在《[Qt Quick 之 QML 与 C++ 混合编程详解](http://blog.csdn.net/foruok/article/details/32698603)》一文中我们讲解了 QML 与 C++ 混合编程的方方面面的内容，这次我们通过一个图像处理应用，再来看一下 QML 与 C++ 混合编程的威力，同时也为诸君揭开美图秀秀、魔拍之类的相片美化应用的底层原理。

    项目的创建过程请参考《[Qt Quick 之 Hello World 图文详解](http://blog.csdn.net/foruok/article/details/28850879)》，项目名称为 imageProcessor ，创建完成后需要添加两个文件： imageProcessor.h 和 imageProcessor.cpp 。

    本文是作者 Qt Quick 系列文章中的一篇，其它文章在这里：

* [Qt Quick 简介](http://blog.csdn.net/foruok/article/details/28620525)
* [QML 语言基础](http://blog.csdn.net/foruok/article/details/28634543)
* [Qt Quick 之 Hello World 图文详解](http://blog.csdn.net/foruok/article/details/28850879)
* [Qt Quick 简单教程](http://blog.csdn.net/foruok/article/details/28859415)
* [Qt Quick 事件处理之信号与槽](http://blog.csdn.net/foruok/article/details/30028711)
* [Qt Quick事件处理之鼠标、键盘、定时器](http://blog.csdn.net/foruok/article/details/30474299)
* [Qt Quick 事件处理之捏拉缩放与旋转](http://blog.csdn.net/foruok/article/details/32078761)
* [Qt Quick 组件与对象动态创建详解](http://blog.csdn.net/foruok/article/details/32730911)
* [Qt Quick 布局介绍](http://blog.csdn.net/foruok/article/details/33738227)
* [Qt Quick 之 QML 与 C++ 混合编程详解](http://blog.csdn.net/foruok/article/details/32698603)

**实例效果**

    先看一下示例的实际运行效果，然后我们再来展开。

    图 1 是在电脑上打开一个图片后的初始效果：

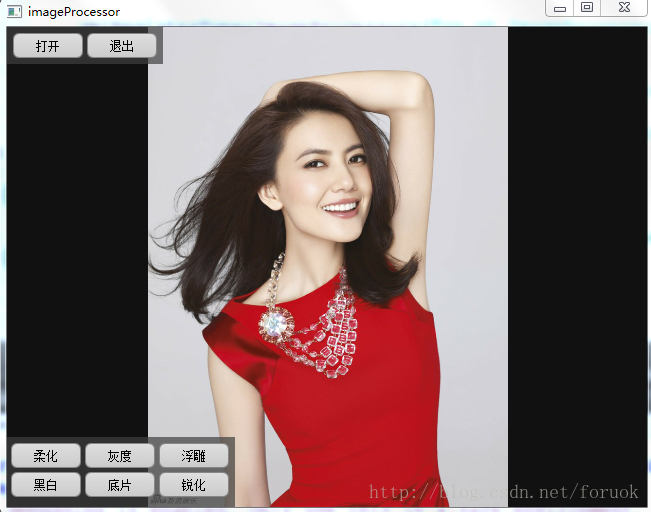


                  图 1 初始效果

    图 2 是应用柔化特效后的效果：

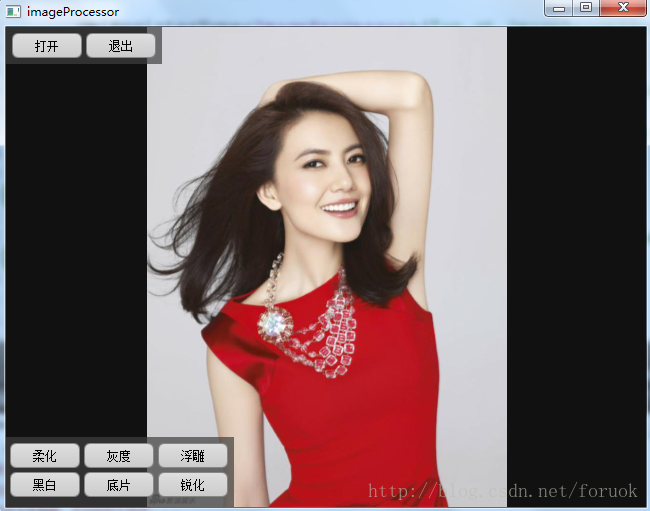


               图 2 柔化特效

    图 3 是应用灰度特效后的截图：

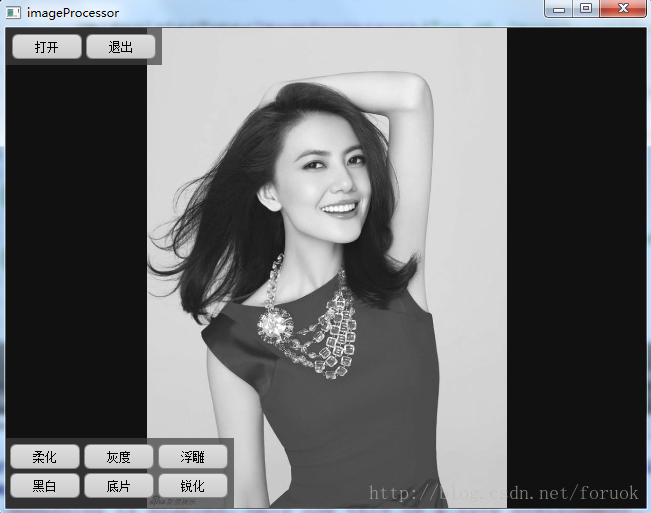


                   图 3 灰度特效

    图 4 是浮雕特效：

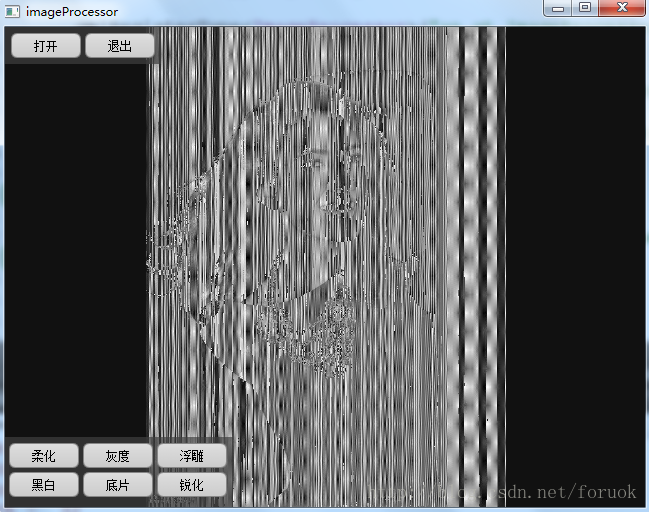


             图 4 浮雕特效

    图 5 是黑白特效：



                图 5 黑白特效

    图 6 是应用底片特效后的截图：

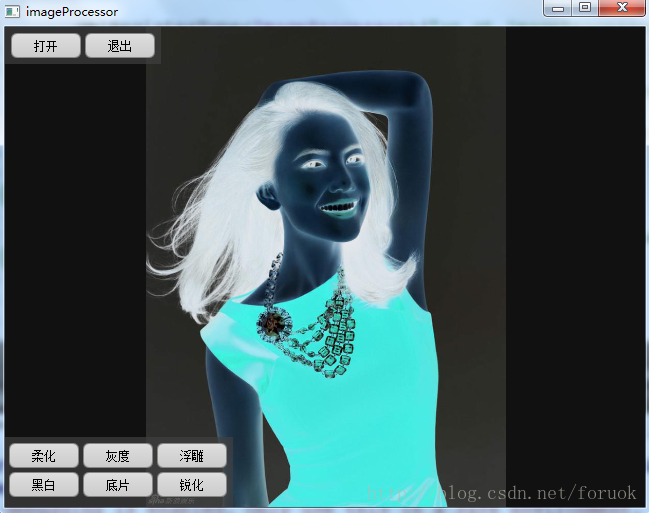


                  图 6 底片特效

    如果你注意到我博客的头像……嗯，木错，它就是我使用本文实例的底片特效做出来的。

    图 7 是应用锐化特效后的截图：



                  图 7 锐化特效

    特效展示完毕，那么它们是怎么实现的呢？这就要说到图像处理算法了。

**图像处理算法**

    imageProcessor 实例提供了"柔化"、"灰度"、"浮雕"、"黑白"、"底片"、"锐化"六种图像效果。算法的实现在 imageProcessor.h / imageProcessor.cpp 两个文件中，我们先简介每种效果对应的算法，然后看代码实现。

**柔化**

    柔化又称模糊，图像模糊算法有很多种，我们最常见的就是均值模糊，即取一定半径内的像素值之平均值作为当前点的新的像素值。

    为了提高计算速度，我们取 3 为半径，就是针对每一个像素，将周围 8 个点加上自身的 RGB 值的平均值作为像素新的颜色值置。代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. **static** **void** \_soften(QString sourceFile, QString destFile)
2. {
3. QImage image(sourceFile);
4. **if**(image.isNull())
5. {
6. qDebug() << "load " << sourceFile << " failed! ";
7. **return**;
8. }
9. **int** width = image.width();
10. **int** height = image.height();
11. **int** r, g, b;
12. QRgb color;
13. **int** xLimit = width - 1;
14. **int** yLimit = height - 1;
15. **for**(**int** i = 1; i < xLimit; i++)
16. {
17. **for**(**int** j = 1; j < yLimit; j++)
18. {
19. r = 0;
20. g = 0;
21. b = 0;
22. **for**(**int** m = 0; m < 9; m++)
23. {
24. **int** s = 0;
25. **int** p = 0;
26. **switch**(m)
27. {
28. **case** 0:
29. s = i - 1;
30. p = j - 1;
31. **break**;
32. **case** 1:
33. s = i;
34. p = j - 1;
35. **break**;
36. **case** 2:
37. s = i + 1;
38. p = j - 1;
39. **break**;
40. **case** 3:
41. s = i + 1;
42. p = j;
43. **break**;
44. **case** 4:
45. s = i + 1;
46. p = j + 1;
47. **break**;
48. **case** 5:
49. s = i;
50. p = j + 1;
51. **break**;
52. **case** 6:
53. s = i - 1;
54. p = j + 1;
55. **break**;
56. **case** 7:
57. s = i - 1;
58. p = j;
59. **break**;
60. **case** 8:
61. s = i;
62. p = j;
63. }
64. color = image.pixel(s, p);
65. r += qRed(color);
66. g += qGreen(color);
67. b += qBlue(color);
68. }
70. r = (**int**) (r / 9.0);
71. g = (**int**) (g / 9.0);
72. b = (**int**) (b / 9.0);
74. r = qMin(255, qMax(0, r));
75. g = qMin(255, qMax(0, g));
76. b = qMin(255, qMax(0, b));
78. image.setPixel(i, j, qRgb(r, g, b));
79. }
80. }
82. image.save(destFile);
83. }

    这样处理的效果不是特别明显，采用高斯模糊算法可以获取更好的效果。

**灰度**

    把图像变灰，大概有这么三种方法：

1. 最大值法，即 R = G = B = max(R , G , B)，这种方法处理过的图片亮度偏高
2. 平均值法，即 R = G = B = (R + G + B) / 3 ，这种方法处理过的图片比较柔和
3. 加权平均值法，即 R = G = B = R\*Wr + G\*Wg + B\*Wb ，因为人眼对不同颜色的敏感度不一样，三种颜色权重也不一样，一般来说绿色最高，红色次之，蓝色最低。这种方法最合理的取值，红、绿、蓝的权重依次是 0.299 、0.587 、 0.114 。为了避免浮点运算，可以用移位替代。

    Qt 框架有一个 qGray() 函数，采取加权平均值法计算灰度。 qGray() 将浮点运算转为整型的乘法和除法，公式是  (r \* 11 + g \* 16 + b \* 5)/32 ，没有使用移位运算。

    我使用 qGray() 函数计算灰度，下面是代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. **static** **void** \_gray(QString sourceFile, QString destFile)
2. {
3. QImage image(sourceFile);
4. **if**(image.isNull())
5. {
6. qDebug() << "load " << sourceFile << " failed! ";
7. **return**;
8. }
9. qDebug() << "depth - " << image.depth();
11. **int** width = image.width();
12. **int** height = image.height();
13. QRgb color;
14. **int** gray;
15. **for**(**int** i = 0; i < width; i++)
16. {
17. **for**(**int** j= 0; j < height; j++)
18. {
19. color = image.pixel(i, j);
20. gray = qGray(color);
21. image.setPixel(i, j,
22. qRgba(gray, gray, gray, qAlpha(color)));
23. }
24. }
26. image.save(destFile);
27. }

    qGray() 计算灰度时忽略了 Alpha 值，我在实现时保留原有的 Alpha 值。

**浮雕**

    "浮雕" 图象效果是指图像的前景前向凸出背景。

    浮雕的算法相对复杂一些，用当前点的 RGB 值减去相邻点的 RGB 值并加上 128 作为新的 RGB 值。由于图片中相邻点的颜色值是比较接近的，因此这样的算法处理之后，只有颜色的边沿区域，也就是相邻颜色差异较大的部分的结果才会比较明显，而其他平滑区域则值都接近128左右，也就是灰色，这样就具有了浮雕效果。

    看代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. **static** **void** \_emboss(QString sourceFile, QString destFile)
2. {
3. QImage image(sourceFile);
4. **if**(image.isNull())
5. {
6. qDebug() << "load " << sourceFile << " failed! ";
7. **return**;
8. }
9. **int** width = image.width();
10. **int** height = image.height();
11. QRgb color;
12. QRgb preColor = 0;
13. QRgb newColor;
14. **int** gray, r, g, b, a;
15. **for**(**int** i = 0; i < width; i++)
16. {
17. **for**(**int** j= 0; j < height; j++)
18. {
19. color = image.pixel(i, j);
20. r = qRed(color) - qRed(preColor) + 128;
21. g = qGreen(color) - qGreen(preColor) + 128;
22. b = qBlue(color) - qBlue(preColor) + 128;
23. a = qAlpha(color);
24. gray = qGray(r, g, b);
25. newColor = qRgba(gray, gray, gray, a);
26. image.setPixel(i, j, newColor);
27. preColor = newColor;
28. }
29. }
30. image.save(destFile);
31. }

    在实现 \_emboss() 函数时 ，为避免有些区域残留“彩色”杂点或者条状痕迹，我对新的 RGB 值又做了一次灰度处理。

**黑白**

    黑白图片的处理算法比较简单：对一个像素的 R 、G 、B 求平均值，average = (R + G + B) / 3 ，如果 average 大于等于选定的阈值则将该像素置为白色，小于阈值就把像素置为黑色。

    示例中我选择的阈值是 128 ，也可以是其它值，根据效果调整即可。比如你媳妇儿高圆圆嫌给她拍的照片黑白处理后黑多白少，那可以把阈值调低一些，取 80 ，效果肯定就变了。下面是代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. **static** **void** \_binarize(QString sourceFile, QString destFile)
2. {
3. QImage image(sourceFile);
4. **if**(image.isNull())
5. {
6. qDebug() << "load " << sourceFile << " failed! ";
7. **return**;
8. }
9. **int** width = image.width();
10. **int** height = image.height();
11. QRgb color;
12. QRgb avg;
13. QRgb black = qRgb(0, 0, 0);
14. QRgb white = qRgb(255, 255, 255);
15. **for**(**int** i = 0; i < width; i++)
16. {
17. **for**(**int** j= 0; j < height; j++)
18. {
19. color = image.pixel(i, j);
20. avg = (qRed(color) + qGreen(color) + qBlue(color))/3;
21. image.setPixel(i, j, avg >= 128 ? white : black);
22. }
23. }
24. image.save(destFile);
25. }

    代码的逻辑简单，从文件加载图片，生成一个 QImage 实例，然后应用算法，处理后的图片保存到指定位置。

**底片**

    早些年的相机使用胶卷记录拍摄结果，洗照片比较麻烦，不过如果你拿到底片，逆光去看，效果就很特别。

    底片算法其实很简单，取 255 与像素的 R 、 G、 B 分量之差作为新的 R、 G、 B 值。实现代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. **static** **void** \_negative(QString sourceFile, QString destFile)
2. {
3. QImage image(sourceFile);
4. **if**(image.isNull())
5. {
6. qDebug() << "load " << sourceFile << " failed! ";
7. **return**;
8. }
9. **int** width = image.width();
10. **int** height = image.height();
11. QRgb color;
12. QRgb negative;
13. **for**(**int** i = 0; i < width; i++)
14. {
15. **for**(**int** j= 0; j < height; j++)
16. {
17. color = image.pixel(i, j);
18. negative = qRgba(255 - qRed(color),
19. 255 - qGreen(color),
20. 255 - qBlue(color),
21. qAlpha(color));
22. image.setPixel(i, j, negative);
23. }
24. }
25. image.save(destFile);
26. }

**锐化**

    图像锐化的主要目的是增强图像边缘，使模糊的图像变得更加清晰，颜色变得鲜明突出，图像的质量有所改善，产生更适合人眼观察和识别的图像。

    常见的锐化算法有微分法和高通滤波法。微分法又以梯度锐化和拉普拉斯锐化较为常用。本示例采用微分法中的梯度锐化，用差分近似微分，则图像在点(i,j)处的梯度幅度计算公式如下：

**[plain]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. G[f(i,j)] = abs(f(i,j) - f(i+1,j)) + abs(f(i,j) - f(i,j+1))

    为了更好的增强图像边缘，我们引入一个阈值，只有像素点的梯度值大于阈值时才对该像素点进行锐化，将像素点的 R 、 G、 B 值设置为对应的梯度值与一个常数之和。常数值的选取应当参考图像的具体特点。我们的示例为简单起见，将常数设定为 100 ，梯度阈值取 80 ，写死在算法函数中，更好的做法是通过参数传入，以便客户程序可以调整这些变量来观察效果。

    好啦，看代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. **static** **void** \_sharpen(QString sourceFile, QString destFile)
2. {
3. QImage image(sourceFile);
4. **if**(image.isNull())
5. {
6. qDebug() << "load " << sourceFile << " failed! ";
7. **return**;
8. }
9. **int** width = image.width();
10. **int** height = image.height();
11. **int** threshold = 80;
12. QImage sharpen(width, height, QImage::Format\_ARGB32);
13. **int** r, g, b, gradientR, gradientG, gradientB;
14. QRgb rgb00, rgb01, rgb10;
15. **for**(**int** i = 0; i < width; i++)
16. {
17. **for**(**int** j= 0; j < height; j++)
18. {
19. **if**(image.valid(i, j) &&
20. image.valid(i+1, j) &&
21. image.valid(i, j+1))
22. {
23. rgb00 = image.pixel(i, j);
24. rgb01 = image.pixel(i, j+1);
25. rgb10 = image.pixel(i+1, j);
26. r = qRed(rgb00);
27. g = qGreen(rgb00);
28. b = qBlue(rgb00);
29. gradientR = abs(r - qRed(rgb01)) + abs(r - qRed(rgb10));
30. gradientG = abs(g - qGreen(rgb01)) +
31. abs(g - qGreen(rgb10));
32. gradientB = abs(b - qBlue(rgb01)) +
33. abs(b - qBlue(rgb10));
35. **if**(gradientR > threshold)
36. {
37. r = qMin(gradientR + 100, 255);
38. }
40. **if**(gradientG > threshold)
41. {
42. g = qMin( gradientG + 100, 255);
43. }
45. **if**(gradientB > threshold)
46. {
47. b = qMin( gradientB + 100, 255);
48. }
50. sharpen.setPixel(i, j, qRgb(r, g, b));
51. }
52. }
53. }
55. sharpen.save(destFile);
56. }

    示例用到的图像处理算法和 Qt 代码实现已经介绍完毕，您看得累吗？累就对了，舒服是留给死人的。擦，睡着了，我敲打……

**源码情景分析**

    上一节介绍了图像特效算法，现在我们先看应用与管理这些特效的 C++ 类 ImageProcessor ，然后再来看 QML 代码。

**ImageProcessor**

    在设计 ImageProcessor 类时，我希望它能够在 QML 环境中使用，因此实用了信号、槽、 Q\_ENUMS 、 Q\_PROPERTY 等特性，感兴趣的话请参考《[Qt Quick 之 QML 与 C++ 混合编程详解](http://blog.csdn.net/foruok/article/details/32698603)》进一步熟悉。

    先看 imageProcessor.h ：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. #ifndef IMAGEPROCESSOR\_H
2. #define IMAGEPROCESSOR\_H
3. #include <QObject>
4. #include <QString>
6. **class** ImageProcessorPrivate;
7. **class** ImageProcessor : **public** QObject
8. {
9. Q\_OBJECT
10. Q\_ENUMS(ImageAlgorithm)
11. Q\_PROPERTY(QString sourceFile READ sourceFile)
12. Q\_PROPERTY(ImageAlgorithm algorithm READ algorithm)
14. **public**:
15. ImageProcessor(QObject \*parent = 0);
16. ~ImageProcessor();
18. **enum** ImageAlgorithm{
19. Gray = 0,
20. Binarize,
21. Negative,
22. Emboss,
23. Sharpen,
24. Soften,
25. AlgorithmCount
26. };
28. QString sourceFile() **const**;
29. ImageAlgorithm algorithm() **const**;
30. **void** setTempPath(QString tempPath);
32. signals:
33. **void** finished(QString newFile);
34. **void** progress(**int** value);
36. **public** slots:
37. **void** process(QString file, ImageAlgorithm algorithm);
38. **void** abort(QString file, ImageAlgorithm algorithm);
39. **void** abortAll();
41. **private**:
42. ImageProcessorPrivate \*m\_d;
43. };
45. #endif

    ImageProcessor 类的声明比较简单，它通过 finished() 信号通知关注者图像处理完毕，提供 process() 方法供客户程序调用，还有 setTempPath() 设置临时目录，也允许你取消待执行的任务……

    下面是实现文件 imageProcessor.cpp ：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. #include "imageProcessor.h"
2. #include <QThreadPool>
3. #include <QList>
4. #include <QFile>
5. #include <QFileInfo>
6. #include <QRunnable>
7. #include <QEvent>
8. #include <QCoreApplication>
9. #include <QPointer>
10. #include <QUrl>
11. #include <QImage>
12. #include <QDebug>
13. #include <QDir>
15. **typedef** **void** (\*AlgorithmFunction)(QString sourceFile,
16. QString destFile);
18. **class** AlgorithmRunnable;
19. **class** ExcutedEvent : **public** QEvent
20. {
21. **public**:
22. ExcutedEvent(AlgorithmRunnable \*r)
23. : QEvent(evType()), m\_runnable(r)
24. {
25. }
26. AlgorithmRunnable \*m\_runnable;
28. **static** QEvent::Type evType()
29. {
30. **if**(s\_evType == QEvent::None)
31. {
32. s\_evType = (QEvent::Type)registerEventType();
33. }
34. **return** s\_evType;
35. }
37. **private**:
38. **static** QEvent::Type s\_evType;
39. };
40. QEvent::Type ExcutedEvent::s\_evType = QEvent::None;
42. **static** **void** \_gray(QString sourceFile, QString destFile);
43. **static** **void** \_binarize(QString sourceFile, QString destFile);
44. **static** **void** \_negative(QString sourceFile, QString destFile);
45. **static** **void** \_emboss(QString sourceFile, QString destFile);
46. **static** **void** \_sharpen(QString sourceFile, QString destFile);
47. **static** **void** \_soften(QString sourceFile, QString destFile);
49. **static** AlgorithmFunction g\_functions[ImageProcessor::AlgorithmCount] = {
50. \_gray,
51. \_binarize,
52. \_negative,
53. \_emboss,
54. \_sharpen,
55. \_soften
56. };
58. **class** AlgorithmRunnable : **public** QRunnable
59. {
60. **public**:
61. AlgorithmRunnable(
62. QString sourceFile,
63. QString destFile,
64. ImageProcessor::ImageAlgorithm algorithm,
65. QObject \* observer)
66. : m\_observer(observer)
67. , m\_sourceFilePath(sourceFile)
68. , m\_destFilePath(destFile)
69. , m\_algorithm(algorithm)
70. {
71. }
72. ~AlgorithmRunnable(){}

75. **void** run()
76. {
77. g\_functions[m\_algorithm](m\_sourceFilePath, m\_destFilePath);
78. QCoreApplication::postEvent(m\_observer,
79. **new** ExcutedEvent(**this**));
80. }
82. QPointer<QObject> m\_observer;
83. QString m\_sourceFilePath;
84. QString m\_destFilePath;
85. ImageProcessor::ImageAlgorithm m\_algorithm;
86. };
88. **class** ImageProcessorPrivate : **public** QObject
89. {
90. **public**:
91. ImageProcessorPrivate(ImageProcessor \*processor)
92. : QObject(processor), m\_processor(processor),
93. m\_tempPath(QDir::currentPath())
94. {
95. ExcutedEvent::evType();
96. }
97. ~ImageProcessorPrivate()
98. {
99. }
101. **bool** event(QEvent \* e)
102. {
103. **if**(e->type() == ExcutedEvent::evType())
104. {
105. ExcutedEvent \*ee = (ExcutedEvent\*)e;
106. **if**(m\_runnables.contains(ee->m\_runnable))
107. {
108. m\_notifiedAlgorithm = ee->m\_runnable->m\_algorithm;
109. m\_notifiedSourceFile =
110. ee->m\_runnable->m\_sourceFilePath;
111. emit m\_processor->finished(ee->m\_runnable->m\_destFilePath);
112. m\_runnables.removeOne(ee->m\_runnable);
113. }
114. **delete** ee->m\_runnable;
115. **return** **true**;
116. }
117. **return** QObject::event(e);
118. }
120. **void** process(QString sourceFile, ImageProcessor::ImageAlgorithm algorithm)
121. {
122. QFileInfo fi(sourceFile);
123. QString destFile = QString("%1/%2\_%3").arg(m\_tempPath)
124. .arg((**int**)algorithm).arg(fi.fileName());
125. AlgorithmRunnable \*r = **new** AlgorithmRunnable(sourceFile,
126. destFile, algorithm, **this**);
127. m\_runnables.append(r);
128. r->setAutoDelete(**false**);
129. QThreadPool::globalInstance()->start(r);
130. }
132. ImageProcessor \* m\_processor;
133. QList<AlgorithmRunnable\*> m\_runnables;
134. QString m\_notifiedSourceFile;
135. ImageProcessor::ImageAlgorithm m\_notifiedAlgorithm;
136. QString m\_tempPath;
137. };
139. ImageProcessor::ImageProcessor(QObject \*parent)
140. : QObject(parent)
141. , m\_d(**new** ImageProcessorPrivate(**this**))
142. {}
143. ImageProcessor::~ImageProcessor()
144. {
145. **delete** m\_d;
146. }
148. QString ImageProcessor::sourceFile() **const**
149. {
150. **return** m\_d->m\_notifiedSourceFile;
151. }
153. ImageProcessor::ImageAlgorithm ImageProcessor::algorithm() **const**
154. {
155. **return** m\_d->m\_notifiedAlgorithm;
156. }
158. **void** ImageProcessor::setTempPath(QString tempPath)
159. {
160. m\_d->m\_tempPath = tempPath;
161. }
163. **void** ImageProcessor::process(QString file, ImageAlgorithm algorithm)
164. {
165. m\_d->process(file, algorithm);
166. }
168. **void** ImageProcessor::abort(QString file, ImageAlgorithm algorithm)
169. {
170. **int** size = m\_d->m\_runnables.size();
171. AlgorithmRunnable \*r;
172. **for**(**int** i = 0; i < size; i++)
173. {
174. r = m\_d->m\_runnables.at(i);
175. **if**(r->m\_sourceFilePath == file && r->m\_algorithm == algorithm)
176. {
177. m\_d->m\_runnables.removeAt(i);
178. **break**;
179. }
180. }
181. }

    为避免阻塞 UI 线程，我把图像处理部分放到线程池内完成，根据 QThreadPool 的要求，从 QRunnable 继承，实现了 AlgorithmRunnable ，当 run() 函数执行完时发送自定义的 ExecutedEvent 给 ImageProcessor ，而 ImageProcessor 就在处理事件时发出 finished() 信号。关于 QThreadPool 和自定义事件，请参考 Qt 帮助了解详情。

    算法函数放在一个全局的函数指针数组中， AlgorithmRunnable 则根据算法枚举值从数组中取出相应的函数来处理图像。

    其它的代码一看即可明白，不再多说。

    要想在 QML 中实用 ImageProcessor 类，需要导出一个 QML 类型。这个工作是在 main() 函数中完成的。

**main() 函数**

    main() 函数就在 main.cpp 中，下面是 main.cpp 的全部代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. #include <QApplication>
2. #include "qtquick2applicationviewer.h"
3. #include <QtQml>
4. #include "imageProcessor.h"
5. #include <QQuickItem>
6. #include <QDebug>
8. **int** main(**int** argc, **char** \*argv[])
9. {
10. QApplication app(argc, argv);
12. qmlRegisterType<ImageProcessor>("an.qt.ImageProcessor", 1, 0,"ImageProcessor");
14. QtQuick2ApplicationViewer viewer;
15. viewer.rootContext()->setContextProperty("imageProcessor", **new** ImageProcessor);
17. viewer.setMainQmlFile(QStringLiteral("qml/imageProcessor/main.qml"));
18. viewer.showExpanded();
20. **return** app.exec();
21. }

    我使用 qmlRegisterType() 注册了 ImageProcessor 类，包名是 an.qt.ImageProcessor ，版本是 1.0 ，所以你在稍后的 main.qml 文档中可以看到下面的导入语句：

**[javascript]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. **import** an.qt.ImageProcessor 1.0

    上了贼船，就跟贼走，是时候看看 main.qml 了 。

**main.qml**

    main.qml 还是比较长的哈，有 194 行代码：

**[javascript]** [view plaincopy](http://blog.csdn.net/foruok/article/details/37740583)[在CODE上查看代码片](https://code.csdn.net/snippets/427768)

1. **import** QtQuick 2.2
2. **import** QtQuick.Controls 1.1
3. **import** QtQuick.Dialogs 1.1
4. **import** an.qt.ImageProcessor 1.0
5. **import** QtQuick.Controls.Styles 1.1
7. Rectangle {
8. width: 640;
9. height: 480;
10. color: "#121212";
12. BusyIndicator {
13. id: busy;
14. running: **false**;
15. anchors.centerIn: parent;
16. z: 2;
17. }
19. Label {
20. id: stateLabel;
21. visible: **false**;
22. anchors.centerIn: parent;
23. }
25. Image {
26. objectName: "imageViewer";
27. id: imageViewer;
28. asynchronous: **true**;
29. anchors.fill: parent;
30. fillMode: Image.PreserveAspectFit;
31. onStatusChanged: {
32. **if** (imageViewer.status === Image.Loading) {
33. busy.running = **true**;
34. stateLabel.visible = **false**;
35. }
36. **else** **if**(imageViewer.status === Image.Ready){
37. busy.running = **false**;
38. }
39. **else** **if**(imageViewer.status === Image.Error){
40. busy.running = **false**;
41. stateLabel.visible = **true**;
42. stateLabel.text = "ERROR";
43. }
44. }
45. }
47. ImageProcessor {
48. id: processor;
49. onFinished: {
50. imageViewer.source = "file:///" +newFile;
51. }
52. }
54. FileDialog {
55. id: fileDialog;
56. title: "Please choose a file";
57. nameFilters: ["Image Files (\*.jpg \*.png \*.gif)"];
58. onAccepted: {
59. console.log(fileDialog.fileUrl);
60. imageViewer.source = fileDialog.fileUrl;
61. }
62. }
64. Component{
65. id: btnStyle;
66. ButtonStyle {
67. background: Rectangle {
68. implicitWidth: 70
69. implicitHeight: 25
70. border.width: control.pressed ? 2 : 1
71. border.color: (control.pressed || control.hovered) ? "#00A060" : "#888888"
72. radius: 6
73. gradient: Gradient {
74. GradientStop { position: 0 ; color: control.pressed ? "#cccccc" : "#e0e0e0" }
75. GradientStop { position: 1 ; color: control.pressed ? "#aaa" : "#ccc" }
76. }
77. }
78. }
79. }
81. Button {
82. id: openFile;
83. text: "打开";
84. anchors.left:  parent.left;
85. anchors.leftMargin: 6;
86. anchors.top: parent.top;
87. anchors.topMargin: 6;
88. onClicked: {
89. fileDialog.visible = **true**;
90. }
91. style: btnStyle;
92. z: 1;
93. }
95. Button {
96. id: quit;
97. text: "退出";
98. anchors.left: openFile.right;
99. anchors.leftMargin: 4;
100. anchors.bottom: openFile.bottom;
101. onClicked: {
102. Qt.quit()
103. }
104. style: btnStyle;
105. z: 1;
106. }
108. Rectangle {
109. anchors.left: parent.left;
110. anchors.top: parent.top;
111. anchors.bottom: openFile.bottom;
112. anchors.bottomMargin: -6;
113. anchors.right: quit.right;
114. anchors.rightMargin: -6;
115. color: "#404040";
116. opacity: 0.7;
117. }
118. Grid {
119. id: op;
120. anchors.left: parent.left;
121. anchors.leftMargin: 4;
122. anchors.bottom: parent.bottom;
123. anchors.bottomMargin: 4;
124. rows: 2;
125. columns: 3;
126. rowSpacing: 4;
127. columnSpacing: 4;
128. z: 1;
130. Button {
131. text: "柔化";
132. style: btnStyle;
133. onClicked: {
134. busy.running = **true**;
135. processor.process(fileDialog.fileUrl, ImageProcessor.Soften);
136. }
137. }
139. Button {
140. text: "灰度";
141. style: btnStyle;
142. onClicked: {
143. busy.running = **true**;
144. processor.process(fileDialog.fileUrl, ImageProcessor.Gray);
145. }
146. }
148. Button {
149. text: "浮雕";
150. style: btnStyle;
151. onClicked: {
152. busy.running = **true**;
153. processor.process(fileDialog.fileUrl, ImageProcessor.Emboss);
154. }
155. }
156. Button {
157. text: "黑白";
158. style: btnStyle;
159. onClicked: {
160. busy.running = **true**;
161. processor.process(fileDialog.fileUrl, ImageProcessor.Binarize);
162. }
163. }
165. Button {
166. text: "底片";
167. style: btnStyle;
168. onClicked: {
169. busy.running = **true**;
170. processor.process(fileDialog.fileUrl, ImageProcessor.Negative);
171. }
172. }
174. Button {
175. text: "锐化";
176. style: btnStyle;
177. onClicked: {
178. busy.running = **true**;
179. processor.process(fileDialog.fileUrl, ImageProcessor.Sharpen);
180. }
181. }
182. }
184. Rectangle {
185. anchors.left: parent.left;
186. anchors.top: op.top;
187. anchors.topMargin: -4;
188. anchors.bottom: parent.bottom;
189. anchors.right: op.right;
190. anchors.rightMargin: -4;
191. color: "#404040";
192. opacity: 0.7;
193. }
194. }

    图片的显示使用一个充满窗口的 Image 对象，在 onStatusChanged 信号处理器中控制加载提示对象 BusyIndicator 是否显示。我通过 Z 序来保证 busy 总是在 imageViewer 上面。

    你看到了，我像使用 QML 内建对象那样使用了 ImageProcessor 对象，为它的 finished 信号定义了 onFinished 信号处理器，在信号处理器中把应用图像特效后的中间文件传递给 imageViewer 来显示。

    界面布局比较简陋，打开和退出两个按钮放在左上角，使用锚布局。关于锚布局，请参考《[Qt Quick 布局介绍](http://blog.csdn.net/foruok/article/details/33738227)》或《[Qt Quick 简单教程](http://blog.csdn.net/foruok/article/details/28859415)》。图像处理的 6 个按钮使用 Grid 定位器来管理， 2 行 3 列，放在界面左下角。 Grid 定位器的使用请参考《[Qt Quick 布局介绍](http://blog.csdn.net/foruok/article/details/33738227)》。

    关于图像处理按钮，以黑白特效做下说明，在 onClicked 信号处理器中，调用 processor 的 process() 方法，传入本地图片路径和特效算法。当特效运算异步完成后，就会触发 finished 信号，进而 imageViewer 会更新……

    好啦好啦，我们的图像处理实例就到此为止了。秒懂？