**Qt Book**

目录

[说明 1](#_Toc495308804)

[Core Internals 1](#_Toc495308805)

[Qt Core 1](#_Toc495308806)

[Qt Quick 1](#_Toc495308807)

[Qt Quick中的重要概念 1](#_Toc495308808)

[C++扩展(C++ Extensions Points) 2](#_Toc495308809)

[Reference Documentation 2](#_Toc495308810)

[Qt Quick Examples and Tutorials 2](#_Toc495308811)

[Qt QML 2](#_Toc495308812)

[Getting Started 2](#_Toc495308813)

[QML and QML Types 2](#_Toc495308814)

[JavaScript Environment for QML Applications 3](#_Toc495308815)

[Integrating QML with C++ Applications 3](#_Toc495308816)

[其他框架(Additional Frameworks) 3](#_Toc495308817)

[Licenses and Attributes 3](#_Toc495308818)

[Qt Application 3](#_Toc495308819)

[什么是QML？(what is QML?) 3](#_Toc495308820)

[什么是Qt Quick？(what is Qt Quick?) 4](#_Toc495308821)

[QML用户界面(QML User Interfaces) 4](#_Toc495308822)

[特效(Speical Effects) 5](#_Toc495308823)

[在QML应用中浏览网络内容(Viewing Web Content in QML Application) 5](#_Toc495308824)

[Sensors,Gestures, and Touch Interfaces 5](#_Toc495308825)

[多媒体(Multimedia Content) 5](#_Toc495308826)

[移动设备(Mobile Devices) 5](#_Toc495308827)

[Code Samples and Demos 6](#_Toc495308828)

[应用开发高级话题(Advanced Application Development Topics) 6](#_Toc495308829)

[其他QML模块 6](#_Toc495308830)

[相关话题 6](#_Toc495308831)

[单词索引 6](#_Toc495308832)

# 说明

翻译中有写在中文后面保留有英文原文，多半情况下都是因为自己觉得翻译不准确而保留的，以后在看的过程中，知道了正确翻译或者更好的翻译，可以随时改正。

到【导入语句】[从模块中导入JavaScript资源]处。但暂时不继续从这里下去了，先跳过【导入语句】部分，后面在逐渐完善。

# qmake Manual

qmake工具能够帮组开发这简化跨平台的项目开发。它能够自动配置Makefile文件，开发者只需要填写很少几行就可以了。你可以将qmake应用到任何软件工程中去，即使工程不是Qt项目。

qmake根据工程中文件的信息来配置Makefile文件。工程文件是由开发这自己创建的，通常来说比较简单，但是更复杂的工程文件可以用到更为复杂的工程当中。

qmake包含了其他特性，用以支持Qt开发，它会自动包含使用moc和uic构建工程的规则。

qamke还可以为Microsoft Visual Studio配置工程，开发者因此不需要对工程文件做任何改动。

# Core Internals

## Qt Core 模块

### 本模块简介

Qt中所有模块都依赖于该模块。要使用该模块需要在源文件中使用以下指令：

#include <QtCore>

如果使用qmake来构建工程，Qt Core模块和Qt GUI模块将自动包含到工程中。

#### 核心功能(Core Functionalities)

Qt向C++中添加了这些特性：

1 信号与槽机制，用于对象间通信；

2 可查询和可设置的对象属性；

3 以层次化和可查询方式组织的对象树；

4 具有保护指针的自然的对象所有权方式；

5 跨越库边界的动态类型转换(a dynamic cast that works across library boundaries)；

下面列举的章节中详细介绍了各个Qt核心功能：

《The Meta-Object System》

《The Property System》

《Object Model》

《Object Trees & Ownership》

《Signals & Slots》

#### 线程和并发编程(Threading and Concurrent Programming)

。。。

### 元对象系统(The Meta-Obejct System)

Qt元对象系统提供了对象间通信、运行时类型信息和动态属性系统功能。

元对象系统基于以下三点：

1 QObject类是所有要使用元对象功能的类的基类；

2 Q\_OBJECT宏嵌入到类的私有声明片段中以启用元对象系统特性；

3 Meta-Object Compiler(moc)元对象编译器会为每一个QObject子类都生成一段支持元对象特性的代码。

moc读取C++源文件。如果在一个或者多个类中发现了Q\_OBJECT宏的声明，则moc会为每一个声明该宏的类另外生成一个新的支持元对象特性的源代码文件。这些代码会通过#incude指令包含到类的源文件中，或者，更常见的做法是编译、链接到类的实现中。

如果一个类使用了元对象特性，例如信号与槽机制，在moc生成的元对象代码中除了包含该机制的代码，还有关于以下特性的代码：

1 QObject::metaObejct()返回该类关联的元对象；

2 QMetaObject::className()在运行时返回类名，通过C++编译器不需要本地对RTTI的支持；

3 QObject::inherits()返回该对象的类是否是从指定类派生来的。

4 QObject::tr()和QObject::trUtf8()用与字符串的国际化转换；

5 QObject::setProperty()和QObject::property()可以通过属性名动态设置和获取对象属性；

6 QMetaObject::newInstance()为该类构造新的实例。

。。。。。。

## Qt Widgets 模块

Qt Widgets模块提供了一系列UI元素来创建传统的桌面风格用户交互界面。可以参考《User Interfaces》查看更多关于开发Qt用户交互界面的内容。

## Qt Quick 模块

QtQuick模块是QML语言的标准库，正如STL是C++语言的标准库一样。QtQML模块提供了QML语言引擎以及语言相关的基础架构，参考《Qt QML》内容。QtQuick模块为使用QML语言创建用户界面提供了所有基础且必要的类型。QtQuick模块提供了视图画布(visual canvas)，以及创建可视组件、为组件设置动画，接收用户输入，创建数据模型和视图以及延时对象实例化(delayed object intantiation)。

QtQuick提供了QML API和C++ API。都支持QML.类型，以及使用QML语言编写用户界面。C++ API为能将QML扩展到C++代码中。

提示：除了QtQuick模块提供的基础类型，QtQuickControls模块还提供了大量用于编写用户界面的控件类。参考《Qt Quick Controls》

对于初次接触QML以及Qt Quick者，参考《QML Application》了解如何编写QML应用。

### Qt Quick中的重要概念

Qt Quick为创建一个流畅、动态的用户界面提供了应有尽有的东西。Qt Quick能够通过定义一系列行为组件来创建用户界面，并可以定义组件或者界面之间如何连接。Qt Quick提供了一个自带坐标系统和渲染引擎的视图画布。动画和转换效果是Qt Quick中首先需要了解的概念。视图效果(visual effects)可以通过专门的组件来增强粒子(particle)和着色效果(shader effects)。参考：

《The Visual Canvas》

《User Input》

《Positioning》

《States,Transitions and Animations》

《Data-Models,Views and Data Storage》

《Particles and Graphics Effects》

《Convenience Types》

在使用Qt Quick模块进行应用开发之前，你应当先了解如何使用QML语言。尤其是在《QML Application》中关于QML基础和核心内容的介绍。

#### States,Transitions And Animations

在任何现代交互界面中，在状态间转换，以及移动交互界面等功能及其有好处。这些在Qt Quick属于最重要的一类概念。

本节介绍了状态(states)，状态转换(states transitions)和属性动画(property animations)。详细介绍了那些重要的概念，为什么以及如何内部相联系的。

##### States

一个特定的可视项的状态由一系列描述了可视项内的组件如何以及在何处显示的信息，以及与状态相关的数据组成。绝大多数的交互界面中的可视项都拥有有限定的、属性定义完善的状态。

例如，列表中的项可能处于选中或者位选中的状态，如果选中了，它可能是当前唯一选中的项或者是选中组中的一个成员。

每一个状态都可能关联一个可视状态。

Qt Quick提供了State类型，并且状态的改变可以触发行为(Behavior)和动画(Animation)。

###### 相关类型(Related Types)

|  |  |
| --- | --- |
| AnchorChanges | 指定在一个状态中如何改变项的锚(anchors) |
| ParentChange | 指定项在状态变化时如何重新指定其父对象 |
| PropertyChange | 描述了一个状态新的绑定属性或者值 |
| State | 定义一个对象或者属性的配置，即定义一个状态 |
| StateChangeScript | 指定如何在一个状态中运行一个脚本 |
| StateGroup | 为非Item类型提供内嵌的状态支持 |

许多用户界面是由状态驱动(state driven)的；交互界面在不同的状态有不同的配置。例如，交通灯的亮灭就是根据其状态来决定的。在停止状态下，交通灯会亮红色，在通行状态下，交通等会亮绿色。

在QML中，状态是一系列关于属性的配置，定义在State类型中。不同的配置可能是：

1 显示一些UI而隐藏另一些UI；

2 代表用户可以操作的不同动作；

3 开始，暂停或者停止动画；

4 执行在新状态中需要使用的脚本；

5 修改一个特定项的属性值；

6 显示不同的视图或者屏幕。

所有基于Item的对象都有states属性，并且可以通过添加新的State类型对象来指明该Item对象的新的状态。每一个State都有一个独一无二的name属性，该属性默认是空字符串。要改变Item对象的当前状态，只需将需要的状态名称赋给其state属性即可。

非Item对象可以使用StateGroup类型类设置状态。

###### 创建状态(Creating States)

创建新的状态，只需要将State类型添加到Item-based对象的states属性列表中。

一个警告信号可能有两种状态，正常状态NORMAL和警告状态CRITICAL。假设在NORMAL状态信号是绿色，并且警号标志处于关闭状态；在警告状态，信号灯是红色，同时开启警告标志。我们可以使用State类型来管理这些不同的状态，并使用PorpertyChanges类型将那些属性改变成对应状态下的值。

Rectangle {

id: signal

width: 200; height: 200

state: “NORMAL”

states: [

State {

name: “NORMAL”

PropertyChanges { target: signal; color: “green” }

PropertyChanges { target: flag; state: “FLAG\_DOWN”}

},

State {

name: “CRITIAL”

PropertyChanges { target: signal; color: “red”}

PropertyChanges { target: flag; state: “FLAG\_UP”}

}

]

}

PropertyChanges类会改变对象的属性值。其目标通过对象id来指定。在该组件外的对象也是通过id属性来指定的，例如上面例子中改变flag对象的属性值。

定义好对象的各种状态后，我们可能需要在某些状态下切换状态。切换对象的状态只需要将其state属性赋予相应状态的名称，即name属性值即可。

Rectangle {

id: signalswitch

width: 75; height: 75

color: “blue”

MouseArea {

anchors.fill: parent

onClicked: {

if(signal.state == “NORMAL”)

signal.state = “CRITICAL”

else

signal.state = “NORMAL”

}

}

}

State类型不会限制对何种属性值进行修改。除例子展示的方式外，它还可以：

1 使用StateChangeScript运行脚本

2 使用PropertyChanges重载对象已经存在的信号处理函数

3 使用ParentChange改变对象的父类对象

4 使用AnchorChanges改变锚值。

###### 默认状态(The Default State)

所有Item-based类型组件都有state属性和一个默认的状态。默认状态下，state的值是空字符串，并且包含了该对象所有初始化属性的值。使用默认状态来管理属性发生改变之前的属性值十分有用。将state属性赋值一个空字符串将会加载默认状态。

###### State类型的when属性

为了方便，State类型有一个when属性用来绑定到一个表达式上，当表达式的值变为true时，对象就会切换到该状态下。

Rectangle {

id: bell

width: 75; height: 75

color: “yellow”

states:State {

name: “RINGING”

when: (signal.state == “CRITICAL”)

PropertyChanges { target: speaker; paly: “RING!” }

}

}

无论什么时候，当signal.state处于CRITICAL状态时，bell组件都会切换到RINGING状态。

###### 状态改变动画(Animating State Changes)

状态改变时会使得属性值突然发生改变。使用Transition类型可以使得状态的改变可以变得更加平滑。在转换中，可以定义动画和差值行为。《Animation and Transition》一节中详述了关于创建状态动画的内容。

《Using Qt Quick Behaviors with States》一节介绍和解释了在状态改变时使用行为的一些常见问题。

###### 状态快进(State Fast Forwarding)

为使Transition能够正确播放状态改变动画，有时候在状态最终改变之前，让引擎快进和回放状态是很有必要的。其过程如下：

1 让状态快进以确定完整过程中的值；

2 状态回放；

3 通过转换，执行状态改变。

在有些情况下，这可能会导致无法预料的行为。例如，当一个状态改变了视图的模型或者Loader的sourceComponent时，会多次设置这些属性(赋值，回放，再赋值)，这将会导致较大的开销。

在实现中，需要考虑转台的快进问题。这些可能会在后续版本中该进。

##### Animation and Transition in Qt Quick

###### Animation and Transition Types

|  |  |
| --- | --- |
| Transition | 播放状态改变时的转换动画 |
| SequentialAnimation | 顺序播放动画 |
| ParallelAnimation | 并行播放动画 |
| Behavior | 为属性变化定义默认动画 |
| PropertyAction | 设置在动画播放过程中立即改变的属性 |
| PauseAnimation | 在动画中加入暂停 |
| SmoothedAnimation | 允许一个属性平滑地变为某个值 |
| SpringAnimation | 允许一个属性弹性摆动(spring-like)地变为某值 |
| ScriptAction | 在动画播放时执行脚本 |

基于特定类型的属性动画

|  |  |
| --- | --- |
| AnchorAnimation | 锚值改变动画 |
| ParentAnimation | 父对象改变动画 |
| PathAnimation | 沿着指定路径播放动画 |
| ColorAnimation | 颜色改变动画 |
| NumberAnimation | qreal类型属性改变动画 |
| PropertyAnimation | 属性值改变动画 |
| RotationAnimation | 旋转值改变动画 |
| Vector3dAnimation | QVector3d值改变动画 |

动画是通过将动画类型应用到属性值上实现。动画类型会通过计算插值来创建平滑的转换。同时，状态转换也可以将动画应用到状态改变中。

要创建动画，需要将合适的动画类型应用到对应的需要动画的属性，然后根据所需行为应用动画。

###### 触发动画(Triggering Animations)

触发动画的方式有多种。

直接启动属性动画(Direct Property Animation)

。。。

##### Animating Property Assignments

动画不仅可以用于可以用与状态和状态的转换。例如，动画可能被任何其他事件触发，并且不必关联到确切的状态。

。。。

##### Animators

Animator类型是一种特殊的Animation类，可以绕开QML对象直接在场景视图的基础上操作。这样的好处是基于Animator的动画可以在视图场景的渲染线程中运行，并且即使在UI其他部分被锁定了也依旧可以播放动画。

Qt Quick提供了以下类型的Animator类型：

|  |  |
| --- | --- |
| XAnimator | 播放Item的水平位置变化动画 |
| YAnimator | 播放垂直位置变化动画 |
| ScaleAnimator | 播放缩放系数变化动画 |
| RotationAnimator | 播放旋转角度变化动画 |
| OpacityAnimator | 播放透明度变化动画 |
| UniformAnimator | 播放在ShaderEffect中通用类型变化动画 |

##### Animated Sprites

。。。

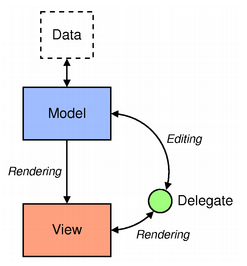
#### Data – Models,Views and Data Storage

大多数应用都有需要显示给用户的数据。这些数据可能来自于很多地方：网络，本地文件，以及数据库等等。

##### Models and Views In QtQuick

在应用当中，使用相似的行为显示相似的数据，基于这种想法，我们使用模型(model)来包含数据，然后使用视图(view)显示基于模型的委托。

简单来说(Simply put)，应用需要组织数据和显示数据。Qt Quick中有models,views和delegats等用来显示数据的概念。他们将数据模块化，一遍开发者能从不同层面控制数据。开发者可以将数据很少改动而在列表视图和网格视图之间转换。类似地，在一个委托中封装的一个数据实例能让开发者指出如何表示和处理这些数据。



**Model** – 包含数据及其数据结构的容器。Qt Quick中有用于创建模型的QML类型。

**View** – 用于显示数据的容器。视图可能显示在列表中或者网格中的数据。

**Delegate** – 用于指出数据在视图中如何显示，委托从模型中获取数据并封装数据。数据可以通过委托进行处理。

要显示数据，将视图的model属性绑定到一个模型，delegate属性绑定到一个组件或者其他可用的类型上。

###### 使用视图显示数据(Displaying Data with Views)

视图是用于项(items)的集合的容器。视图拥有很多特性，可以通过设置这些特性来满足需要的风格或者行为。

Qt Quick基础图形类型组中提供了一组标准视图类型：

ListView – 将项以水平或者垂直方式组织到列表中

GridView – 以网格方式将项组织到网格空间中

PathView – 将项组织到路线中。路线是指依次显示的位置。？

这些类型的属性和行为相互间都是独有的。

修饰视图(Decorating Views)

视图可以通过修饰属性修改显示的样式，例如header、footer、和section属性。通过绑定一个对象——通常是另外的可视对象——到这些属性，视图就可以进行修饰了。footer可能绑定一个Rectangle类型用于展示边界，或者是绑定到header，在列表头用显示logo。

假设有一个俱乐部想用特定的颜色来修饰它的成员列表。成员列表在一个模型中，委托会显示模型的内容。

ListModel {

id: nameModel

ListElement { name: ”Alice” }

ListElement { name: “Bob” }

ListElement { name: “Jane” }

ListElement { name: ”Harray” }

ListElement { name: “Wendy” }

}

Component {

id: nameDelegate

Text {

text: name;

font.pixelSize: 24

}

}

俱乐部想要通过绑定可视对象到header和footer来修饰成员列表。这些可视对象可以声明为内联，在其他文件中，或者在一个Component类型中。

ListView {

anchors.fill: parent

clip: true;

model: nameModel

delegate: nameDelegate

header: bannerComponent

footer: Rectangle {

width: parent.width; height: 30;

gradient: clubcolors;

}

highlight: Rectangle {

width: parent.width:

color: “lightgray”

}

}

Component {

id: bannerComponent

Rectangle{

id: bannner

width: parent.width;height:50

gradient:clubcolors

border { color: “#9eddf2”;width:2}

Text {

anchors.centerIn: parent

text: “Club Members”

font.pixelSize: 32

}

}

}

Gradient {

id: clubcolors

GradientStop { position:0.0; color:” #8EE2FE”}

GradientStop { position :0.66; color:” #7ED2EE”}

}

鼠标和点击处理

视图会处理内容的抓取和点击，然而它们并不处理单独的委托的点击事件。要响应委托的点击，例如要设置currentIndex，委托必须包含MoseArea提供合理的逻辑来提供设置currentIndex的数据。

注意，如果highlightRangeMode被设置成StrictlyEnforceRange，抓取和点击会影响currentIndex，尽管视图总是会确保currentIndex会在声明的高亮范围内？？？

ListView分类(ListView Sections)

ListView视图中的内容可能会组织到不同的段(sections)中，相关的列表项根据其所属的段标注出来。更进一步，这些段可以通过委托进行修修饰。

例如，一个列表可能指明了人们的名字和所属的队伍：

ListModel {

id: nameModel

ListElement { name: “Alice”; team: “Crypto” }

ListElement { name:”Bob”; team: “Crypto” }

ListElement { name:”Jane”; team:”QA”}

ListElement { name:”Victor”; team:”QA”}

ListElement { name:”Wendy”; team:”QA”}

}

Component {

id: nameDelegate

Text {

text: name

font.pixelSize: 24

anchors.left: parent.left

anchors.leftMargin: 2

}

}

ListView拥有名为section的属性，能够将相邻的或者相关的类型关联在一起。section.propert指明了模型列表项中的那个属性将作为段。section.criteria指明如何显示段的名字，section.delegate属性的用法则和视图的委托用法相似。

ListView {

anchors.fill: parent

model: nameModel

delegate: nameDelegate

focus: true

highlight: Rectangle {

color: "lightblue"

width: parent.width

}

section {

property: "team"

criteria: ViewSection.FullString

delegate: Rectangle {

color: "#b0dfb0"

width: parent.width

height: childrenRect.height + 4

Text { anchors.horizontalCenter: parent.horizontalCenter

font.pixelSize: 16

font.bold: true

text: section

}

}

}

}

###### 视图委托(View Delegates)

视图需要一个委托来表示在列表中的项。视图会根据委托定义的模板来显示列表中的每一项。可以根据index来获取在模型中的某一项的序列位置。

Component {

id: petdelegate

Text {

id: label

font.pixelSize: 24

text: if (index == 0)

label.text = type + " (default)"

else

text: type

}

}

通过委托操作视图和模型(Accessing Views and Models from Delegates)

一个视图可以通过绑定的委托，使用ListView.view属性来操作。类似地，GridView视图可以通过委托使用GridView.view属性来操作。相应的模型和及其属性就可以通过ListView.view.model进行使用。另外，任何在模型中定义的方法和信号也可以在委托中使用。

当你想将同一个委托应用到多个视图中时这种机制十分有效。例如，当你想让每个视图的修饰和特性有所不同，并且想把这些不同的设置提供给视图的属性时。类似地，有可能需要处理和显示模型的一些属性。

下面的例子，展示了委托显示模型的language属性，然后每一个项的颜色则是视图的fruit\_color属性提供的颜色：

Rectangle {

width: 200; height: 200

ListModel {

id: fruitModel

property string language: “en”

ListElement {

name: “Apple”

cost: “2.45”

}

ListElement {

name: “Orange”

cost:3.25

}

ListElement {

name: “Banana”

cost: 1.95

}

}

Component {

id: fruitDelegate

Row {

id: fruit

Text { text: “Fruit: ”+name; color: fruit.ListView.view.fruit\_color}

Text { text: “Cost: $”+cost }

Text { text: “Language: ”+fruit.ListView.view.model.language}

}

}

ListView {

property color fruit\_color: “green”

model: fruitModel

delegate: fruitDelegate

anchors.fill: parent

}

}

###### 模型(Models)

数据是通过委托绑定的命名数据元(named data roles)提供的，下面的例子中两个数据元，type和age，然后使用有委托绑定到这些数据元的ListView来显示这些数据元的值：

import QtQuick 2.0

Item {

width: 200; height:200

ListModel {

id: myModel

ListElement { type: “Dog”; age: 8 }

ListElement { type: “Cat”; age: 5 }

}

Component {

id: myDelegate

Text { text: type+”,”+age }

}

ListView {

anchors.fill: parent

model: myModel

delegate: myDelegate

}

}

如果委托的属性和模型的属性间存在命名冲突，数据元可以通过添加model名称来代替。例如，如果Text类型有type或者age属性，text属性显示的值将是Text类型的type和age属性值。在这种情况下，可以使用model.type和model.age来替代，保证委托正确地显示需要的属性值。

模型中的所有项都包含一个特殊的数据元index，它保存了该项在模型中的索引位置。委托可以使用该数据元。注意当模型中的某项从模型中移除后，该项的index将被置为-1.如果要在代码逻辑中使用index数据元，要注意考虑index可能为-1的情况。

如果模型没有命名的数据元，数据会通过modelData数据元提供。当一个模型只有一个数据元的时候，也会提供modelData。在这种情况下，modelData中的数据和命名数据元中的数据相同。

QML提供了几种内嵌到QML类型中的数据模型。另外，模型还可以通过Qt C++创建，然后通过QML组件，是的QQmlEngine可以使用。如何创建这些模型，参考《Using C++ Models with Qt Quick Views》和《creating QML types》

可以使用Repeater类型来设置模型中的项。

列表模型(List Model)

ListModel在QML中是一种简单的类型层次。其数据元是通过ListElement属性来声明的。 ListModel {

id: fruitModel

ListElement {

name: “Apple”

cost: 2.45

}

ListElement {

name: “Orange”

cost: 3.25

}

ListElement {

name: “Banana”

cost: 1.95

}

}

上面的模型有两个数据元，name和cost。这些数据元可以被绑定到ListView委托，例如：

ListView {

anchors.fill: parent

model: fruitModel

delegate: Row {

Text { text: “Fruit: ”+name }

Text { text: “Cost: $”+cost}

}

}

ListModel中提供了直接操作ListModel的JavaScript方法。这种情况下，第一项插入到模型中的项指明模型中可以被视图使用的数据元。例如，如果一个空的模型创建出来，并通过JavaScript方法进行操作，模型中第一项提供的数据元在视图中才是可见的：

ListModel { id: fruitModel }

…

MouseArea {

ancrhos.fill: parent

onClicked: fruitModel.append({“cost”:5.95, “name”:Pizza });

}

当点击MouseArea时，fruitModel会拥有两个数据媛，cost和name。即使随后又添加了新的数据元，之后最开始的两个能够在视图中被处理。要重置数据元，使其在模型中起作用，可以调用ListModel::clear()函数。

XML模型(XML Model)

XmlListModel可以从XML获取数据构造模型。数据元通过XmlRole类型声明，该类型需要使用导入语句：

import QtQuick.XmlListModel 2.0

下面的模型中有三个数据元，title，link和description：

XmlListModel {

id: feedModel

source: http://rss.news.yahoo.com/rss/oceania

query: “/rss/channel/item”

XmlRole { name: “title”; query: “title/string()” }

XmlRole { name: “linke”; query: “link/string()”}

XmlRole { name: “description”; query: “description/string()” }

}

query属性指明了在XML文档中配置的XmlListModel的模型项<Item>的每一项。

《RSS News demo》展示了如何使用XmlListModel显示Rss推送消息。

对象模型(Object Model)

ObejctModel用来包含在视图中使用的可视项。当视图中使用一个ObjectModel时，该视图可以不需要委托，因为ObjectModel中已经包含了可视委托(可视项)。

下面的例子中将放置三个有颜色的矩形到ListView中：

import QtQuick 2.0

import QtQml.Models 2.1

Rectangle {

ObejctModel {

id: itemModel

Rectangle { height: 30; width: 80; color: “red”}

Rectangle { height: 30; width:80; color: “green” }

Rectangle { height: 30; width: 80; color: “blue”}

}

ListView {

anchors.fill: parent

model: itemModel

}

}

**注意：**也可以使用VisualItemModel，但是它只在合适的情况下提供。VisualItemModel允许一个QML项被当作模型来使用。该模型包含了数据和委托；VisualItemModel的子项提供了委托的内容。该模型不会提供任何数据源。？？？？

将整数作为模型(Integers as Models)

一个包含了确切数值类型的整型也可以作为模型使用。在这种情况下，该模型不存在任何数据元。

下面的例子中ListView使用了数字5作为模型：

Item {

width: 200; height: 250

Component {

id: itemDelegate

Text { text: “I am item number: ”+index}

}

ListView {

anchors.fill: parent

model: 5

delegate: itemDelegate

}

}

对象实例作为模型(Object Instances as Models)

一个对象实例可用作模型。对象的属性作为数据元。

下面的例子使用创建了包含一个项的列表，用来显示myText的text属性的颜色。在委托中使用了model.color避免名称冲突。

Rectangle {

width: 200; height: 250

Text {

id: myText

text: “Hello”

color: “#dd44ee”

}

Component {

id: myDelegate

Text { text: model.color }

}

ListView {

anchors.fill: parent

ancrhos.topMargin: 30

model: myText

delegate: myDelegate

}

}

使用C++数据模型(C++ Data Models)/Using C++ Models in Qt Quick Views

模型还可以在C++中定义好，然后在QML中使用。这对于将已经存在的C++数据模型和其他复杂的数据结构暴露给QML非常有用。

使用自定义C++模型提供数据

模型可以通过C++定义，然后在QML中使用。这对于将已经C++中的数据或者其他复杂的数据集暴露给QML，十分有用。

C++模型可以定义为QStringList，QVariantList，QObjectList或者QAbstractItemModel。前三项用于简单数据集十分有用，而QAbstractItemModel为复杂数据集提供了更灵活的解决方案。

QStringList-based Model

一个模型可能就是一个简单的QStringList，通过modelData数据元可以获取列表的内容。

下面是ListView通过委托使用modelData获取其应用的模型的项的值：

ListView {

width: 100; height: 100

model: myModel

delegate: Rectangle {

height: 25

width: 100

Text { text: modelData }

}

}

Qt应用可以加载QML文档并设置myModel的值到一个QStringList中：

QStringList dataList;

dataList.append(“Item 1”);

dataList.appden(“Item 2”);

QQuickView view;

QQmlContext \*ctxt = view.rootContext();

ctxt->setContextProperty(“myModel”,QVariant::fromValue(dataList));

**注意：**视图无法知道QStringList的内容发生了改变，如果QStringList发生改变，需要使用QQmlContext::setContextProperty()重置视图中的内容。

《writing:QVariantList-based Model》

###### Repeaters

Repeaters使用来自模型的数据，根据给定模板创建项，通常结合定位器(positioners)来使用。联合Repeaters和positioners可以便捷地创建大量项。在这种情况下，一个Repeater项是放置在一个定位器中的，并且将这些复制产生的项排列在定位器中。

每个复制器(repeaters)通过model属性，来指明要创建的项的数据来源。然后使用声明为Repeater的子对象的模板项来创建各项。复制器创建的项数根据model绑定的数据源中的数据数量来确定。

下面的例子中使用一个复制器和一个网格容器来生成和管理一组Rectangle项。Repeater创建24个矩形，然后放到5x5的网格中：

import QtQuick. 2.0

Rectangle {

width: 400; height: 400; color: “balck”

Grid {

x: 5; y: 5

rows: 5; columns: 5; spacing: 10

Repeater {

model: 24

Rectangle {

width: 70; height: 70

color: “lightgreen”

Text {

text: index

font.pointSize: 30

anchors.centerIn: parent

}

}

}

}

}

Repeater的count属性保存着其创建的项数。但是不可能通过设置该属性来规定要创建的项数。实际上，正如我们在上面例子中展示的，我们将一个整数作为模型来指明创建的项数。

因为实际创建的项数就是根据模型中数据的项数来确定的。

如果模型是一个字符串列表，委托就会暴露给具有只读权限的modelData属性，该属性保存有字符串。例如：

Column {

Repeater {

model: [“apples”,”oranges”,”pears”]

Text { text: “Data: ” + modelData }

}

}

Repeater也可以使用委托作为模板来创建项。通过delegate属性来声明使用的委托。

###### 使用转换功能(Using Transitions)

Transition可以给添加进、移入和移除定位器的项添加动画。

使用Transition为定位器添加项，新增的项将成为定位器的子项。

Trasition从定位器删除项，将会从定位器中移除项，并重新为该项重新设置一个在该文档中的对象作为父类。

注意：改变一项的透明度，并不会导致该项充定位器中消失。可以通过设置visible属性来移除或者重新添加项。

#### 粒子和图形特效(Particles And Graphical Effects)

美观的交互界面比单调的更吸引人。也就是说，界面设计者必须牢记通过视觉特效和用户交流是一种很有效的交流方方式（例如，哪个界面是激活的，用户需要关注什么地方）。但是过度使用特效也会导致用户体验下降。

##### 视图变换(Visual Transformation)

可视对象可以进行变换。如缩放、旋转等。这些不同种类的变换可以提供关于焦点、选择等操作的提示，还可以直观地提示应用正在发生的事件。

### C++扩展点(C++ Extensions Points)

所有QML应用都可以使用C++进行扩展，以使用C++代码实现的功能或者基于C++的QML插件。这个内容在Qt QML模块的《Integrating QML and C++》一节中。

另外，Qt Quick模块为C++开发者提供了几种扩展和整合点，在该模块中使用。实际上，Qt Quick允许C++开发者创建和注册自定义的基于QQuickItem的类在Qt Quick中渲染。该模块还提供了几种场景视图相关的类，使得开发者可以定义自己的渲染设施。

##### 自定义基于QQuickItem的类型(user-Defined QQuickItem-Derived Types)

Qt Quick中已经拥有了大量的可在QML中使用的可视类型，开发者如果要在C++中定义自己的可视类型，然后在QML中使用，最简单的方法就是基于QQuickItem类进行开发。QQuickItem是Qt Quick模块中所有可视类型的基类。参考下节关于QQuickItem的介绍。

###### QQuickItem类

QQuickItem类是Qt Quick模块中所有可视项的基类。

Qt Quick中所有可视类型都是从QQuickItem类派生而来。尽管QQuickItem的实例在场景中是不可见的，但他定义了所有可是对象的通用属性，如位置(x,y)，width，height，锚以及按键事件处理。

你可以通过继承QQuickItem类定义自己的可视类型。

自定义场景项(Custom Scene Graph Items)

所有QML可视项都是使用场景视图来渲染的。场景视图是一种关联OpenGL的底层，高性能的渲染堆栈。可以使用QQuickItem::ItemHasContents标志将QQuickItem的子类的内容添加到视图场景中，然后重载QQuickItem::updatePaintNode()函数。

**警告：**很关键的一点是OpenGL操作和作用于场景视图是发生在渲染线程中的，主要是在updatePaintNode()函数调用期间。最好的的做法是在QQuickItem::updatePaintNode()函数中值使用以”QSG”作为前缀的类。

注意：所有以”QSG”开头的类应该单独在场景视图的渲染线程中使用。参考《Scene Graph and Rendering》

图形资源处理(Graphics Resource Handling)

在场景视图中，推荐的清理图形资源的方式是使用节点的自动清理功能。从QQuickItem::updatePaintNode()函数返回的QSGNode对象会在正确的线程，正确的时间清理。QSGNode实例树通过QSGNode::OwnedByParent自动设置和管理。所以，对于大多数自定义的场景项，都不需要做额外的清理工作。

。。。。。。

### Qt Quick场景视图(Qt Quick Scene Graph)

Qt Quick 2使用一种专门的图形视图和一系列适配器(Adaptation)，默认是OpenGLES 2.0或者OpenGL 2.0来渲染。使用场景视图而不是传统的强制的绘制系统(QPainter和类似的功能)，意味着可以维持帧之间被渲染的场景，所有用于渲染的所有最基本的状态，在开始渲染之前都可知。使用场景视图，将大量基础的操作都提供给了开发者，可以按照自己意愿控制。

例如，一个用户界面有一个包含十个可视项的列表，每个列表都有背景颜色、icon和文本。使用传统的绘制机制，这将需要30次绘制，以及大量类似的状态变化。在场景视图中，可以重新组织将渲染的基元，以使得所有背景都同时绘制，类似地，也可以一次绘制icon，文本。这样，所有的绘制次数只需要三次。批次和状态改变次数的减少可以极大地提高应用的性能。

场景视图是和Qt Quick 2.0关联在一起的，不能单独使用。场景视图的管理和渲染是通过QQuickWindow类进行的。自定义的可视项则可以通过QQuickItem::updatePaintNode()将其图元放置到场景视图中进行渲染。

场景视图是一种用于可视项图形显示，其场景视图结构包含了足够用来渲染其中所有项的信息。一旦建立场景视图结构，就可以独立操作和渲染其中的可视项的状态。在很多平台上，场景视图甚至能够在占用的渲染线程上进行渲染，同时GUI线程已经准备好下一帧的状态。这样子很高效。

**注意：**本章列出的很多内容都默认适配OpenGL的来渲染Qt Quick场景视图的。参考《Scene Graph Adaptaions》

#### Qt Quick场景视图结构(Qt Quick Scene Graph Structure)

场景视图有大量预定义的节点类型组成，每个节点都有特定的目的。尽管我们称之为场景视图，但实际称场景视图为节点树更符合其定义方式。该节点树是在QML场景中使用QQuickItem类建立的，然后场景会被一个渲染器渲染和绘制出来。这些节点本身没有任何用于绘制的代码，也没有虚函数paint().

##### Nodes

对于开发者来说最重要的节点就是QSGGeometryNode。这是通过定义其几何形状和材质来定义自定义图形的。几何图形通过QSGGeomtry定义其图元的形状和网格来实现的。它可是直线、矩形、多边形或者其他互不相连的矩形，或者复杂的3D网格。材质则定义了像素如何填充到形状中。

一个节点可以拥有任意数量的孩子节点，几何节点会按照孩子节点序列顺序几何节点会被渲染，，父节点在所有孩子节点之后。

**注意：**这不是说渲染器中所有实际的渲染顺序都是这样，只针对输出显示的图形是这样。

节点类型有：

|  |  |
| --- | --- |
| QSGClipNode | 在场景视图中实现裁剪功能 |
| QSGGeometryNode | 用于所有在场景视图中被渲染的内容 |
| QSGNode | 场景视图中所有节点的基类 |
| QSGOpacityNode | 用于改变节点的透明度 |
| QSGTransformNode | 实现在场景视图中的各种变换 |

自定义节点通过QQuickItem::updatePaintNode()以及设置QQuickItem::ItemHasContents标志添加到场景视图中的。

**警告：**很关键的一点是OpenGL操作和作用于场景视图是发生在渲染线程中的，主要是在updatePaintNode()函数调用期间。最好的的做法是在QQuickItem::updatePaintNode()函数中值使用以”QSG”作为前缀的类。

###### QSGGeometry类

QSGGeometry类在场景视图中为图元提供了底层的数据空间。该类用于管理顶点数据和索引数据。setDrawingMode()函数设置绘制几何图形的模式，这些设置模式直接对应了OpenGL中的绘制模式，如GL\_LINES,GL\_TRIANGLE\_STRIP,GL\_POINTS等。

顶点数据可以是简单的点，只有代表位置的x,y，也可以是包含向量，纹理坐标和3D坐标的复杂顶点数据。QSGGeometry::AttributeSet就是用来描述如何建立顶点数据的。属性组(Attribute set)只能在QSGGeometry类的构造函数中声明。QSGGeometry类中有一些预定义的默认属性和属性组，如通过defaultAttributes\_Point2D()函数可以获取在普通固定色的矩形几何图形中使用的属性组。defaultAttributes\_TexturedPoint2D()函数可以获取2D纹理几何图形中。顶点数据在类中以void\*形式保存，并且可以通过vertexData()函数获取指向数据的指针。对于默认的属性组，还可以通过vertexDataAsPoint2D()和vertexDataAdTexturedPoint2D()函数获取数据。 顶点数据通过向allocate()函数传入顶点数量来分配内存的。

QSGGeometry可以选择指定索引参数，索引可以是unsigned 32-bit,unsigned 16-bit, unsigned 8-bit的整数。索引参数必须在QSGGeometry的构造函数中指定，而且一旦指定就不可以再改变。

下面一段代码演示了如何创建一个顶点数据包含位置和颜色的几何图形：

struct MyPoint2D {

float x;

float y;

float r;

float g;

float b;

float a;

void set(float \_x, float \_y, float \_r, float \_g, float \_b, float \_a) {

x = \_x; y = \_y; r = \_r; g = \_g; b = \_b; a = \_a;

}

};

QSGGeometry::Attribute MyPoint2D\_Attributes[] = {

QSGGeometry::Attribute::create(0,2,float,true),

QSGGeometry::Attribute::create(1,4,float,false)

};

QSGGeometry::AttributeSet MyPoint2D\_AttributeSet = {

MyPoint2D\_Attributes,

sizeof(MyPoint2D),

2

};

// …

geometry = new geometry(MyPoint2D\_AttributeSet,2);

geometry->setDrawingMode(GL\_LINES);

MyPoint2D \*vertices = static\_cast<MyPoint2D>(geometry->vertexData());

vertices[0].set(0,0,1,0,0,1);

vertices[1].set(width(),height(),0,0,1,1);

QSGGeometry是软件缓冲，在客户端根据OpenGL渲染。

。。。。。。

QSGGeometry::Attribute类

Attribute类描述了QSGGeometry中顶点的一个属性。在Attribute中描述了属性的注册位置，属性对的大小和属性类型。

类中还包含了用于提示着色器属性是否为顶点的位置属性。场景视图着色器可能使用该提示执行优化。

在源文件中Attribute类的声明如下：

class QSGGeometry {

// …

struct Attribute {

int position;

int tupleSize;

int type;

uint isVertextCoordinate : 1;

AttributeType attributeType : 4;

uint reserved : 27;

static Attribute create(int pos, int tupleSize, int primitive, bool isPosition = false);

static Attribute createWithAttributeType(int pos, int tupleSize, int primitiveType, AttributeType attributeType);

};

// …

};

。。。。。

###### （示例）自定义Geometry

自定义几何图形示例展示了如何使用场景视图API创建基于QQuickItem的自定义几何图形。我们将会创建一个BezierCurve项并将其注册到CustomGeometry模块中，然后在QML文件中使用。

BezierCurve类声明

#include <QQuickItem>

class BezierCurve : public QQuickItem

{

Q\_OBJECT

Q\_PROPERTY(QPointF p1 READ p1 WRITE setP1 NOTIFY p1Changed)

Q\_PROPERTY(QPointF p2 READ p2 WRITE setP2 NOTIFY p2Changed)

Q\_PROPERTY(QPointF p3 READ p3 WRITE setP3 NOTIFY p3Changed)

Q\_PROPERTY(QPointF p4 READ p4 WRITE setP4 NOTIFY p4Changed)

Q\_PROPERTY(int segmentCount READ segmentCount WRITE setSegmentCount NOTIFY segmentCountChanged)

public:

BezierCurve(QQuickItem \*parent = 0);

~BezierCurve();

virtual QSGNode \*updatePaintNode(QSGNode \*oldNode, UpdatePaintNodeData \*);

QPointF & p1() const { return m\_p1; }

void setP1(const QPointF &p);

QPointF & p2() const { return m\_p2; }

void setP2(const QPointF &p);

QPointF & p3() const { return m\_p3; }

void setP3(const QPointF &p);

QPointF & p4() const { return m\_p4; }

void setP4(const QPointF &p);

int segmentCount() const { return m\_segmentCount; }

void setSegmentCount(int count);

signals:

void p1Changed(const QPoinfF &p);

void p2Changed(const QPoinfF &p);

void p3Changed(const QPoinfF &p);

void p4Changed(const QPoinfF &p);

void segmentCountChanged(int count);

private:

QPointF m\_p1;

QPointF m\_p2;

QPointF m\_p3;

QPointF m\_p4;

int m\_segmentCount;

};

BezierCurve类继承了QQuickItem类，并且添加了五个属性。其中四个用于贝塞尔曲线控制点，还有一个属性属性用于指定曲线拆分的份数。

每一个属性我们都要设置获取和赋值的方法，以及属性值改变的信号。

QML场景和场景视图的渲染同步点实在虚函数QQuickItem::updatePainNode()中完成的，所有自定义可视场景视图项都要重载该函数。

QSGNode \*updatePaintNode(QSGNode \*oldNode, UpdatePaintNodeData \*updateData);

注意：场景视图在很多平台上都会在单独的线程中进行渲染。因此很关键的一点就是要能够掌控场景视图的行为，一定要在QQuickItem::updatePaintItem()函数中进行场景视图的渲染。

BezierCurve类实现

BezierCurve::BezierCurve(QQuickItem \*parent) : QQuickItem(parent)

, m\_p1(0,0)

, m\_p1(1,0)

, m\_p1(0,1)

, m\_p1(1,1)

, m\_segmentCount(32)

{

setFlag(ItemHasContents, true);

}

BezierCurve构造函数为成员变量设置了默认参数。贝塞尔曲线的坐标默认关联到一个矩形区域。

构造函数中还设置了QQuickItem::ItemHasContents标志位。该标志位告诉画布(canvas)该类中有可视内容，并且当渲染场景视图时，会调用QQuickItem::updatePaintNode()函数和QML场景同步。

BezierCurve::~BezierCurve()

{

}

BerzierCurve没有需要清除的数据成员，所以析构函数中什么也没做。值得提醒的一点是渲染场景视图是通过场景视图自身来管理的，极有可能在不同的线程中，所以不要在一个QQuickItem类中保存QSGNode类的引用，也不要在其中显式地清除这些节点类。

void BezierCureve::setP1(const QPointF & p)

{

if(p == m\_p1)

return;

m\_p1 = p;

emit p1Changed(p);

update();

}

以上是p1属性的设置函数。在属性值发生变化的时候会发出属性值改变的信号。接着调用了QQuickItem::update()函数。这个函数会提示渲染场景视图，该对象的状态已经发生改变，并且需要使用渲染场景视图和QML场景进行同步了。调用udpate()函数将会导致过后QQuickItem::updatePaintNode()函数的调用。

QQuickItem::updatePaintNode()函数是最主要的用于同步渲染场景视图和QML场景的函数。该函数会获取上一次该函数调用时返回的QSGNode节点。如果是第一次调用，该节点将会是空。我们就需要自己创建一个QSGGeomtryNode节点。

QSGNode \*BezierCurve::updatePaintNode(QSGNode \*oldNode, UpdatePaintNodeData \*)

{

QSGGeomtryNode \*node = Q\_NULLPTR;

QSGGeomtry \*geometry = Q\_NULLLPTR;

if(!oldNode) {

node = new QSGGeomtryNode;

我们需要创建一个几何图形QSGGeometry，然后添加到节点中。QSGGeometry类构造函数的第一个参数用于定义顶点类型，称为“attribute set”，属性组。Since the graphics often used in QML centers around a few common standard attribute sets, these are provided by default.我们在这里使用Point2D 属性组，Point2D有两个浮点数，一个表示x坐标，另一个表示y坐标。第二个参数表示顶点的个数。

当然也可以自定义属性组，但是不在本示例讨论范围。

如果我们对QSGGeometry的内存管理没有特殊的需求，应当将其所有权交给QSGGeometryNode。通过QSGGeometryNode::setFlag(QSGNode::OwnsGeometry)设置。

geomtry = new QSGGeomtry(QSGGeomtry::defaultAttributes\_Point2D(),m\_segmentCount);

geomtry->setLineWidth(2);

geomtry->setDrawingMode(QSGGeometry::DrawLineStrip);

node->setGeomtry(geometry);

node->setFlag(QSGNode::OwnsGeometry);

为了尽可能减少内存的分配以及提升性能，也可以将QSGGeometry类作为QSGGeometryNode的子类的成员，在这种情况下我们不必设置QSGGeomtryNode::OwnsGeomtry标志。

场景视图API中提供了一些常用的材质的实现。在本示例中我们使用QSGFlatColorMaterial类材质，该材质将使用固定色彩填充几何图形。定义材质时，我们需要再一次设置材质的所属权归节点所有，这样我们可以靠场景视图自己负责清理工作。

QSGFlatColorMaterial \*material = new QSGFlatColorMaterial;

material->setColor(QColor(250,0,0));

node->setMaterial(material);

node->setFlag(QSGNode::OwnsMaterial);

如果QQuickItem::updatePaintNode()函数不是第一次调用，我们可能只需要对已经存在的节点做适当的修改。我们将oldNode转换为QSGGeometryNode类型节点并获取其几何图形。一旦贝塞尔曲线的片段数量发生了改变，我们就可以调用QSGGeometry::allocate()来保证该几何图形拥有正确的节点数量。

} else {

node = static\_cast<QSGGeometryNode \*>(oldNode);

geometry = node->geometry();

geometry->allocate(m\_segmentCount);

}

接下来将顶点数据赋给几何图形QSGGeometry。首先我们从一个顶点数组获取数据。如果我们使用的默认属性组，我们可以便捷地使用QSGGeometry::vertexDataAsPoint2D()函数来设置几何图形的顶点数据。然后我们逐片计算其位置和为其赋值。

QRectF bounds = boundingRect(); // boundingRect()是自定义函数，用来指定矩形区域

QSGGeometry::Point2D \*vertices = geometry->vertexDataAsPoint2D();

for(int i = 0; I < m\_segmentCount; ++i) {

qreal t = i / qreal(m\_segmentCount - 1);

qreal invt = 1 – t;

QPointF pos = invt \* invt \* invt \* m\_p1

+ 3 \* invt \* invt \* t \* m\_p2

+ 3 \* invt \* t \* t \* m\_p3

+ t \* t \* t \* m\_p4;}

float x = bounds.x() + pos.x() \* bounds.width();

float y = bounds.y() + pos.y() \* bounds.height();

vertices[i].set(x,y);

}

node->markDirty(QSGNode::DirtyGeometry);

在函数的最后，返回该节点，让场景视图进行渲染。

return node;

}

程序进入点(Application Entry-Point)

int main(int argc, char \*argv[])

{

QGuiApplication app(argc, argv);

qmlRegisterType<BezierCurve>(“CustomItem”,1,0,”BezierCurve”);

QQuickView view;

QSurfaceFormat format = view.format();

format.setSamples(16);

view.setFormat(format);

view.setSource(QUrl("qrc:///scenegraph/customgeometry/main.qml"));

view.show();

app.exec();

}

我们的示例应用是一个很简单的QML应用，使用QGuiApplication和QQuickView加载QML文件即可。我们还将我们的BezierCurve类注册到了CustomItem模块中。贝塞尔曲线是使用GL\_LINE\_STRIP绘制的，我们指定视图使用多重采样以实现反锯齿效果。这并不是必须的，但是这会使曲线有更好的效果。多重采样默认是没有开启的，因为这样需要更高的内存开销。

使用自定义项(Using Custom Item)

import QtQuick 2.7

import CustomItem 1.0

Item {

width: 300

height: 200

BezierCurve {

id: line

anchors.fill: parent;

anchors.margins: 20

property real t

SequentialAnimation on t {

NumberAnimation { to: 1; duration: 2000; easing.type: Easing.InOutQuad }

NumberAnimation { to: 0; duration: 2000; easing.type: Easing.InOutQuad }

loops: Animation.Infinite

}

p2: Qt.point(t, 1 - t);

p2: Qt.point(1 - t, t);

}

}

###### 预处理(Preprocessing)

节点类中有一个虚函数QSGNode::preprocess()，会在场景视图渲染之前调用。节点的子类可以设置QSGNode::UsePreprocess标志以及重载QSGNode::preprocess()函数为节点的渲染做最后的准备。例如，将一个贝塞尔曲线拆分成当前系数下正确的等级，或者更新一个纹理的片段。

###### 节点所有权(Node Ownership)

节点的所有权可以显示地指定，也可以有场景视图通过QSGNode::OwnedByParent标志设置。当场景视图不在GUI线程中时，将所有权交给场景视图是可以实现比较简便的清理方式。

##### 材质(Meterials)

材质描述了几何形状节点QSGGeometryNode内部是如何进行填充的。材质封装了一个OpenGL着色程序，并提供了足够的灵活性来保证可以达到各种效果。尽管大多数Qt Quick项只使用了很基本的材质，如固态色彩(solid color)，纹理填充。

对于那些只想在QML Item类型上添加自定义的着色，可以直接在QML中使用ShaderEffect类型来实现。

以下是完整的材质类列表：

|  |  |
| --- | --- |
| QSGMaterial | 封装了着色器程序的着色状态 |
| QSGMaterialShader | 表示着色器中的OpenGL着色器程序 |
| QSGMaterialType | 结合QSGMaterial使用作为独有的类型标记 |
| QSGFlatColorMaterial | 在场景视图中渲染固定色彩的几何图形的便捷方式 |
| QSGSimpleMaterial | 用来保存在QSGSimpleMaterialShader中使用的状态的通用模板类 |
| QSGSimpleMaterialShader | 适于为场景视图构建基于OpenGL的自定义材质 |
| QSGOpaqueTextureMaterial | 适于渲染场景视图中有纹理的几何图形 |
| QSGTextureMaterial | 同样适于渲染场景视图中有纹理的几何图形 |
| QSGVertexColorMaterial | 适于渲染场景中逐顶点着色的几何图形 |

参考下节《Scene Graph – Simple Material》

##### （示例）简单材质的应用(Simple Material)

在本示例中，我们将使用QSGSimpleMaterialShader类填充场景视图中的几何图形。使用这个类创建材质的时候可以减少很多构建基础样板的代码的编写。大大提高编码效率。

一个简单的材质有两部分组成：材质状态和材质shader。材质的着色器在每一个场景视图中都有一个实例，并且包含了真正的OpenGL着色器程序和相关的属性(attributes)和uniforms。材质的状态是赋给每一个独立节点的，这样就可以给它们设置不同的颜色了。

struct State

{

QColor color;

int compare(const State \*other) const {

uint rgb = color.rgba();

uint otherRgb = other->color.rgba();

if(rgb == otherRgb) {

return 0;

}else if( rgb < otherRgb) {

return -1;

} else {

return -1;

}

}

};

在创建一个材质的时候，我们的第一件事情就是为其创建一个状态类。在本示例中，State类只包含了一个成员变量QColor。此外，类中还定义了一个成员函数，用于场景视图对节点渲染顺序进行重排序。

class Shader : public QSGSimpleMaterialShader<State>

{

QSG\_DECLARE\_SIMPLE\_COMPARABLE\_SHADER(Shader,State);

接下来就是定义材质的另一组成部分——材质shader。通过继承使用State实例化的模板类QSGSimpleMaterialShader。

然后在类中我们使用ASG\_DECLARE\_SIMPLE\_COMPARE\_SHADE宏来声明为该类配置一段底层的代码模板。只要我们的State类有比较成员函数，我们声明的状态就可以进行比较了。还可以通过声明QSG\_DECLARE\_SIMPE\_SHADER宏来代替State::compare()比较函数，但是这样做会降低应用的性能。

状态类被用作模板的参数，用于自动配置QSGMaterialType。所以，将shader和状态类分开，写成不同的类是很重要的。将相同的状态用于多个shader中，可能会导致未定义的行为发生。

public:

const char \*vertexShader() const {

return “attribute highp vec4 aVertex;”

“attribute highp vec2 aTextCoord;”

“uniform highp mat4 qt\_Matrix;”

“varying highp vec2 texCoord;”

“void main() {”

“ gl\_Position = qt\_Matrix \* aVertex;”

“ texCoord = aTexCoord;”

“}”;

}

const char \*fragmentShader() const {

return “uniform lowp float qt\_Opacity;”

“uniform lowp vec4 color;”

“varying highp vec2 texCoord;”

“void main() {”

“ gl\_FragColor = textCoord.y \* texCoord.x \* color \* qt\_Opacity;”

“}”;

}

上面是声明了shader源代码，顶点着色器和片元着色器。我们的简单材质假设在顶点着色器中存在qt\_Matrix，以及在片元着色器中存在qt\_Opacity数据。

QList<QByteArray> attributes() const {

return QList<QByteArray() << “aVertex” << “aTexCoord”;

}

我们重载了attributes函数，用其返回aVertex和aTexCoord属性的名字。这些属性都会映射到0到1之间。

void resolveUniforms() {

id\_color = program()->uniformLocation(“color”);

}

private:

int id\_color;

Uniforms类型属性既可以通过名称，也可以通过索引获取，通过索引比通过名称更快。通过重载resolveUniforms()函数，我们可以找到color uniform属性。我们不必关注如何获取qt\_Opacity和qt\_Matrix属性，这些会通过基类默认处理。

void updateState(const State \*state, const State \*) {

proram()->setUniformValue(id\_color, state->color);

}

每一个不同的状态都会调用一次updateState()函数，并使用当前的颜色更新着色器程序。旧的状态值通过第二个参数传入，这样就可以只在状态发生改变的时候更新状态。每一个节点都会调用一次updateState()函数。

class ColorNode : public QSGGeometryNode

{

public:

ColorNode() : m\_geometry(QSGGeometry::defaultAttributes\_TexturePoint2DD(),4) {

setGeometry(&m\_geometry);

QSGGeometryMaterial<State> \*material = Shader::createMaterial();

material->setFlag(QSGMaterial::Blending);

setMaterial(material);

setFlag(OwnsMaterial);

}

QSGGeometry m\_geometry;

};

ColorNode类用来支持使用该材质来绘制，我们通过继承QSGGeometryNode来自定一个节点。

我们的shader需要位置和纹理坐标，我们使用默认的属性组QSGGeometry::defaultAttributes\_TexturedPoint2D()以及声明组成该几何图形所需的顶点数量。为了避免分配内存，我们让QSGGeometry成为QSGGeometryNode的成员。

当我们使用QSG\_DECLARE\_SIMPLE\_COMPARABLE\_SHADER()宏的时候，它定义了一个createMaterial()函数，用于为State类实例化材质。

如果我们要在为我们的自定义材质中使用透明度，那就需要设置QSGMaterial::Blending标志。场景视图在绘制节点或者重排节点顺序的时候可能会开启或者禁止GL\_BLEND功能。

最后，我们将材质的所有权交给节点，这样就可以避免需要手动进行内存管理的问题了。

class Item : public QQuickItem

{

Q\_OBJECT

Q\_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)

public:

Item (){

setFlag(ItemHasContents, true);

}

QColor color() const { return m\_color; }

void setColor(const QColor & color) {

if(m\_color == color) return;

m\_color = color;

emit colorChanged(m\_color);

update();

}

protected:

QSSGNode \*updatePaintNode(QSGNode \*oldNode, UpdatePaintNodeData \*) {

colorNode \*node = static\_cast< ColorNode\*>(oldNode);

if(oldNode == Q\_NULLPTR) {

node = new ColorNode();

QSGGeometry::updateTextureRectGeometry(node->geometry(), boundingRect(), QRectF(0,0,1,1));

static\_cast<QSGSimpleMaterial<State>\*>(node-material())->state()->color= m\_color;

node->markDirty(QSGNode::DirtyGeometry | QSGNode::DirtyMaterial);

return node;

}

}

signals:

void colorChanged(QColor color);

private:

QColor m\_color;

};

注意：

##### 便捷节点(Convenience Nodes)

场景视图API是非常底层的东西，它更多关注的是性能，而非易用性。编写自定义几何图形和材质，即使是很基本的东西，也需要大量有难度的代码。因此，API中提供了几个便捷的类可以直接用于自定义节点。

1 QSGSimpleRectNode – 一个QSGGeometryNode子类，定义了矩形的几何图形，和固定色彩的材质。

2 QSGSimpleTextureNode – 一个QSGGeometryNode雷子，定义了一个有纹理材质的矩形几何图形。

#### 场景视图和渲染(Scene Graph and Rendering)

场景视图的渲染是在QQuickWindow类中进行的。并且没有共有API接口可以处理它。但是，在渲染管线中有几处地方是可以附着到代码中的。这样可以添加一些自定义的场景视图内容或者渲染器未加工的OpenGL内容。这些整合点是在其他渲染循环中定义的。

关于场景视图渲染器是如何工作的，参考《Qt Quick Scene Graph Render》

场景视图中有三个渲染循环变量：basic,windows和threaded。basic和windows是单线程的，threaded则在专门的渲染线程中执行渲染。Qt会根据平台和图形驱动选择合适的渲染循环。也可以通过QSG\_RENDDER\_LOOP环境变量强制指定使用那种渲染循环。如果要知道是那种渲染循环，可以开启qt.scenegraph.general logging category.

注意：theaded和windows渲染循环需要OpenGL的支持，

### Reference Documentation

其他与Qt Quick相关的内容

Qt Quick C++ Classes – 由Qt Quick模块提供的C++ API接口

Qt Quick QML Types – QtQuick标准库提供的QML类型列表

### Qt Quick Examples and Tutorials

### Qt Quick模块的类型相关

#### Window类型(QQuickWindow类)

Window对象类型可以为Qt Qucik场景创建新的顶级窗口。它会自动建立使用QtQuick 2.x模块图形类型的窗口。导入语句：

import QtQuick.window 2.2

如果忽略这句导入语句，将会建立一个不能使用该类型窗口特性的QML环境。

一个Window对象可以嵌入到一个项或者另外的Window对象中；嵌入的Window对象会自动变成为外部窗口的建立的零时窗口：意思是，嵌入的窗口会在显示时自动显示在外部窗口的中央，在不同的平台上可能还会有其他行为，这取决于flags属性。例如嵌入的窗口会显示成一个对话框，那么应该将其flags属性设置为Qt.Dialg标志，表示该窗口以对话框形式显示，因为有些窗口管理系统没有设置的标志，该窗口可能无法显示。你也可以在一个低级为QtObject的对象中声明多个窗口，这样这些窗口之间就没有临时关系。

可以通过为窗口设置或者绑定x,y属性动态确定Window对象在屏幕上的位置。

当用户尝试关闭窗口时，会发出closing信号。你可以通过实现onClosing函数和设置close.accepted = false，来强制让窗口保持显示状态。

#### ApplicationWindow类型(继承Window类型)

ApplicationWindow类型是派生子Window类型，添加了窗口常用的项，如MenuBar，ToolBar和StatusBar等与平台无关的行为。

注意：默认情况下ApplicationWindow是不可见的。

## Qt QML 模块

Qt QML模块提供了使用QML语言进行应用开发的QML库和框架。该模块定义并实现了QML语言和语言引擎，并且提供了接口能够使用自定义的类型扩展QML语言，QML代码可以整合Javascript和C++代码。Qt QML模块提供了QML API和C++ API。

Qt QML模块和Qt Quick模块的关系是：Qt QML模块提供了QML语言和语言引擎，而Qt Quick模块则提供了用于开发应用界面的各种可视组件，模型/视图，动画框架，以及其他用于创建界面的类型。

新接触QML和Qt Quick的人，建议参考《QML Application》文档。

### Getting Started

要在C++代码中使用Qt QML模块提供的C++接口，使用下面的指令：

#include <QtQml>

要在QML代码中使用Qt QML模块提供的QML接口，使用下面的指令：

import QtQml 2.0

要链接Qt QML模块，需要在qmake的.pro文件中添加该库链接符号：

QT += qml

### QML and QML Types

Qt QML模块中包含了QML语言框架以及重要的QML类型。QML语言的结构在《The QML Reference》文档中有详细描述。

除了QML基础类型之外，该模块还有一下QML对象类型：

《Component》

《QtObject》

《Binding》

《Connections》

《Timer》

全局对象《Qt》为各种QML类型提供了许多重要的枚举和函数。

#### Lists and Models

模型类型在QtQml.Models模块中，参考《Qt QML Models》文档了解更多内容。

### JavaScript Environment for QML Applications

在QML代码中使用JavaScript语句可以为QML提供代码的逻辑。Qt QML模块提供了能够在QML代码中运行JavaScript的框架，这些框架是有C++完成的。参考《The QML Reference》文档了解这部分内容。

### Integrating QML with C++ Applications

Qt QML模块自然也有支持运行QML应用的框架。该模块使得QML代码能够嵌入JavaScript代码，以及和C++代码进行交互。参考：

《Important C++ Classes Provided By The Qt QML Module》

《Integrating QML and C++》

#### Qt QML模块中重要的C++类(Important C++ Classes Provided By The Qt QML Module)

Qt QML模块提供了诸多C++类来实现QML框架。客户端可以使用这些类和QML运行时交互（例如数据传输或者调用对象方法），以及初始化QML文档中的对象树。Qt QML模块提供的C++类远不止此节提到的，然而这里介绍的这些是QML运行时的基础和QML的核心概念。

##### QML运行时

一个典型的QML应用，在C++程序进入点将会初始化一个QQmlEngine对象，然后使用QQmlComponent对象加载一个QML文档。该QML引擎会提供一个默认的QQmlContext对象作为顶级的计算上下文(evaluation context)来计算QML文档中的函数和表达式。在QML文档中的对象树的实例化是通过调用QQmlComponent的create函数进行的，一定要确保在文档加载过程中没有发生错误，所以加载QML文档后要做加载成功与否的判断。

客户端可能需要修改QML引擎提供的QQmlContext，向该上下文添加属性或者对象。客户端可以调用QQmlEngine::rootContext()函数来调用顶层上下文。

在实例化对象树之后，客户端可能要通过应用的事件循环来控制这些对象，使其可以处理用户输入。

**注意：**Qt Quick模块中有QQuickView类，该类提供了QML运行时和一个用于QML应用显示的视图窗口。

###### QQmlEngine类

###### QQmlContext类

QQmlContext类提供了用于对象初始化和表达式计算的上下文。所有的对象都是在特定的上下文中初始化的，以及所有表达式都是在程序运行的特定上下文中进行计算的。上下文定义了符号如何解析，以及表达式的值是如何进行计算的。

上下文允许通过QML引擎实例化的数据提供给QML组件。每一个QQmlContext都包含了一系列的属性，不同于它的QObject属性，QQmlContext的属性允许其数据通过名称显式地绑定到上下文。上下文属性通过调用QQmlContext::setContextProerty()函数定义和更新。下面的例子示例了将一个Qt模型绑定到一个上下文中，然后在QML文件中调用：

QQmlEngine engine;

QStringListModel modelData;

QQmlContext \*context = new QQmlContext(engine.rootContext());

context->setContextProperty(“myModel”,&modelData);

QQmlConponent component(&engine);

component.setData(“import QtQuick 2.0\nListView{model:myModel}”,QUrl());

QObject \*window = component.create(context);

需要注意的是创建QQmlContext的对象在不使用的时候要负责删除该对象。如上例，当context对象不再使用时，window对象被销毁时，context对象必须被显式地销毁，最简单的方式是将window设为context对象的父对象。

为了简化绑定和维护更大的数据，一个上下文对象可以放到QQmlContext类对象中。该上下文对象的属性可以通过名称进行操作，尽管它们是独自通过QQmlContext::setContextProerty()函数添加到上下文中的。属性的值发生改变可以通过提醒信号发现。设置一个上下文对象比手动添加和管理上下文属性值要更快更简单。下面的例子和上一例的效果是一样的，但是下例使用了上下文对象：

class MyDataSet : … {

…

Q\_PROPERTY(QAbstractItemModel \*myModel READ model NOTIFY modelChanged)

…

};

MyDataSet myDataSet;

QQmlEngine engine;

QQmlContext \*context = new QQmlContext(engine.rootContext());

context->setContextObejct(&myDataSet);

QQmlConmponent component(&engine);

component.setData(import QtQuick 2.0\nListView { model: myModel });

component.create(context);

将上下文属性放到一个上下文对象中，然后将对象放到QQmlContext对象中，就可以统一将所有上下文属性一起放入QQmlContext中了。

上下文层次(Context Hierarchy)

上下文以层次方式组织。层次的根就是QML引擎的rootContext。子上下文从父上下文中继承上下文属性。如果子上下文的上下文属性在父上下文中已经存在了，子上下文会重载父上下文的属性。

下面的例子中有两个上下文-context1和context2.第二个上下文重载了从第一个上下文继承的“b”上下文属性：

QQmlEngine engine;

QQmlContext \*context1 = new QQmlContext(engine.rootContext());

QQmlContext \*context2 = new QQmlContext(context1);

context1->setProperty(“a”,12);

context1->setProperty(“b”,12);

context2->setContextProperty(“b”,15);

### 其他框架(Additional Frameworks)

《The Declarative State Machine Framework》

### Licenses and Attributes

### QML类型系统

##### 基本类型

##### JavaScript类型

##### QML对象类型

###### 使用QML文档定义对象类型

###### 使用Component定义无名类型(Anonimous Types)

另一种在QML中创建对象类型的方式是使用Component类。使用Component可以不必使用单独的QML文档，直接在使用类型的QML文档中定义对象类型：

Item {

id: root

width: 500; height: 500

Component {

id: myComponent

Rectangle { width: 100; height: 100; color: “grey”}

}

Component.onCompleted: {

myComponent.createObject(root);

myComponent.createObejct(root,{“x”:200});

}

}

myComponent对象创建了一个匿名类型，并通过Component::createObject()方法实例化了两个该匿名类型对象。

Inline components share all the characteristics of regular top-level components and use the same import list as their containing QML document.

注意：每一个Component对象会创建自己的组件作用域。任何在该Component对象的作用域中声明和使用的id必须是唯一的。但是这个id在整个QML文档中可以不是唯一的。即在Component类对象中有对象使用了id:root，在该Component类对象外，还是可以使用id:root。

详解(Detailed Description)

在QML中组件是一类可重用，接口定义封装良好的QML类型。

组件通常在QML文档中定义。Component类一般都是在一个存在的QML文档中定义，而不是单独使用一个QML文档。这样做的好处是，将QML文档中较小的、或者独立的组件声明成Component类型，使其能够重用或者在代码逻辑中定义和使用。

下面的例子中有一个Component对象，它包含了一个自对象，有多个Loader对象使用了该组件对象：

Item {

width: 100; height: 100

Component {

id: redSquare

Rectangle {

color: “read”

width: 10; height: 10

}

}

Loader { sourceComponent: redSquare }

Loader { sourceComponent: reaSquare; x:20 }

}

注意：尽管Rectangle类型会自动渲染和显示，但并不试用于上面的情况。一个组件封装的QML类型，如果它们是被定义在另一个QML文档中，只有在使用该组件的时候，组件和QML类型才会加载。因为组件并没有在该QML文档的对象树中，你不能使用锚将其他Item绑定到组件上。

## Qt GUI 模块

### 本模块简介

Qt GUI模块提供了窗口系统整合，事件处理，OpenGL和OpenGL ES整合，2D图形，基本图像，字体和文本相关的类。这些类通常是用于Qt的用户界面接口技术，当然也可以直接使用底层的OpenGL ES图形API来实现。

应用开发者使用顶层的用户界面接口，如Qt Quick更合适一些，而不是直接使用该模块的这些功能。

要使用该模块的类，需要在源文件中使用以下指令：

#include <QtGUI>

如果是使用qmake来开发Qt应用，工程将默认包含该模块。如果不需要该模块，可以在.pro文件中去卸载该模块：

QT -= gui

#### 应用窗口(Application Windows)

在Qt GUI模块中，QGuiApplication和QWindow是最重要的两个类。Qt应用想在屏幕上显示图形内容，就必须要使用到它们。QGuiApplication包含了主事件循环，所有事件都是在该循环中处理和调度的。同时它还管理了应用的初始化和结束的工作。

QWindow类则代表了一个窗口对应的底层窗口系统。它提供了大量虚函数处理来自窗口系统的事件(QEvent)，例如点击输入，聚焦，按键，位置变化，显示等等。

#### 2D图形(2D Graphics)

Qt GUI模块中包含了支持2D图形、图像，字体和高级排版格式的类。

使用QSurface::RasterSurface类型创建的QWindow可以和Qt的高度可定制的2D矢量图形API ，QBackingStore和QPainter类相结合使用。QPainter类支持绘制直线、多边形、矢量路径、图形和文本。更多信息，参考《Paint System》和《Raster Window Example》

Qt可以使用QImage和QPixmap类加载和保存图像。默认情况下，Qt支持大多数常用的图像格式。开发者可以使用QImageIOPlugin类来支持其他Qt不支持的图像格式。参考《Reading and Writing Image Files》

Qt中的排版是QTextDocument类使用QPainter类API，然后结合Qt的字体类，主要是QFont类来实现的。除此之外，应用还需要很多底层的接口来处理文本和字体，如QRawFont和QGlyphRun等类。

#### 整合OpenGL和OpenGL ES

QWindow支持使用OpenGL或者OpenGL ES渲染，具体使用哪一个取决于实际运行平台。要使用OpenGL进行渲染，需要将QWindow的表面类型设置为QSurface::OpenGLSurface，使用QSurfaceFormat类选择格式属性，然后使用创建QOpenGLContext类来管理本地OpenGL上下文。另外，Qt中有一个QOpenGLPaintDevice类，能够启用OpenGL加速了的QPainter渲染，同时还有许多能够简化OpenGL代码编写和隐藏扩展处理的复杂性，也隐藏了OpenGL和OpenGL ES 2之间的差别。有些类包含了QOpenGLFunctions类，使得开发者能够在桌面开发应用中使用所有OpenGL ES 2的函数，这样开发好的应用可以直接移植到移动平台。还有的类将OpenGL功能封装成更为简单的Qt API，如：

1 QOpenGLBuffer

2 QOpenGLFrameBufferObejct

3 QOpenGLShaderProgram

4 QOpenGLTexture

5 QOpenGLDebugLogger

6 QOpenGLTimerQuery

7 QOpenGLVertexArrayObject

最后，为了对新版本的OpenGL和OpenGL ES 提供更好的支持，Qt还支持版本函数封装机制：QOpenGLFunction\_N\_N类族可以对应不同版本的OpenGL函数。参考《OpenGL Window Example》

QSurface::OpenGLSurface类型的QWindow可以结合QPainter和QOpenGLPaintDevice类使用OpenGL硬件加速2D图形的绘制，只需要损失一点视觉上的质量。

#### Qt 5.0之前的Qt GUI模块

。。。。。。

#### Drag and Drop

。。。。。。

### Paint System

Qt的绘图系统可以使用相同的API在屏幕，打印机等不同平台上绘图，这些功能主要基于QPainter,QPaintDevice,QPaintEngine类实现。

QPainter类是用于执行绘制操作的，QPaintDevice是一个抽象的二维空间，表示QPainter可以绘制的目标。QPaintEngine则为QPainter类提供类绘制图形的接口。QPaintEngine类在QPainter和QPaintDevice内部使用，并对开发者隐藏了，除非开发者要定义自己的设备类型。



这样的好处是所有的绘制操作都使用相同的绘制管线，这样可以很容易为不支持的设备添加新的特性。

#### 绘制设备和后端(Paint Devices and Backends)

##### 创建绘制设备(Creating a Paint Device)

QPaintDevice类是所有可被绘制的设备的基类。QPainter类可以在任何QPaintDevice的子类上绘制图形。通过实现QWidget,QImage,QPixmap,QPicture,QPrinter和QOpenGLPaintDevice，QPaintDevice可以兼容很多绘制情况。。。

Widget

### （示例）OpenGL Window Example

**注意：**本示例演示了使用QWindow和OpenGL等底层接口的使用方法。在实际开发过程中应该考虑使用更高层的QOpenGLWindow类。

#### OpenGLWindow父类

我们首先定义一个OpenGLWindow类作为来提供通用接口，然后通过继承该类来负责实际的渲染。该类拥有响应render()函数调用请求的函数，可以是立即响应渲染请求的rendNow()函数，也可以是在一次事件循环结束时才调用的renderLater()函数。OpenGLWindow的子类可以为基于OpenGL的渲染重载render()，或者使用QPainter进行渲染。如果底层OpenGL驱动开启了垂直同步(vertical sync)，可以使用OpenGLWindow::setAnimating(true)为render()函数设置垂直刷新速率。

在使用OpenGL函数来渲染的类中，一般需要继承QOpenGLFunctions类。正如我们的OpenGLWindow类要使用与平台无关的OpenGL ES 2.0的函数。通过继承QOpenGLFucntions类，类中包含的OpenGL函数会根据优先级来使用，开发这不必担心自己的应用是否在使用OpenGL或者OpenGL ES 2.0工作。

OpenGLWindow的定义：

class OpenGLWindow : public QWindow, protected QOpenGLFunctions

{

Q\_OBJECT

public:

explicit OpenGLWindow(QWindow \*parent = 0);

~OpenGLWindow();

virtual render(QPainter \*painter);

virtual render();

virtual void initialize();

void setAnimating(bool animating);

public slots:

void renderLater();

void renderNow();

protected:

bool event(QEvent \*event) override;

void exposeEvent(QExposeEvent \*event);

private:

bool m\_animating;

QOpenGLContext \*m\_context;

QOpenGLPaintDevice \*m\_device;

};

窗口的表面类型必须设置为QSurface::OpenGLSurface来指明该窗口是用于OpenGL渲染，而不是使用QPainter在QBackingStore中渲染光栅化内容。

OpenGLWindow::OpenGLWindow(QWindow \*parent)

: QWindow(parent)

, m\_animating(false)

, m\_context(0)

, m\_device(0)

{

setSurfaceType(QWindow::OpenGLSurface);

}

任何OpenGL的初始化都可以在initialize()函数中在render()函数调用之前，使用当前的QOpenGLContext来初始化。在下面的代码片段中可以看到，默认情况下render(QPainter\*)和initialize()函数体中是空的，同时在render()函数的实现中初始化了一个QOpenGLPaintDevice，然后接着调用了render(QPainter\*)函数。

void OpenGLWindow::render(QPainter\* painter)

{

Q\_UNUSED(painter);

}

void OpenGLWindow::initialzie()

{

}

void OpenGLWindow::render()

{

if(!m\_device)

m\_device = new QOpenGLPaintDevice;

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);

m\_device->setSize(size());

QPainter painter(m\_device);

render(&painter);

}

renderLater()函数则只需要调用QWindow::requestUpdate()函数即可，该函数请求的渲染，会在系统准备重新绘制的时候进行。

当我们的应用接收到窗口暴露的事件时，就会调用renderNow()函数。exposeEvent()函数是对窗口暴露与否，即在屏幕上是否可见的事件的响应。当窗口暴露事件发生时，可以通过QWindow::isExposed()来查询是暴露显示窗口还是隐藏不可见窗口。在窗口接收到其第一个暴露事件之前，不应该渲染或者调用QOpenGLContext::swapBuffers()作用于该窗口，因为在此之前可能不知道窗口的大小，而且还有可能在屏幕上的渲染还没有结束。

void OpenGLWindow::renderLater()

{

requestUpdate();

}

void OpenGLWindow::event(QEvent\* event)

{

switch(event->type) {

case QEvent::UpdateRequest:

renderNow();

return true;

default:

return QWindow::event(event);

}

}

void OpenGLWindow::exposeEvent(QExposeEvent\* event)

{

Q\_UNUSED(event);

if(isExposed())

renderNow();

}

在renderNow()函数中，如果当前暴露的的不是我们的窗口，我们就会延迟渲染，直到获取到了一个暴露消息。如果我们没有创建OpenGL上下文，我们就需要在相同QSurfaceFormat的OpenGLWindow中创建QOpenGLContext，并为子类调用initialize()函数，以及为QOpenGLFunctions父类调用initializeOpenGLContext()以将其与QOpenGLContext相关联。在任何情况下，我们都使用QOpenGLContext::makeCurrent()函数来创建当前的上下文，以及调用render()函数来进行实际的渲染，最后将OpenGLWidnow作为参数来调用QOpenGLContext::swapbuffers()，并安排显示渲染的内容。

Once the rendering of a frame using an OpenGL context is initiated by calling QOpenGLContext::makeCurrent(), giving the surface on which to render as a parameter, OpenGL commands can be issued. The commands can be issued either directly by including <qopengl.h>, which also includes the system's OpenGL headers, or as by using QOpenGLFunctions, which can either be inherited from for convenience, or accessed using QOpenGLContext::functions(). QOpenGLFunctions gives access to all the OpenGL ES 2.0 level OpenGL calls that are not already standard in both OpenGL ES 2.0 and desktop OpenGL. For more information about the OpenGL and OpenGL ES APIs, refer to the official OpenGL Registry and Khronos OpenGL ES API Registry.

如果使用OpenGLWindow::setAnimating(true)开启了动画，我们就可以调用renderLater()来安排新的刷新请求了。

void OpenGLWindow::renderNow()

{

if(!isExposed())

return;

bool needsInitialize = false;

if(!m\_context) {

m\_context = new QOpenGLContext(this);

m\_context->setFormat(requestFormat());

m\_context->create();

needsInitialize = true;

}

m\_context->makeCurrent(this);

if(needsInitialize) {

initialzieOpenGLFunctions();

initialize();

}

render();

m\_context->swapBuffers(this);

if(m\_animating)

renderLater();

}

开启动画也会发出新的更新请求：

void OpenGLWindow::setAnimating(bool animating)

{

m\_animating = animating;

if(animating)

renderLater();

}

#### 创建子类示例

该子类用于展示如何使用OpenGL来绘制一个三角形，通过间接继承QOpenGLFunctions，我们可以调用所有OpenGL ES 2.0的OpenGL功能。

class TriangleWindow : public OpenGLWindow

{

public:

TriangleWindow();

void initialize() override;

void render() override;

private:

GLuint m\_posAttr;

GLUint m\_colorAttr;

GLuint m\_matrixUniform;

QOpenGLShaderProgram \*m\_program;

int m\_frame;

};

TriangleWindow::TriangleWindow()

: m\_program(0)

, m\_frame(0)

{

}

我们在main函数中初始化QGuiApplication以及实例化TriangleWindow。我们给窗口的QSurfaceFormat指明我们采用多重采样以及采样次数。我们还可以使用我们类中的setAnimating()函数来开启动画。

int main(int argc, char\* arg[])

{

QGuiApplication(argc, argv);

QSurfaceFormat format;

format.setSample(16);

TriangleWindow window;

window.setFormat(format);

window.resize(640,480);

window.show();

window.setAnimating(true);

return app.exec();

}

下面的是OpenGL着色器的代码。顶点着色器和片源着色器都非常简单，进行顶点转换以及做颜色线性插值。

static const char \*vertextShaderSource =

“attribute highp vec4 posAttr;\n”

“attribute lowp vec4 colAttr;\n”

“varying lowp vec4 col;\n”

“uniform hhighp mat4 matrix;\n”

“void main(){\n”

“ col = colAttr;\n”

“ gl\_Position = matrix \* posAttr; \n”

“};\n”;

static const char \*fragmentShaderSource =

“varying lowp vec4 col;\n”

“void main() {\n”

“ gl\_FragColor = col;\n”

“};\n”;

下面是加载着色器和初始化着色器程序。我们使用的是QOpenGLShaderProgram来代替原生的OpenGL，这样可以便捷地将highp,mediump和lowp声明从桌面OpenGL中剥离，它们不属于标准的一部分。我们将属性和uniform位置存放在成员变量中，这样就不用每一帧都要查询顶点位置。

void TriangleWindow::initialize()

{

m\_program = new QOpenGLShaderProgram(this);

m\_program->addShaderFromSourceCode(QOpenGLShader::Vertex,vertextShaderSource);

m\_program->addShaderFromSourceCode(QOpenGLShader::Fragment,fragmentShaderSource);

m\_program->link();

m\_posAttr = m\_program->attributeLocation(“posAttr”);

m\_matrixUnifrom = m\_program->uniformLocation(“matrix”);

}

最后，我们将为子类实现render()函数，我们通过OpenGL建立视口，清除背景，以及渲染一个旋转的三角形。

void TriangleWindow::render()

{

const qreal retinaScale = devicePixelRatio();

glViewPort(0,0,width() \* retinaScale, height() \* retinaScale);

glClear(GL\_COLOR\_BUFFER\_BIT);

QMatrix4x4 matrix;

matrix.perspective(60.0f,4.0f/3.0f,0.1f,100.0f);

matrix.translate(0,0,-2);

matrix.rotate(100.0f \* m\_frame / screen()->refreshRate(), 0,1,0);

m\_program->setUniformValue(m\_matrixUniform,matrix);

GLfloat vertices[] = {

0.0f,0.707f,

-0.5f,-0.5f,

0.5f,-0.5f

};

GLfloat colors[] = {

1.0f,0.0f,0.0f,

0.0f,0.0f,0.0f,

0.0f,0.0f,1.0f

};

glVertexAttributePointer(m\_posAttr,2,GL\_FLOAT,GL\_FALSE,0,vertices);

glVertexAttributePointer(m\_coloAttr,2,GL\_FLOAT,GL\_FALSE,0,colors);

glEnableVertexAttributeArray(0);

glEnableVertexAttributeArray(1);

glDrawArrays(GL\_TRIANGLES,0,3);

glDisableVertexAttribArray(1);

glDisableVertexAttribArray(0);

m\_program->release();

++m\_frame;

}

## Qt Network 模块

使用Qt Network模块可以编写TCP/IP客户端和服务器应用。该模块既提供了底层类如QTcpSocket、QTcpServer以及QUdpSocket类，也提供了顶层的类如QNetworkRequest、QNetworkReply和QNetworkAccessManager类，使用常用的网络协议执行网络操作，这些只提供了通用的或者顶层概念的类、

## The QML Language Reference

QML是一种用于编写具有高动态性应用的多范式(multi-paradigm)编程语言。使用QML，应用构建各个块，如通过声明建立的UI组件，各种属性则用于定义应用的行为。应用的行为还可以通过JavaScript进一步定义。在QML中还可以直接使用其他诸多Qt特性。

《The QML Reference》指南讲解了关于QML语言的各种特性。其中介绍的许多QML类型来源于Qt QML模块和Qt Quick模块。

### QML语法基础

QML语言是一种多范式编程语言，通过指定对象的属性，与其他对象的关系，以及对属性改变的响应来定义一个对象。不同于纯粹的命令式代码，属性或者行为的改变需要通过一系列语句进行一步步地处理。QML的声明式语法直接将属性和行为的改变之间整合到一个对象的定义当中。这些属性定义可以在使用的过程中包含命令式代码，在这种情况下就需要复杂的自定义应用行为。

QML源代码一般都是QML引擎通过加载QML文档进行解析的。使用QML文档编写一个单独的QML类型，可以提高QML代码的重用率。

#### 导入语句

QML文档中的导入语句告诉QML引擎该文档使用了哪些模块，JavaScript文件或者其他组件的路径。文档中能够使用那些类型，取决于该文档导入的模块，JavaScript文件以及其他资源和路径。

##### 导入方式

一个QML文档的第一行可能会有一行或者多行导入语句。导入语句有三种方式：

1 模块（名称空间）导入。

2 目录导入。

3 JavaScript资源文件导入。

###### 模块导入

模块导入是最常用的导入方式。客户端可以导入QML模块。这些模块在给定的名称空间中定义了QML对象类型或者JavaScript资源。

模块导入的通用格式如下：

import <模块声明符> <主版本号.子版本号> [as <修饰符>]

<模块声明符> 是使用带点的URI表示法定义的声明符。模块声明符能够唯一地表示该模块的名称空间。

<主版本号.子版本号> 定义了该导入的模块中那些对象类型和JS资源是可以使用的。

<修饰符> 是一个可选的参数。它使用来定义导入到当前文档的模块中的类型和JS资源将被安装到的新的局部名称空间。如果忽略该声明，模块中的类型和JS资源将安装到全局名称空间中。

一个导入唯一的模块的例子：

import QtQuick 2.0

该导入语句使得开发者能在QML文档中使用QtQuick模块提供的所有类型。由于没有使用名称空间修饰符，我们能够直接使用这些类型。

下面是使用了名称空间修饰符的导入语句：

import QtQuick 2.0 as QtQuick

这种导入方式允许多个包含同种类型定义的模块分别使用不同的名称空间同时导入和使用这些定义名称冲突的类型。使用名称空间修饰符导入后，使用类型时，必须在其前面加上名称空间修饰符。以保证引擎能正确解析该类型。

使用修饰符导入模块的例子：

import QtQuick 2.0 as QtQuick

QtQuick.Rectangle {

width:200

height:200

color:”red”

}

注意，如果在QML文档中在没有导入模块，而又企图使用该模块中的类型，将会发生错误。例如，在没有导入QtQuick模块的情况下，是不能使用该模块的Rectangle类型的。

当这种错误发生的时候，QML引擎会发出一个错误，并拒绝加载该QML文档。

无名称空间导入(Non-module Namespace Imports)

类型也可以直接通过C++中的各种类型注册函数直接将类型安装到名称空间中。以这种方式将类型注册到一个名称空间后，如果该名称空间是一个模块声明符，则可以通过导入语句将该类型导入到QML文档中，然后使用该类型。

这是一种最常用的将自定义的C++类型注册到QML类型系统中，成为自定义的QML类型的方式。

导入到局部名称空间(Importing into a Qualified Local Namespace)

import语句还可以选择添加as关键字来声明将导入的类型导入到该QML文档中的局部名称空间中，以解决不同导入模块间可能的类型名字冲突。当使用这种方式之后，在QML文档中使用的所有该模块类型都必须加上该局部名称空间作为类型的前缀。如果没有使用，将会发生错误。

import QtQuick 2.0 as CoreItems

CoreItems.Rectangle {

width:200;height:200

CoreItems.Text { text: “Hello!” }

// Rectangle { width:100; height:100; anchors.centerIn:parent }

}

在上面的例子中，QtQuick模块中的类型都被导入到了局部名称空间CoreItems中，所以在QML代码中使用该模块中的类型时，必须加上该局部名称空间作为类型的前缀。上面注释中的Rectangle类型因为没有添加前缀，所以是错误的。

使用有名称空间的导入方式一个很大的作用就是解决不同导入模块中同名类型的冲突问题。

import QtQuick 2.0 as CoreItems

import “../MyRect” as MyItems

CoreItems.Rectangle {

width:200; height:200

MyItems.Rectangle {

width:100; height:100

anchors.centerIn:parent

}

}

如上，导入的两个模块中都有Rectangle类型，未解决类型名称冲突，就是用有名称空间的导入方式，然后在使用不同模块中的类型时，在类型前加上所属名称空间作为前缀，就可以明确，使用的是那个Rectangle了。

注意：不同的模块在导入时还可以同时导入到相同的名称空间，正如不同模块都导入到全局名称空间一样。

import QtQuick 2.0 as Project

import QtMultimedia 5.0 as Project

Project.Rectangle {

width:100; height:100

Project.Audio {

source: “music.wav”

autoPlay: true

}

}

###### 路径导入

一个包含了其他QML文档的目录也可以直接导入到QML文档中。这样就可以很容易将那些可重用的QML类型放置在一起，然后在使用时直接导入即可。

这种导入方式的格式是：

import “<目录路径>” [as <修饰符>]

**注意：**导入目录是网络透明的，意思是：应用可以像从本地导入文档一样从远端导入文档。查看《Network Tranparency》了解在QML文档中的通用URL解析规则。如果目录是在远端的，则必须包含目录导入列表qmldir文件，如果没有包含qmldir文件，则QML引擎无法获取远程目录中的内容。

导入QML文档目录

本地的QML文件可以直接导入而不必进行其他额外配置。远程的QML文件也可以导入，但需要先配置qmldir文件。本地的目录也可以包含qmldir文件，用于定义哪些类型、哪些JavaScript资源可以导入到客户端。

导入本地目录

本地QML文件能够通过import语句导入本地包含QML文件的目录。这样就可以直接使用放置在这些目录下的QML类型了。

如果导入的目录下存在qmldir文件，在qmldir文件中声明了类型名的QML文档才可以被导入使用；否则，他们只能通过QML文档的文件名派生出来的类型名进行使用。如果没有使用qmldir文件，则只有文件名以大写字母开始，以.qml作为后缀的文件可以直接作为类型来使用。

举例来说：

假设有如下的QML工程目录结构。在顶层的目录myapp下，有一些UI组件保存在mycomponents目录下，另外的应用主要代码保存在main目录下：

myapp

| - mycomponents

| - CheckBox.qml

| - DialogBox.qml

| - Slider.qml

| - main

| - application.qml

main/application.qml文档可以导入mycomponents目录，然后就可以在该文档中使用mycomponents下的QML对象类型了：

import “../mycomponents”

DialogBox {

CheckBox {

// …

}

Slider {

// …

}

}

当然，还可以将该导入的目录下的类型安装到一个局部名称空间中，然后在使用这些类型的时候加上名称空间修饰符即可：

import “../mycomponents” as MyComponents

MyComponents.DialogBox {

// …

}

导入远程目录

如果远程目录下包含有qmldir文件，我们也可以将该远程目录导入到QML文档中。

例如前面的myapp目录存放在“http://www.my-example-server.com”,而子目录mycomponents中包含qmldir文件，文件内容如下：

CheckBox CheckBox.qml

DialogBox DialogBox.qml

Slider Slider.qml

那么，在一个QML文档中就可以导入这些目录，并可以使用在qmldir中指明的类型:

import “http://www.my-example-server.com/myapp/mycomponents”

DialogBox {

CheckBox {

// …

}

Slider {

// …

}

}

需要注意的是，当需要从网络中导入文件时，仅可以导入在qmldir文件中指定的该目录下的QML文件和JavaScript文件。

**警告：**从网络上导入目录时，要确保加载的代码是安全可靠的，以防导入了漏洞或者病毒代码。

目录列表qmldir文件

目录列表qmldir文件显然不同于模块定义qmldir文件，参考《module definition qmldir file》目录列表qmldir文件能够快速和容易地分享共享一组QML文档，但是qmldir文件既不能为QML类型定义名称空间，也不支持QML对象类型的版本声明。

目录列表qmldir文件的语法如下：

|  |  |  |
| --- | --- | --- |
| 命令 | 语法 | 描述 |
| 对象类型声明 | <类型名> <QML文档名> | 将一个QML文档声明为指定名称的类型，然后以该类型名称在QML中使用，例如：RoundeButton RoundeButton.qml |
| 内嵌(internal)对象类型声明 | internal <类型名> <QML文件名> | 内嵌对象类型声明能够将一个QML文档声明为只能在该导入目录下的其他QML文档中使用的类型。例如：  internal HighlightedButton HighlightedButton.qml |
| JavaScript资源声明 | <声明符> <JS文件名> | JavaScript资源声明使得JavaScript文件中的内容可以通过声明符访问。如：  MathFunctions mathFucntions.js |

本地目录可以选择包含qmldir文件，或者不包含。包含时就可以使引擎只加载文件中指定的类型到导入的QML文件中。另外，该目录下的JavaScript资源不会自动提供给客户端，除非在qmldir中声明，或者在QML文件中直接导入。

###### JavaScript资源导入

JavaScript资源可以直接导入到QML文档中。所有导入到QML文档中的JavaScript资源必须要使用声明符。通过声明符才可以访问JavaScript文件中的类型和函数。

导入JavaScript资源的通用格式：

import “<JavaScriptFile>” as <声明符>

注意<声明符>在导入的QML文件中必须是唯一的。这一点不同与模块导入时，可以使用相同的名称空间。

从模块中导入JavaScript资源

JavaScript文件也可以由模块来提供，通过在qmldir文件中添加模块声明符来声明来定义该模块。

例如，如果在qmldir文件中定义了projects.MyQMLProject.MyFunctions模块，并且文件放在QML导入路径中：

module projects.MyQMLProject.MyFunctions

SystemFunctions 1.0 SystemFucntions.js

UserFunctions 1.0 UserFunctions.js

客户端可以将声明为模块的JavaScript资源导入到QML文档中，并可以使用注册到该模块中的类型：

import QtQuick 2.0

import projects.MyQMLProject.MyFunctions 1.0

Item {

Component.onCompleted: { SystemFunctions.cleanUp(); }

}

如果该模块导入时生命了局部名称空间，则在使用模块中的类型时，必须加上该名称空间作为前缀：

import QtQuick 2.0

import projects.MyQMLProject.MyFunctions 1.0 as MyFuncs

import org.example.Functions 1.0 as TheirFuncs

Item {

Component.onCompleted: {

MyFuncs.SystemFunctions.cleanUp();

TheirFuncs.SystemFunctions.shutdown();

}

}

更多关于JavaScript资源的内容可以参考《defining JavaScript resources in QML》以及《importing JavaScript resources in QML》

###### QML导入路径

###### Debugging

#### 对象声明

#### 子对象

#### 注释

### 属性绑定(Property Binding)

一个对象的属性可以被赋予一个静态的值，只在显式重新赋值之后在改变。然而，在QML中，要实现QML对对象动态行为的支持，QML属性还需要支持属性绑定才行。

属性绑定是QML的一个核心特性，能够让不同对象的对象属性关联起来。当一个属性关联的属性值发生了改变，该属性也会自动按照关联规则更新属性值。在场景后端，QML引擎会监测属性间的依赖关系（根据属性的绑定表达式）。当发现属性值变动时，QML引擎会重新计算绑定表达式，并更新属性值。

#### 概述

要创建属性绑定，使用JavaScript表达式计算获得的值为属性赋值。最简单的属性绑定，可能是绑定到另一个属性。例如：下面将子对象的height属性绑定到了父对象的height属性：

Rectangle {

width:200; height:200

Rectangle {

width:100

height:parent.height

}

}

无论什么时候改变父对象的height属性值，子对象的height属性值都会跟着发生改变。

属性绑定可以包含任何合法的JavaScript表达式和语句。

### 整合QML和JavaScript(Integrating QML and JavaScript)

#### 在JavaScript中动态创建QML对象

QML支持在JavaScript代码中动态创建QML对象。这可以通过推迟一些对象的实例化，直到必要时才加载，来加速应用的启动速度。动态创建还支持相应用户的输入或者其他事件。

本节内容可以参考例程《Dynamic Scene example》

##### 动态创建对象(Creating Objects Dynamically)

有两种在JavaScript中动态创建QML对象的方法。一是使用Qt.createComponent()动态创建Component对象，二是使用Qt.createQmlObject()创建对象。当QML文档中已经有Component类型，而你想动态加载该组件时，使用第一种方式比较好。否则，使用第二种方式，传入QML字符串来创建对象是更常用的方式。

###### 动态创建组件(Creating a Component Dynamically)

要动态加载在QML文件中已经定义的组件，需要使用Qt.createComponent()函数。该函数需要QML文件的URL作为参数，然后创建一个Component对象。

通过createComponent创建好一个组件类型后，就可以调用Component::createObject()函数实例化一个组件对象了。createObject()函数有两个参数：

第一个参数是该对象的父类对象。父对象可以是可视对象(QtObejct及其子类)或者不可见对象(QObject及其子类)。需要注意的是，可视对象要设置了可视的父类对象才可以被渲染到场景中。如果希望之后再为该新建的对象设置父对象，可以传入null。

第二个参数可选。它可以使该对象的任意属性的属性-值对(“property”:value)映射。该参数的属性值会在对象创建完成之前获取，要确保属性绑定不会发生错误。相比于在对象创建完之后为属性赋值和绑定属性，这种方式会有一些额外的开销。下面是示例：

首先新建一个QML文档，名为Sprite.qml。该文档定义了将作为组件的QML类型。

import QtQuick 2.0

Rectangle { width: 200; height: 200; color: “red” }

在main.qml文档中，通过调用JavaScript代码创建Sprite对象：

import QtQuick 2.0

import “componentCreattion.js” as MyScript

Rectangle {

id: appWindow

width: 300; height: 300

Component.onCompleted: MyScipt.creationSpriteObjects()

}

下面是componentCreation.js文件的实现。注意，在使用createObejct()创建对象前要使用组件status属性检查组件是否准备好，即是否等于Component.Ready.

var component;

var sprite;

function createSpriteObjects() {

component = Qt.createComponent(“Sprite.qml”);

if(component.status == Component.Ready)

finishCreation();

else

component.statusChanged.connect(finishCreation);

}

function finishCreation() {

if(component.status == Component.Ready) {

sprite = component.createObejct(appWindow,{“x”:100, “y”:100});

if(sprite == null) {

console.log(“Error creating obejct”);

}

}else if(component.status == Component.Error) {

console.log(“Error loading component: ”,component.errorString());

}

}

如果你确定加载的是本地QML文件，那么应该忽略finishCreation()函数调用而直接使用Component::creatObejct()函数。

function createSpriteObjects() {

component = Qt.createComponent(“Sprite.qml”);

sprite = component.createObject(appWindow,{“x”:100, “y”:100});

if(sprite == null) {

console.log(“Error creating object”);

}

}

注意上面两种实例，createObject()调用时都将appWindow作为了其父对象参数。这样一旦动态创建的对象是一个Qt Quick的可视对象，该对象就会成为appwindow的一个子对象，进而可以岁appWindow一起显示出来。

当使用加载文件的方式时，文件的路径应该和Qt.createComponent()方法调用的文件在相同的路径下。

要连接信号到动态对象，需要使用connect()方法。参考《Connecting Singals to Methods and Signals》

实例化组件也可以使用Component::incubateObject()方法。

###### 使用QML代码字符串创建对象(Creating an object from a String of QML)

如果没有提前定义好的QML类，可以使用Qt.createQmlObject()函数直接使用表示QML代码的字符串作为参数来创建QML对象。就像下面这样：

var newObejct = Qt.createQmlObejct(‘import QtQuick 2.0; Rectangle {color:”red”;width:100;height:100}’,parentItem,”errorFilepath”);

第一个参数是用于创建对象的QML代码，其写法和在一个独立的QML文档中是一样的。第二个参数是设置该对象的父对象。第三个参数是用于对象创建错误生成错误报告文件的路径。

### QML类型系统(The QML Type System)

在QML文档中使用的类型，可能有多个来源：

1 QML语言类型；

2 QML模块中通过C++注册的类型；

3 QML模块提供的QML文档；

开发者也可以使用自定义类型，要么通过注册C++类型，要么在QML文档中定义可重用的类型，然后导入使用。

无论这些类型来自于哪里，QML引擎都会保证属性和实例的类型安全。

#### 基本类型(Basic Types)

QML语言支持常用的基本类型，包括整型，双精度浮点数，字符串，以及布尔型。这些类型相对于QML对象类型，对象类型拥有属性，信号，方法等等；基本类型不能声明对象，例如像这样使用：int{}.

基本对象可以：

1 作为一个简单值；

2 包含属性-值对的值。

##### Qt支持的基本类型(Supported Basic Types)

一些基本类型由QML引擎默认支持，而有些则需要使用import语句导入支持类型的模块。下面列出的所有基本属性，除了list和enumeration外，都可以在QML文档中用作属性声明。

list类型必须和QML对象结合使用。

enumeration类型并不能直接声明和使用，必须用已经定义的QML对象类型使用<ObjectType>.<value>的格式使用。

###### QML语言中的基本基本类型(Basic Types Provided By The QML Language)

bool、double、enumeration、int、list、real(相当于float)、string、url、var(通用类型)

###### QML模块提供的基本类型(Basic Types Provided By QML Engine)

QML模块可能会扩展QML语言的基本类型。例如，有QtQuick模块提供的基本类型：

date 时间值

point 包含x，y变量

rect 包含x,y,width,height变量

size 包含width,height变量

Qt全局对象提供了很多有用的操作基本类型的值的函数。

当前只有Qt提供的QML模块才支持它们自己的基本类型，在以后的Qt QML模块中可能发生改变。要使用有特定模块提供的基本类型，必须在QML文档中导入该模块。

##### 改变基本类型的属性行为(Property Change Behavior for Basic Types)

有些基本类型有属性，例如font类型有pixelSize,family以及bold属性。不同与对象类型的属性，基本类型的属性没有属性改变信号。只可能为基本类型创建一个属性改变信号处理函数。

然而，需要注意的是，只要基本类型的属性发生改变，无论是什么，基本类型的变化信号都会发送。

相反地，一个对象类型的属性都会有自己的属性值改变信号，而且对象类型的属性的属性改变信号处理函数只会在属性值发生改变的时候调用。

#### JavaScript 类型

QML引擎支持JavaScript对象和数组。任何标准JavaScript类型都可以使用var类型来表示和存储。

【write with:JavaScript Expressions in QML Documents】

#### QML对象类型

QML对象类型是可以实例化对象的类型。QML对象类型是从QtObejct类派生而来，有QML模块提供。应用可以通过导入相应模块使用模块提供的对象类型。QtQuick模块提供了大多数常用的用于创建用户界面的对象类型。

最后，每一个QML文档都隐式地定义了QML对象类型，这些文档可以在其他QML文档中作为对象类型重复使用。

。。。。。。

## Qt Application

QML是一种声明式编程语言。我们可以用它来描述用户界面的构成方式，以及组成用户界面的视图组件间如何关联，如何相互影响。通过QML，我们能够编写极具易读性的代码，使组件之间可以以动态方式相互联系，使用QML编写的组件同时还具有高度的重用性和可定制性。使用QtQuick模块，UI设计师和程序员可以十分容易地使用QML创建流畅的UI界面动画。还可以选择使用后端的C++库。

### 什么是QML？(what is QML?)

QML是一门编程语言，同时也是Qt应用的图形用户界面规范。程序员和设计师可以使用QML创建同样具有高性能、流畅度好以及极具视觉吸引力的应用。QML具有高度易读，声明式，类JSON语法以及支持直接绑定javascript语言表达式到动态属性的特性。

QML语言以及该语言的引擎架构有Qt QML模块提供。更进一步关于QML语言的内容，参考《Qt QML》文档内容。

下面的内容包含了更多关于QML的信息：

《First Steps with QML》 – 使用QML语言的参考例子

《Creating Qt Quick Projects in Qt Creator》

《The QML Reference》 – QML参考文档

《QML Coding Conventions》

《Glossary of QML Terms》

### 什么是Qt Quick？(what is Qt Quick?)

Qt Quick是QML语言标准库。其中包括了可视类型，交互类型，动画，模型和视图，粒子特效和着色器特效。QML应用开发者，可以通过导入语句import QtQuick x.x使用所有以上类型和功能。

QML的QtQuick库有Qt Quick模块提供。关于Qt Quick支持的QML类型和功能，参考《Qt Quick》内容。

### QML用户界面(QML User Interfaces)

要创建和定制用户界面，Qt Quick提供了各种Qt QML模块提供的标准QML类型之外的可视类型，动画类型。从Qt Creator 2.7开始整合了Qt Quick Designer，并且支持Qt Quick 2。

《Visual types in QML》

《Responding to User Input in QML》

《Animations in QML》

《Displaying Text in QML》

《Layouts in QML》

《Style and Theme Support》

《Integrating JavaScript in QML》

《Scalability》

Buttons,Menus,and other Controls

对于UI中常用的一些控件，如：按键、菜单等等，已经在《Qt Quick Controls》模块中实现。这些控件的具体行为与具体平台有关。

《Qt Quick Controls Overviews》

《Styles》

《Dialogs》

《Layouts》

《Extras》

另一类UI控件，Qt Quick Contorls 2，是设计用于硬件资源有限的嵌入式系统。更多关于这两套UI控件的内容，参考《Differences between Qt Quick Controls》

《Qt Quick Controls 2 Guidelines》

《Qt Quick Tempaltes 2》

### 特效(Speical Effects)

Qt中有几个为应用提供创建特效功能的模块。分别是：

《Particle Effects》

《Graphics Effects》

### 在QML应用中浏览网络内容(Viewing Web Content in QML Application)

QML的WebEngineView和WebView类型可以渲染和显示动态的网页内容。两者的不同之处在于，后者会尝试使用本地的网页视图(web view)来显示内容。这两种类型都可以加载URL和HTML字符串。详细内容参考《Qt WebEngineView》和《Qt WebView》

### Sensors,Gestures, and Touch Interfaces

《Qt Sensors》模块让应用可以从传感器获取信息，如距离传感器和角度传感器。QML API中包含在不同平台中常用的接口，当然还可以使用C++进行扩展。

《Qt Sensors QML Types》

《Qt Sensors Examples》

《Compability Map》 – 列出对不同手机系统平台的支持程度

### 多媒体(Multimedia Content)

《Qt Multimedia》模块能够让应用使用相应的QML类型支持众多的媒体设备和文件。这些QML类型依旧可以通过C++进行扩展。

《Multimedia》

《Audio Overview》

《Video Overview》

《Camera Overview》

《Radio Overview》

《Qt Audio Engine》 – 对3D环绕音效文件的支持和管理

### 移动设备(Mobile Devices)

Qt中还有几个支持网络和移动设备的模块。这些模块提供了对蓝牙，近场通信(Near-Field Communication)，以及GPS的支持。

《Qt Positioning》

《Qt Location》

《Qt Bluetooth》

《Qt NFC》

更多先关内容则可以参考《Networking and Connectivity》和《Mobile APIs》页面。

### Code Samples and Demos

要更进一步了解QML代码，下面有几例关于QML类型的示例代码可供参考。另外，还有几例在应用中使用QML代码的例子。

《Getting Started Programming with Qt Quick》 – 关于创建一个简单的文本编辑器的示例教程。

《Qt Quick Examples and Tutorials》

### 应用开发高级话题(Advanced Application Development Topics)

《Integraing QML and C++》

《Deploying QML Applications》

《Performance Considerations and Suggestions》

《Internationalization and Location》

Testing and Debugging

《Prototyping with qmlscene》

《Debugging QML Applications》

《Qt Quick Test：QML Unit Testing Framework》

### 其他QML模块

Qt Quick仅仅提供了基本的可视类型，还有其他大量的Qt功能则由其他模块提供给QML。如果需要这些模块的功能，你可以通过《All QML APIs by Module》查看所需模块的文档。

### 相关话题

《Porting QML Applications to Qt 5》

# User Interfaces

# Graphics

Qt 5中的图形主要使用QPainter的API完成，或者通过Qt的声明是UI语言，Qt Quick和其场景视图后端来完成的。Qt 5图形还兼容打印，加载和保存各种图像格式。

## 2D Graphics with QPainter

QPainter提供了API支持绘制矢量图形，文本和图像到不同的表面或者是QPaintDevice，如QImage,QOpenGLPaintDevice,QWidget和QPrinter。实际的绘制发生在QPaintDevice的QPaintEngine中。软件光栅化和OpenGL ES 2.0 后端是最重要的两种QPiantEngine实现。光栅化绘制引擎是Qt的软件光栅化器，它是用来绘制QImage和QWidget的。相比于OpenGL绘制引擎，它的好处是在开启抗锯齿的时候可以实现更高质量的图形。

Qt GUI Paint System – QPainter类和架构介绍

。。。。。。

# C++11 Notes

## 伪随机数生成器(pseudo-random number generation)

随机数库提供了生成随机数和伪随机数的类。这些类包括：

1 随机数引擎（包括伪随机数生成器，能够产生按照均匀分布(uniform distribution)的整型数值序列；以及真实的随机数生成器）。

2 随机数分布（例如，均匀分布，正态分布或者泊松分布），能够将随机数引擎的输出转化为各种统计学上的分布状态。

### 随机数引擎(Random number engines)

随机数引擎通过设置随机数种子生成随机数。几种不同的伪随机数生成的算法类被设计成了模板，使其能够根据需要来修改。

具体选用哪种引擎，需要根据实际取舍：

1 线性乘同余算法(Linear congruential)引擎是速度较快，计算中存储空间消耗较少的一种方式。

2 滞后斐波那契数列(The lagged Fibonacci)引擎速度非常快，即使是在没有高级算术指令集的处理器上，但需要更大的运算空间，而且有时可能无法得到期望的频谱特特性。

3 梅森旋转算法(Mersenne Twister)引擎是一种更慢以及需要消耗更多运算空间的算法，但是使用合理的参数可以获得最长无重复的序列，而且可以获得最符合（提供的期望）的频谱特性。

所有随机数引擎都满足均匀随机数位生成器(UniformRandomBitGenerator)的要求（即区间内每个数出现的概率一致）。

linear\_congruential\_engine类：线性乘同余算法引擎。

mersenne\_twister\_engine类：梅森旋转算法引擎

sutract\_with\_carry\_engine类：带进位减法运算算法（一种滞后斐波那契算法）引擎

#### 随机数引擎适配器(Random number engine adaptors)

随机数引擎适配器通过将其他随机数引擎作为熵源(entropy source)来生成随机数。他们通常用来改变底层引擎的频谱特性。

discard\_block\_engine类：丢弃随机数引擎的部分输出

independent\_bits\_engine类：将随机数引擎的输出打包成指定位数的块。

shuffle\_order\_engine类：将随机数引擎的输出序列打乱输出。

#### 预定义的随机数生成器

下面是一些预定义的使用较广的随机数算法类。

|  |  |
| --- | --- |
| 类型 | 定义 |
| minstd\_rand0 | std::linear\_congruential\_engine<std::uint\_fast32\_t,16807,0,2147483648> |
| minstd\_rand | std::linear\_congruential\_engine<std::uint\_fast32\_t,48271,012147483647> |
| mt19937 | std::mersenne\_twister\_engine<std::uint\_fast32\_t,32,624,397,31,  0x9908b0df,11,  0xffffffff,7,  0x9d2c5680,15,  0xefc60000,18,1812433253> |
| mt19937\_64 | std::mersenne\_twister\_engine<std::uint\_fast64\_t,64,312,156,31,  0xb5026f5aa96619e9, 29, 0x5555555555555555, 17, 0x71d67fffeda60000, 37, 0xfff7eee000000000, 43, 6364136223846793005> |
| ranlux24\_base | std::subtract\_with\_carry\_engine<std::uint\_fast32\_t,24,10,24> |
| ranlux48\_base | std::subtract\_with\_carry\_engine<std::uint\_fast64\_t, 48, 5, 12> |
| ranlux24 | std::discard\_block\_engine<std::ranlux24\_base, 223, 23> |
| ranlux48 | std::discard\_block\_engine<std::ranlux48\_base, 389, 11> |
| knuth\_b | std::shuffle\_order\_engine<std::minstd\_rand0, 256> |
| default\_random\_engine | 取决于具体实现 |

# 单词索引

infrastructure 基础设施，基础架构，基础建设

supplemented 补充

specialized 专门的，专业的，专科的，专用的

In particular 尤其，特别是

interact 相互影响，相互作用，交互

interactive 交互式，互动，互动式

interconnect 相互联系，相互连接，互联

specification 规格，规范，说明书，产品规格，规格说明

alike to 与…相同，与…一样

imperative 强制的

in addtion to 加之，除…之外，除了

onward 向前的，继续的，一路向前

mimic 模仿，模拟，假装的

respective 各自的，分别的

paradigm 范例，模型，范式，典范

in terms of 从…方面，根据，在…方面，按照

contrast 对比，对照，形成对比，反差

generally 大体上，通常地，普遍地，一般

semantics 语义的，（单词，短语等的）含义，语意

notation 符号，记号，标记法，表示法

omitt 忽略，遗漏，删除，漏掉，省略

precede 在…之上，优于，比…重要，先于

transparent 透明的，清澈的，透明色，透明模式，易识破的，显而易见的

transprency 透明度，透明性，透明化，透明效果

resolution 解决，解析，决议，分辨率，解析度，决心

distinctly 显然，清楚地，明显地

distinct from 不同于…

detect 发现，查明，探测，察觉

evaluate 评估，求…的值，计算，估计

desire 希望，欲望，渴求，情欲，心愿

inspect 检查，视察，查看，审查

inject （给…）注射（药物等），增加（某品质），注入，喷射，注满

inejct sth(into sth) 将…注入（到…）

maintian 维护，保养，抚养，坚持

precedence 优先级，优先顺序

encapsulate 封装，压缩，简述，概括，压缩

essentially 基本上，本质上，根本上；本来，本质，基本地

characteristic 特征，特点，特色；有特性的，…特有的

omit 忽略，省略，删去

entropy 熵

tradeoff 取舍，权衡，折中

moderate 适度的，温和的，中度的，温和

conjunction 联合，结合，连词

in conjunction with 与…共同，连带着，与…协力

datum 数据，资料，基准