素选择器.....	10
格式.....	10
注意点.....	10
一般用法.....	10
属性选择器.....	11
格式.....	11
注意点.....	11
一般用法.....	12
并集选择器.....	12
格式.....	12
注意点.....	12
一般用法.....	12
两个特殊的选择器.....	13
子控件选择器.....	13
伪类选择器.....	14
没有选择器的情况.....	15
选择器的匹配规则.....	15
Qss 的特性.....	17

层叠性.....	17
继承性(Qt-Version >= 5.7).....	17
优先级.....	18
Qt 官方关于冲突解决的说明.....	19
盒模型	21
什么是盒模型?.....	21
盒模型中的宽度与高度.....	22
属性	22
背景属性 background.....	23
background-color	23
background-image	23
background-repeat.....	23
background-position	24
background-attachment.....	25
background-clip.....	26
background-origin	28
背景属性的连写格式.....	29
前景属性 color	30
边框属性 border.....	31
属性.....	31
格式.....	40
字体属性 font.....	43
font-style	43
font-weight.....	43
font-size.....	44
font-family	44
连写格式.....	44
文本属性.....	45
text-align	45
text-decoration.....	46
padding 和 margin	47
padding	47
margin	47
width 与 height	49
width, height	49
max-width min width 与 max-height min-height	49
outline	50
属性结语.....	50
Brush 类型介绍	50
Color.....	50
Gradient	51
与 CSS 的对比.....	54
选择器对比.....	54
属性对比.....	55

简介

为了书写方便，文中一律使用 Qss 代替 Qt style sheet.

首先来看 Qt 的官方介绍：除了子类化 QStyle 以外，Qss 是一个非常强大的用于自定义控件外观的机制。它的概念、术语以及语法都是受到了 HTML CSS 的启发(实际上就是 CSS 的语法)，但可以适应全局窗口部件(这一句我没明白是什么意思)。

就是说，Qss 是用来设置界面样式的，设置的方法就是类似于 CSS, 通过以一定规则组织的字符串来给界面设置样式，而不用我们调用控件对象的接口或子类化 QStyle 去设置样式。这种组织字符串的规则就是它的语法，以下的总结也是主要讲一些 Qss 的语法，包含少量的经验以及官方文档中没有说明的内容。

个人认为使用 Qss 的好处：

- (1) 可读性高并且非常直观, 便于设置界面样式;
- (2) 在每个平台上都有相同的显示效果.
- (3) 可以在界面代码中省去与显示效果相关的大量代码, 将界面逻辑独立出来.
- (4) 在界面风格(配色, 字体等)改变的情况下, 可以不用修改 c++源码就可以实现.

语法

基本语法

样式表由一系列的样式规则组成。一条样式规则由一个选择器和一个声明语句组成，选择器指明了哪个（或者说是哪种）控件将会受规则影响，而声明语句则指明了哪些属性会设置到这个（这些）控件。语法如下：

```
selector { attribute: value; }
```

在上面这条语句中，selector 代表选择器，指明了哪个(或者说是哪种)控件将会受到规则影响。{attribute: value;}代表声明语句，其中 attribute 表示属性，value 表示该属性的值，属性与它的值之间必须以冒号(：)隔开，属性值后面必须以分号(；)结束，表示这条属性已经设置完成。整条语句加起来的意思是，在整个应用程序中，被 selector 匹配的控件，它们控件的 attribute 属性的值应该被设置为 value。

例如：

QPushButton{ color: red;}表示将我们的应用程序中所有的 QPushButton 对象以及它们的子类对象应该使用 red 作为它们的前景色(即字体的颜色)；

由此看来，我们要学会如何使用 qss 来控制我们的界面样式，只需要学会两个方面的内容，即选择器和属性，下面将用专门的章节来讲解这两部分的内容。

选择器

通用选择器

又叫通配符选择器

格式

***** { 属性: 值; }

通用选择器用 (*) 来表示, 它表示匹配程序中所有的 widget.

注意点

由于通用选择器会匹配程序中所有的 widgets, 效率较低, 因此应该尽量减少或者不使用

一般用法

通用选择器一般用来给应用程序设置统一的字体, 例如

```
*{font: normal 20px "微软雅黑";}
```

这条语句表示将程序中所有 widget 的字体大小都设置为 20px 大小, 字体采用微软雅黑.

类型选择器

格式

类名 { 属性: 值; }

类名即 Widget 类名, 由 `QObject::metaObject()::className()` 获取, 类型选择器匹配所有该类以及该类的派生类的对象. 例如:

```
QPushButton{  
    color: blue;  
}
```

这条语句表示，程序中所有的 QPushButton 类和它的派生类的对象，它们的前景色(即文字颜色)被设置为蓝色。

注意点

Qt 样式表使用 widget 的 QObject::className()来决定何时应用类型选择器。当自定义控件在命名空间之中(或它是一个嵌套类)，QObject::className()会返回(::)，这与后面介绍的子控件选择器相冲突。为了解决这个问题，当为命名空间中 widget 使用类型选择器时，我们必须将 "::" 替换成 "--"，下面即将介绍的类选择器也是一样。例子如下：

```
namespace ns {
    class MyPushButton : public QPushButton {
        // ...
    }
}

// ...
qApp->setStyleSheet("ns--MyPushButton { background: yellow; }");
```

一般用法

类型选择器会匹配所有该类以及该类的派生类的对象，所以我们在程序中，有时为了统一一些具有相似性的控件的样式，可以使用类型选择器，如，我们想要为 QSpinBox, QDoubleSpinBox, QDateTimeEdit, QTimeEdit, QDateEdit 等这些编辑框的控件设置一些相同的样式，因为它们都是 QAbstractSpinBox 类的派生类，因此可以如下写：

```
QAbstractSpinBox{
    min-height: 30px;
    max-height: 30px;
    border-width: 1px;
    border-style: solid;
    border-color: gray;
    padding: 0px;
}
```

类选择器

格式

.类名 { 属性: 值; }

这里的类名与类型选择器中的类名一样, 不同的是, 类选择器的类名前面有一个(.), 这种选择器只会匹配该类的所有对象, 而不会匹配其派生类的对象.

```
.QPushButton{  
    color: blue;  
}
```

一般用法

类选择器提供了一种匹配所有该类的对象但不会匹配派生类的方法, 通常用来特例化拥有派生类的类对象, 但不仅限于此.例如在在我的应用程序中, 我用 QFrame 来作为容器 widget, 此时我想对它设置一些样式, 但又不想影响它的子类对象(QLabel, QAbstractScrollArea 等等), 那么我可以使用类选择器给所有的 QFrame 设置相同的样式 . 例如:

```
.QFrame{  
    padding: 15px 25px;  
}
```

ID 选择器

格式

#id{ 属性: 值; }

这里的 id 指的是 objectName, 每个 QObject 类及其派生类都有一个属性, “#” + objectName 构成了我们的 ID 选择器, 它匹配所有 objectName 为 ID 选择器所指定的名称的对象, 为其设置样式. 例如:

```
#button_1{  
    color: red;  
}
```

注意点:

1. `objectName` 是大小写敏感的.
2. “#”与 ID 之间不可以有空格
3. 由于 `objectName` 是所有 `QObject` 类对象的一个属性, 在运行过程中可以改变, 所以一般情况下, 要使用 ID 选择器时, 保证 `objectName` 不要在运行时被改变, 否则重新加载 `stylesheet` 文件时, 对应的 ID 选择器将不会匹配到原来的控件.
4. 由于 `objectName` 允许字符串中含有空格, 但是 ID 选择器中, ID 是从紧跟#后的第一个字符开始直到遇到空格或“{”之间的字符串, 因此, 如果是为了使用 ID 选择器而设置 `objectName`, 则 `objectName` 中不能含有空格
5. 由于任何对象的 `objectName` 都可以出现重复, 因此在设置 `objectName` 时, 尽量保持其唯一性
6. Qt 官方给出的 ID 选择器的格式为:

类名#id{ 属性: 值; }

但实际上不加类名也是可以的(加了类名的 ID 选择器在 CSS 中被称为交集选择器), 在正式开发中, 还是建议加上类名, 因为这样可以看出这个 id 选择器所匹配的对象类型, 有利于提高阅读性.

基于以上特点, 我们在设置 `objectName` 时, 一般使用下划线“_”连接的多个单词表明此对象的功能.

一般用法

ID 选择器一般用于为比较特殊的控件设置样式, 例如在我的某个页面中, 需要突出一个重要的按钮, 那么此时我可以给这个按钮设置一个独特的样式用以提醒用户, 如:

```
QPushButton#settings_popup_fileDialog_button{
    min-height: 31px;
    min-width: 70px;
    border: 1px solid black;
    color: #F0F0F0;
    min-height: 10px;
    border-radius: 3px;
    background: qlineargradient(spread:pad, x1:0, y1:0, x2:0, y2:1,
stop:0 #454648, stop:1 #7A7A7A);
}
```


后代选择器

格式

选择器 1 选择器 2{ 属性: 值; }

这个选择器表示: 在选择器 1 匹配的所有对象中, 找到选择器 2 所匹配的所有后代对象, 并给它们设置样式.

注意点:

1. 后代选择器必须用空格隔开每个选择器
2. 后代选择器可以通过空格一直延续下去, 例如:
选择器 1 选择器 2 选择器 3 ... 选择器 N{ 属性: 值; }
3. 顾名思义, 后代选择器不仅包含“儿子”, 还包含“孙子”, “重孙子”等, 一般来说, 只要 B 控件显示在 A 控件上, 那么 B 控件就是 A 控件的后代.
4. 后代选择器不仅可以使⤵用类型选择器, 还可以使用类选择器, id 选择器等.
5. Qt 中, 各控件的父子关系:
通过简单的验证, 各控件的父子关系并非我们在创建对象时所指定的那样, 实际父子关系取决于如何布局.

一般用法

后代选择器一般用于指定特定类的后代的样式, 例如在我的应用程序中, 有很多个相似的对话框, 它们中包含一些样式相同的按钮, 那么我可以使用后代选择器为他们指定样式, 例如:

```
BaseDialog QPushButton{  
    min-width: 120px;  
    min-height: 40px;  
    max-width: 120px;  
    max-height: 40px;  
    font-size: 20px;  
    padding: 0px;  
}
```

子元素选择器

格式

选择器 1 > 选择器 2 { 属性: 值; }

子元素选择器表示找到指定选择器所匹配的对象中的所有特定直接子元素然后设置属性, 即找到选择器 1 匹配到的对象中的被选择器 2 匹配到的直接子元素然后设置属性

注意点

1. 子元素选择器必须用">"连接, ">"两边有没有空格都可以, 但是不建议写空格, 因为会与后代选择器的连接符混淆.
2. 子元素选择器只会查找"儿子", 不会查找其他后代.
3. 子元素选择器不仅可以使用类型选择器, 还可以使用类选择器, id 等选择器
4. 子元素选择器不能通过">"一直延续下去, 只能有一个">"
5. 由于 Qt 中有继承关系的 Widgets 较多, 在使用子元素选择器时, 请特别注意继承关系, 比如我只想选中 QGroupBox 中的 QPushButton, 那么我即可以写成

QWidget > QPushButton {color: red;} ①

也可以写成

QGroupBox > QPushButton {color: red;} ②

这是因为 QGroupBox 是 QWidget 的派生类, 类型选择器 QWidget 会选中所有它的派生类对象, 这些对象中包括 QGroupBox, 因此写法 ① 会将所有的 QPushButton 的前景色设置为红色.

鉴于此种情况, 我推荐在使用子元素选择器时, 使用类选择器替代类型选择器

一般用法

子元素选择器一般用于一些特定布局条件中的控件, 例如我想给直接布局在 QGroupBox 的 QCheckBox 设置一些特定属性, 那么可以这样做:

```
.QGroupBox > .QCheckBox {  
    color: blue;  
}
```

属性选择器

格式

[attribute=value]{ 属性: 值; }

[attribute|=value]{ 属性:值; }

[attribute~=value]{ 属性:值; }

attribute=value 表示匹配有特定属性 **attribute**, 并且值为 **value** 的所有控件, 然后设置样式;
attribute|=value 表示匹配有特定属性 **attribute**, 并且值以 **value** 开头的所有控件, 然后设置样式;
attribute~=value 表示匹配有特定属性 **attribute**, 并且值包含 **value** 的所有控件, 然后设置样式;

注意点

1. **attribute|=value** 表示 **attribute** 属性的值以 **value** 开头, 无论 **value** 后面还有没有值, 或者 **value** 后面是什么, 均能匹配到, 例如:

```
[objectName|="button"]{  
    color: red;  
}
```

这表示将 **objectName** 属性以 **button** 开头的所有控件的前景色设置为红色.

attribute~=value 表示 **attribute** 属性的值中包含 **value**, 这里要注意的是:**value** 必须是独立的单词, 也就是包含 **value** 并且 **value** 是被空格隔开的, 例如:

```
[objectName~="button"]{  
    color: red;  
}
```

而我在代码中, 设置的 **objectName** 的语句为:

```
pBtn1->setObjectName("button12345");  
pBtn2->setObjectName("abc button 2");
```

结果是只匹配到了 **pBtn2** 所指的對象

2. 官方文档的解释:

通常情况下, 这里的属性指的是, 使用 `Q_PROPERTY` 宏所声明的属性, 并且属性类型要受 `QVariant::toString()` 支持.

这个选择器类型也可以用来判断动态属性, 要了解更多使用自定义动态属性的细节, 请参考使用自定义动态属性。

除了使用 `=`, 你还可以使用 `~=` 来判断一个 `QStringList` 中是否包含给定的 `QString`。

警告: 如果在设置了样式表后, 相应的属性值发生了改变(如: `flat` 变成了 `"true"`), 则有必要重新加载样式表, 一个有效的方法是, 取消样式表, 再重新设置一次, 下面的代码是其中一种方式:

```
style()->unpolish(this);
style()->polish(this);
```

一般用法

属性选择器一般不常用, 如果要用, 请参照官方文档的方法使用

并集选择器

格式

选择器 1,选择器 2,选择器 3{ 属性: 值; }

并集选择器表示, 将每个单独选择器匹配到的控件放在同一个结果集中, 并给结果集中的每个控件都设置声明语句中的样式.

注意点

1. 并集选择器必须使用 `,` 来连接不同的选择器
2. 并集选择器可以使用类选择器, 类型选择器, `id` 选择器, 属性选择器等.

一般用法

主要用于给具有相同属性并且外观相似的的控件设置样式, 例如:

```
.QLineEdit, .QComboBox {
    border: 1px solid gray;
    background-color: white;
}
```

两个特殊的选择器

子控件选择器

格式

类型选择器::子控件{ 属性: 值; }

类选择器.子控件{ 属性: 值; }

表示对类型选择器或类选择器指定的所有控件的子控件设置样式;

Qt 官方说明

为了样式化你的复杂 widget，很有必要使用 widget 的 subcontrol，比如 QComboBox 的 drop-down 部分或者是 QSpinBox 的上和下箭头。选择器也许会包含 subcontrols 用于限制 widget 的 subcontrols，举个例子：

```
QComboBox::down-arrow {  
    image: url(:/res/arrowdown.png);  
}
```

上述规则样式化所有 QComboBox 的 drop-down 部分，虽然双冒号(::)让人联想到 CSS3 的伪元素语法，但是 Qt 的 Sub-Controls 跟它是不一样的。

Sub-Controls 始终相对于另一个元素来定位 - 一个参考元素。这个参考元素可以是一个 Widget 又或者是另一个 Sub-Control。举个例子，QComboBox 的::drop-down 默认被放置于 QComboBox 的 Padding rectangle(盒子模型)的右上角。::drop-down 默认会被放置于另一个::drop-down Sub-Control 的中心。查看可样式化的 Widget 列表以了解更多使用 Sub-Control 来样式化 Widget 和初始化其位置的内容。

源 rectangle 可以使用 subcontrol-origin 来改变。举个例子，如果我们想要把 drop-down 放置于 QComboBox 的 margin rectangle 而不是默认的 Padding rectangle，我们可以像下面这样指定：

```
QComboBox {  
    margin-right: 20px;  
}  
QComboBox::drop-down {  
    subcontrol-origin: margin;  
}
```

drop-down 在 Margin rectangle 内的排列方式可以由 subcontrol-position 来改变。

width 和 height 属性可以用来控制 Sub-control 的 size。需要注意的是，设置了 image 就隐

式的设置了 Sub-control 的 size 了。

相对定位方案 (position: relative), 允许 Sub-Control 的位置从它的初始化位置作出偏移。举个例子, 当 QComboBox 的 drop-down 按钮被 pressed 时, 我们也许想要那个箭头作出位移以显示一种 “pressed” 的效果, 为了达到目标, 我们可以像下面那样指定

```
QComboBox::down-arrow {  
    image: url(down_arrow.png);  
}  
QComboBox::down-arrow:pressed {  
    position: relative;  
    top: 1px; left: 1px;  
}
```

绝对定位方案(position : absolute), 使得 Sub-control 的 position 和 size 基于其参考元素而改变。

一旦定位, 它们将会与 widget 同等对待并且可以使用盒子模型来样式化。

查看 Sub-Control 列表以了解那些 sub-control 是被支持的, 并且可以查看自定义 QPushButton 的菜单指示器 Sub-Control 来了解一个实际的使用例子。

注意: 像 QComboBox 和 QScrollBar 这样的复杂部件, 如果 sub-control 的一项属性是自定义的, 那么其他所有的属性跟 sub-control 也都应该自定义。

伪类选择器

格式

类型选择器:状态{ 属性: 值; }

类选择器:状态{ 属性: 值; }

表示对类型选择器或类选择器指定的所有控件设置它在指定状态时的样式。

Qt 官方说明

选择器也许会包含基于 widget 的 state 的程序限制规则的伪状态。伪状态以冒号(:)作为分隔紧跟着选择器。举个例子, 下面的规则在鼠标悬浮在 QPushButton 的上方时生效:

```
QPushButton:hover { color: white }
```

伪状态可以使用感叹号进行取反, 下面一条规则在鼠标没有悬浮在 QRadioButton 上方时生效:

```
QRadioButton:!hover { color: red }
```

伪状态可以链接，在这样的情况下，隐式地包含了逻辑与。举个例子，下面一条规则在鼠标悬浮到一个已 check 的 QCheckBox 上时生效：

```
QCheckBox:hover:checked { color: white }
```

伪状态的取反也可以出现在伪状态链中，举个例子，下面的规则在鼠标悬浮到一个没有被 press 的 QPushButton 上时生效：

```
QPushButton:hover:!pressed { color: blue; }
```

如果有需要，可以使用逗号来表示逻辑或，即并集选择器

```
QCheckBox:hover, QCheckBox:checked { color: white }
```

伪状态可以与 subcontrol 组合使用，举个例子：

```
QComboBox::drop-down:hover { image: url(dropdown_bright.png) }
```

没有选择器的情况

如果在 c++ 的代码中直接调用控件对象的 setStyleSheet 函数来设置样式，但样式中没有任何选择器，例如下面这样

```
pBtn1->setStyleSheet("color: green;");
```

即使这种写法可以生效，但它不符合语法规则，因此不推荐使用。

经过测试，这样的语句被忽略的选择器相当于通用选择器或下面例子中的选择器，假如 pBtn1 是一个 QPushButton 对象的指针，那么这条语句等价于

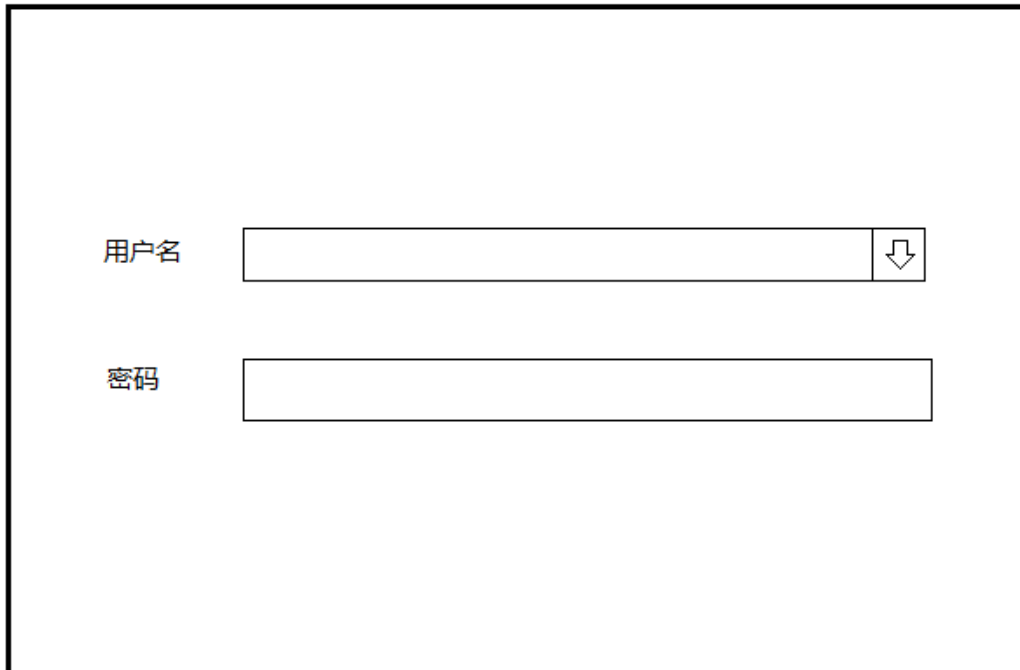
```
pBtn1->setStyleSheet("QPushButton, QPushButton *{color: green;}");
```

选择器的匹配规则

之所以要把这个问题单独作为一个小节来写，是因为在日常开发过程中，经常会犯这方面的错误。

下面看一个例子：

假如需要完成一个界面，如下图所示



这里用户名输入框是一个 `QComboBox` 对象, 密码输入框是一个 `QLineEdit` 对象, 它们的父控件是一个 `QDialog`, 有这样一个需求: 给这两个输入框设置相同的边框属性: 1 个像素宽的蓝色实线框, 为了方便更改风格, 我有一个 `css` 文件, 将所有样式都写在了这个文件里, 这时, 观察发现, 这两个控件都是 `QDialog` 的子控件, 于是可以用后代选择器或者子元素选择器, 如下: 第一种:

```
QDialog QComboBox, QLineEdit {  
    border: 1px solid blue;  
}
```

第二种:

```
QDialog>QComboBox, QLineEdit {  
    border: 1px solid blue;  
}
```

当我写完并运行程序后, 发现无论采用哪种写法 `QComboBox` 是正常的, 但是我的程序界面中, 其他所有的 `QLineEdit` 的边框都变成了 1 个像素宽的蓝色实线框, 而这并不是我想要的效果.

因此对于上面的现象, 我们很容易得出结论: **多个选择器组合使用时, 它们的结合方向是自右向左, 而不是我们认为的自左向右.** 也就是说, 这两个选择器分别被理解为 `(QDialog QComboBox)`, `QLineEdit` 和 `(QDialog>QComboBox)`, `QLineEdit`.

其实, 这应该与 `CSS` 的选择器匹配规则是一样的, 是为了提高效率的一种做法, 具体原因这里不细谈.

正确的写法应该是

```
QDialog QComboBox, QDialog QLineEdit{  
    border:1px solid blue;  
}
```

或

```
QDialog>QComboBox, QDialog>QLineEdit{  
    border:1px solid blue;  
}
```

Qss 的特性

层叠性

qss 的语法来源于 css, 而 css 的全称是 Cascading StyleSheet, 翻译过来叫做层叠样式表, 也叫级联样式表, 本文中一律使用层叠样式表。

层叠性是 css 处理冲突的一种能力。

只有在多个选择器匹配到同一个控件时才会发生层叠性, 如下面的例子:

```
pBtn1->setStyleSheet("QPushButton{color: blue;}");  
pBtn1->setStyleSheet(".QPushButton{color: green;}");
```

这两个选择器匹配到了同一个按钮, 结果是后面的样式覆盖掉了前面的, 这就是层叠现象。

继承性(Qt-Version >= 5.7)

在典型的 CSS 中, 如果一个标签的字体和颜色没有显式设置, 它会自动从其父亲获得。当使用 Qt 样式表时, 控件**不会**从其父亲继承字体和颜色的设置(请注意, 父亲和父类、孩子和子类都是不同的概念, 不要搞混)

举个例子, 考虑一个 QGroupBox 内有一个 QPushButton:

```
qApp->setStyleSheet("QGroupBox{ color: red; }");
```

QPushButton 没有任何显式的 color 设置。因此, 它会获得系统的颜色而不是从父亲继承 color 的值。如果我们要设置 QGroupBox 及其所有孩子的 color, 我们可以这样写:

```
qApp->setStyleSheet("QGroupBox,QGroupBox * { color: red; }");
```

注意 QGroupBox 和*之间的空格。

与此相反,使用 `QWidget::setFont()` 可以设置字体包括孩子的字体,使用 `QWidget::setPalette()` 可以设置调色板包括孩子的调色板。

如果想要字体和调色板被孩子继承,可以给 `QApplication` 设置 `Qt::AA_UseStyleSheetPropagationInWidgetStyles` (Qt5.7 加入) 属性,例如:

```
QCoreApplication::setAttribute(Qt::AA_UseStyleSheetPropagationInWidgetStyles, true);
```

优先级

为什么要有优先级?

当一个控件被多个选择器选中并且设置了相同的属性(值不同)时,不能仅仅根据设置样式语句出现的先后顺序进行层叠,那么控件的样式如何确定,于是引出了选择器的优先级问题。

一般通过下面两步进行选择器优先级的判定。

第一步： 设置方式所产生的优先级问题

在 CSS 中,有如下层叠优先级规则:

内联样式 > 内部样式 > 外部样式 > 浏览器缺省

而在 Qss 中,这个规则表现为:

给控件直接设置的样式 > 给 QApplication 设置的样式

就是说,调用控件的 `setStyleSheet` 设置的样式的优先级永远高于给 `QApplication` 设置的样式,即使 `QApplication` 中的选择器优先级更高

第二步： 样式表本身的优先级问题

当设置方式相同,且几个样式规则为同一个控件的同一个属性指定不同的值时,就产生了冲突.此时,如何层叠就由选择器的优先级来确定。

一般而言,选择器越特殊,它的优先级越高。也就是选择器指向的越准确,它的优先级就越高。

优先级判断的三种方式

1. 间接选中

间接选中就是指继承,也就是在 Qt5.7 及以上版本,程序中给 `QApplication` 对象设置了 `Qt::AA_UseStyleSheetPropagationInWidgetStyles` 属性时,才会有间接选中。

如果是间接选中,那么最终的样式就是离目标最近的那个,这里的近指的是两个控件的父子关系.例如一个 `QPushButton` 对象被布局在 `QGroupBox` 中,而 `QGroupBox` 又被布局

在 QWidget 中, 此时如果给 QGroupBox 和 QWidget 都设置了 color 属性的颜色, 那么无论设置顺序如何, QPushButton 的前景色总是表现为 QGroupBox 设置的颜色, 因为 QGroupBox 显然是离 QPushButton 最近的那一个.

2. 相同选择器(直接选中)

如果都是直接选中, 并且都是同类型的选择器, 那么写在后面的样式会覆盖掉前面的样式, 例如

```
pBtn1->setStyleSheet("QPushButton{color: green;}");
pBtn1->setStyleSheet("QPushButton{color: blue;}");
```

显而易见, pBtn1 的前景色是蓝色.

3. 不同选择器(直接选中)

如果都是直接选中, 并且不是相同类型的选择器, 那么就会按照选择器的优先级来层叠. 具体的优先级如下:

id > 类 > 类型 > 通配符 > 继承 > 默认

优先级权重

为什么会有优先级权重?

当多个选择器混合在一起使用时, 我们可以通过计算权重来判断谁的优先级最高, 从而确定控件的样式.

注意点: 只有选择器是直接选中控件时才需要计算权重, 否则直接选择器高于一切间接选中的选择器

优先级权重的计算方式:

1. 计算选择器中的 id 选择器数量[=a]
2. 计算选择器中类选择器的数量+属性选择器的数量[=b]
3. 计算选择器中类型选择器的数量[=c]
4. 忽略子控件选择器

串联这三个数字 a-b-c 就得到优先级权重, 数字越大优先级越高.

这里给出我写的一个例子, qss 文件位于资源文件中:



Qt 官方关于冲突解决的说明

当几个样式规则为同一个属性指定不同的值时, 就产生了冲突. 请考虑下面的样式表:

```
QPushButton#okButton { color: gray; }
QPushButton { color: red; }
```

两条规则都匹配名为 okButton 的 QPushButton 实例并且冲突于颜色属性. 为了解决冲突, 我们必须考虑到选择器的特殊性. 在上面的例子中, QPushButton#okButton 被视为比

QPushButton 更特殊, 因为它 (通常) 指向一个单一的对象而不是 QPushButton 的所有实例。

相似的, 指定了伪状态的选择器比没有指定伪状态的更特殊。从而, 下面的样式表指明了当鼠标悬浮到 QPushButton 上方时其字体颜色应该为白色, 而其余情况则为红色:

```
QPushButton:hover { color: white; }
QPushButton { color: red; }
```

接下来看一个很有意思的:

```
QPushButton:hover { color: white; }
QPushButton:enabled { color: red; }
```

两个选择器都有相同的特殊性, 所以当鼠标悬浮在一个 enabled 的按钮上时, 第二条规则优先。如果在这种情况下我们想要文字变成白色, 我们可以像下面那样重新排布一下样式规则:

```
QPushButton:enabled { color: red; }
QPushButton:hover { color: white; }
```

另外, 我们可以使第一条规则更特殊一些:

```
QPushButton:hover:enabled { color: white; }
QPushButton:enabled { color: red; }
```

相似的问题出现在相互配合的类型选择器上。考虑以下情况:

```
QPushButton { color: red; }
QAbstractButton { color: gray; }
```

两条规则都应用于 QPushButton 的实例 (因为 QPushButton 继承于 QAbstractButton) 并且冲突于 color 属性。因为 QPushButton 继承于 QAbstractButton, 这让人不禁想到 QPushButton 比 QAbstractButton 更特殊。然而, 对于样式表的运算, 所有的类型选择器都具有同等的特殊性, 并且出现在更后面的规则优先级更高。换句话说, QAbstractButton 的 color 会被设置成灰色, 包括 QPushButton。如果我们确实想要 QPushButton 字体颜色设置为红色, 我们总是可以使用重新排列样式表规则顺序的方式实现。

为确定规则的特殊性, Qt 样式表跟随 CSS2 规范

一个选择器的特殊性由下面的方式计算:

- 计算选择器中 ID 属性的数量[=a]
- 计算选择器中其他属性和伪类的数量[=b]
- 计算选择器中元素名字的数量[=c]
- 忽略伪原素[如:subcontrol]

串联这三个数字 a-b-c (在一个大基数的数字系统) 就得到了特殊性等级。

举个例子:

```
* {} /* a=0 b=0 c=0 -> specificity = 0 */
LI {} /* a=0 b=0 c=1 -> specificity = 1 */
UL LI {} /* a=0 b=0 c=2 -> specificity = 2 */
UL OL+LI {} /* a=0 b=0 c=3 -> specificity = 3 */
```

```
H1 + *[REL=up]{} /* a=0 b=1 c=1 -> specificity = 11 */  
  
UL OL LI.red {} /* a=0 b=1 c=3 -> specificity = 13 */  
  
LI.red.level {} /* a=0 b=2 c=1 -> specificity = 21 */  
  
#x34y {} /* a=1 b=0 c=0 -> specificity = 100 */
```

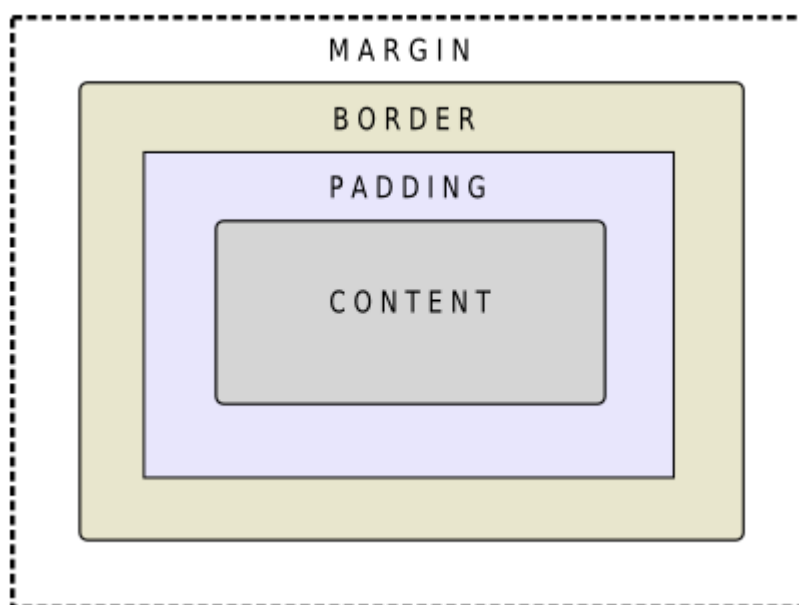
盒模型

在讲解属性之前，我们必须了解一下 QSS 的盒模型。

什么是盒模型？

盒模型仅仅是一个形象的比喻，所有的 **widget** 都被看做是一个“盒子”，一个盒子包括：外边距，边框，内边距，和实际内容。它们可以看作是有包含关系的矩形，并且这种包含关系是固定不变的。

如图所示



- **Margin**（外边距） - 与其他盒子之间的距离。

- **Border（边框）** - 外边距与内边距之间的区域,边框有自己的颜色不会受到盒子的背景颜色影响
- **Padding（内边距）** - 内容和边框之间的区域,会受到背景颜色影响.
- **Content（内容）** - 盒子的内容,显示文本,图像或其他控件

除了内容外, 其他三个部分均不能单独设置颜色, 只能设置其宽度, 并且是以像素为单位

对比一张生活中的一张照片墙来看会更容易理解



盒模型中的宽度与高度

在属性中将要学到的 `width`, `height` 两个属性, 设置的均是盒子的**内容**的宽高, 而我们在 `c++` 代码中的窗口的 `width` 与 `height` 指的是**整个盒子的宽度与高度**, 这一点非常重要.

整个盒子的宽度应该等于:

左外边距 + 左边框 + 左内边距 + 内容宽度 + 右内边距 + 右边框 + 右外边距,
同理, 整个盒子的高度也是上下外边距,内边距,边框和内容高度的和.

属性

属性即控件的具体外观样式, 比如背景颜色, 边框宽度等等. 本节主要列举一些常用的属性, 并介绍它们的具体格式或取值

注意:一个属性并不是被所有 **widget** 都支持的, 要想查看什么 **widget** 支持哪些属性, 或一个属性被哪些 **widget** 支持, 请查看文档后面给出的官方链接.

背景属性 **background**

背景共有 7 个属性, 既可以每个属性单独设置, 也可以连写, 下面将对他们逐一进行分析, 并在最后给出其连写格式

background-color

取值: **Brush** 类型(**Brush** 类型介绍见本节最后)

作用: 设置控件的背景颜色, 默认是 **border** 之内的矩形区域, 包括内边距和内容矩形.

background-image

取值: **Url** 类型, 格式是 **url(filename)**, **filename** 是一个本地文件路径或 **Qt** 资源文件路径, 不支持网络图片

作用: 设置控件的背景图片. 可以与背景颜色共存, 背景图片会覆盖背景颜色, 在没有被图片覆盖的区域, 显示背景颜色.

background-repeat

取值:

repeat-x: 在水平方向上平铺

repeat-y: 在垂直方向上平铺

repeat: 在水平和垂直方向上都平铺, 这是默认值(但是 **Qt** 好像有 **bug**, 设置了 **repeat** 反而不会平铺, 不设置才平铺)

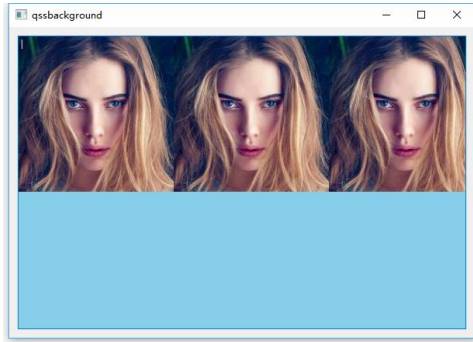
no-repeat: 不平铺

作用: 设置背景图片的平铺方式.

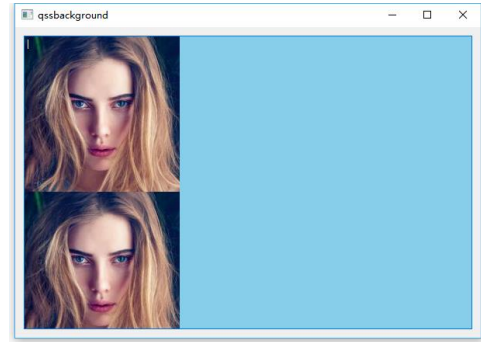
例子:

给一个 **QTextEdit** 设置背景图片, 代码和效果分别如下

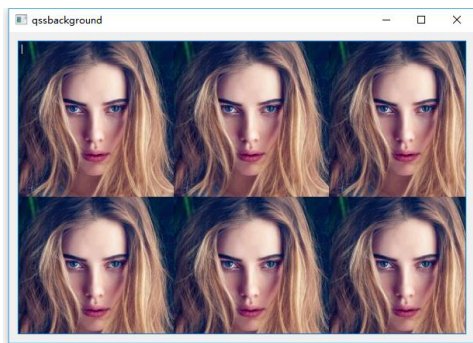
```
QTextEdit{
    background-color: skyblue;
    background-image: url(:/resource/girl.jpg);
    background-repeat: no-repeat;/*repeat-x; repeat-y; repeat;*/
}
```



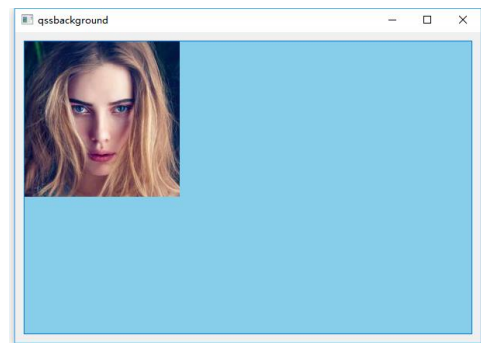
repeat-x



repeat-y



repeat



no-repeat

background-position

取值:

top: 向上对齐

left: 向左对齐

bottom: 向下对齐

right: 向右对齐

center: 居中

格式:

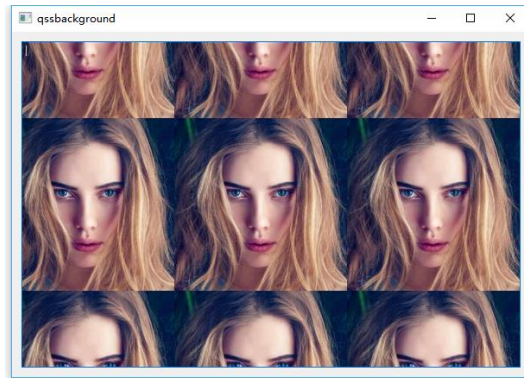
background-position: 水平对齐方式 垂直对齐方式;

这是 css 规定的标准顺序, 而 qss 并未严格规定, 但建议按照 css 的顺序写

作用: 设置背景图片的对齐方式, 默认情况下向左向上对齐,

举例: 水平向左垂直居中对齐, 代码和效果如下

```
QTextEdit{
    background-color: skyblue;
    background-image: url(:/resource/girl.jpg);
    background-position: right center;
}
```

background-attachment

取值:

scroll: 滚动, 这是默认值

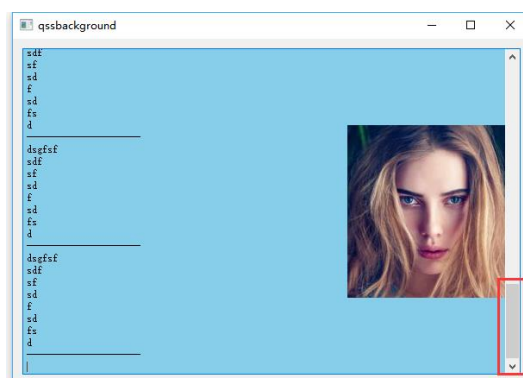
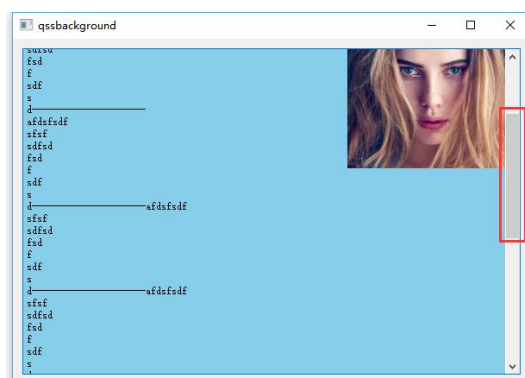
fixed: 固定

作用: 设置背景图片在带滚动条的控件(QAbstractScrollArea)中是固定在一个位置还是随着滚动条滚动.

比如: css 代码分别如下

```
QTextEdit{
    background-color: skyblue;
    background-image: url(/resource/girl.jpg);
    background-repeat: no-repeat;/*repeat-x; repeat-y; repeat;*/
    background-position: right center;
    background-attachment: scroll;
}
```

```
QTextEdit{
    background-color: skyblue;
    background-image: url(/resource/girl.jpg);
    background-repeat: no-repeat;/*repeat-x; repeat-y; repeat;*/
    background-position: right center;
    background-attachment: fixed;
}
```



background-clip

取值:

margin: 外边距矩形

border: 边框矩形

padding: 内边距矩形

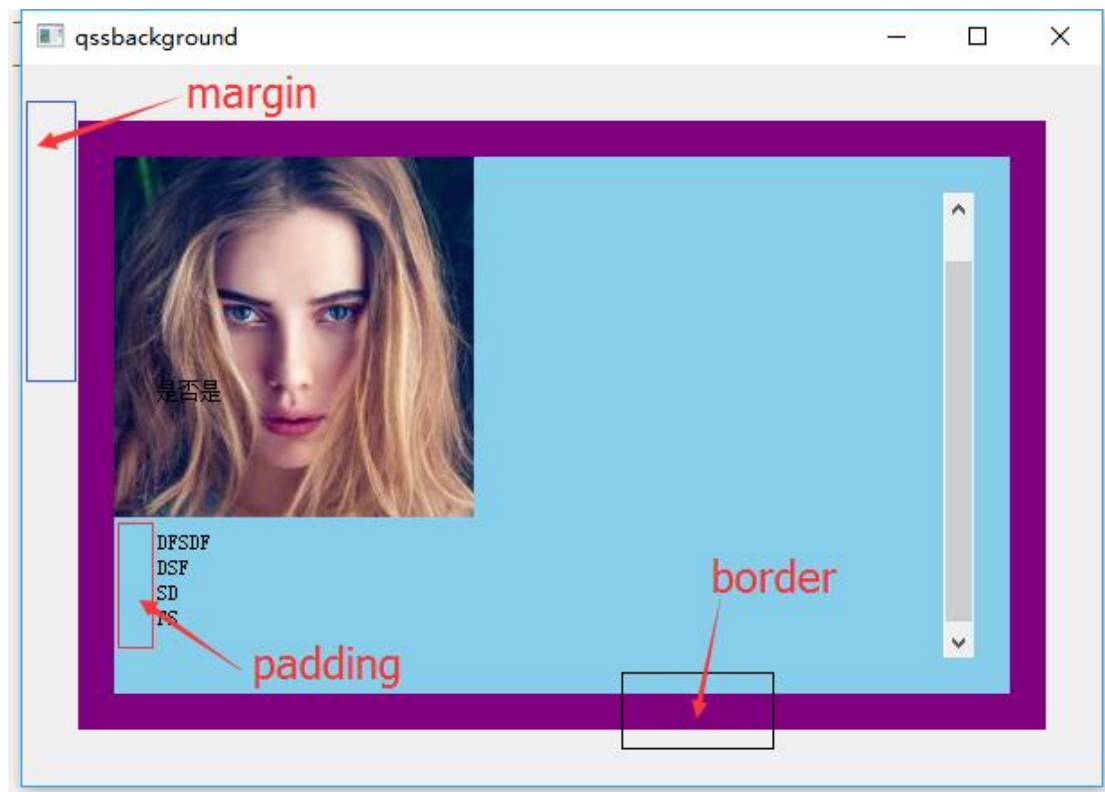
content: 内容矩形

作用: 设置背景颜色覆盖的区域, 默认情况下背景只覆盖内边距矩形, 如果没有指定, 默认为 border

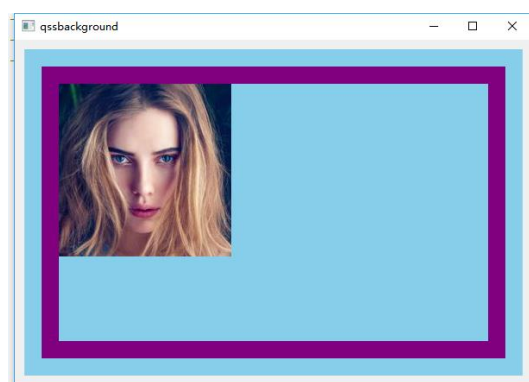
例子: 为了区别各矩形, 我们先给 QTextEdit 设置边框, 内边距和外边距, 为了区别明显, 我们将边框宽度设置大一点

代码和图片如下:

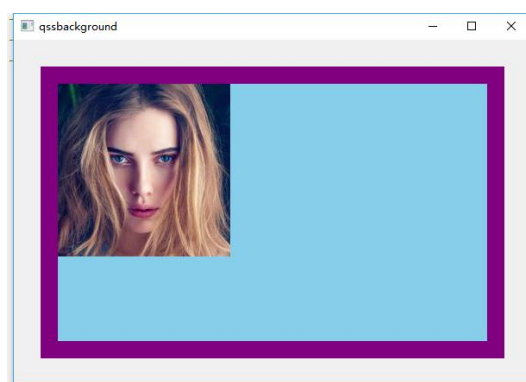
```
QTextEdit{
    background-color: skyblue;
    border: 20px solid purple;
    padding: 20px;
    margin: 20px;
    background-image: url(:/resource/girl.jpg);
    background-repeat: no-repeat;/*repeat-x; repeat-y; repeat;*/
    background-position: left top;
    background-attachment: fixed;
}
```



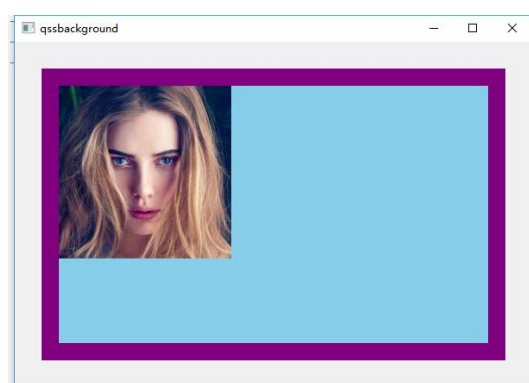
分别将 background-clip 属性的值设为 margin, border, padding, content, 效果图如下



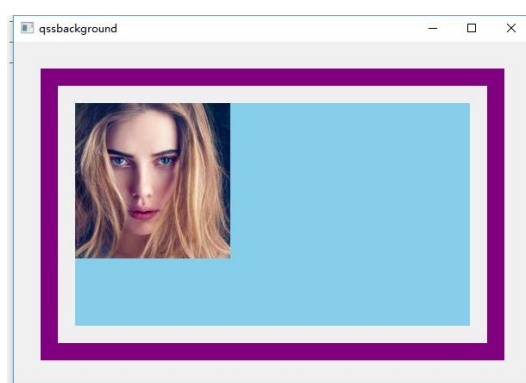
margin



border



padding



content

可见, `background-clip` 属性只对背景的渲染区域有关系, 背景图片始终是靠 `padding` 边上

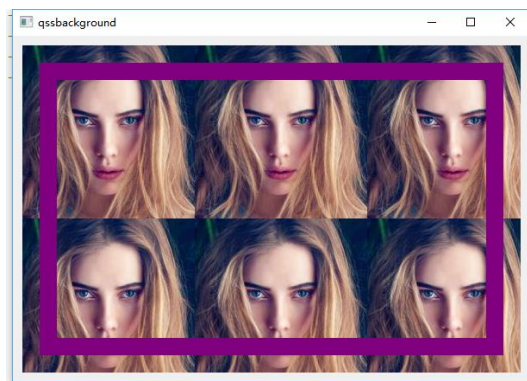
background-origin

取值: 与 `background-clip` 一样

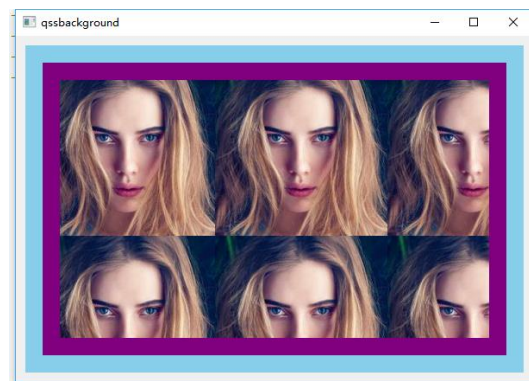
作用: 与 `background-position` 和 `background-image` 一起使用, 指明背景图片的覆盖范围矩形, 如果没有指定, 默认为 `padding`

下面是分别设置为 `margin`, `border`, `padding` 和 `content` 的代码和效果图

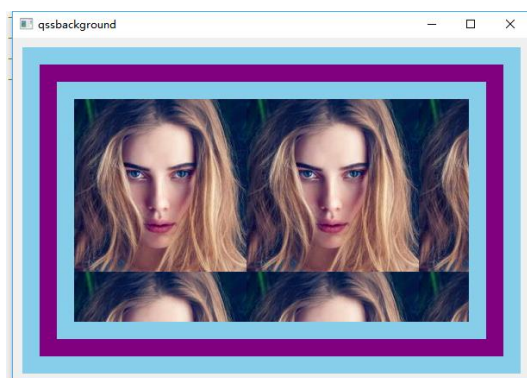
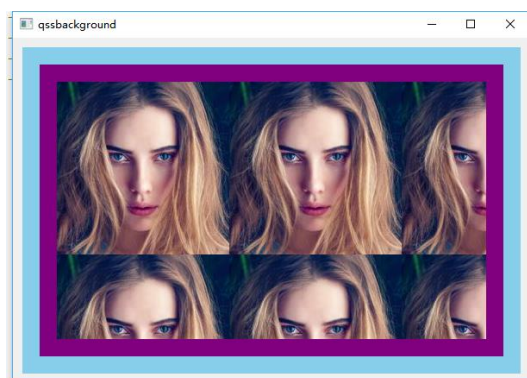
```
QTextEdit{  
    background-color: skyblue;  
    border: 20px solid purple;  
    padding: 20px;  
    margin: 20px;  
    background-image: url(/resource/girl.jpg);  
    /*background-repeat: no-repeat;*/repeat-x; repeat-y; repeat;*/  
    background-position: left top;  
    background-attachment: fixed;  
    background-clip: margin; /*border; padding; content;*/  
    background-origin: margin; /*border; padding; content;*/  
}
```



margin



border



padding

content

注意观察 border 与 padding 的图片，它们是不同的，差别就是 border 的 20 个像素所造成的不一致

背景属性的连写格式

格式:

background: color image repeat position;

在这种连写格式中，只能包含着四个属性，其他几个仍然需要单独写，而且这四个属性可以省略任何一个，最多可以省略三个，也就是最少需要保留一个，即属性值不能为空

另外让人比较迷惑的是，在这种连写方式中，repeat 确实是平铺了图片，而单独写时，它又是不平铺的，具体原因还未找到。

举个例子:

```
QTextEdit{
    border: 20px solid purple;
    padding: 20px;
    margin: 20px;
    background: skyblue url(/resource/girl.jpg) repeat left top;
}
```

效果图



前景属性 color

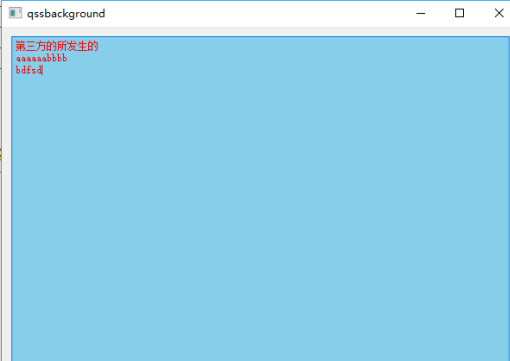
与背景相对应，背景设置的是控件的最底层的颜色，作为背景，但 `color` 设置的前景色，也就是控件文字的颜色，`color` 属性是被所有 `widget` 都支持的。

格式:

color: Brush 类型的值;

举个例子:

```
QTextEdit{
    color: red;
    background-color: skyblue;
}
```



边框属性 border

边框属性有四种书写方式, 同样, 先逐一进行分析, 最后给出书写格式

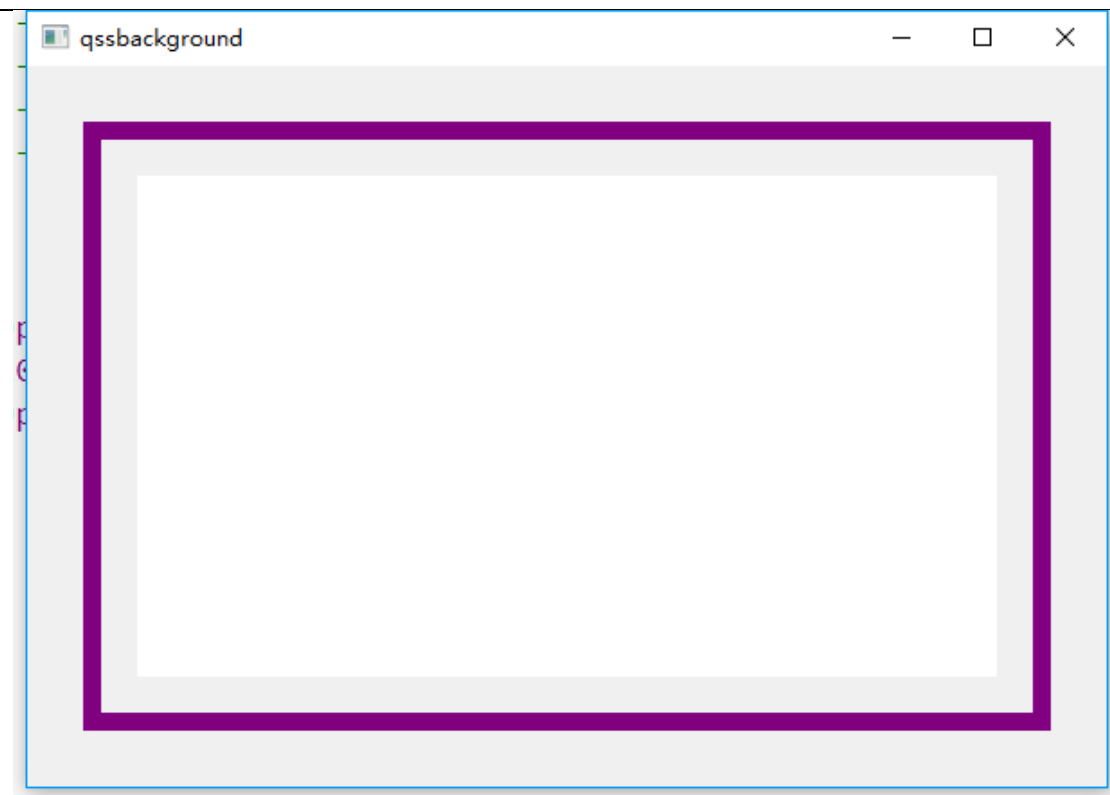
属性

border-width

取值: ?px 像素宽度, 数值后面一定要加上像素单位 **px**, 也有其他单位, 但不推荐使用
作用: 用于边框宽度

例子:

```
QTextEdit{  
    border-width: 10px;  
    border-style: solid;  
    border-color: purple;  
    padding: 20px;  
    margin: 20px;  
}
```



border-style

设置边框的渲染样式.

取值以及效果如下:

dashed



dot-dash



dot-dot-dash



dotted



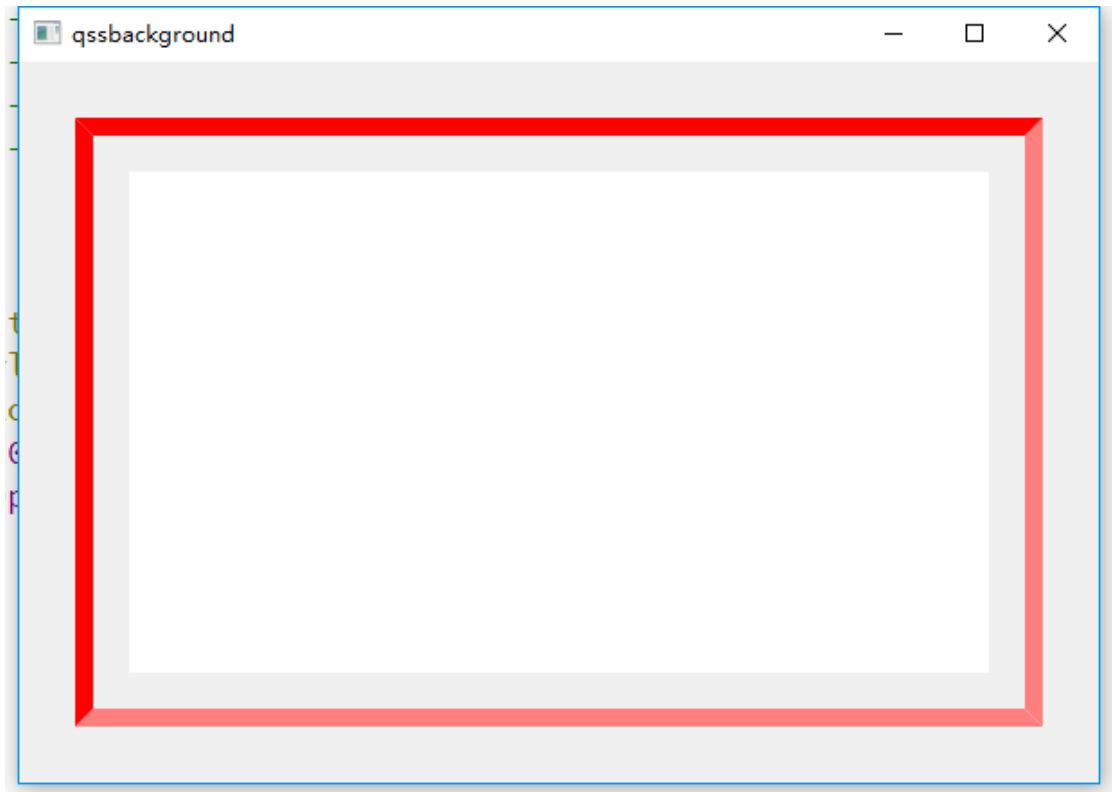
double



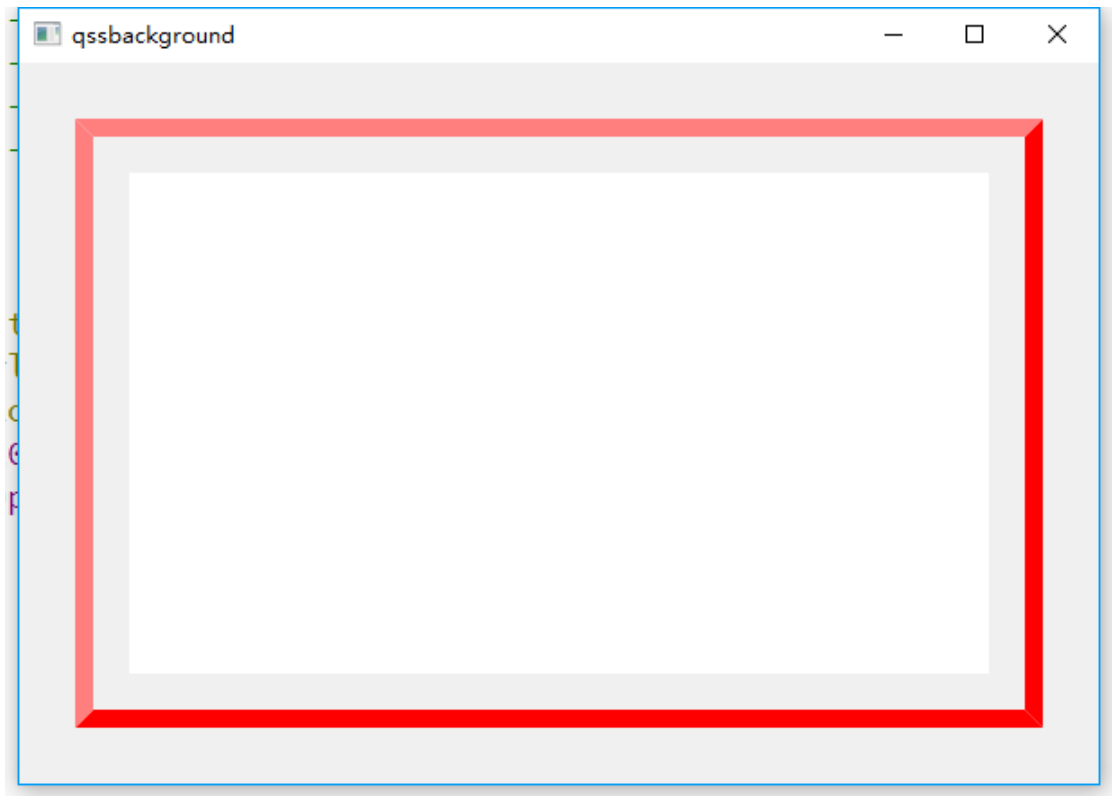
groove



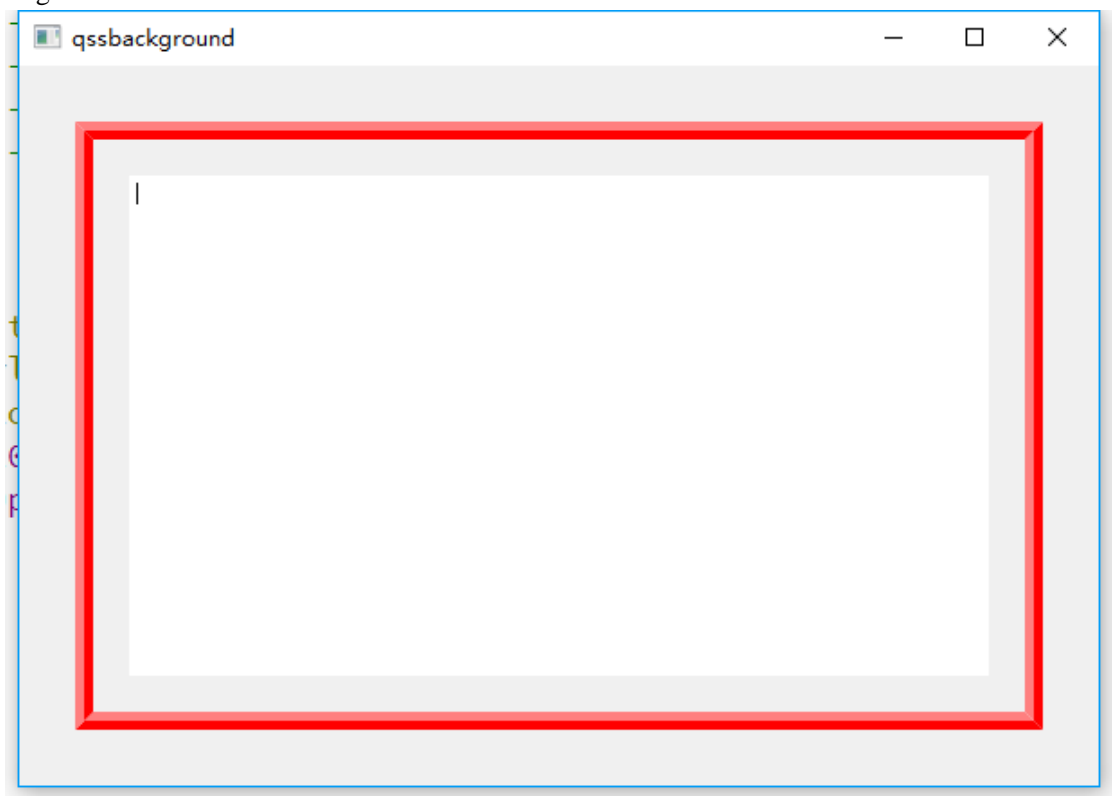
inset



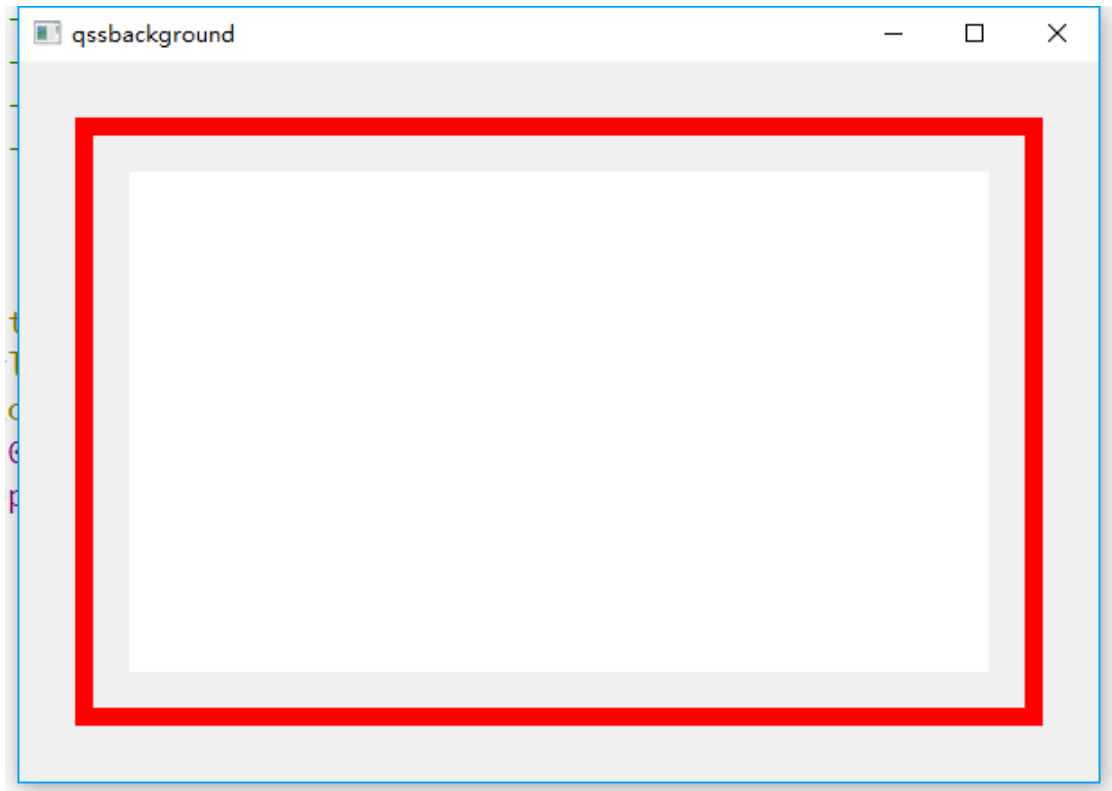
outset



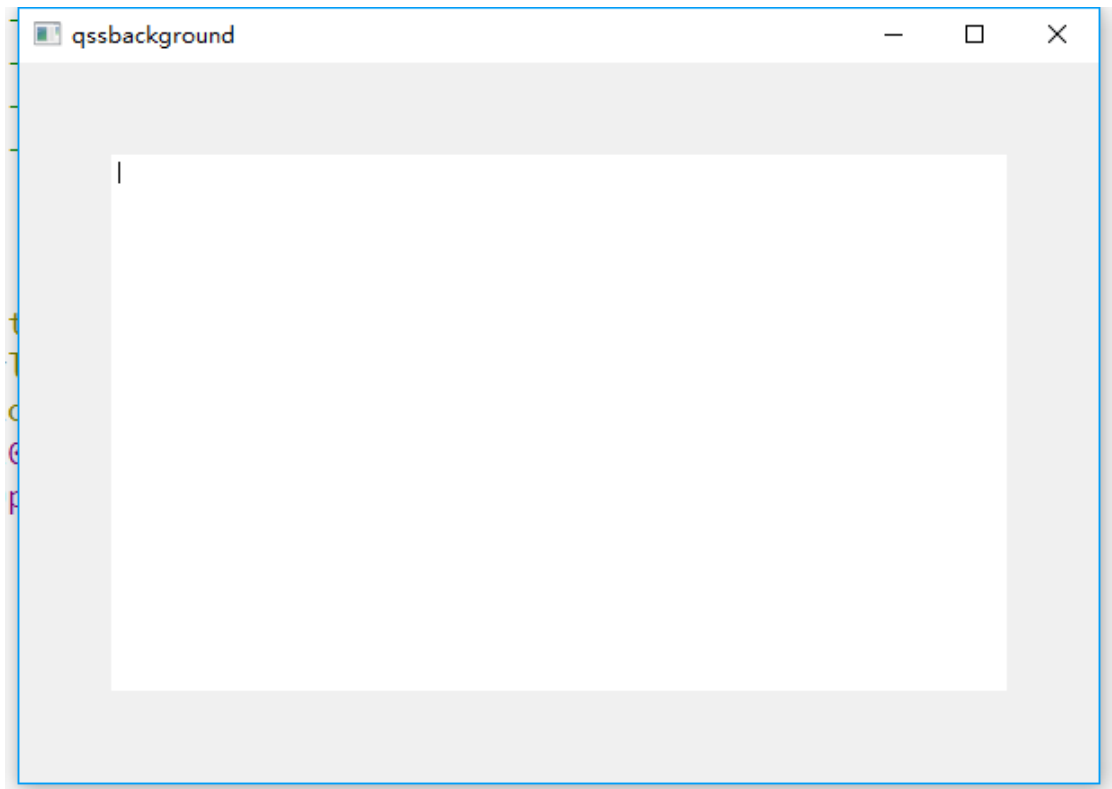
ridge



solid



none



border-color

取值: Brush 类型

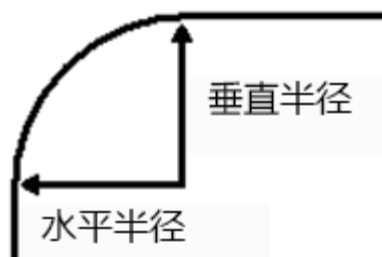
作用: 设置边框的颜色

border-radius

取值: 水平半径 垂直半径;

均是以像素为单位, 值必须带 **px**, 第二个值是可选的, 如果只有一个值, 表示同时水平半径和垂直半径, 如果有两个值, 则第一个代表水平半径, 第二个代表垂直半径.

示例图



作用: 设置边框四个角的弧度

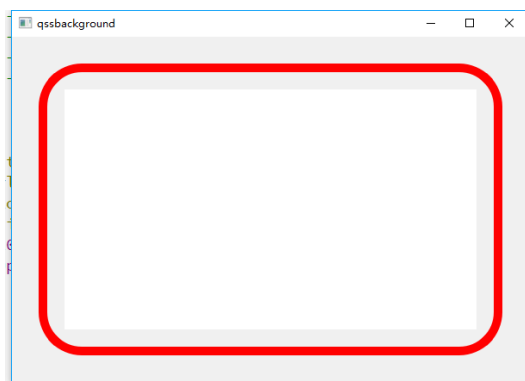
举个例子:

代码 1:

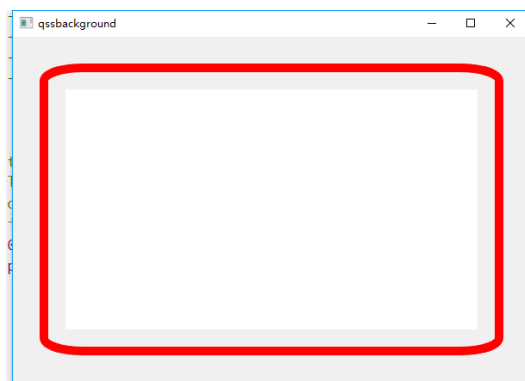
```
QTextEdit{  
    border-width: 10px;  
    border-style: solid;  
    border-color: red;  
    border-radius: 50px;  
    padding: 20px;  
    margin: 20px;  
}
```

代码 2:

```
QTextEdit{  
    border-width: 10px;  
    border-style: solid;  
    border-color: red;  
    border-radius: 50px 20px;  
    padding: 20px;  
    margin: 20px;  
}
```



代码 1 效果图



代码 2 效果图

border-image

取值: 这是一个连写格式, 下面给出具体的书写格式, 由于 Qt 对这个属性支持不是很好, 因此不建议使用, 下面简要介绍一下

格式:

CSS 的连写格式如下, 每一项分别代表分开写时的一个属性.

border-image: border-image-source border-image-slice (fill)/ border-image-width / border-image-outset border-image-repeat

其中, fill, border-image-width 和 border-image-outset 在 Qt 中不被支持, 而且 Qt 只支持连写格式, 因此在 Qt 中, 我们实际的代码格式是

border-image: border-image-source border-image-slice border-image-repeat

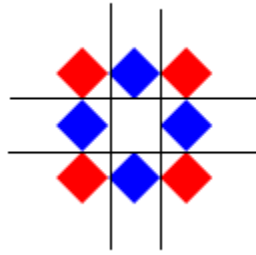
下面简略说一下每项的含义:

border-image-source: 图片路径, 还是只支持本地路径和 Qt 资源文件路径

border-image-slice: 图片切片, 单位只能是像素值, 因此数值不必带单位 px, 它最多可以指定 4 个值, 按照顺序分别代表上右下左, 最少指定 1 个值, 左省略时和右相同, 下省略时和上相同, 右省略时和上相同; 它们的含义是, 距图片顶部, 右侧, 下部, 左侧分别按照指定的像素值进行切片, 将图片分成 四个角(左上, 右上, 右下, 左下)+ 四个边(上右下左)+ 中间部分 = 共 9 个部分, 在 CSS 中, 如果指定了 fill, 则中间部分会覆盖元素(控件)的背景, 否则中间部分默认被省略

border-image-repeat: 最多两个值最少一个值, 第一个值表示水平方向, 第二个值表示垂直方向. 作用是指定边框图片的四条边和四个角的平铺方式, 不包括中间部分, 有三种取值, 分别为 **stretch**(默认), **round**(均分平铺), **repeat**(平铺). **stretch** 表示拉伸四条边相应的切片图片, 来填补边框的间隙.**round** 是把四个角和四条边分成均等区域然后用背景图片切好能铺满整个边框空隙, 不能多也不能少, 正好合适.**repeat** 是做直接复制填满空隙.

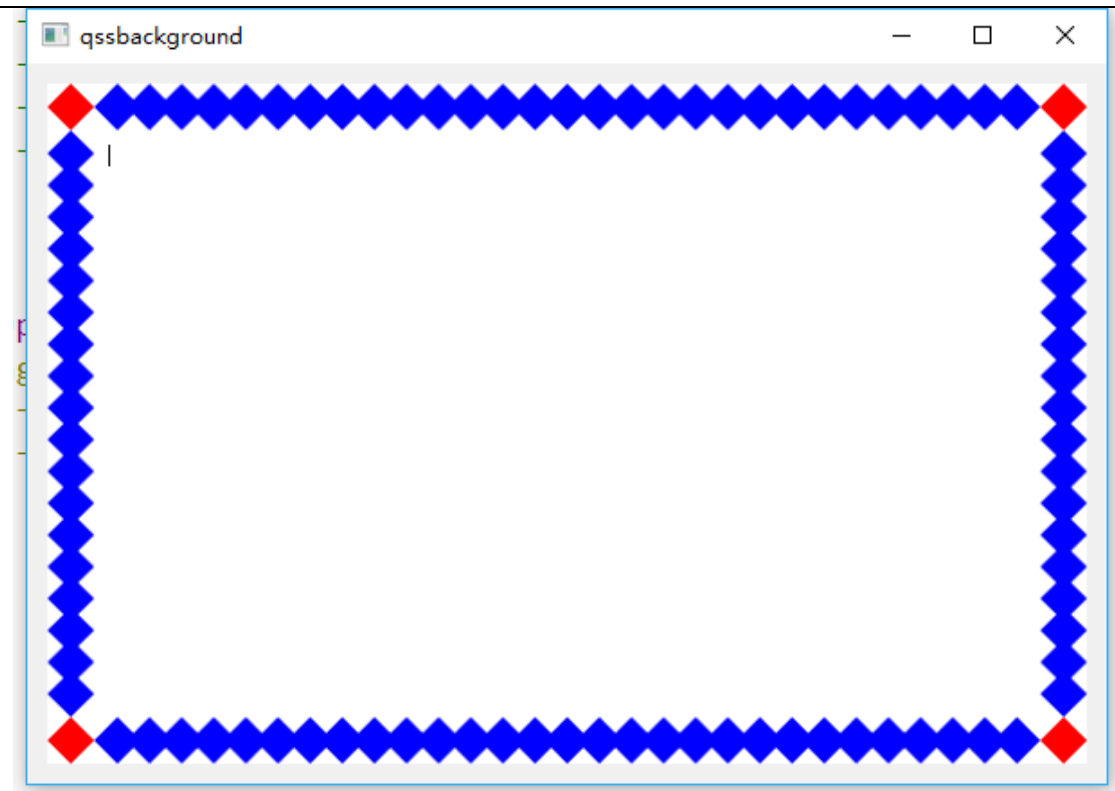
下面是一个切片的具体示例:



利用上面这张图片,来看一下 `border-image` 的一些效果图

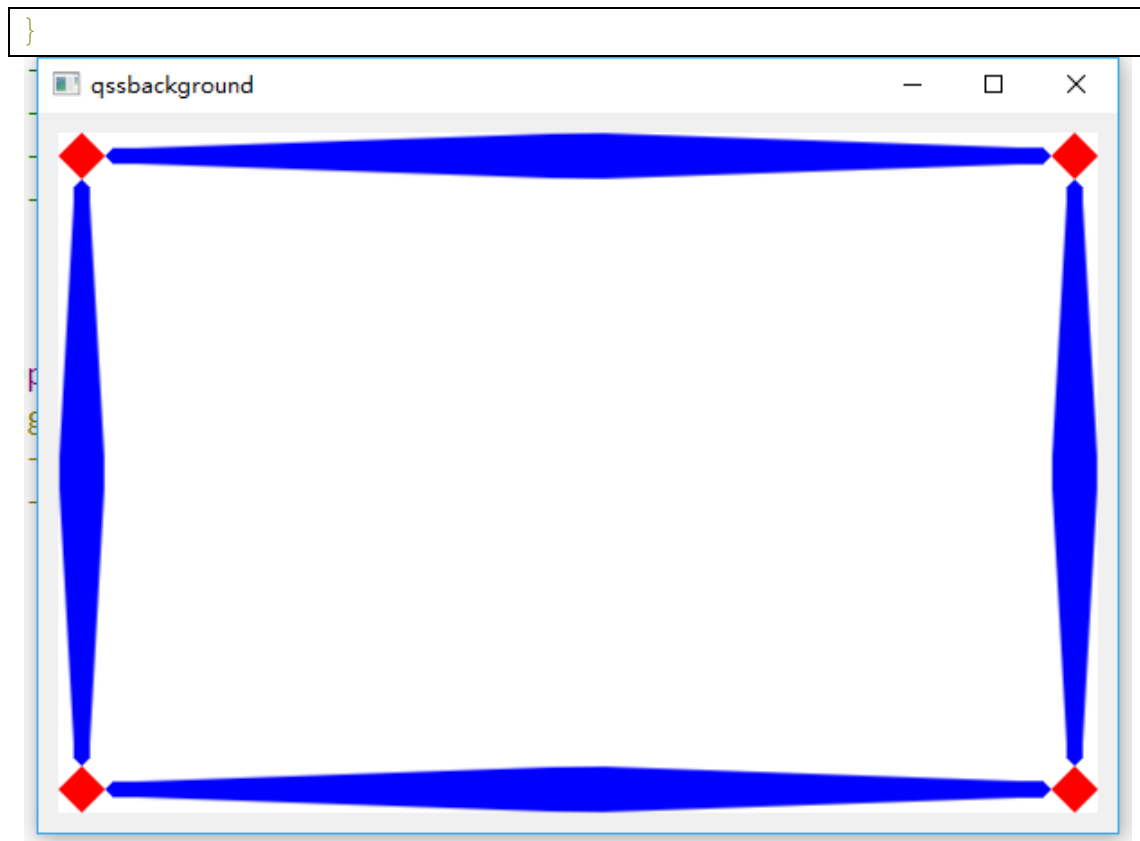
效果 1:

```
QTextEdit{  
    border: 30px solid red;  
    border-image: url(/resource/border.png) 30 round;  
    background-color: skyblue;  
    background-image: url(/resource/girl.jpg);  
}
```



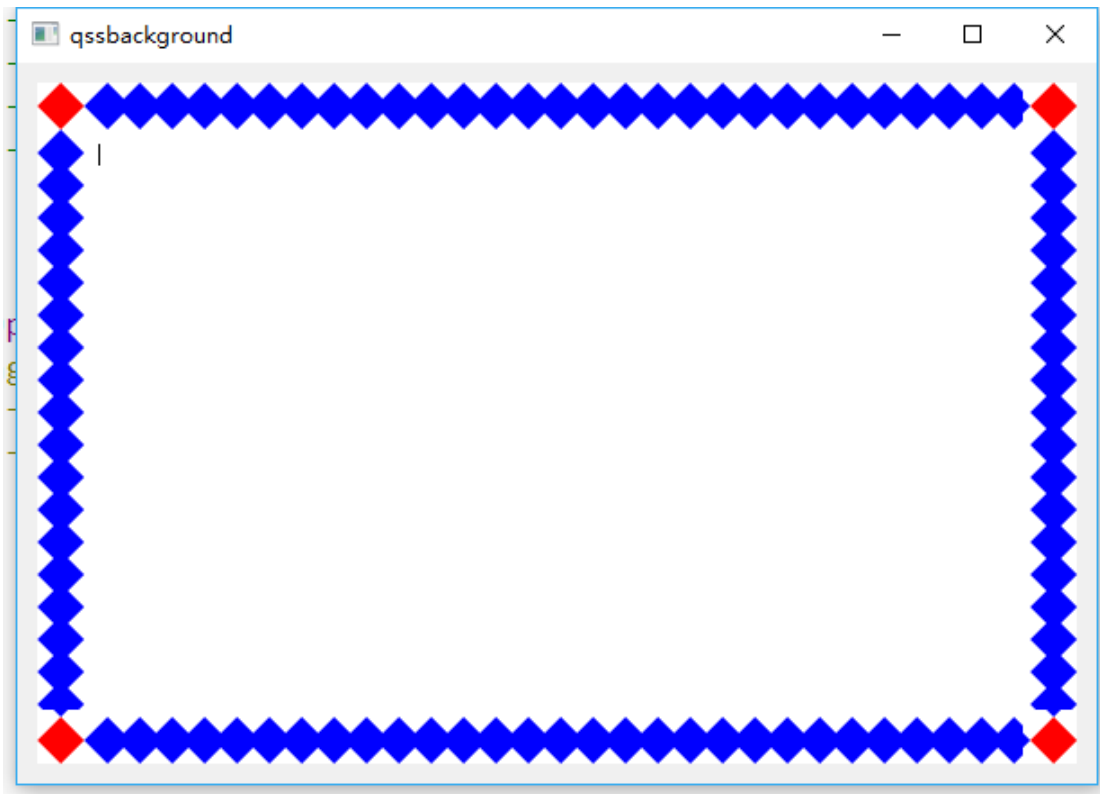
效果 2:

```
QTextEdit{  
    border: 30px solid red;  
    border-image: url(/resource/border.png) 30 stretch;  
    background-color: skyblue;  
    background-image: url(/resource/girl.jpg);  
}
```



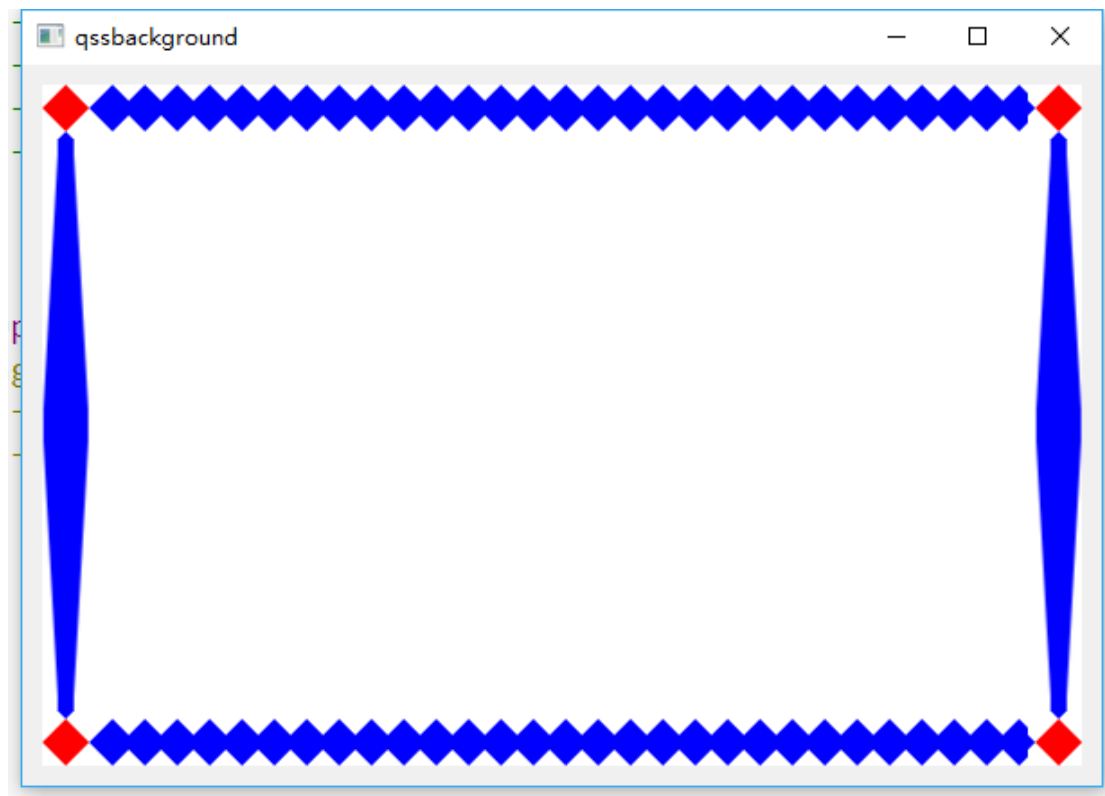
效果 3:

```
QTextEdit{  
    border: 30px solid red;  
    border-image: url(:/resource/border.png) 30 repeat;  
    background-color: skyblue;  
    background-image: url(:/resource/girl.jpg);  
}
```



效果 4:

```
QTextEdit{  
    border: 30px solid red;  
    border-image: url(:/resource/border.png) 30 repeat stretch;  
    background-color: skyblue;  
    background-image: url(:/resource/girl.jpg);  
}
```



格式

`border` 属性的 `style`, `color`, `width` 可以连写也可以单独写, 并且可以分别设置四条边的边框, 下面进行详细介绍

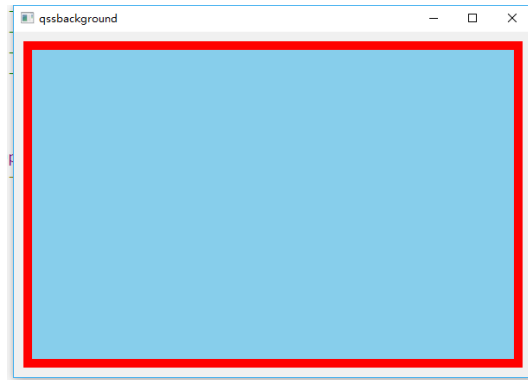
连写格式 1

```
border: width style color;
```

这种格式将四条边框的宽度, 风格, 颜色全部设置为一样.

例子如下

```
QTextEdit{  
    border: 10px solid red;  
    background-color: skyblue;  
}
```

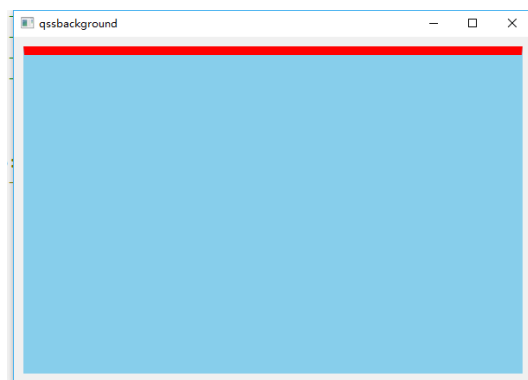
连写格式 2

这种格式设置指定方向的边框的样式, 可以只设置一条边, 格式如下

```
border-top: width style color;  
border-right: width style color;  
border-bottom: width style color;  
border-left: width style color;
```

例子如下:

```
QTextEdit{  
    border-top: 10px solid red;  
    background-color: skyblue;  
}
```



连写格式 3

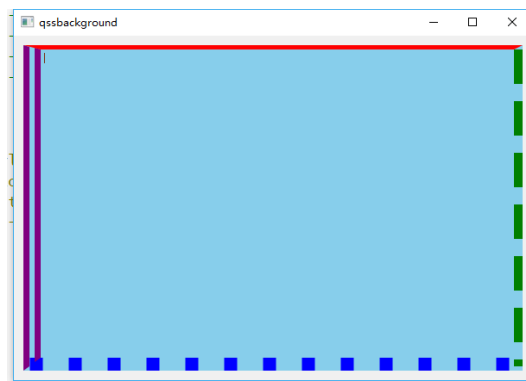
这种连写格式是指定一种属性, 按照上右下左四个方向进行设置边框, 格式如下

```
border-style: 上 右 下 左;  
border-width: 上 右 下 左;  
border-color: 上 右 下 左;
```

其中, 后三个可以省略, 左省略则与右相同, 下省略则与上相同, 右省略与上相同
这里给两个示例:

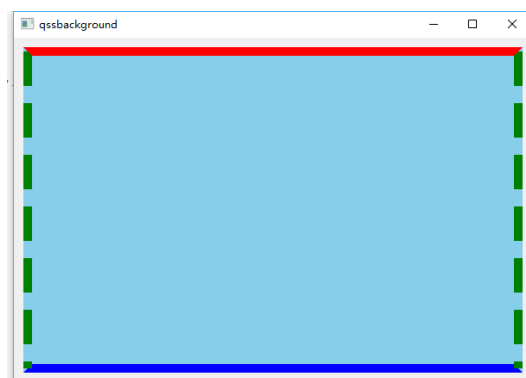
示例 1:

```
QTextEdit{  
    border-style: solid dashed dotted double;  
    border-color: red green blue purple;  
    border-width: 5px 10px 15px 20px;  
    background-color: skyblue;  
}
```



示例 2:

```
QTextEdit{  
    border-style: solid dashed ;  
    border-color: red green blue;  
    border-width: 10px;  
    background-color: skyblue;  
}
```



单写格式

单写格式指的是, 对每条边框的每个属性分别进行设置, 格式如下

border-top-width	border-top-style	border-top-color
border-right-width	border- right -style	border- right -color
border-bottom-width	border- bottom -style	border- bottom -color
border-left-width	border- left -style	border- left -color

字体属性 font

font-style

作用: 设置字体风格

取值:

normal: 正常

italic: 斜体

oblique: 倾斜的字体

关于 italic 和 oblique 的区别:

他们都是向右倾斜的文字, 大多数情况下看不出任何区别, 但是原理却不一样.

要搞清楚这个问题, 首先要明白字体是怎么回事。一种字体有粗体、斜体、下划线、删除线等诸多属性。

但是并不是所有字体都做了这些, 一些不常用的字体, 或许就只是个正常体, 如果你用 Italic, 就没有效果了, 这时候就要用 Oblique.

可以理解成 Italic 是使用文字的斜体, Oblique 是让没有斜体属性的文字倾斜!

font-weight

作用: 设置文字的粗细

取值:

它有两种取值, 一种是单次表示:

normal: 正常粗细

bold: 加粗

另一种是整数表示, 整数越大, 字体越粗

100, 200, 300, ..., 900

font-size

作用: 设置字体大小

取值:

字体大小的取值是一个数值加上单位, 它的单位有 `px`, `pt`, 但一般都使用 `px`, 表示多少个像素, 如 `20px`, 表示字体的宽和搞

说明:

注意, 实际上它设置的是字体中字符框的高度; 实际的字符字形可能比这些框高或矮 (通常会矮)。

各关键字对应的字体必须比一个最小关键字相应字体要高, 并且要小于下一个最大关键字对应的字体。

font-family

作用: 设置文字字体

取值: 各种字体名称

如果字体是中文, 尽量用双引号括起来

QSS 中 `font-family` 只能指定一种字体

连写格式

字体属性可以单写, 也可以连写, 连写格式如下:

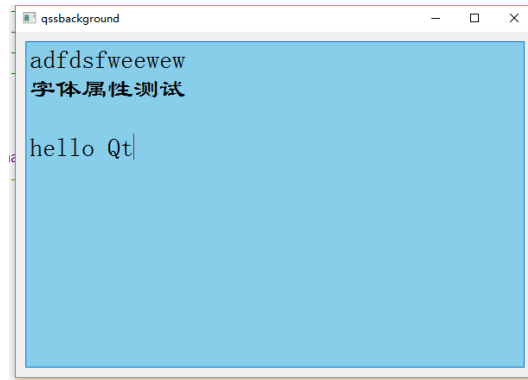
font: style weight size family

这种书写格式中的注意点:

1. `style` 和 `weight` 的位置可以交换, 并且可以省略;
2. `size` 不能被省略
3. `size` 和 `family` 必须写在其他两个属性的后面, 并且位置不能交换
4. `family` 可以省略, 省略后使用默认字体

举个例子:

```
QTextEdit{  
    font: normal normal 30px "隶书";  
    background-color: skyblue;  
}
```



文本属性

text-align

作用:设置文本的对齐方式

取值:

top
bottom
left
right
center

注意点: 支持这个属性的控件目前只有 QPushButton 和 QProgressBar.

格式:

text-align: 水平对齐方式(left, right, center) 垂直对齐方式(top bottom center);

举个例子

```
QPushButton{  
    background-color: pink;  
    min-height: 80px;  
    text-align: left top;  
}
```

这是一个按钮

text-decoration

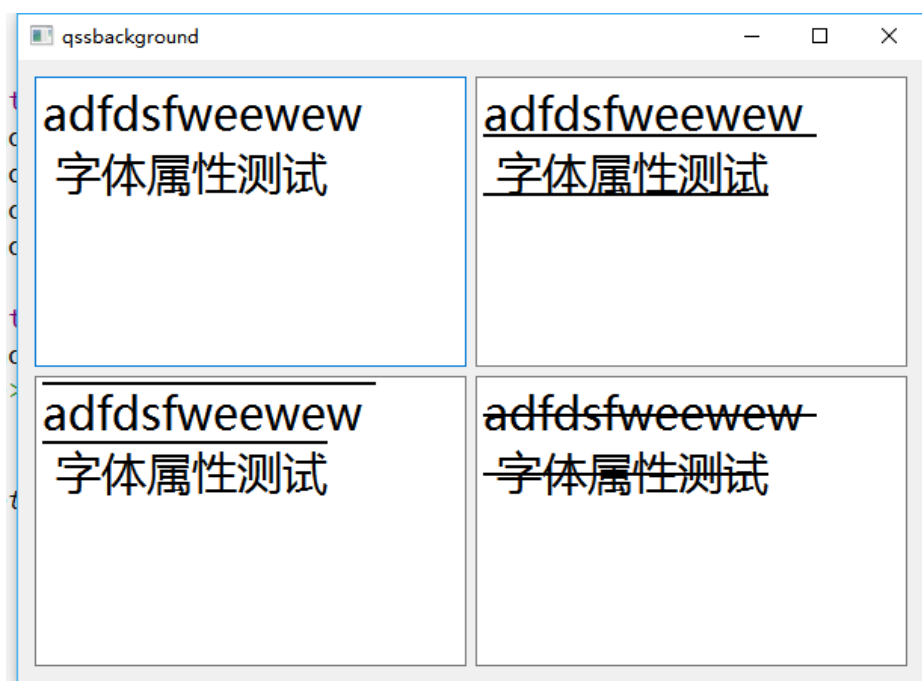
作用: 给文本添加装饰

取值:

- none: 没有装饰
- underline: 下划线
- overline: 上划线
- line-through: 删除线

举个例子:

```
QTextEdit{  
    font: normal normal 30px "微软雅黑";  
}  
  
#text_edit_1{  
    text-decoration: none;  
}  
#text_edit_2{  
    text-decoration: underline;  
}  
#text_edit_3{  
    text-decoration: overline;  
}  
#text_edit_4{  
    text-decoration: line-through;  
}
```



padding 和 margin

盒模型中的 padding 和 margin 都可以连写, 也可以单独写, 它们都能完成四个方向上的边距设置, 默认情况下都是 0.

与边框和其他连写格式一样, 如果它们连写时, 最多可以指定 4 个值, 最少指定 1 个值, 指定 4 个时, 分别表示上右下左方向的边距, 省略时, 也有相同的效果, 即左省略时默认和右一样, 下省略时默认和上一样, 右省略时和上一样.

设置边距时, 数值后面必须要带像素单位, 即 px;

padding

padding 既可以连写也可以分四个方向单独设置, 格式如下

```
padding: 上 右 下 左;  
或  
padding-top: ?px;  
padding-right: ?px;  
padding-bottom: ?px;  
padding-left: ?px;
```

margin

格式与 padding 类似, 具体如下

```
margin: 上 右 下 左;  
或  
margin-top: ?px;  
margin-right: ?px;  
margin-bottom: ?px;  
margin-left: ?px;
```

由于 Qt 将整个盒子看做是一个控件, 因此在布局时, 不会考虑每个盒子的垂直方向的外边距是否有合并现象等, 所以一个控件的外边距只会对自己产生影响, 不会对其他的控件产生影响.

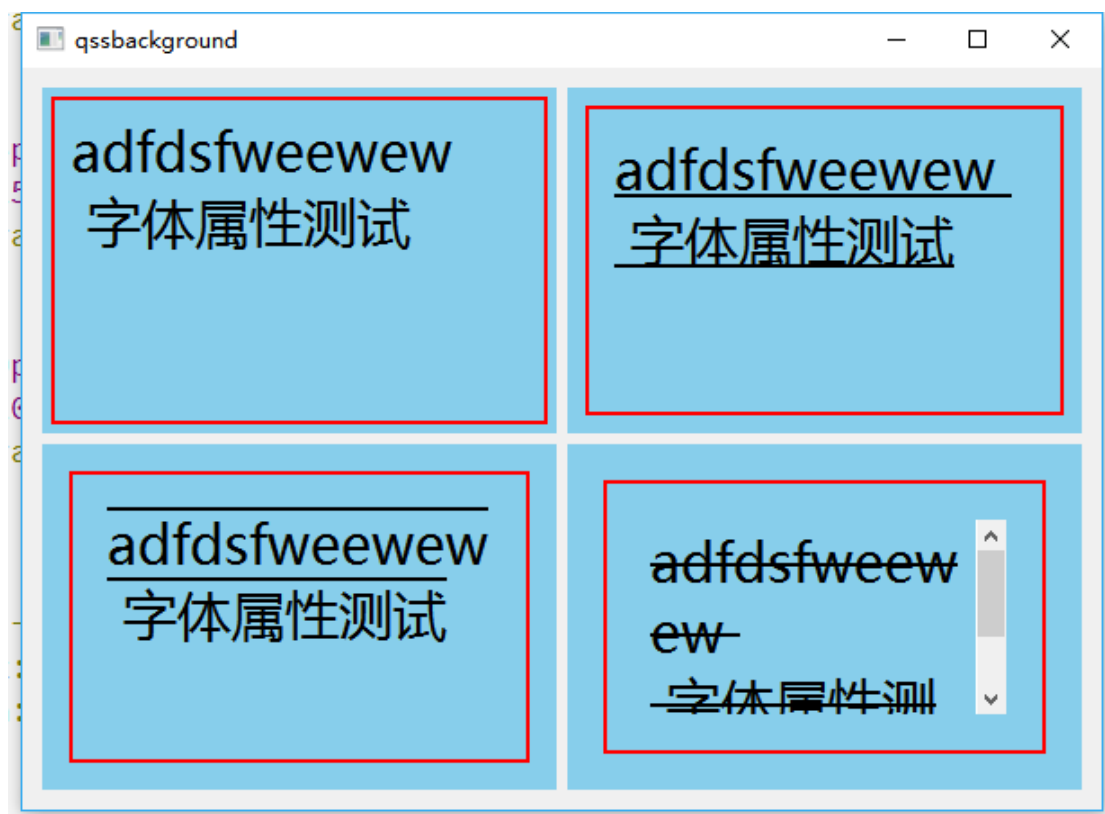
下面看一个例子:

```
QTextEdit{  
    border: 2px solid red;  
    background: skyblue;  
    background-clip: margin;  
    font: normal normal 30px "微软雅黑";  
}
```

```

#text_edit_1{
    margin: 5px;
    padding: 5px;
    text-decoration: none;
}
#text_edit_2{
    margin: 10px;
    padding: 10px;
    text-decoration: underline;
}
#text_edit_3{
    margin: 15px;
    padding: 15px;
    text-decoration: overline;
}
#text_edit_4{
    margin: 20px;
    padding: 20px;
    text-decoration: line-through;
}

```



width 与 height

width, height

这两个属性设置的是盒子内容的宽高.

这两个属性只对子控件选择器选中的对象有效

这两个属性的取值均是像素值, 即数字加像素单位 `px`;

max-width min width 与 max-height min-height

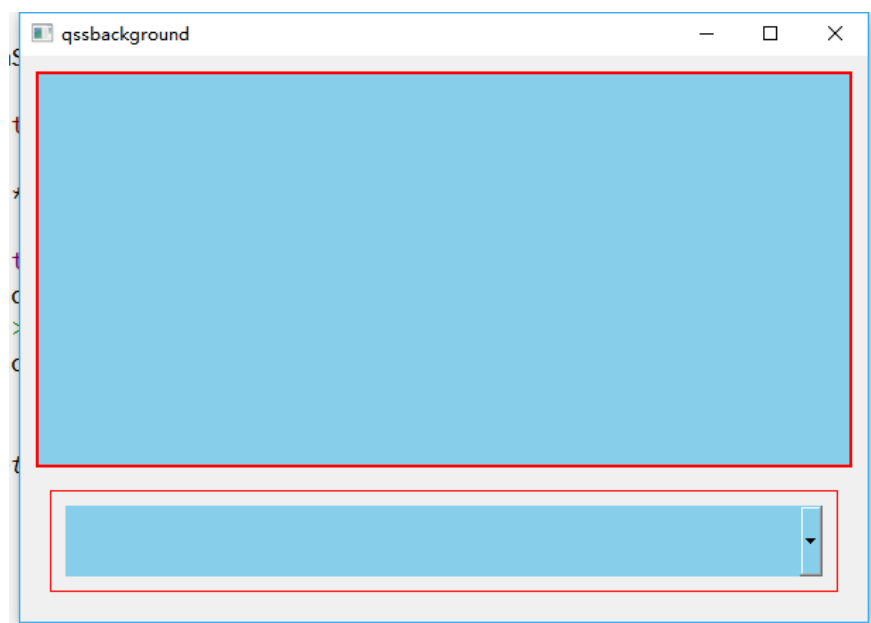
这四个属性对所有的 `widget` 都有效, 用来设置盒子内容的最小或最大尺寸

当最小宽度与最大宽度相等时, 意味着给这个盒子的内容设置了一个固定宽度.

当最小高度与最大高度相等时, 意味着给这个盒子的内容设置了一个固定高度.

举个例子:

```
QComboBox {  
    background-color: skyblue;  
    border: 1px solid red;  
    padding: 10px;  
    margin: 10px;  
    background-clip: content;  
    min-height: 50px;  
    max-height: 50px;  
}
```



outline

outline（轮廓）是控件有焦点时，绘制在边框边缘的外围,可起到突出作用,轮廓线不占据控件，也不一定是矩形.

它有如下属性,

outline

outline-color

outline-offset

outline-style

outline-radius

outline-bottom-left-radius

outline-bottom-right-radius

outline-top-left-radius

outline-top-right-radius

这里对这些属性不做详细介绍，只需要知道，当我们想在一个控件有焦点时，不绘制轮廓，只需要这样做，

outline: none;

属性结语

这一小节主要列举了一些常用的属性，并列举了它们的用法和取值等，还有其他一些属性并未介绍，但很可能在开发中需要用到。这里啰嗦一下，这种总结性质的东西，不可能尽善尽美罗列所有，如果有需要，还是要去自己查看 **Qt** 官方给的资料，查阅文档搜寻有用的知识也是一种很重要的技能。

Brush 类型介绍

brush 一般用来设置颜色，其取值有 3 种，分别是 **Color**, **Gradient** 和 **PaletteRole**，下面简单介绍一下

Color

color 本身又支持很多格式，列举所有格式，如下：

rgb(r, g, b) 每个数字表示每个通道的值，依次分别是红绿蓝

rgba(r, g, b, a) 与 **rgb** 相同, **a** 代表 α 通道，是一个范围 **0~1** 的浮点数，表示透明度, **1** 代表不透明, **0** 表示全透明

hsv(h, s, v)

hsva(h, s, v, a)

#rrggb: 16 进制表示的 rgb 值, 每个值占两位, 但如果每个通道的两位都一样, 可以简写为 #rgb, 如#66FFAA 可以简写为 #6FA, 并且大小写不敏感.

name: 常见的表示颜色的单次, 如 red, green, blue, yellow, purple 等

常用的是 rgb, rgba, #rrggb, name

Gradient

可实现渐变效果, 三种取值:

qlineargradient 线性渐变

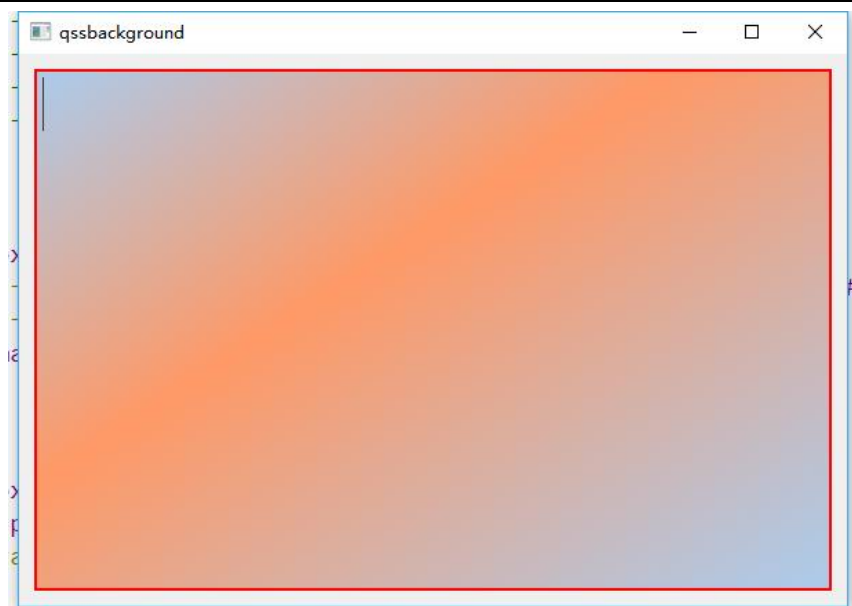
qradialgradient 径向渐变

qconicalgradient 锥形渐变

它们分别对应了 Qt 的 3 个类, QLinearGradient, QRadialGradient, QConicalGradient, 参数可以参考它们的函数, 这里不多赘述

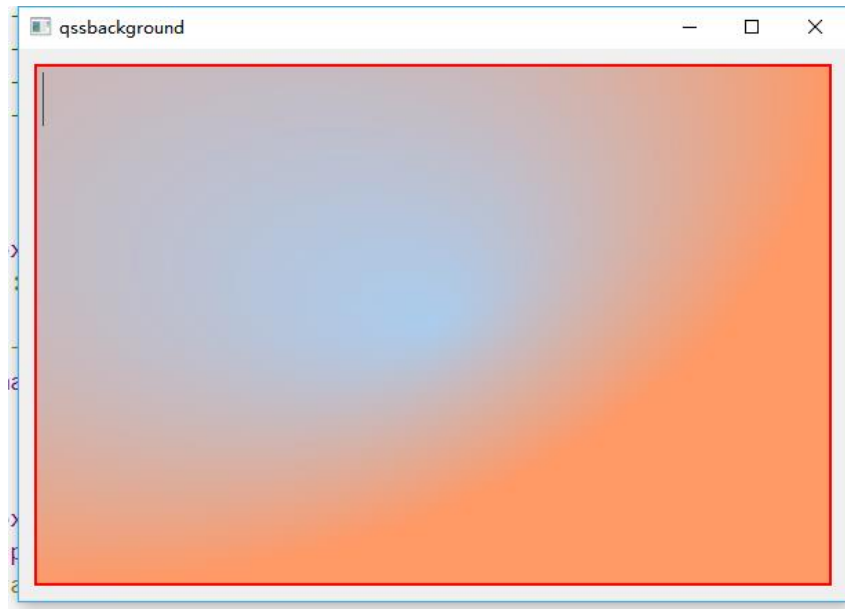
直接看示例:

```
QTextEdit{
    border: 2px solid red;
    background-color: qlineargradient(x1:0, y1:0, x2:1, y2:1, stop:0
                                     #ace, stop: 0.4 #f96, stop:1 #ace);
    background-clip: margin;
    font: normal normal 30px "微软雅黑";
}
```

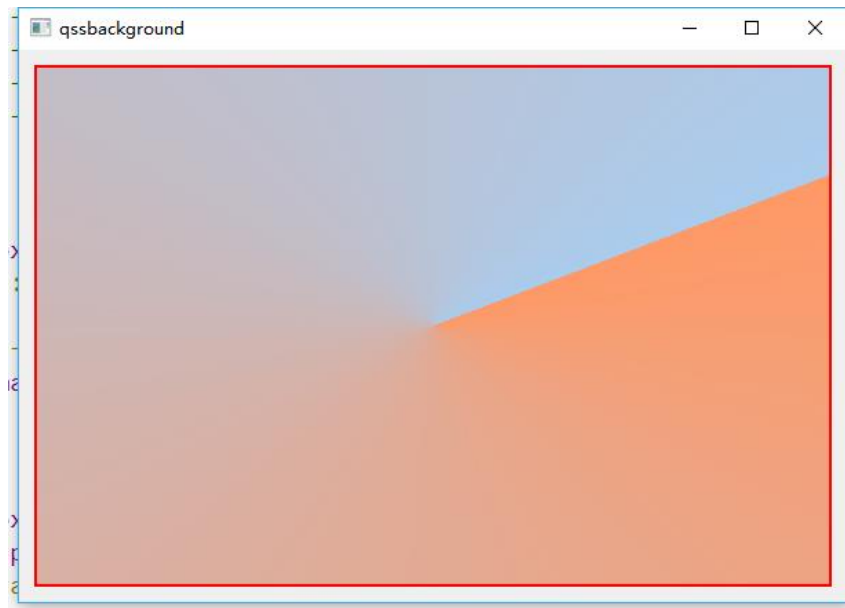


```
QTextEdit{
```

```
border: 2px solid red;
background: qradialgradient(cx:0, cy:0, radius: 1,
                           fx:0.5, fy:0.5, stop:0 #ace, stop:1 #f96);
background-clip: margin;
font: normal normal 30px "微软雅黑";
}
```



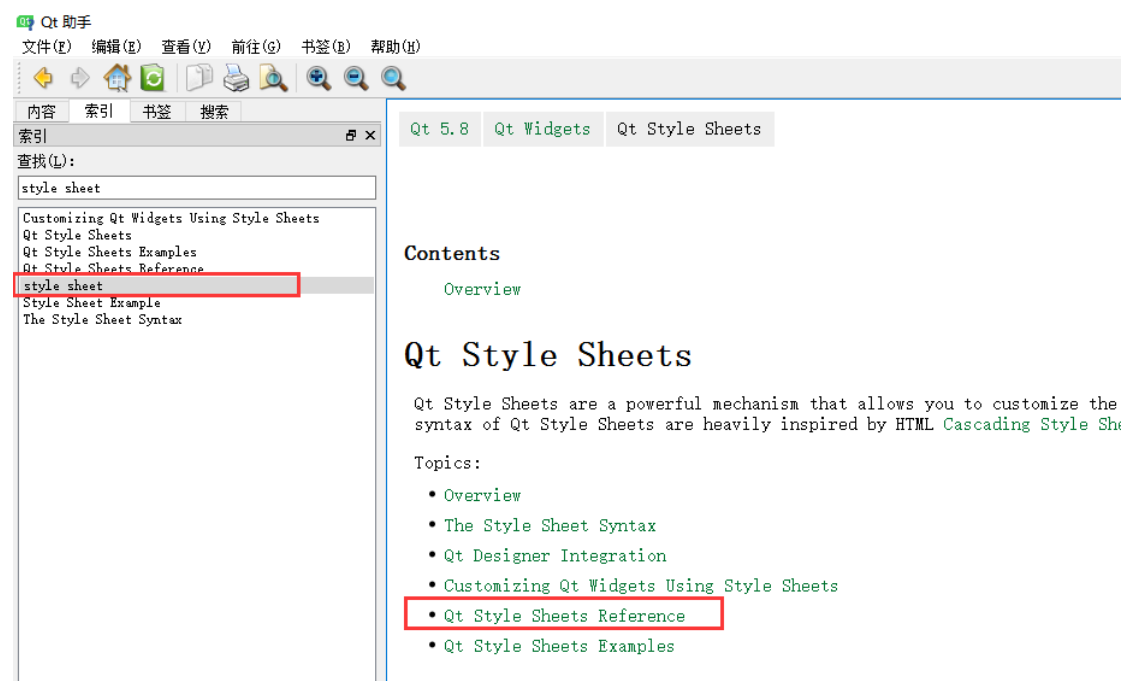
```
QTextEdit{
    border: 2px solid red;
    background: qconicalgradient(cx:0.5, cy:0.5, angle:30,
                                stop:0 #ace, stop:1 #f96);
    background-clip: margin;
    font: normal normal 30px "微软雅黑";
}
```



Qt 官方链接

链接地址: <http://doc.qt.io/qt-5/stylesheet.html>

assistant 搜索方法: 输入 style sheet 即可, 注意两个单词之间有空格, 然后进入 Qt Style Sheets Reference 专题, 如图:



与 CSS 的对比

CSS 与 Qss 的基本语法一模一样，这里主要对比选择器与属性

选择器对比

QSS 只实现了 CSS2 中的选择器，并不包含 CSS3 中新增的选择器，具体对比如下：

QSS 选择器	对应的 CSS 选择器	区别
通用选择器	通用选择器	没有区别
类型选择器	标签选择器	一个选择的是该类型所有的对象，一个选择的是所有该标签，某种程度上是没有区别的
类选择器	类选择器	qss 的类选择器中的类名指的是类型名称前加一个(.) 并且一个控件只可能有一个类名，而 CSS 中，类名指的是通过标签的 class 属性指定的类名，一个标签可能有多个类名
ID 选择器	ID 选择器	QSS 的选择器中的 id 指的是对象的 objectName，可以重复，而在 CSS 中，ID 是通过便签的 ID 属性设置的，并且同一个页面中是不能存在相同 ID 的元素的
后代选择器	后代选择器	QSS 中，后代关系指的是控件之间的父子关系，而 CSS 中后代关系指的是标签之间的包含关系，也可以理解为父子关系，因此在概念上是相同的
子元素选择器	子元素选择器	在 QSS 中,子元素选择器只能有一级，也就是对于一个控件，只能找到它的子控件，无法找到子控件的子控件，而在 CSS 中，子元素选择器是可以无限延伸下去的
属性选择器	属性选择器	QSS 的属性指的是用 Q_PROPERTY 定义的属性，CSS 的属性指的是标签的股友属性，而且一般只用于表单标签
并集选择器	并集选择器	没有任何区别
子控件选择器	伪元素选择器	qss 中子控件选择器的连接符是(::) 而 CSS 伪元素选择器的连接符是(:), CSS 中的伪元素并不是一个真正的元素，比如一段文字的第一行被当做一个元素，显然它并不是一个

		html 元素, 而 qss 中的子控件大多数情况下却是一个真正的控件, 也就是一个独立的对象.
伪类选择器	伪类选择器	指的都是某种状态下的对象/标签

属性对比

关于属性, QSS 其实只支持了一部分的 CSS 属性, 并且有些属性还做了简化(如 border-image)

在 CSS 中, 元素是被分为块级元素(独占一行, 可以设置宽高), 行内元素(不会独占一行, 不能设置宽高), 行内块级元素(即不独占一行, 也可以设置宽高), 并且它们都是可以通过 display 属性相互转换, 而在 QSS 中, 并没有这种分类, 这种表现出来的形式, 可以通过布局器实现.

而且在 qss 中, 有些属性是参照 CSS2 的, 有少部分的属性参照了 CSS3, 还有个别的属性是 QSS 独有的, 这里不做一一对比, 因为实在没有什么必要. 而且就目前来说, qss 对很多属性支持的还不够好, 但如果能熟练使用, 也足以满足我们的需求.

源码阅读--QSS 是如何实现的/会不会有新发现?

这里先看看 QApplication::setStyleSheet 的实现:

```
void QApplication::setStyleSheet(const QString& styleSheet)
{
    QApplicationPrivate::styleSheet = styleSheet;
    QStyleSheetStyle *proxy =
qobject_cast<QStyleSheetStyle*>(QApplicationPrivate::app_style);
    if (styleSheet.isEmpty()) { // application style sheet removed
        if (!proxy)
            return; // there was no stylesheet before
        setStyle(proxy->base);
    } else if (proxy) { // style sheet update, just repolish
        proxy->repolish(qApp);
    } else { // stylesheet set the first time
        QStyleSheetStyle *newProxy = new
QStyleSheetStyle(QApplicationPrivate::app_style);
        QApplicationPrivate::app_style->setParent(newProxy);
    }
}
```

```

        setStyle(newProxy);
    }
}

```

可以看出, 这里外观 `widgets` 的外观主要在于 `QStyleSheetStyle::repolish()` 函数,

```

void QStyleSheetStyle::repolish(QApplication *app)
{
    Q_UNUSED(app);
    const QList<const QObject*> allObjects =
styleSheetCaches->styleRulesCache.keys();
    styleSheetCaches->styleSheetCache.remove(qApp);
    styleSheetCaches->styleRulesCache.clear();
    styleSheetCaches->hasStyleRuleCache.clear();
    styleSheetCaches->renderRulesCache.clear();
    updateObjects(allObjects);
}

```

先清空了之前的样式, 然后再更新每个窗口, 主要看最后一个函数:

```

static void updateObjects(const QList<const QObject*>& objects)
{
    if (!styleSheetCaches->styleRulesCache.isEmpty()
|| !styleSheetCaches->hasStyleRuleCache.isEmpty()
|| !styleSheetCaches->renderRulesCache.isEmpty()) {
        for (int i = 0; i < objects.size(); ++i) {
            const QObject *object = objects.at(i);
            styleSheetCaches->styleRulesCache.remove(object);
            styleSheetCaches->hasStyleRuleCache.remove(object);
            styleSheetCaches->renderRulesCache.remove(object);
        }
    }

    QWidgetList widgets;
    foreach (const QObject *object, objects) {
        if (QWidget *w =
qobject_cast<QWidget*>(const_cast<QObject*>(object)))
            widgets << w;
    }

    QEvent event(QEvent::StyleChange);
    foreach (QWidget *widget, widgets) {
        widget->style()->polish(widget);
    }
}

```



```

        QApplication::sendEvent(widget, &event);
    }
}

```

这里 `stylesheetCaches` 的类型是 `QStyleSheetStyleCaches`, 看一下这个类的定义

```

class QStyleSheetStyleCaches : public QObject
{
    Q_OBJECT
public Q_SLOTS:
    void objectDestroyed(QObject *);
    void styleDestroyed(QObject *);
public:
    QHash<const QObject *, QVector<QCss::StyleRule> >
styleRulesCache;
    QHash<const QObject *, QHash<int, bool> > hasStyleRuleCache;
    typedef QHash<int, QHash<quint64, QRenderRule> > QRenderRules;
    QHash<const QObject *, QRenderRules> renderRulesCache;
    QHash<const void *, QCss::StyleSheet> styleSheetCache; // parsed
style sheets
    QSet<const QWidget *> autoFillDisabledWidgets;
    // widgets whose palettes and fonts we have tampered. stored
value pair is
    // QPair<old widget value, resolve mask of stylesheet value>
    QHash<const QWidget *, QPair<QPalette, uint> >
customPaletteWidgets;
    QHash<const QWidget *, QPair<QFont, uint> > customFontWidgets;
};

```

Ok, 看到这里, 去看一下 `QCss` 这个命名空间, 正好发现了它的解析器类,

```

class Q_GUI_EXPORT Parser
{
public:
    Parser();
    explicit Parser(const QString &css, bool file = false);

    void init(const QString &css, bool file = false);
    bool parse(StyleSheet *styleSheet, Qt::CaseSensitivity
nameCaseSensitivity = Qt::CaseSensitive);
    Symbol errorSymbol();

    bool parseImport(ImportRule *importRule);
    bool parseMedia(MediaRule *mediaRule);
    bool parseMedium(QStringList *media);
    bool parsePage(PageRule *pageRule);

```

```

    bool parsePseudoPage(QString *selector);
    bool parseNextOperator(Value *value);
    bool parseCombinator(BasicSelector::Relation *relation);
    bool parseProperty(Declaration *decl);
    bool parseRuleset(StyleRule *styleRule);
    bool parseSelector(Selector *sel);
    bool parseSimpleSelector(BasicSelector *basicSel);
    bool parseClass(QString *name);
    bool parseElementName(QString *name);

    .....
    .....
};

```

大概看了一下，这里面很复杂，就不一一展开了，继续回头看 updateObjects 函数，可以看出，到了最后，QApplication 给每个 widget 都发送了一个 **StyleChange** 事件，接下来去看一下 QWidget 的事件处理函数，这个函数非常大，大概有 500 行，截取一段来看

```

bool QWidget::event(QEvent *event)
{
    .....

    case QEvent::ToolBarChange:
    case QEvent::ActivationChange:
    case QEvent::EnabledChange:
    case QEvent::FontChange:
    case QEvent::StyleChange:          /*****/
    case QEvent::PaletteChange:
    case QEvent::WindowTitleChange:
    case QEvent::IconTextChange:
    case QEvent::ModifiedChange:
    case QEvent::MouseTrackingChange:
    case QEvent::ParentChange:
    case QEvent::LocaleChange:
    case QEvent::MacSizeChange:
    case QEvent::ContentsRectChange:
    case QEvent::ThemeChange:
    case QEvent::ReadOnlyChange:
        changeEvent(event);
        break;
    .....
}

```

ok，继续追踪

```

void QWidget::changeEvent(QEvent * event)
{

```

```

        switch(event->type()) {
        case QEvent::EnabledChange: {
            update();
#ifdef QT_NO_ACCESSIBILITY
            QAccessible::State s;
            s.disabled = true;
            QAccessibleStateChangeEvent event(this, s);
            QAccessible::updateAccessibility(&event);
#endif
            break;
        }

        case QEvent::FontChange:
        case QEvent::StyleChange: {           /*******/
            Q_D(QWidget);
            update();
            updateGeometry();
            if (d->layout)
                d->layout->invalidate();
            break;
        }
    }

```

好了，最终调用了 update，毫无疑问，肯定是进入了 paintEvent 事件，看看 paintEvent 是怎么实现的，由于 Qt 只支持背景属性，不是很全，所以我们去看一个支持比较全的，QFrame

```

void QFrame::paintEvent(QPaintEvent *)
{
    QPainter paint(this);
    drawFrame(&paint);
}

```

继续追踪

```

void QFrame::drawFrame(QPainter *p)
{
    QStyleOptionFrame opt;
    initStyleOption(&opt);
    style()->drawControl(QStyle::CE_ShapedFrame, &opt, p, this);
}

```

最后，又回到了 Qstyle，由于 QStyle 是一个抽象类，这个接口刚好是纯虚函数，我们随便找一个它的派生类，看看它的 drawControl 等函数如何实现，这里我选取了 QCommonStyle 类，

```

case CE_ShapedFrame:
    if (const QStyleOptionFrame *f = qstyleoption_cast<const QStyleOptionFrame *>(opt)) {
        int frameShape = f->frameShape;
        int frameShadow = QFrame::Plain;
        if (f->state & QStyle::State_Sunken) {
            frameShadow = QFrame::Sunken;
        } else if (f->state & QStyle::State_Raised) {
            frameShadow = QFrame::Raised;
        }

        int lw = f->lineWidth;
        int mlw = f->midLineWidth;
        QPalette::ColorRole foregroundRole = QPalette::WindowText;
        if (widget)
            foregroundRole = widget->foregroundRole();

        switch (frameShape) {
        case QFrame::Box:
            if (frameShadow == QFrame::Plain) {
                qDrawPlainRect(p, f->rect, f->palette.color(foregroundRole), lw);
            } else {
                qDrawShadeRect(p, f->rect, f->palette, frameShadow == QFrame::Sunken, lw, mlw);
            }
            break;
        case QFrame::StyledPanel:
            //keep the compatibility with Qt 4.4 if there is a proxy style.
            //be sure to call drawPrimitive(QStyle::PE_Frame) on the proxy style
            if (widget) {
                widget->style()->drawPrimitive(QStyle::PE_Frame, opt, p, widget);
            } else {

```

注意看，这里真正用于绘制样式的颜色等值，最终还是从 `QStyleOption` 对象中获得，可以想象，Parser 解析出来的各种样式相关数据最终被放在了 `QStyleOption` 和它的派生类对象中，当 `widget` 需要绘制的时候，调用 `style` 函数获取 `style` 对象，再获取所需要的数据，到这里我们大致了解了 `stylesheet` 从设置到完成绘制的大概过程，再看看 `widget` 中的 `style` 对象是从哪里来的，

```

QStyle *QWidget::style() const
{
    Q_D(const QWidget);

    if (d->extra && d->extra->style)
        return d->extra->style;
    return QApplication::style();
}

```

ok，这里有两种情况，先看看 `extra->style` 是如何得到的，经过搜索，它只在这里进行过赋值，继续追踪，再看看这个 `setStyle_helper` 在哪里被调用，

```

void QWidgetPrivate::setStyle_helper(QStyle *newStyle, bool propagate, bool
#if 0 // Used to be included in Qt4 for Q_WS_MAC
    metalHack
#endif
)
{
    Q_Q(QWidget);
    QStyle *oldStyle = q->style();
#ifdef QT_NO_STYLE_STYLESHEET
    QPointer<QStyle> origStyle;
#endif

#if 0 // Used to be included in Qt4 for Q_WS_MAC
    // the metalhack boolean allows Qt/Mac to do a proper re-polish depending
    // on how the Qt::WA_MacBrushedMetal attribute is set. It is only ever
    // set when changing that attribute and passes the widget's CURRENT style.
    // therefore no need to do a reassignment.
    if (!metalHack)
#endif
    {
        createExtra();

#ifdef QT_NO_STYLE_STYLESHEET
        origStyle = extra->style.data();
#endif
        extra->style = newStyle;
    }
}

```

继续搜索, 发现 `setStyle_helper` 被调用过 8 次, 其中两次都在 `setStyleSheet` 中, 如图

```

void QWidget::setStyleSheet(const QString& styleSheet)
{
    Q_D(QWidget);
    if (data->in_destructor)
        return;
    d->createExtra();

    QStyleSheetStyle *proxy = qobject_cast<QStyleSheetStyle *>(d->extra->style);
    d->extra->styleSheet = styleSheet;
    if (styleSheet.isEmpty()) { // stylesheet removed
        if (!proxy)
            return;

        d->inheritStyle();
        return;
    }

    if (proxy) { // style sheet update
        if (d->polished)
            proxy->repolish(this);
        return;
    }

    if (testAttribute(Qt::WA_SetStyle)) {
        d->setStyle_helper(new QStyleSheetStyle(d->extra->style), true);
    } else {
        d->setStyle_helper(new QStyleSheetStyle(0), true);
    }
}

```

ok, 看到这里, 我们好像终于找到了一些有价值的线索: 调用控件的 `setStyleSheet` 可能会产生一个新的 `QStyle` 对象, OK, 写个简单的程序验证一下.

假如我们的应用程序控件很多, 几乎每个都单独设置一遍 `styleSheet`, 会不会创建很多个 `style`, 通过一个 demo, 我得出的结论是: 直接调用控件的 `setstylesheet`, 结果是每个控件 `style` 返回的对象都是不同的(地址不同足以证明是不同的对象), 而只给 `QApplication` 对象

`setStyleSheet`, 每个控件的 `style` 函数返回的对象都是相同的.

基于以上原因, 在开发时, 无论是出于维护的便捷性, 还是节省内存资源的考虑, 都应该有一个 `qss` 文件来存放所有的样式表, 而不应该将 `setStyleSheet` 写的到处都是.