**QQuickView 提供Qt Quick显示的窗口**

The QQuickView class provides a window for displaying a Qt Quick user interface

**QQmlEngine 提供QML组件实例化的环境**

The QQmlEngine class provides an environment for instantiating QML components.

**QQmlComponent 封装组件定义**

The QQmlComponent class encapsulates a QML component definition

**QQmlContext 给QMLengine提供一个上下文环境**

The QQmlContext class defines a context within a QML engine.

QML API有三个主要成员——[QDeclarativeEngine](http://doc.qt.nokia.com/4.7/qdeclarativeengine.html)，[QDeclarativeComponent](http://doc.qt.nokia.com/4.7/qdeclarativecomponent.html)和[QDeclarativeContext](http://doc.qt.nokia.com/4.7/qdeclarativecontext.html)。

[QDeclarativeEngine](http://doc.qt.nokia.com/4.7/qdeclarativeengine.html)提供了QML的运行环境。 [QDeclarativeComponent](http://doc.qt.nokia.com/4.7/qdeclarativecomponent.html)封装了[QML Documents](http://doc.qt.nokia.com/4.7/qdeclarativedocuments.html)。 [QDeclarativeContext](http://doc.qt.nokia.com/4.7/qdeclarativecontext.html)允许程序使用QML组件显示数据。

QML包含一个非常好用的API——[QDeclarativeView](http://doc.qt.nokia.com/4.7/qdeclarativeview.html)。通过它，应用程序可以很方便的把QML组件嵌入到[QGraphicsView](http://doc.qt.nokia.com/4.7/qgraphicsview.html)中。[QDeclarativeView](http://doc.qt.nokia.com/4.7/qdeclarativeview.html)主要用于在应用程序开发过程中进行快速原型开发，它的主要特性将在下面讨论。

如果你正打算用QML改造现有的Qt应用程序，请参考[QML与Qt UI代码整合](http://doc.qt.nokia.com/4.7/qml-integration.html)。

**基本用法**

若想将QML与C++程序结合，程序中至少需要一个[QDeclarativeEngine](http://doc.qt.nokia.com/4.7/qdeclarativeengine.html)。只有程序中需要使用多个不同的QML组件实例时，才需要多个[QDeclarativeEngine](http://doc.qt.nokia.com/4.7/qdeclarativeengine.html)。为了使所有QML组件实例可以工作，[QDeclarativeEngine](http://doc.qt.nokia.com/4.7/qdeclarativeengine.html)为他们提供全局配置，[QDeclarativeEngine](http://doc.qt.nokia.com/4.7/qdeclarativeengine.html)对于C++中使用QML的作用如同[QNetworkAccessManager](http://doc.qt.nokia.com/4.7/qnetworkaccessmanager.html)对于网络通信、路径对于持久化存储的作用。

可以使用[QDeclarativeComponent](http://doc.qt.nokia.com/4.7/qdeclarativecomponent.html)加载[QML Documents](http://doc.qt.nokia.com/4.7/qdeclarativedocuments.html)。每一个[QDeclarativeComponent](http://doc.qt.nokia.com/4.7/qdeclarativecomponent.html)实例对应一个QML document。

可以传递一个Document URL或者表示Document内容的原始文本给[QDeclarativeComponent](http://doc.qt.nokia.com/4.7/qdeclarativecomponent.html)。Document URL可以是本地文件系统URL，或者任何[QNetworkAccessManager](http://doc.qt.nokia.com/4.7/qnetworkaccessmanager.html)支持的网络URL。

可以通过[QDeclarativeComponent::create()](http://doc.qt.nokia.com/4.7/qdeclarativecomponent.html#create)方法创建QML组件实例。下面的代码演示了如何加载一个QML Document并创建一个实例：

QDeclarativeEngine \*engine = new QDeclarativeEngine(parent);  
QDeclarativeComponent component(engine, QUrl::fromLocalFile("main.qml"));  
QObject \*myObject = component.create();

**暴露数据（Exposing Data）**

QML组件在[QDeclarativeContext](http://doc.qt.nokia.com/4.7/qdeclarativecontext.html)中实例化。**一个上下文（context）允许程序暴露数据给QML组件实例。一个[QDeclarativeContext](http://doc.qt.nokia.com/4.7/qdeclarativecontext.html)可用于创建应用程序中用到的所有对象实例，如果需要精确控制为每个实例暴露的数据，可以创建多个[QDeclarativeContex](http://doc.qt.nokia.com/4.7/qdeclarativecontext.html)**。如果上下文（context）没有传递给[QDeclarativeComponent::create()](http://doc.qt.nokia.com/4.7/qdeclarativecomponent.html" \l "create)方法，将默认使用[QDeclarativeEngine](http://doc.qt.nokia.com/4.7/qdeclarativeengine.html)的根上下文（[root context](http://doc.qt.nokia.com/4.7/qdeclarativeengine.html" \l "rootContext)），这时数据通过跟上下文（root context）暴露给所有对象实例。

**简单数据（Simple Data）**

向QML组件实例暴露数据，通过QML属性绑定（Property Bindings）和JavaScript对象访问应用程序设置上下文属性（context properties）。下面的例子展示了如何通过[QDeclarativeView](http://doc.qt.nokia.com/4.7/qdeclarativeview.html)暴露背景颜色给QML文件：

[复制代码](javascript:void(0);)

// main.cpp  
 #include <QApplication>  
 #include <QDeclarativeView>  
 #include <QDeclarativeContext>  
   
 int main(int argc, char \*argv[])  
 {  
 QApplication app(argc, argv);  
   
 QDeclarativeView view;  
 QDeclarativeContext \*context = view.rootContext();  
 context->setContextProperty("backgroundColor",  
 QColor(Qt::yellow));  
 //利用QDeclarativeContext 给QML提供一个全局的变量backgroundColor 颜色为黄色  
 view.setSource(QUrl::fromLocalFile("main.qml"));  
 view.show();  
   
 return app.exec();  
 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

// main.qml  
   
 import QtQuick 1.0  
   
 Rectangle {  
 width: 300  
 height: 300  
   
 color: backgroundColor  
   
 Text {  
 anchors.centerIn: parent  
 text: "Hello Yellow World!"  
 }  
 }

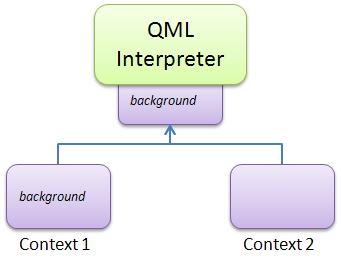
[复制代码](javascript:void(0);)

如果你只希望在main.cpp里创建组件，不想显示在[QDeclarativeView](http://doc.qt.nokia.com/4.7/qdeclarativeview.html)中，需要使用[QDeclarativeEngine::rootContext()](http://doc.qt.nokia.com/4.7/qdeclarativeengine.html" \l "rootContext)来创建[QDeclarativeContext](http://doc.qt.nokia.com/4.7/qdeclarativecontext.html)实例：

QDeclarativeEngine engine;  
 QDeclarativeContext \*windowContext = new QDeclarativeContext(engine.rootContext());  
 windowContext->setContextProperty("backgroundColor", QColor(Qt::yellow));  
   
 QDeclarativeComponent component(&engine, "main.qml");  
 QObject \*window = component.create(windowContext);

上下文属性（Context Properties）的工作方式同QML绑定（QML bindings）中的普通属性（normal properties）一样，在上面的例子中，当上下文属性（context poperty）backgroundColor变为红色时，组件对象实例都会自动更新。请注意创建者有责任删除它创建的[QDeclarativeContext](http://doc.qt.nokia.com/4.7/qdeclarativecontext.html)。当销毁Window组件时，windowContext必须被显式的销毁（手动delete），或者用一种更简单的方法——设置windowContext的父类为window（父对象释放时，会自动释放所有子对象）。

QDeclarativeContexts以树状结构组织，除了根上下文（root context）外，每个[QDeclarativeContext](http://doc.qt.nokia.com/4.7/qdeclarativecontext.html)都有一个父对象。QDeclarativeContexts子对象有效的继承父对象的上下文属性（context properties），这使得应用程序更加灵活的在不同的QML对象实例间暴露（exposed）数据。如果[QDeclarativeContext](http://doc.qt.nokia.com/4.7/qdeclarativecontext.html)设置了一个与父对象相同的上下文属性（context property），父对象的这个属性将被“隐藏”。如下面的例子所示，Context1中的上下文属性（context property）background“隐藏”了根上下文（root context）中的background属性。



**结构化数据（Structed Data）**

上下文属性（context property）还可用于向QML对象暴露结构化可写数据。除了[QVariant](http://doc.qt.nokia.com/4.7/qvariant.html)已经支持的所有类型外，派生自[QObject](http://doc.qt.nokia.com/4.7/qobject.html)的类型也可分配给上下文属性（context property）。[QObject](http://doc.qt.nokia.com/4.7/qobject.html)上下文属性（context property）允许暴露结构化的数据，并允许QML对这些数据设值。 下面的例子创建了一个CustomPalette对象，并将它设为名为palette的上下文属性（context property）：

[复制代码](javascript:void(0);)

class CustomPalette : public QObject  
 {  
 Q\_OBJECT  
 Q\_PROPERTY(QColor background READ background WRITE setBackground NOTIFY backgroundChanged)  
 Q\_PROPERTY(QColor text READ text WRITE setText NOTIFY textChanged)  
   
 public:  
 CustomPalette() : m\_background(Qt::white), m\_text(Qt::black) {}  
   
 QColor background() const { return m\_background; }  
 void setBackground(const QColor &c) {  
 if (c != m\_background) {  
 m\_background = c;  
 emit backgroundChanged();  
 }  
 }  
   
 QColor text() const { return m\_text; }  
 void setText(const QColor &c) {  
 if (c != m\_text) {  
 m\_text = c;  
 emit textChanged();  
 }  
 }  
   
 signals:  
 void textChanged();  
 void backgroundChanged();  
   
 private:  
 QColor m\_background;  
 QColor m\_text;  
 };  
   
 int main(int argc, char \*argv[])  
 {  
 QApplication app(argc, argv);  
   
 QDeclarativeView view;  
 view.rootContext()->setContextProperty("palette", new CustomPalette);  
   
 view.setSource(QUrl::fromLocalFile("main.qml"));  
 view.show();  
   
 return app.exec();  
 }

[复制代码](javascript:void(0);)

下面的QML文件使用了palette对象及它的属性来设置背景和文字颜色。当窗口被点击时，palette的颜色将被改变，窗口文本也会被相应的更新：

[复制代码](javascript:void(0);)

import QtQuick 1.0  
   
 Rectangle {  
 width: 240  
 height: 320  
 color: palette.background  
   
 Text {  
 anchors.centerIn: parent  
 color: palette.text  
 text: "Click me to change color!"  
 }  
   
 MouseArea {  
 anchors.fill: parent  
 onClicked: {  
 palette.text = "blue";  
 }  
 }  
 }

[复制代码](javascript:void(0);)

在这个例子中，当检测到C++属性值（CustomPalette的文本）改变时，该属性必须有一个相应的NOTIFY信号，当属性值改变时发送NOTIFY信号。实现的时候需要注意，仅当属性值改变时才发送信号，从而避免发生死循环。访问一个绑定的属性时，如果没有NOTIFY信号将会导致QML产生一个运行时的警告。

**动态结构数据（Dynamic Structured Data）**

如果一个应用程序在编译期具有很多QObject类型的动态结构化数据，可以使用[QDeclarativePropertyMap](http://doc.qt.nokia.com/4.7/qdeclarativepropertymap.html)在运行期动态的创建这些结构化数据。

**在QML中调用C++方法**

QML中可以调用[QObject](http://doc.qt.nokia.com/4.7/qobject.html)及其派生类对象中的public slot的方法或标记为Q\_INVOKABLE的方法。

上述的C++方法可以具有参数和返回值，QML支持下列数据类型：

[复制代码](javascript:void(0);)

\* bool  
 \* unsigned int, int  
 \* float, double, qreal  
 \* QString  
 \* QUrl  
 \* QColor  
 \* QDate，QTime，QDateTime  
 \* QPoint，QPointF  
 \* QSize, QSizeF  
 \* QRect，QRectF  
 \* QVariant

[复制代码](javascript:void(0);)

下面的例子演示了当MouseArea被点击时，触发“Stopwatch”对象的start()/stop()：

[复制代码](javascript:void(0);)

// main.cpp  
   
 class Stopwatch : public QObject  
 {  
 Q\_OBJECT  
 public:  
 Stopwatch();  
   
 Q\_INVOKABLE bool isRunning() const;  
   
 public slots:  
 void start();  
 void stop();  
   
 private:  
 bool m\_running;  
 };  
   
 int main(int argc, char \*argv[])  
 {  
 QApplication app(argc, argv);  
   
 QDeclarativeView view;  
 view.rootContext()->setContextProperty("stopwatch",  
 new Stopwatch);  
   
 view.setSource(QUrl::fromLocalFile("main.qml"));  
 view.show();  
   
 return app.exec();  
 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

// main.qml  
   
 import QtQuick 1.0  
   
 Rectangle {  
 width: 300  
 height: 300  
   
 MouseArea {  
 anchors.fill: parent  
 onClicked: {  
 if (stopwatch.isRunning())  
 stopwatch.stop()  
 else  
 stopwatch.start();  
 }  
 }  
 }

[复制代码](javascript:void(0);)

请注意，在这个特殊的例子中，有一个更好的方法来达到同样的结果——main.qml中可以用一个“运行中“属性（property），更漂亮的QML代码如下：

[复制代码](javascript:void(0);)

// main.qml  
 import QtQuick 1.0  
   
 Rectangle {  
 MouseArea {  
 anchors.fill: parent  
 onClicked: stopwatch.running = !stopwatch.running  
 }  
 }

[复制代码](javascript:void(0);)

此外，还可以调用[functions declared in QML from C++](http://doc.qt.nokia.com/4.7/qml-extending-types.html#adding-new-methods)描述的方法。