  Qt Quick 技术的引入，使得你能够快速构建 UI ，具有动画、各种绚丽效果的 UI 都不在话下。但它不是万能的，也有很多局限性，原来 Qt 的一些技术，比如低阶的网络编程如 QTcpSocket ，多线程，又如 XML 文档处理类库 QXmlStreamReader / QXmlStreamWriter 等等，在 QML 中要么不可用，要么用起来不方便，所以呢，很多时候我们是会基于这样的原则来混合使用 QML 和 C++： QML 构建界面， C++ 实现非界面的业务逻辑和复杂运算。

    版权所有 foruok ，转载请注明出处：<http://blog.csdn.net/foruok> 。

    那这篇呢，我们就来看看 QML 和 C++ 之间如何交互。

    其实话说回来， QML 的很多基本类型原本就是在 C++ 中实现的，比如 Item 对应 QQuickItem ， Image 对应 QQuickImage ， Text 对应 QQuickText  ，……这样看来，在 QML 中访问 C++ 对象必然不成问题。然也！反过来，在 C++ 中其实也可以使用 QML 对象。

    对于这两种情景，我们分别讲述。先看如何在 QML 中使用 C++ 类和对象。

    首先我们需要创建一个 Qt Quick App ，请参考《[Qt Quick 之 Hello World 图文详解](http://blog.csdn.net/foruok/article/details/28850879)》建立一个名为 colorMaker 的项目，接下来我们的学习将会伴随 colorMaker 项目进行，等我们讲完，一个完整的 colorMaker 项目也会完成。需要新建两个文件， colorMaker.h 和 colorMaker.cpp 。

    colorMaker 只是一个示例项目，我在 C++ 中实现一个 ColorMaker 类，它可以被注册为一个 QML 类型供 QML 像内建类型一样使用，它的实例也可以导出为 QML 上下文属性在 QML 中访问。我们的示例只是在界面顶部显示当前时间（时间文字的颜色随时间变化而变化），在界面中间显示一个变色矩形，在界面底部放置几个按钮来控制颜色如何变化。

    图 1 是效果图：

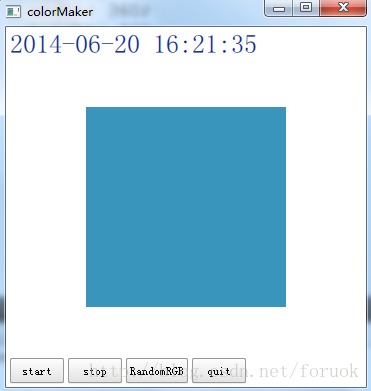


               图 1 colorMaker 效果图

**在 QML 中使用 C++ 类和对象**

    我们知道， QML 其实是对 JavaScript 的扩展，融合了 Qt Object 系统，它是一种新的解释型的语言， QML 引擎虽然由 Qt C++ 实现，但 QML 对象的运行环境，说到底和 C++ 对象的上下文环境是不同的，是平行的两个世界。如果你想在 QML 中访问 C++ 对象，那么必然要找到一种途径来在两个运行环境之间建立沟通桥梁。

    Qt 提供了两种在 QML 环境中使用 C++ 对象的方式：

1. 在 C++ 中实现一个类，注册到 QML 环境中， QML 环境中使用该类型创建对象
2. 在 C++ 中构造一个对象，将这个对象设置为 QML 的上下文属性，在 QML 环境中直接使用改属性

    不管哪种方式，对要导出的 C++ 类都有要求，不是一个类的所有方法、变量都可以被 QML 使用，因此我们先来看看怎样让一个方法或属性可以被 QML 使用。

**实现可以导出的 C++ 类**

**前提条件**

     要想将一个类或对象导出到 QML 中，下列前提条件必须满足：

* 从 QObject 或 QObject 的派生类继承
* 使用 Q\_OBJECT 宏

    看起来好像和使用信号与槽的前提条件一样……没错，的确是一样的。这两个条件是为了让一个类能够进入 Qt 强大的元对象系统（meta-object system）中，只有使用元对象系统，一个类的某些方法或属性才可能通过字符串形式的名字来调用，才具有了在 QML 中访问的基础条件。

    一旦你导出了一个类，在 QML 中就必然要访问该类的实例的属性或方法来达到某种目的，否则我真想不来你要干什么……而具有什么特征的属性或方法才可以被 QML 访问呢？

**信号，槽**

    只要是信号或者槽，都可以在 QML 中访问，你可以把 C++ 对象的信号连接到 QML 中定义的方法上，也可以把 QML 对象的信号连接到 C++ 对象的槽上，还可以直接调用 C++ 对象的槽或信号……所以，这是最简单好用的一种途径。

    下面初始 ColorMaker 类的声明：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **class** ColorMaker : **public** QObject
2. {
3. Q\_OBJECT
5. **public**:
6. ColorMaker(QObject \*parent = 0);
7. ~ColorMaker();
9. signals:
10. **void** colorChanged(**const** QColor & color);
11. **void** currentTime(**const** QString &strTime);
13. **public** slots:
14. **void** start();
15. **void** stop();
17. };

    我们定义了 start() / stop() 两个槽， colorChanged() / currentTime() 两个信号，都可以在 QML 中使用。

**Q\_INVOKABLE 宏**

    在定义一个类的成员函数时使用 Q\_INVOKABLE 宏来修饰，就可以让该方法被元对象系统调用。这个宏必须放在返回类型前面。

    我给 ColorMaker 添加了两个使用 Q\_INVOKABLE 宏修饰的方法，现在 ColorMaker 类的声明变成了这个样子：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **class** ColorMaker : **public** QObject
2. {
3. Q\_OBJECT
5. **public**:
6. ColorMaker(QObject \*parent = 0);
7. ~ColorMaker();
9. Q\_INVOKABLE GenerateAlgorithm algorithm() **const**;
10. Q\_INVOKABLE **void** setAlgorithm(GenerateAlgorithm algorithm);
12. signals:
13. **void** colorChanged(**const** QColor & color);
14. **void** currentTime(**const** QString &strTime);
16. **public** slots:
17. **void** start();
18. **void** stop();
19. };

    一旦你使用 Q\_INVOKABLE 将某个方法注册到元对象系统中，在 QML 中就可以用 ${Object}.${method} 来访问，colorMaker 的 main.qml 中有使用 algorithm() 和 setAlgorithm() 的 QML 代码 ：

**[javascript]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. Component.onCompleted: {
2. colorMaker.color = Qt.rgba(0,180,120, 255);
3. colorMaker.setAlgorithm(ColorMaker.LinearIncrease);
4. changeAlgorithm(colorAlgorithm, colorMaker.algorithm());
5. }

    我在根元素创建完成时初始化 colorMaker 对象，给它设定了颜色生成算法，同时改变界面上切换颜色生成算法的按钮的文字。

**Q\_ENUMS**

    如果你要导出的类定义了想在 QML 中使用枚举类型，可以使用 Q\_ENUMS 宏将该枚举注册到元对象系统中。

    ColorMaker 类定义了 GenerateAlgorithm 枚举类型，支持 RandomRGB / RandomRed 等颜色生成算法。现在 ColorMaker 类的声明变成了这个样子：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **class** ColorMaker : **public** QObject
2. {
3. Q\_OBJECT
4. Q\_ENUMS(GenerateAlgorithm)
6. **public**:
7. ColorMaker(QObject \*parent = 0);
8. ~ColorMaker();
10. **enum** GenerateAlgorithm{
11. RandomRGB,
12. RandomRed,
13. RandomGreen,
14. RandomBlue,
15. LinearIncrease
16. };
18. Q\_INVOKABLE GenerateAlgorithm algorithm() **const**;
19. Q\_INVOKABLE **void** setAlgorithm(GenerateAlgorithm algorithm);
21. signals:
22. **void** colorChanged(**const** QColor & color);
23. **void** currentTime(**const** QString &strTime);
25. **public** slots:
26. **void** start();
27. **void** stop();
28. };

    一旦你使用 Q\_ENUMS 宏注册了你的枚举类型，在 QML 中就可以用 ${CLASS\_NAME}.${ENUM\_VALUE} 的形式来访问，比如 ColorMaker.LinearIncrease ，上节展示的 QML 代码片段已经使用了导出的枚举类型。

**Q\_PROPERTY**

    Q\_PROPERTY 宏用来定义可通过元对象系统访问的属性，通过它定义的属性，可以在 QML 中访问、修改，也可以在属性变化时发射特定的信号。要想使用 Q\_PROPERTY 宏，你的类必须是 QObject 的后裔，必须在类首使用 Q\_OBJECT 宏。

    下面是 Q\_PROPERTY 宏的原型：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. Q\_PROPERTY(type name
2. (READ getFunction [WRITE setFunction] |
3. MEMBER memberName [(READ getFunction | WRITE setFunction)])
4. [RESET resetFunction]
5. [NOTIFY notifySignal]
6. [REVISION **int**]
7. [DESIGNABLE **bool**]
8. [SCRIPTABLE **bool**]
9. [STORED **bool**]
10. [USER **bool**]
11. [CONSTANT]
12. [FINAL])

    是不是很复杂？你可以为一个属性命名，可以设定的选项数超过10个……我是觉得有点儿头疼。不过，不是所有的选项都必须设定，看一个最简短的属性声明：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. Q\_PROPERTY(**int** x READ x)

    上面的声明定义了一个类型为 int 名为 x 的属性，通过方法 x() 来访问。

    type name 这两个字段想必不用细说了吧？ type 是属性的类型，可以是 int / float / QString / QObject / QColor / QFont 等等， name 就是属性的名字。

    其实我们在实际使用中，很少能够用全 Q\_PROPERTY 的所有选项，就往 QML 导出类这种场景来说，比较常用的是 READ / WRITE / NOTIFY 三个选项。我们来看看都是什么含义。

* READ 标记，如果你没有为属性指定 MEMBER 标记，则 READ 标记必不可少；声明一个读取属性的函数，该函数一般没有参数，返回定义的属性。
* WRITE 标记，可选配置。声明一个设定属性的函数。它指定的函数，只能有一个与属性类型匹配的参数，必须返回 void 。
* NOTIFY 标记，可选配置。给属性关联一个信号（该信号必须是已经在类中声明过的），当属性的值发生变化时就会触发该信号。信号的参数，一般就是你定义的属性。

    其它标记的含义，请参考 Qt SDK 。

    QML 中的 Text 类型对应 C++ 中的 QQuickText 类，下面是我摘取的部分代码，可以看到 Q\_ENUMS 和 Q\_PROPERTY 的使用：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **class** QQuickText : **public** QQuickImplicitSizeItem
2. {
3. Q\_OBJECT
4. Q\_ENUMS(HAlignment)

7. Q\_PROPERTY(QString text READ text WRITE setText NOTIFY textChanged)
8. Q\_PROPERTY(QFont font READ font WRITE setFont NOTIFY fontChanged)
9. Q\_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
10. ...
12. **public**:
13. **enum** HAlignment { AlignLeft = Qt::AlignLeft,
14. AlignRight = Qt::AlignRight,
15. AlignHCenter = Qt::AlignHCenter,
16. AlignJustify = Qt::AlignJustify };
17. ...
18. QString text() **const**;
19. **void** setText(**const** QString &);
21. QFont font() **const**;
22. **void** setFont(**const** QFont &font);
24. QColor color() **const**;
25. **void** setColor(**const** QColor &c);
26. ...
27. };

    现在给我们的 ColorMaker 类添加一些属性，以便 QML 可以获取、设置颜色值。新的 ColorMaker 类如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **class** ColorMaker : **public** QObject
2. {
3. Q\_OBJECT
4. Q\_ENUMS(GenerateAlgorithm)
5. Q\_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
6. Q\_PROPERTY(QColor timeColor READ timeColor)
8. **public**:
9. ColorMaker(QObject \*parent = 0);
10. ~ColorMaker();
12. **enum** GenerateAlgorithm{
13. RandomRGB,
14. RandomRed,
15. RandomGreen,
16. RandomBlue,
17. LinearIncrease
18. };
20. QColor color() **const**;
21. **void** setColor(**const** QColor & color);
22. QColor timeColor() **const**;
24. Q\_INVOKABLE GenerateAlgorithm algorithm() **const**;
25. Q\_INVOKABLE **void** setAlgorithm(GenerateAlgorithm algorithm);
27. signals:
28. **void** colorChanged(**const** QColor & color);
29. **void** currentTime(**const** QString &strTime);
31. **public** slots:
32. **void** start();
33. **void** stop();
35. **protected**:
36. **void** timerEvent(QTimerEvent \*e);
38. **private**:
39. GenerateAlgorithm m\_algorithm;
40. QColor m\_currentColor;
41. **int** m\_nColorTimer;
42. };

    现在我们的 ColorMaker 已经是一个完整的类了，有信号、有槽、有使用 Q\_INVOKABLE 注册的方法，还导出了枚举类型，小小麻雀五脏俱全。

    是时候看看它的实现了。翠花，上代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. #include "colorMaker.h"
2. #include <QTimerEvent>
3. #include <QDateTime>
5. ColorMaker::ColorMaker(QObject \*parent)
6. : QObject(parent)
7. , m\_algorithm(RandomRGB)
8. , m\_currentColor(Qt::black)
9. , m\_nColorTimer(0)
10. {
11. qsrand(QDateTime::currentDateTime().toTime\_t());
12. }
14. ColorMaker::~ColorMaker()
15. {
16. }
18. QColor ColorMaker::color() **const**
19. {
20. **return** m\_currentColor;
21. }
23. **void** ColorMaker::setColor(**const** QColor &color)
24. {
25. m\_currentColor = color;
26. emit colorChanged(m\_currentColor);
27. }
29. QColor ColorMaker::timeColor() **const**
30. {
31. QTime time = QTime::currentTime();
32. **int** r = time.hour();
33. **int** g = time.minute()\*2;
34. **int** b = time.second()\*4;
35. **return** QColor::fromRgb(r, g, b);
36. }
38. ColorMaker::GenerateAlgorithm ColorMaker::algorithm() **const**
39. {
40. **return** m\_algorithm;
41. }
43. **void** ColorMaker::setAlgorithm(GenerateAlgorithm algorithm)
44. {
45. m\_algorithm = algorithm;
46. }
48. **void** ColorMaker::start()
49. {
50. **if**(m\_nColorTimer == 0)
51. {
52. m\_nColorTimer = startTimer(1000);
53. }
54. }
56. **void** ColorMaker::stop()
57. {
58. **if**(m\_nColorTimer > 0)
59. {
60. killTimer(m\_nColorTimer);
61. m\_nColorTimer = 0;
62. }
63. }
65. **void** ColorMaker::timerEvent(QTimerEvent \*e)
66. {
67. **if**(e->timerId() == m\_nColorTimer)
68. {
69. **switch**(m\_algorithm)
70. {
71. **case** RandomRGB:
72. m\_currentColor.setRgb(qrand() % 255, qrand() % 255, qrand() % 255);
73. **break**;
74. **case** RandomRed:
75. m\_currentColor.setRed(qrand() % 255);
76. **break**;
77. **case** RandomGreen:
78. m\_currentColor.setGreen(qrand() % 255);
79. **break**;
80. **case** RandomBlue:
81. m\_currentColor.setBlue(qrand() % 255);
82. **break**;
83. **case** LinearIncrease:
84. {
85. **int** r = m\_currentColor.red() + 10;
86. **int** g = m\_currentColor.green() + 10;
87. **int** b = m\_currentColor.blue() + 10;
88. m\_currentColor.setRgb(r % 255, g % 255, b % 255);
89. }
90. **break**;
91. }
92. emit colorChanged(m\_currentColor);
93. emit currentTime(QDateTime::currentDateTime().toString("yyyy-MM-dd hh:mm:ss"));
94. }
95. **else**
96. {
97. QObject::timerEvent(e);
98. }
99. }

    我使用一个周期为 1000 的定时器来产生颜色，定时器触发时根据算法来构造新的颜色值，发射 colorChanged 信号，同时也发送一个 currentTime 信号。

**注册一个 QML 中可用的类型**

    看过了怎样实现一个可供 QML 访问的类，这节我们看看怎样将一个 C++ 类型注册为 QML 类型以及怎样在 QML 中使用这个类型。

    要达到这种目的，大概可以分四步：

1. 实现 C++ 类
2. 注册 QML 类型
3. 在 QML 中导入类型
4. 在 QML 创建由 C++ 导出的类型的实例并使用

    ColorMaker 已经就绪了，现在看看怎样将其注册为 QML 可以使用的类型。

**注册 QML 类型**

    要注册一个 QML 类型，有多种方法可用，如 qmlRegisterSingletonType() 用来注册一个单例类型， qmlRegisterType() 注册一个非单例的类型， qmlRegisterTypeNotAvailable() 注册一个类型用来占位， qmlRegisterUncreatableType() 通常用来注册一个具有附加属性的附加类型，……好吧，我这里只说常规的类型注册，其它的，请您参考 Qt SDK 吧。

    qmlRegisterType() 是个模板函数，有两个原型：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **template**<**typename** T>
2. **int** qmlRegisterType(**const** **char** \*uri, **int** versionMajor, **int** versionMinor, **const** **char** \*qmlName);
4. **template**<**typename** T, **int** metaObjectRevision>
5. **int** qmlRegisterType(**const** **char** \*uri, **int** versionMajor, **int** versionMinor, **const** **char** \*qmlName);

    前一个原型一般用来注册一个新类型，而后一个可以为特定的版本注册类型。后面这个牵涉到 Qt Quick 的类型和版本机制，三言两语不能尽述，咱们单说前一个原型的使用。要使用 qmlRegisterType 需要包含 QtQml 头文件。

    先说模板参数 typename ，它就是你实现的 C++ 类的类名。

    qmlRegisterType() 的第一个参数 uri ，让你指定一个唯一的包名，类似 Java 中的那种，一是用来避免名字冲突，而是可以把多个相关类聚合到一个包中方便引用。比如我们常写这个语句 "import QtQuick.Controls 1.1" ，其中的 "QtQuick.Controls" 就是包名 uri ，而 1.1 则是版本，是 versionMajor 和 versionMinor 的组合。 qmlName 则是 QML 中可以使用的类名。

    下面是 colorMaker 示例的 main.cpp 文件：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. #include <QtGui/QGuiApplication>
2. #include "qtquick2applicationviewer.h"
3. #include <QtQml>
4. #include "colorMaker.h"

7. **int** main(**int** argc, **char** \*argv[])
8. {
9. QGuiApplication app(argc, argv);
10. qmlRegisterType<ColorMaker>("an.qt.ColorMaker", 1, 0, "ColorMaker");
12. QtQuick2ApplicationViewer viewer;
13. viewer.setMainQmlFile(QStringLiteral("qml/colorMaker/main.qml"));
14. viewer.showExpanded();
16. **return** app.exec();
17. }

    上面的代码将 ColorMaker 类注册为 QML 类 ColorMaker ，主版本为 1 ，次版本为 0 ，而我起的包名则是 an.qt.ColorMaker 。注册动作一定要放在 QML 上下文创建之前，否则的话，木有用滴。

**在 QML 中导入 C++ 注册的类型**

    一旦你在 C++ 中注册好了 QML 类型，就可以在 QML 文档中引入你注册的包，然后使用注册的类型。要引入包，使用 import 语句。比如要使用我们注册的 ColorMaker 类，可以在 QML 文档中加入下面的 import 语句：

**[javascript]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **import** an.qt.ColorMaker 1.0

**在 QML 中创建 C++ 导入类型的实例**

    引入包后，你就可以在 QML 中创建 C++ 导入类型的对象了，与 QML 内建类型的使用完全一样。如下是创建一个 ColorMaker 实例的代码：

**[javascript]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. Rectangle {
2. width: 360;
3. height: 360;
5. ColorMaker {
6. id: colorMaker;
7. color: Qt.green;
8. }
9. }

    如你所见，ColorMaker 的使用与 Retangle 没什么区别。如果你想在别处引用 ColorMaker 的实例，可以给实例指定一个唯一的 id ，就像上面的代码中那样。

**完整的 colorMaker 实例**

    如何定义一个可以导出到 QML 中的 C++ 类、如何注册 QML 类型、如何在 QML 中使用 C++ 导出的类型，都介绍完了，现在来看看完整的 colorMaker 。

    如果你对如何在 QML 中使用信号与槽还有疑问，请温习下《[Qt Quick 事件处理之信号与槽](http://blog.csdn.net/foruok/article/details/30028711)》再往下看。如果你对 QML 的基本语法忘了，请温习《[QML 语言基础](http://blog.csdn.net/foruok/article/details/28634543)》。如果你不记得 Qt Quick 的基本元素，看《[Qt Quick 简单教程](http://blog.csdn.net/foruok/article/details/28859415)》吧。

   C++ 代码前面都展示过了，这里看下完整的 main.qml 文档：

**[javascript]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **import** QtQuick 2.0
2. **import** QtQuick.Controls 1.1
3. **import** an.qt.ColorMaker 1.0
5. Rectangle {
6. width: 360;
7. height: 360;
8. Text {
9. id: timeLabel;
10. anchors.left: parent.left;
11. anchors.leftMargin: 4;
12. anchors.top: parent.top;
13. anchors.topMargin: 4;
14. font.pixelSize: 26;
15. }
16. ColorMaker {
17. id: colorMaker;
18. color: Qt.green;
19. }
21. Rectangle {
22. id: colorRect;
23. anchors.centerIn: parent;
24. width: 200;
25. height: 200;
26. color: "blue";
27. }
29. Button {
30. id: start;
31. text: "start";
32. anchors.left: parent.left;
33. anchors.leftMargin: 4;
34. anchors.bottom: parent.bottom;
35. anchors.bottomMargin: 4;
36. onClicked: {
37. colorMaker.start();
38. }
39. }
40. Button {
41. id: stop;
42. text: "stop";
43. anchors.left: start.right;
44. anchors.leftMargin: 4;
45. anchors.bottom: start.bottom;
46. onClicked: {
47. colorMaker.stop();
48. }
49. }
51. **function** changeAlgorithm(button, algorithm){
52. **switch**(algorithm)
53. {
54. **case** 0:
55. button.text = "RandomRGB";
56. **break**;
57. **case** 1:
58. button.text = "RandomRed";
59. **break**;
60. **case** 2:
61. button.text = "RandomGreen";
62. **break**;
63. **case** 3:
64. button.text = "RandomBlue";
65. **break**;
66. **case** 4:
67. button.text = "LinearIncrease";
68. **break**;
69. }
70. }
72. Button {
73. id: colorAlgorithm;
74. text: "RandomRGB";
75. anchors.left: stop.right;
76. anchors.leftMargin: 4;
77. anchors.bottom: start.bottom;
78. onClicked: {
79. **var** algorithm = (colorMaker.algorithm() + 1) % 5;
80. changeAlgorithm(colorAlgorithm, algorithm);
81. colorMaker.setAlgorithm(algorithm);
82. }
83. }
85. Button {
86. id: quit;
87. text: "quit";
88. anchors.left: colorAlgorithm.right;
89. anchors.leftMargin: 4;
90. anchors.bottom: start.bottom;
91. onClicked: {
92. Qt.quit();
93. }
94. }
96. Component.onCompleted: {
97. colorMaker.color = Qt.rgba(0,180,120, 255);
98. colorMaker.setAlgorithm(ColorMaker.LinearIncrease);
99. changeAlgorithm(colorAlgorithm, colorMaker.algorithm());
100. }
102. Connections {
103. target: colorMaker;
104. onCurrentTime:{
105. timeLabel.text = strTime;
106. timeLabel.color = colorMaker.timeColor;
107. }
108. }
110. Connections {
111. target: colorMaker;
112. onColorChanged:{
113. colorRect.color = color;
114. }
115. }
116. }

    main.qml 的界面分成了三部分，参看图 1 。顶部是一个 Text ，用来显示由 ColorMaker 提供的时间，我使用 Connections 对象，指定 target 为 colorMaker ，在 onCurrentTime 信号处理器中改变 timeLabel 的文本和颜色。这里使用 ColorMaker 的 timeColor 属性，该属性的读取函数是 timeColor ，回看一下 colorMaker.cpp 中的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. QColor ColorMaker::timeColor() **const**
2. {
3. QTime time = QTime::currentTime();
4. **int** r = time.hour();
5. **int** g = time.minute()\*2;
6. **int** b = time.second()\*4;
7. **return** QColor::fromRgb(r, g, b);
8. }

    timeColor() 函数获取当前时间，取时、分、秒转换为 R 、 G 、 B 值，构造一个 QColor 对象。

    我构造了ColorMaker 类的一个实例， id 为 colorMaker ，初始化颜色值为 green 。

    colorMaker 实例界面的中间是一个 Rectangle 对象，id 是 colorRect 。我使用 Connections 对象，指定 target 为 colorMaker ，在 onColorChanged 信号处理器中改变 colorRect 的颜色。

    界面的底部就是几个按钮，使用锚布局把它们排成一行。 start 按钮的 onClicked 信号处理器调用 colorMaker 的 start() 槽，启动颜色生成器。 stop 按钮的 onClicked 信号处理器调用 colorMaker 的 stop() 槽，停止颜色生成器。而 colorAlgorithm 按钮则每点击一次就切换一个颜色生成算法，同时调用 changeAlgorithm() 函数，根据算法改变按钮上的文字。 quit 按钮一点就退出应用。

    main.qml 还引入了一个新内容：定义函数。这个可以参考 JavaScript 的教程。我们定义的 changeAlgorithm 函数，接受两个参数， button 和 algorithm 。如果你是 C++ 程序猿，可能有点儿不适应：怎么参数就木有类型呢哈…… JavaScript 就是酱紫滴，拥有动态类型，一个变量在赋值时决定其类型。

    这就是 colorMaker 的全部了。

    好啦，现在再来看看怎样导出一个对象到 QML 中。

**导出一个 C++ 对象为 QML 的属性**

    上面看了怎样导出一个 QML 类型在 QML 文档中使用，你还可以把 C++ 中创建的对象作为属性传递到 QML 环境中，然后在 QML 环境中访问。我们还是以 colorMaker 为例，对其代码做适当修改来适应本节的内容。

**注册属性**

    要将一个对象注册为属性很简单，colorMaker 的 main.cpp 修改后如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. #include <QtGui/QGuiApplication>
2. #include "qtquick2applicationviewer.h"
3. #include <QtQml>
4. #include "colorMaker.h"

7. **int** main(**int** argc, **char** \*argv[])
8. {
9. QGuiApplication app(argc, argv);
11. QtQuick2ApplicationViewer viewer;
13. **viewer.rootContext()->setContextProperty("colorMaker", new ColorMaker);**
15. viewer.setMainQmlFile(QStringLiteral("qml/colorMaker/main.qml"));
16. viewer.showExpanded();
18. **return** app.exec();
19. }

    注意我在 viewer 变量后增加的新语句：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. viewer.rootContext()->setContextProperty("colorMaker", **new** ColorMaker);

  **正式这行代码从堆上分配了一个 ColorMaker 对象，然后注册为 QML 上下文的属性，起了个名字就叫 colorMaker 。**

**viewer.rootContext() 返回的是 QQmlContext 对象。 QQmlContext 类代表一个 QML 上下文，它的 setContextProperty() 方法可以为该上下文设置一个全局可见的属性。要注意的是，你 new 出来的对象， QQmlContext 只是使用，不会帮你删除，你需要自己找一个合适的时机来删除它。**

    还有一点要说明，因为我们去掉了 qmlRegisterType() 调用，所以在 main.qml 中不能再访问 ColorMaker 类了，比如你不能通过类名来引用它定义的 GenerateAlgorithm 枚举类型， colorMaker.setAlgorithm(ColorMaker.LinearIncrease) 语句会导致下面的报错：

**[javascript]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. ReferenceError: ColorMaker is not defined

    现在来看如何在 QML 中使用我们导出的属性

**在 QML 中使用关联到 C++ 对象的属性**

    一旦调用 setContextProperty() 导出了属性，就可以在 QML 中使用了，不需要 import 语句哦。下面是 main.qml 修改后的代码：

**[javascript]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **import** QtQuick 2.0
2. **import** QtQuick.Controls 1.1
3. //[1]
4. //import an.qt.ColorMaker 1.0
6. Rectangle {
7. width: 360;
8. height: 360;
9. Text {
10. id: timeLabel;
11. anchors.left: parent.left;
12. anchors.leftMargin: 4;
13. anchors.top: parent.top;
14. anchors.topMargin: 4;
15. font.pixelSize: 26;
16. }
17. /\* [2]
18. ColorMaker {
19. id: colorMaker;
20. color: Qt.green;
21. }
22. \*/
24. Rectangle {
25. id: colorRect;
26. anchors.centerIn: parent;
27. width: 200;
28. height: 200;
29. color: "blue";
30. }
32. Button {
33. id: start;
34. text: "start";
35. anchors.left: parent.left;
36. anchors.leftMargin: 4;
37. anchors.bottom: parent.bottom;
38. anchors.bottomMargin: 4;
39. onClicked: {
40. colorMaker.start();
41. }
42. }
43. Button {
44. id: stop;
45. text: "stop";
46. anchors.left: start.right;
47. anchors.leftMargin: 4;
48. anchors.bottom: start.bottom;
49. onClicked: {
50. colorMaker.stop();
51. }
52. }
54. **function** changeAlgorithm(button, algorithm){
55. **switch**(algorithm)
56. {
57. **case** 0:
58. button.text = "RandomRGB";
59. **break**;
60. **case** 1:
61. button.text = "RandomRed";
62. **break**;
63. **case** 2:
64. button.text = "RandomGreen";
65. **break**;
66. **case** 3:
67. button.text = "RandomBlue";
68. **break**;
69. **case** 4:
70. button.text = "LinearIncrease";
71. **break**;
72. }
73. }
75. Button {
76. id: colorAlgorithm;
77. text: "RandomRGB";
78. anchors.left: stop.right;
79. anchors.leftMargin: 4;
80. anchors.bottom: start.bottom;
81. onClicked: {
82. **var** algorithm = (colorMaker.algorithm() + 1) % 5;
83. changeAlgorithm(colorAlgorithm, algorithm);
84. colorMaker.setAlgorithm(algorithm);
85. }
86. }
88. Button {
89. id: quit;
90. text: "quit";
91. anchors.left: colorAlgorithm.right;
92. anchors.leftMargin: 4;
93. anchors.bottom: start.bottom;
94. onClicked: {
95. Qt.quit();
96. }
97. }
99. Component.onCompleted: {
100. colorMaker.color = Qt.rgba(0,180,120, 255);
101. //[3]
102. //colorMaker.setAlgorithm(ColorMaker.LinearIncrease);
103. colorMaker.setAlgorithm(2);
104. changeAlgorithm(colorAlgorithm, colorMaker.algorithm());
105. }
107. Connections {
108. target: colorMaker;
109. onCurrentTime:{
110. timeLabel.text = strTime;
111. timeLabel.color = colorMaker.timeColor;
112. }
113. }
115. Connections {
116. target: colorMaker;
117. onColorChanged:{
118. colorRect.color = color;
119. }
120. }
121. }

    main.qml 代码主要修改了三处，我已经使用方括号标注出来了。因为我将导出的属性命名为 colorMaker ，和导出 ColorMaker 类时构造的实例的 id 一样，所以改动少了些。

    你看到了，导出的属性可以直接使用，与属性关联的对象，它的信号、槽、可调用方法（使用 Q\_INVOKABLE 宏修饰的方法）、属性都可以使用，只是不能通过类名来引用枚举值了。

**在 C++ 中使用 QML 对象**

    看过了如何在 QML 中使用 C++ 类型或对象，现在来看如何在 C++ 中使用 QML 对象。

    我们可以使用 QML 对象的信号、槽，访问它们的属性，都没有问题，因为很多 QML 对象对应的类型，原本就是 C++ 类型，比如 Image 对应 QQuickImage ， Text 对应 QQuickText……但是，这些与 QML 类型对应的 C++ 类型都是私有的，你写的 C++ 代码也不能直接访问。肿么办？

    Qt 最核心的一个基础特性，就是元对象系统，通过元对象系统，你可以查询 QObject 的某个派生类的类名、有哪些信号、槽、属性、可调用方法等等信息，然后也可以使用 QMetaObject::invokeMethod() 调用 QObject 的某个注册到元对象系统中的方法。而对于使用 Q\_PROPERTY 定义的属性，可以使用 QObject 的 property() 方法访问属性，如果该属性定义了 WRITE 方法，还可以使用 setProperty() 修改属性。所以只要我们找到 QML 环境中的某个对象，就可以通过元对象系统来访问它的属性、信号、槽等。

**查找一个对象的孩子**

    QObject 类的构造函数有一个 parent 参数，可以指定一个对象的父亲， QML 中的对象其实借助这个组成了以根 item 为父的一棵对象树。

    而 QObject 定义了一个属性 objectName ，这个对象名字属性，就可以用于查找对象。现在该说到查找对象的方法了： findChild() 和 findChildren() 。它们的函数原型如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. T QObject::findChild(**const** QString & name = QString(),\
2. Qt::FindChildOptions options = \
3. Qt::FindChildrenRecursively) **const**;
4. QList<T> QObject::findChildren(**const** QString & name = \
5. QString(), Qt::FindChildOptions options = \
6. Qt::FindChildrenRecursively) **const**;
7. QList<T> QObject::findChildren(**const** QRegExp & regExp, \
8. Qt::FindChildOptions options = \
9. Qt::FindChildrenRecursively) **const**;
10. QList<T> QObject::findChildren(**const** QRegularExpression & re,\
11. Qt::FindChildOptions options = \
12. Qt::FindChildrenRecursively) **const**;

    都是模板方法，从命名上也可以看出，一个返回单个对象，一个返回对象列表。闲话少说，现在让我们看看如何查询一个或多个对象，我们先以 Qt Widgets 为例来说明用法哈。

    示例 1 ：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. QPushButton \*button = parentWidget->findChild<QPushButton \*>("button1");

    查找 parentWidget 的名为 "button1" 的类型为 QPushButton 的孩子。

    示例 2 ：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. QList<QWidget \*> widgets = parentWidget.findChildren<QWidget \*>("widgetname");

    返回 parentWidget 所有名为 "widgetname" 的 QWidget 类型的孩子列表。

**使用元对象调用一个对象的方法**

    QMetaObject 的 invokeMethod() 方法用来调用一个对象的信号、槽、可调用方法。它是个静态方法，其函数原型如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **bool** QMetaObject::invokeMethod(QObject \* obj, /
2. **const** **char** \* member,/
3. Qt::ConnectionType type, /
4. QGenericReturnArgument ret, /
5. QGenericArgument val0 = QGenericArgument( 0 ), /
6. QGenericArgument val1 = QGenericArgument(), /
7. QGenericArgument val2 = QGenericArgument(), /
8. QGenericArgument val3 = QGenericArgument(), /
9. QGenericArgument val4 = QGenericArgument(), /
10. QGenericArgument val5 = QGenericArgument(), /
11. QGenericArgument val6 = QGenericArgument(), /
12. QGenericArgument val7 = QGenericArgument(), /
13. QGenericArgument val8 = QGenericArgument(), /
14. QGenericArgument val9 = QGenericArgument()) [**static**]

    其实 QMetaObject 还有三个 invokeMethod() 函数，不过都是上面这个原型的重载，所以我们只要介绍上面这个就 OK 了。

    先说返回值吧，返回 true 说明调用成功。返回 false ，要么是因为没有你说的那个方法，要么是参数类型不匹配。

    第一个参数是被调用对象的指针。

    第二个参数是方法名字。

    第三个参数是连接类型，看到这里你就知道， invokeMethod 为信号与槽而生，你可以指定连接类型，如果你要调用的对象和发起调用的线程是同一个线程，那么可以使用 Qt::DirectConnection 或 Qt::AutoConnection 或 Qt::QueuedConnection ，如果被调用对象在另一个线程，那么建议你使用 Qt::QueuedConnection 。

    第四个参数用来接收返回指。

    然后就是多达 10 个可以传递给被调用方法的参数。嗯，看来信号与槽的参数个数是有限制的，不能超过 10 个。

    对于要传递给被调用方法的参数，使用 QGenericArgument 来表示，你可以使用 Q\_ARG 宏来构造一个参数，它的定义是：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. QGenericArgument Q\_ARG( Type, **const** Type & value)

    返回类型是类似的，使用 QGenericReturnArgument 表示，你可以使用 Q\_RETURN\_ARG 宏来构造一个接收返回指的参数，它的定义是：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. QGenericReturnArgument Q\_RETURN\_ARG( Type, Type & value)

    好啦，总算把这个天杀的函数介绍完了，下面我们看看怎么用。

    假设一个对象有这么一个槽 compute(QString, int, double) ，返回一个 QString 对象，那么你可以这么调用（同步方式）：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. QString retVal;
2. QMetaObject::invokeMethod(obj, "compute", Qt::DirectConnection,
3. Q\_RETURN\_ARG(QString, retVal),
4. Q\_ARG(QString, "sqrt"),
5. Q\_ARG(**int**, 42),
6. Q\_ARG(**double**, 9.7));

    如果你要让一个线程对象退出，可以这么调用（队列连接方式）：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. QMetaObject::invokeMethod(**thread**, "quit",
2. Qt::QueuedConnection);

    更多细节请参考 Qt 帮助 中的 QMetaObject 手册。

**callQml 示例**

    现在让我们创建一个新的项目，名字是 callQml ，添加 changeColor.h 、 changeColor.cpp 两个文件。 main.qml 内容如下：

**[javascript]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **import** QtQuick 2.0
2. **import** QtQuick.Controls 1.1
4. Rectangle {
5. objectName: "rootRect";
6. width: 360;
7. height: 360;
8. Text {
9. objectName: "textLabel";
10. text: "Hello World";
11. anchors.centerIn: parent;
12. font.pixelSize: 26;
13. }
15. Button {
16. anchors.right: parent.right;
17. anchors.rightMargin: 4;
18. anchors.bottom: parent.bottom;
19. anchors.bottomMargin: 4;
20. text: "quit";
21. objectName: "quitButton";
22. }
23. }

    我们给根元素起了个名字 "rootRect" ，给退出按钮起了个名字 "quitButton" ，给文本起了名字 "textLabel" ，我们会在 C++ 代码中通过这些个名字来查找对应的对象并改变它们。

    现在来看看 main.cpp ：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. #include <QtGui/QGuiApplication>
2. #include "qtquick2applicationviewer.h"
3. #include <QQuickItem>
4. #include "changeColor.h"
5. #include <QMetaObject>
6. #include <QDebug>
7. #include <QColor>
8. #include <QVariant>
10. **int** main(**int** argc, **char** \*argv[])
11. {
12. QGuiApplication app(argc, argv);
14. QtQuick2ApplicationViewer viewer;
15. viewer.setMainQmlFile(QStringLiteral("qml/callQml/main.qml"));
16. viewer.showExpanded();
18. QQuickItem \* rootItem = viewer.rootObject();
19. **new** ChangeQmlColor(rootItem);
20. QObject \* quitButton = rootItem->findChild<QObject\*>("quitButton");
21. **if**(quitButton)
22. {
23. QObject::connect(quitButton, SIGNAL(clicked()), &app, SLOT(quit()));
24. }
26. QObject \*textLabel = rootItem->findChild<QObject\*>("textLabel");
27. **if**(textLabel)
28. {
29. //1. failed call
30. **bool** bRet = QMetaObject::invokeMethod(textLabel, "setText", Q\_ARG(QString, "world hello"));
31. qDebug() << "call setText return - " << bRet;
32. textLabel->setProperty("color", QColor::fromRgb(255,0,0));
33. bRet = QMetaObject::invokeMethod(textLabel, "doLayout");
34. qDebug() << "call doLayout return - " << bRet;
35. }
37. **return** app.exec();
38. }

    在一开始我通过 viewer.rootObject() ，获取到了作为根元素的 Rectangle ，然后把它交给一个 ChangeQmlColor 对象，该对象会内部通过一个定时器，一秒改变一次传入对象的颜色。

    紧接着，我使用 QObject 的 findChild() 找到了 quitButton 按钮，把它的 clicked() 信号连接到 QGuiApplication 的 quit() 槽上。所以你点击这个按钮，应用就退出了。

    后来，我又通过名字 "textLabel" 找到了 textLabel 对象。首先我企图使用 invodeMethod() 调用 setText() 方法来改变 textLabel 的文本，这个注定是会失败的，因为 QML 中的Text 对象对应 C++ QQuickText 类，而 QQuickText 没有名为 setText 的槽或者可调用方法。我查看了头文件 qquicktext\_p.h ，发现它有一个使用 Q\_INVOKABLE 宏修饰的 doLayout() 的方法，所以后来我又调用 doLayout() ，这次成功了。

    图 2 是运行效果：

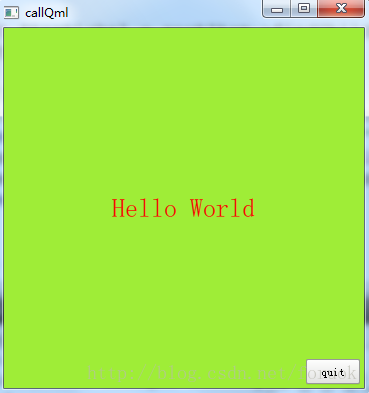


            图 2 callQml 运行效果图

    Hello World 这行字变成了红色，是因为我在 main() 函数中使用 setProperty 修改了 textLabel 的 color 属性。

    下面是 Qt Creator 应用程序输出窗口的信息，可以验证对 Text 方法的调用是否成功：

**[plain]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. Starting D:\projects\...\release\callQml.exe...
2. QMetaObject::invokeMethod: No such method QQuickText::setText(QString)
3. call setText return -  false
4. call doLayout return -  true

    好啦，最后看看界面背景为么变成了浅绿色。这正是下面这行代码的功劳：

**[csharp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. **new** ChangeQmlColor(rootItem);

    它以 rootItem 为参数创建了一个 ChangeQmlColor 对象，而 ChangeQmlColor 类会改变传给它的对象的颜色。

    ChangeQmlColor 类定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. #ifndef CHANGECOLOR\_H
2. #define CHANGECOLOR\_H
3. #include <QObject>
4. #include <QTimer>
6. **class** ChangeQmlColor : **public** QObject
7. {
8. Q\_OBJECT
9. **public**:
10. ChangeQmlColor(QObject \*target, QObject \*parent = 0);
11. ~ChangeQmlColor();
13. **protected** slots:
14. **void** onTimeout();
16. **private**:
17. QTimer m\_timer;
18. QObject \*m\_target;
19. };
21. #endif

    实现文件 changeColor.cpp ：

**[cpp]** [view plaincopy](http://blog.csdn.net/foruok/article/details/32698603)[在CODE上查看代码片](https://code.csdn.net/snippets/416486)

1. #include "changeColor.h"
2. #include <QDateTime>
3. #include <QColor>
4. #include <QVariant>
6. ChangeQmlColor::ChangeQmlColor(QObject \*target, QObject \*parent)
7. : QObject(parent)
8. , m\_timer(**this**)
9. , m\_target(target)
10. {
11. qsrand(QDateTime::currentDateTime().toTime\_t());
12. connect(&m\_timer, SIGNAL(timeout()), **this**, SLOT(onTimeout()));
13. m\_timer.start(1000);
14. }
16. ChangeQmlColor::~ChangeQmlColor()
17. {}
19. **void** ChangeQmlColor::onTimeout()
20. {
21. QColor color = QColor::fromRgb(qrand()%256, qrand()%256, qrand()%256);
22. m\_target->setProperty("color", color);
23. }

    很简单，不说啦。

    版权所有 foruok ，转载请注明出处：<http://blog.csdn.net/foruok> 。