Qt QML 与 C++混合编程

目　录

正　文

1．概述

Qt 是一个跨平台的应用程序和 UI 框架，支持 **C++**和一种类似 CSS、JavaScript 的 **QML** 语言，Qt Creator 便是供 Qt 开发者使用的 IDE。Qt Quick 是 Qt 中一项现代化的 UI 技术，使用它可以把描述性的 UI 设计和命令式的业务逻辑分开，应用程序的表示层可以不再使用传统的 C++ API，取而代之的是 Qt 特有的描述性语言——QML。

这样，一个 Qt Quick 应用程序就可以用 QML 和 C++进行混合编程：QML 高效构建 UI，例如动画、状态机、三维图形、OpenGL 着色器、图形效果、粒子、多点触控手势等，而 C++实现核心业务逻辑和复杂算法。

本文主要介绍 Qt QML 与 C++混合编程，即两者间交互的方法与技巧。

2．QML 访问 C++

首先，新建一个 Qt Quick Application，工程名字为 **RGBGame**，然后添加 C++文件，RGBGame.h 和 RGBGame.cpp。RGBGame 是一个示例工程，在 C++中实现了 RGBGame 类然后通过 QML 来访问。RGBGame 示例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，在界面底部有几个按钮来控制颜色变化。

由于 QML 引擎与 QML 元对象系统的紧密结合，QML 很容易从 C++中得到扩展，在一定的条件下，任何 QML 代码都可以访问 QObject 派生类的成员，例如：槽函数、信号、枚举类型、成员函数、属性等。

QML 访问 C++有两个方法：一是在 QML 元对象系统中注册 C++类，然后在 QML 代码中实例化、访问。二是在 C++代码中实例化，把这个类对象设置为 QML 的上下文属性，然后在 QML 代码中直接使用。与后者相比，前者可以使 C++类在 QML 中作为一个数据类型，例如函数参数或属性，也可以使用其枚举类型、单例等，功能更强大。

2.1 如何实现可以被 QML 访问的 C++类

C++类，以 RGBGame 类为例，将其导出到 QML 中，首先必须满足两个条件：一是派生自 QObject 类或 QObject 类的子类，二是使用 Q_OBJECT 宏。在定义一个 C++类时，Q_OBJECT 宏必须在 private 区声明，位置一般在语句块首行（C++默认为 private），用来声明信号和槽，进入 Qt 元对象系统，以便 QML 访问。

2.1.1 C++类的槽与信号

RGBGame.h 初始声明：

```
class RGBGame : public QObject
{
    Q_OBJECT

public:
    RGBGame(QObject *parent = 0);
    ~RGBGame();

public slots:
    void start();
    void stop();

signals:
    void colorChanged(const QColor &color);
    void currentTime(const QString &strTime);
};
```

RGBGame 类中的槽 start()、stop()，信号 colorChanged()、currentTime()都可以在 QML 中直接调用或者与 QML 中的信号、函数连接，例子参照 main.qml。

槽必须声明为 public。

QML 引擎会为每一个信号自动创建一个可以在 QML 中使用的信号处理器 on<Signal>，Signal 首字母大写，信号中的所有参数在信号处理器中都是可用的。值得注意的是，如果信号的参数类型未注册到 Qt 元对象系统，不被 QML 引擎支持，但不会出错，只是这个参数不能通过信号处理器来访问。

2.1.2 C++类的枚举类型

RGBGame.h 添加枚举类型：

```
class RGBGame : public QObject
{
    Q_OBJECT
    Q_ENUMS(GenerateAlgorithm)

public:
    RGBGame(QObject *parent = 0);
    ~RGBGame();

    enum GenerateAlgorithm {
        RandomRGB,
        RandomRed,
        RandomGreen,
        RandomBlue,
        LinearIncrease
    };

public slots:
    void start();
    void stop();

signals:
    void colorChanged(const QColor &color);
    void currentTime(const QString &strTime);
};
```

RGBGame 类定义了 GenerateAlgorithm 枚举类型，在 QML 中可用${CLASS_NAME}.${ENUM_VALUE}来访问，例如 RGBGame.RGBRed，例子参照 main.qml。

在 QML 中使用 C++类中自定义的枚举类型时，可用 Q_ENUMS()宏将该枚举类型注册到 Qt 元对象系统中。

2.1.3 C++类的成员函数

RGBGame.h 添加成员函数：

```
class RGBGame : public QObject
{
    Q_OBJECT
    Q_ENUMS(GenerateAlgorithm)

public:
    RGBGame(QObject *parent = 0);
    ~RGBGame();

    enum GenerateAlgorithm {
        RandomRGB,
        RandomRed,
        RandomGreen,
        RandomBlue,
        LinearIncrease
    };

    Q_INVOKABLE GenerateAlgorithm algorithm() const;
    Q_INVOKABLE void setAlgorithm(GenerateAlgorithm algorithm);

public slots:
    void start();
    void stop();

signals:
    void colorChanged(const QColor &color);
    void currentTime(const QString &strTime);
};
```

RGBGame 类的成员函数 algorithm()、setAlgorithm()，在 QML 代码中使用时，可以使用${Object}.${method}来访问，例子参照 main.qml。

类 public 成员函数用 Q_INVOKABLE 宏标识时，就可以通过 Qt 元对象系统来调用，但这个宏必须放在返回类型前面。

函数参数：如果函数参数的类型是 QObject*，在 QML 代码中使用时，函数参数可以通过对象 id 来传递，也可以通过指向对象的 JavaScript var 类型的值来传递。

函数重载：QML 支持 C++函数重载，通过函数参数列表的不同进行匹配，但不识别多个同名不同参数的信号，这种情况下，只有最后一个信号可以通过 QML 来访问。

函数返回值：通过 QML 中的 JavaScrpt 表达式访问 C++函数时，函数返回值会自动转换为 JavaScrpt 类型的值。

例子：添加 MyParameter.h、MyParameter.cpp、QMLParameter.qml 进行测试。MyParameter.h 列出了 QML 支持的几种情况：

```
class Message : public QObject
{
    Q_OBJECT

public:
    Message();
    ~Message();

    Q_INVOKABLE void refresh() const;

    Q_INVOKABLE QString stringMessage() const;
    Q_INVOKABLE void setStringMessage(QString msg);

    Q_INVOKABLE void setStringReferenceMessage(const QString &msg);

    Q_INVOKABLE Message* objectPointerMessage() const;
    Q_INVOKABLE void setObjectPointerMessage(Message *msg);
private:
    QString m_message;
    QString *m_pMessage;
};
```

QMLParameter.qml 介绍了 QML 支持函数参数类型为自定义类 Message*的方法，可以通过对象 id 来传递，例子中是 message，也可以通过指向对象的 JavaScript var 类型的值来传递，例子中是 qmlMessage：

```
Message {
    id: message
}

var qmlMessae = message.objectPointerMessage()
message.setObjectPointerMessage(qmlMessae)

message.setObjectPointerMessage(message)
```

2.1.4 C++类的属性

RGBGame.h 添加属性：

```
class RGBGame : public QObject
{
    Q_OBJECT
    Q_ENUMS(GenerateAlgorithm)
    Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
    Q_PROPERTY(QColor timeColor READ timeColor)

public:
    RGBGame(QObject *parent = 0);
    ~RGBGame();

    enum GenerateAlgorithm {
        RandomRGB,
        RandomRed,
        RandomGreen,
        RandomBlue,
        LinearIncrease
    };

    QColor color() const;
    void setColor(const QColor &color);
    QColor timeColor() const;

    Q_INVOKABLE GenerateAlgorithm algorithm() const;
    Q_INVOKABLE void setAlgorithm(GenerateAlgorithm algorithm);

public slots:
    void start();
    void stop();

signals:
    void colorChanged(const QColor &color);
    void currentTime(const QString &strTime);

protected:
    void timerEvent(QTimerEvent *e);

private:
    GenerateAlgorithm m_algorithm;
    QColor m_currentColor;
    int m_nColorTimer;
};
```

RGBGame 类中的 color 属性可以在 QML 代码中进行访问、修改，color 属性值改变时还可发送一个信号来自动更新 QML 中与其绑定的属性值，timeColor 属性也可以在 QML 代码中进行访问，但不能修改，也没有信号，例子参照 main.qml。

Q_PROPERTY()宏用来在 QObject 派生类中声明属性，这个属性如同类的数据成员一样，但它又有一些额外的特性可通过 Qt 元对象系统来访问。

下面是 Q_PROPERTY()宏的原型：

```
Q_PROPERTY()(type name
            (READ getFunction [WRITE setFunction] |
             MEMBER memberName [(READ getFunction | WRITE setFunction)])
            [RESET resetFunction]
            [NOTIFY notifySignal]
            [REVISION int]
            [DESIGNABLE bool]
            [SCRIPTABLE bool]
            [STORED bool]
            [USER bool]
            [CONSTANT]
            [FINAL])
```

属性的 **type**、**name** 是必需的，其它是可选项，常用的有 READ、WRITE、NOTIFY。属性的 type 可以是 QVariant 支持的任何类型，也可以是自定义类型，包括类类型、列表类型、组属性等。另外，属性的 READ、WRITE、RESET 是可以被继承的，也可以是虚函数，这些特性并不常用。

READ：读取属性值，如果没有设置 MEMBER 的话，它是必需的。一般情况下，函数是个 const 函数，返回值类型必须是属性本身的类型或这个类型的 const 引用，没有参数。

WRITE：设置属性值，可选项。函数必须返回 void，有且仅有一个参数，参数类型必须是属性本身的类型或这个类型的指针或引用。

NOTIFY：与属性关联的可选信号。这个信号必须在类中声明过，当属性值改变时，就可触发这个信号，可以没有参数，有参数的话只能是一个类型同属性本身类型的参数，用来记录属性改变后的值。

RGBGame 类定义部分已经完成，有 public 槽、信号、枚举类型、Q_INVOKABLE 宏标识的成员函数、属性等，这些都可以在 QML 代码中访问。

RGBGame 类实现部分如下 RGBGame.cpp 所示：

```
#include <QTimerEvent>
#include <QDateTime>
#include "RGBGame.h"

RGBGame::RGBGame(QObject *parent)
    : QObject(parent)
    , m_algorithm(RandomRGB)
    , m_currentColor(Qt::black)
    , m_nColorTimer(0)
{
    qsrand(QDateTime::currentDateTime().toTime_t());
}

RGBGame::~RGBGame()
{

}


QColor RGBGame::color() const
{
    return m_currentColor;
}

void RGBGame::setColor(const QColor &color)
{
    m_currentColor = color;
    emit colorChanged(m_currentColor);
}

QColor RGBGame::timeColor() const
{
    QTime time = QTime::currentTime();
    int r = time.hour();
    int g = time.minute()*2;
    int b = time.second()*4;
    return QColor::fromRgb(r, g, b);
}


RGBGame::GenerateAlgorithm RGBGame::algorithm() const
{
    return m_algorithm;
}

void RGBGame::setAlgorithm(GenerateAlgorithm algorithm)
{
    m_algorithm = algorithm;
}
```

```
void RGBGame::start()
{
    if(m_nColorTimer == 0)
    {
        m_nColorTimer = startTimer(1000);
    }
}

void RGBGame::stop()
{
    if(m_nColorTimer > 0)
    {
        killTimer(m_nColorTimer);
        m_nColorTimer = 0;
    }
}
```

```
void RGBGame::timerEvent(QTimerEvent *e)
{
    if(e->timerId() == m_nColorTimer)
    {
        switch(m_algorithm)
        {
        case RandomRGB:
            m_currentColor.setRgb(qrand() % 255, qrand() % 255, qrand() % 255);
            break;
        case RandomRed:
            m_currentColor.setRed(qrand() % 255);
            break;
        case RandomGreen:
            m_currentColor.setGreen(qrand() % 255);
            break;
        case RandomBlue:
            m_currentColor.setBlue(qrand() % 255);
            break;
        case LinearIncrease:
            {
                int r = m_currentColor.red() + 10;
                int g = m_currentColor.green() + 10;
                int b = m_currentColor.blue() + 10;
                m_currentColor.setRgb(r % 255, g % 255, b % 255);
            }
            break;
        }
        emit colorChanged(m_currentColor);
        emit currentTime(QDateTime::currentDateTime().toString("yyyy-MM-dd hh:mm:ss"));
    }
    else
    {
        QObject::timerEvent(e);
    }
}
```

2.2 QML 类型注册

QObject 派生类可以注册到 Qt 元对象系统，使得该类在 QML 中同其它内建对象一样，可以作为一个数据类型来使用。QML 引擎允许注册可实例化的类型，也可以是不可实例化的类型。

常见的注册函数有：

qmlRegisterInterface()
qmlRegisterRevision()
qmlRegisterSingletonType()
qmlRegisterType()
qmlRegisterTypeNotAvailable()
qmlRegisterUncreatableType()

这些注册函数各有其用，可根据需要选择，使用时需要添加#include <QtQml>。常用的为 qmlRegisterType()，它有三个重载函数，这里只介绍其一：

```
template<typename T>
int qmlRegisterType(const char *uri,
                    int versionMajor,
                    int versionMinor,
                    const char *qmlName);
```

这个模板函数注册 C++类到 Qt 元对象系统中，uri 是需要导入到 QML 中的库名，versionMajor 和 versionMinor 是其版本数字，qmlName 是在 QML 中可以使用的类名。

2.2.1 在 C++中注册 QML 类型

使用 qmlRegisterType()将 RGBGame 类注册到 Qt 元对象系统中。

main.cpp:

```
#include <QGuiApplication>
#include <QQuickView>
#include <QtQml>
#include "RGBGame.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    qmlRegisterType<RGBGame>("suntec.tools.RGBGame", 1, 0, "RGBGame");

    QQuickView view;;
    view.setSource(QUrl(QStringLiteral("qrc:///main.qml")));
    view.show();

    return app.exec();
}
```

main.cpp 将 RGBGame 类注册为在 QML 中可以使用的 RGBGame 类型，主版本为 1，次版本为 0，库的名字是 suntec.tools.RGBGame。

注册动作必须在 QML 上下文创建之前，否则无效。

另外：QQuickView 为 Qt Quick UI 提供了一个窗口，可以方便地加载 QML 文件并显示其界面。QApplication 派生自 QGuiApplication，而 QGuiApplication 又派生自 QCoreApplication，这三个类是常见的管理 Qt 应用程序的类。QQmlApplicationEngine 可以方便地从一个单一的 QML 文件中加载应用程序，它派生自 QQmlEngine，QQmlEngine 则提供了加载 QML 组件的环境，可以与 QQmlComponent、QQmlContext 等一起使用。

单例注册：注册单例要用 qmlRegisterSingletonType()。有三个重载函数，单例类型可以是 **QObject** 或 **QJSValue**，也可以在 QML 文件中添加”**pragma Singleton**”，以这个 QML 文件的路径注册一个单例。注册成功后，我们不必亲自去构造一个实例，却可以直接使用一个类对象。这里介绍一个单例类型为 QObject 的例子。

首先，自定义一个派生自 QObject 的 MySingelton 类，参照 MySingleton.h：

```
class MySingleton : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int age READ age WRITE setAge NOTIFY ageChanged)

public:
    MySingleton(QObject *parent = 0)
        : QObject(parent), m_age(0) {}

    ~MySingleton() {}

    Q_INVOKABLE int changeAge() { setAge(100); return m_age; }
```

```
    int age() const { return m_age; }
    void setAge(int newAge) { m_age = newAge; emit ageChanged(newAge); }

signals:
    void ageChanged(int newAge);

private:
    int m_age;
};
```

然后，定义一个提供单例的回调函数 singletonProvider()：

```
static QObject* singletonProvider(QQmlEngine *engine, QJSEngine *scriptEngine)
{
    Q_UNUSED(engine)
    Q_UNUSED(scriptEngine)

    MySingleton *mySingleton = new MySingleton();
    return mySingleton;
}
```

最后，注册单例：

```
  qmlRegisterSingletonType<MySingleton>("suntec.tools.MySingleton",
                                        1, 0, "MySingleton", singletonProvider);
```

这样，我们就可以在 MySingleton.qml 中使用了：

```
import QtQuick 2.3
import QtQuick.Window 2.0
import suntec.tools.MySingleton 1.0

Window {
    visible: true
    width: 360; height: 360
    title: "Using MySingleton"
    color: "lightblue"

    Item {
        id: item
        width: parent.width; height: parent.height
        property int qmlAge: MySingleton.age

        MouseArea{
            anchors.fill: parent
            onPressed: console.log(item.qmlAge)
            onReleased: {
                item.qmlAge = MySingleton.changeAge()
                console.log(item.qmlAge)
            }
        }
    }
}
```

2.2.2 在 QML 中访问 C++类对象

RGBGame 类已经在 C++中注册为 QML 类型，在 QML 文件中使用 import 语句导入后，就可以使用它了，例子参照 main.qml：

```
import QtQuick 2.3
import QtQuick.Controls 1.2
import suntec.tools.RGBGame 1.0

Item {
    width: 360; height: 360;

    Text {
        id: timeLabel
        anchors.left: parent.left
        anchors.leftMargin: 10
        anchors.top: parent.top
        anchors.topMargin: 10
        font.pixelSize: 26
    }


    RGBGame {
        id: rgbGame;
        color: Qt.blue
    }

    Rectangle {
        id: colorRect
        anchors.centerIn: parent
        width: 200
        height: 200
        color: "yellow"
    }


    Button {
        id: start
        text: "start"
        anchors.left: parent.left
        anchors.leftMargin: 10
        anchors.bottom: parent.bottom
        anchors.bottomMargin: 10

        onClicked: rgbGame.start()
    }

    Button {
        id: stop
        text: "stop"
        anchors.left: start.right
        anchors.leftMargin: 5
        anchors.bottom: start.bottom

        onClicked: rgbGame.stop()
    }
```

```
    function changeAlgorithm(button, algorithm) {
        switch(algorithm)
        {
        case 0:
            button.text = "RandomRGB";
            break;
        case 1:
            button.text = "RandomRed";
            break;
        case 2:
            button.text = "RandomGreen";
            break;
        case 3:
            button.text = "RandomBlue";
            break;
        case 4:
            button.text = "LinearIncrease";
            break;
        }
    }


    Button {
        id: colorAlgorithm
        text: "RandomRGB"
        anchors.left: stop.right
        anchors.leftMargin: 5
        anchors.bottom: start.bottom

        onClicked: {
            var algorithm = (rgbGame.algorithm() + 1) % 5
            changeAlgorithm(colorAlgorithm, algorithm)
            rgbGame.setAlgorithm(algorithm)
        }
    }

    Button {
        id: quit
        text: "quit"
        anchors.left: colorAlgorithm.right
        anchors.leftMargin: 5
        anchors.bottom: start.bottom

        onClicked: Qt.quit()
    }


    Component.onCompleted: {
        rgbGame.color = Qt.rgba(0, 180, 120, 255)
        rgbGame.setAlgorithm(rgbGame.LinearIncrease)
        changeAlgorithm(colorAlgorithm, rgbGame.algorithm())
    }

    Connections {
        target: rgbGame
        onCurrentTime: {
            timeLabel.text = strTime
            timeLabel.color = rgbGame.timeColor
        }
    }

    Connections {
        target: rgbGame
        onColorChanged: colorRect.color = color
    }
}
```

main.qml 中使用 RGBGame 构造了一个对象，id 为 rgbGame ，这样就可以借助 id 来访问 C++类对象了 ， 这里还定义了一个 changeAlgorithm 函数，参数 button 和 algorithm 拥有动态类型。

界面由三部分组成。顶部是一个 Text ，用来显示由 RGBGame 提供的时间，这里使用 Connections 对象来连接目标对象和信号处理器，指定 target 为 RGBGame ，在 onCurrentTime 信号处理器中改变 timeLabel 的文本和颜色，颜色用到了 RGBGame 的 timeColor 属性，该属性的读取函数是 timeColor 。

界面中间是一个 Rectangle 对象，id 是 colorRect 。这里使用 Connections 对象，指定 target 为 rgbGame ，在 onColorChanged 信号处理器中改变 colorRect 的颜色。

界面底部是几个按钮，使用锚布局把它们排成一行。 start 按钮的 onClicked 信号处理器调用 rgbGame 的 start() 槽，启动颜色生成器。 stop 按钮的 onClicked 信号处理器调用 rgbGame 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 按钮点击时退出应用。

2.3 QML 上下文属性设置

在 C++应用程序加载 QML 对象时，我们可以直接嵌入一些 C++数据来给 QML 代码使用，这里需要用到 QQmlContext::setContextProperty()，即设置 QML 上下问属性，它可以是一个简单的类型，也可以是任何我们自定义的类对象。

main.cpp 中把 RGBGame 类对象设置为 QML 上下文属性：

```
#include <QGuiApplication>
#include <QQuickView>
#include "RGBGame.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQuickView view;
    RGBGame myRGBGame;
    view.rootContext()->setContextProperty("myRGBGame", &myRGBGame);
    view.setSource(QUrl(QStringLiteral("qrc:///main.qml")));
    view.show();

    return app.exec();
}
```

在 C++中构建一个 RGBGame 类对象 myRGBGame，然后设置为 QML 上下文属性，名字是 myRGBGame，这样就可以在 QML 文件中通过 myRGBGame 来访问在 C++实例化的对象 myRGBGame，与 qmlRegisterType()不同的是：不能在 QML 文件中使用 RGBGame 类进行实例化，不能访问 RGBGame 中的枚举类型，也不能作为函数参数，其它用法基本相同。

3、C++访问 QML

同样，在 C++代码中也可以访问 QML 对象中的属性、函数和信号。

3.1 在 C++中加载 QML 对象

在 C++中加载 QML 文件可以用 QQmlComponent 或 QQuickView，然后就可以在 C++中访问 QML 对象了。QQuickView 提供了一个显示用户界面的窗口，而 QQmlComponent 没有。

3.1.1 使用 QQmlComponent

使用 QQmlComponent 时，需调用 QQmlComponent::create()来给组件构造一个新的实例，如果想显示窗口，可在 QML 文件中使用 Window 对象。main.cpp 如下：

```
#include <QGuiApplication>
#include <QtQml>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQmlEngine engine;
    QQmlComponent component(&engine, QUrl(QStringLiteral("qrc:///MyQQmlComponent.qml")));
    component.create();

    return app.exec();
}
```

MyQQmlComponent.qml 是新添加的 QML 文件，如下所示：

```
import QtQuick 2.3
import QtQuick.Window 2.0

Window {
    visible: true
    width: 600; height: 600
    title: "Using QQmlComponent"
    color: "lightblue"

    Rectangle {
        width: 500; height: 500
        anchors.centerIn: parent
        color: "yellow"

        Rectangle {
            objectName: "rect"
            width: 100; height: 100
            anchors.centerIn: parent
            color: "blue"
        }
    }
}
```

3.1.2 使用 QQuickView

使用 QQuickView 时，会自动创建一个组件实例，可以通过 QQuickView::rootObject() 来获取。main.cpp 如下：

```
#include <QGuiApplication>
#include <QQuickView>
#include <QtQml>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQuickView view;
    view.setSource(QUrl(QStringLiteral("qrc:///MyQQuickView.qml")));
    view.show();

    return app.exec();
}
```

MyQQuickView.qml 是新添加的 QML 文件， 如下所示：

```
import QtQuick 2.3

Rectangle {
    width: 500
    height: 500
    color: "yellow"
}
```

3.2 在 C++中访问 QML 对象的属性

在 C++中加载了 QML 文件并进行组件实例化后，就可以在 C++中访问、修改这个实例的属性值了，可以是 QML 内建属性，也可以是自定义属性，方法如下：

```
#include <QGuiApplication>
#include <QtQml>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQmlEngine engine;
    QQmlComponent component(&engine, QUrl(QStringLiteral("qrc:///MyQQmlComponent.qml")));
    QObject *object = component.create();

    object->setProperty("width", 510);
    QQmlProperty(object, "height").write(510);

    return app.exec();
}
```

使用了 QObject::setProperty()和 QQmlProperty 来分别修改 **width** 和 **height**。

```
#include <QGuiApplication>
#include <QQuickView>
#include <QQuickItem>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQuickView view;
    view.setSource(QUrl(QStringLiteral("qrc:///MyQQuickView.qml")));
    view.show();
    QQuickItem *item = view.rootObject();

    item->setWidth(300);

    return app.exec();
}
```

使用了 QQuickItem::setWidth()来修改 **width**。

QML 组件有时候是一个复杂的树型结构，包含兄弟组件和孩子组件，我们可以使用 QObject::findchild()或 QObject::findchildren()，通过 QObject::objectName 来访问，使用方法如下：

```
QObject *rect = object->findChild<QObject *>("rect");
if(rect)
{
    rect->setProperty("color", "black");
}
```

MyQQmlComponent.qml 中有个孩子 Rectangle 的 objectName 是 rect，通过 rect 来查找，找到把它的 **color** 属性由 yellow 改为 black。

3.3 在 C++中访问 QML 对象的函数与信号

在 C++中，使用 QMetaObject::invokeMethod()可以调用 QML 中的函数，从 QML 中传递过来的函数参数和返回值会被转换为 C++中的 QVariant 类型， 成功返回 true，参数不正确或被调用函数名错误返回 false，invokeMethod()共有四个重载函数，用法相似。必须使用 **Q_ARG()**宏来声明函数参数，用 **Q_RETURN_ARG()**宏来声明函数返回值，其原型如下：

QGenericArgument	Q_ARG(Type, const Type & value)
QGenericReturnArgument	Q_RETURN_ARG(Type, Type & value)

使用 QObject::connect()可以连接 QML 中的信号，connect()共有四个重载函数，它们都是静态函数。必须使用 **SIGNAL()**宏来声明信号，**SLOT()**宏声明槽函数。

使用 QObject::disconnect()可以解除信号与槽函数的连接。

添加 CallQML.qml，自定义一个 **qmlSignal()**信号和一个 **myQmlFunction()**函数，设置一个 **MouseArea**，鼠标点击时发送 qmlSignal()信号，代码如下：

```
import QtQuick 2.3
import QtQuick.Window 2.0
import suntec.tools.MyClass 1.0

Window {
    visible: true
    width: 360; height: 360
    title: "Using QML methods/signals in C++"
    color: "lightblue"

    Rectangle {
        objectName: "rect"
        id:button
        width: 100; height: 100
        anchors.centerIn: parent
        color: "lightyellow"

        signal qmlSignal(string msg)

        function myQmlFunction(msg) {
            console.log("Got message:", msg)
            return "Return value from QML"
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
                button.qmlSignal("Hello from QML")
            }
        }
    }
}
```

添加 MyClass.h、MyClass.cpp，实现一个 public 槽函数 **cppSlot()**，用于连接 CallQML.qml 的信号 qmlSignal()，代码如下：

```
class MyClass : public QObject
{
    Q_OBJECT

public slots:
    void cppSlot(const QString &msg)
    {
        qDebug() << "MyClass::cppSlot called with:" << msg;
    }
};
```

main.cpp 中，使用 **QObject::connect()**连接了 QML 中的 qmlSignal()信号和 C++中的 cppSlot()槽函数，QML 中设置的 MouseArea 被点击时就会触发 qmlSignal()信号，然后执行与之连接的 cppSlot()槽函数；使用 **QMetaObject::invokeMethod()** 调用了 QML 中的 myQmlFunction()函数，验证了函数参数及返回值的正确性，代码如下：

```
#include <QGuiApplication>
#include <QtQml>
#include "MyClass.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    qmlRegisterType<MyClass>("suntec.tools.MyClass", 1, 0, "MyClass");

    QQmlEngine engine;
    QQmlComponent component(&engine, QUrl(QStringLiteral("qrc:///CallQML.qml")));
    QObject *object = component.create();

    QObject *rect = object->findChild<QObject *>("rect");
    MyClass myClass;
    QObject::connect(rect, SIGNAL(qmlSignal(QString)),
                     &myClass, SLOT(cppSlot(QString)));

    QVariant returnedValue;
    QVariant msg = "Hello from C++";
    QMetaObject::invokeMethod(rect, "myQmlFunction",
            Q_RETURN_ARG(QVariant, returnedValue),
            Q_ARG(QVariant, msg));
    qDebug() << "QML function returned:" << returnedValue.toString();

    //delete object;

    return app.exec();
}
```

4、总结

本文主要介绍了 Qt QML 与 C++混合编程常用的方法与技巧，在使用过程中有几点值得注意：

自定义类一定要派生自 QObject。

使用 Q_OBJECT 宏。

注册自定义类到 Qt 元对象系统或设置自定义类对象为 QML 上下文属性是必须的。

两者交互进行数据传递时，要符合 QML 与 C++间数据类型的转换规则。QML 引擎支持部分 C++数据类型，如果是自定义数据类型，那就必须注册到 QML 类型系统。