Qt Quick

简介

Qt Quick 提供了一套高动态，丰富的 QML 元素来定制用户界面的说明性框架。Qt Quick 有助于程序开发员与界面设计员的合作为便携式设备建立流畅的用户界面，例如：移动电话、媒体播放器，机顶盒以及上网本等。Qt Quick 包含了 QtDeclarative C++模块，QML 并且它们全被整合到 Qt Create IDE 中。使用 QtDeclarative C++模块可以从你的 QT 应用程序中载入 QML 文件并与之互动。

QML 是对 JavaScript 一种扩展，它提供一种机制使用 QML 元素来说明构建一个对象树。QML 对 JavaScript 与 Qt 现有的 QObjec-tbase 类型系统进行整合改善；增加了自动属性绑定的支持并提供在语言级别的网络透明度。

QML 元素是一套先进的图形，就像搭积木方式那样构建界面。这些不同的元素是通过 QML 文档来绑在一起的，从简单的按钮与滑块到复杂完整的应用程序，例如一个受欢迎的 Flickr 照片共享网站上的照片浏览器。

Qt Quick 是建立在 Qt 固有优势的基础上。QML 可被用于逐步扩展现有的程序或创建全新的应用程序。QML 通过 QtDeclarative 模块来完全扩展 C++功能。

入门

QML 语言入门

QML 是一个说明性语言，用来描述程序的用户界面：它的外观以及它的行为。在 QML 中，一个用户界面作为一个带有属性的对象树。

本篇是为有很少或没有编程经验的人准备的。JavaScript 是用来作为 QML 的脚本语言，所以你可能想对它了解多一点（JavaScript：The Definitive Guide），然后再深入 QML。另外如 HTML 与 CSS 等 Web 技术的基础原理也是有帮助的；但它不是必需的。

QML 语言基础

QML 类似这样：

```
import Qt 4.7

Rectangle {
```

```
    width: 200
    height: 200
    color: "blue"

    Image {
        source: "pics/logo.png"
        anchors.centerIn: parent
    }
}
```

对象是指定的类型，紧随其后是一对大括号，对象类型总是首字母为大写。在上面的示例中，有两个对象，一个 Rectangle [矩形] 与一个 Image [图像] 。在大括号之间是关于该对象特定的信息，例如它们的属性。

属性是以 property [属性] : value [值] 形式指定的。在上面的示例中，我们可以看到图像有个名为 source [来源] 的属性，它的值被指定为”pics/logo.png”。该属性与它的值是由冒号来分隔。

属性是由单行来输写：

```
Rectangle {
    width: 100
    height: 100
}
```

或单行输写多个属性：

```
Rectangle { width: 100; height: 100 }
```

当单行输写多属性/值时，必须由分号来分隔开来。

Import 语句导入 `Qt` 模块，它包含所有标准的 `QML` 元素。如果没有 Import 语句；那么 `Rectangle` 与 `Image` 元素将无效。

表达式

除了赋值属性外；你也可以用 `JavaScript` 编写的表达式来指定。

```
Rotation {
    angle: 360 * 3
}
```

这些表达式可以包含其它的对象与属性的引用，在这种情况下就会建立约束关联：当该表达式改变值

时；该属性值将自动更新。

```
Item {
    Text {
        id: text1
        text: "Hello World"
    }
    Text {
        id: text2
        text: text1.text
    }
}
```

在上面的示例中，text2 对象将显示与 text1 相同的文本。如果 text1 改变值；那么 text2 也将自动更新为相同的值。

注意这里我们通过使用 id 值引用其它的对象。（详情请参见 id 属性信息）

QML 注释

在 QML 中的注释类似于 JavaScript。

单选注释以 // 开始。

多行注释以 /* 开始，以 */ 结束。

```
import Qt 4.7
```

注释是不被执行的，添加注释可对代码进行解释或者提高其可读性。

注释同样还可用于防止代码执行，这对跟踪问题是非常有用的。

```
Text {
    text: "Hello world!"
    //opacity: 0.5
}
```

在上面的示例中，这个 Text 对象将正常显示，一旦 opacity: 0.5 这行关掉注释，这是以半透明方式显示文本。

属性

属性命名

属性命名以首字母为小字（附加属性除外）。

属性类型

QML 支持许多类型的属性（参阅 QML 基本类型）。基本类型包括整型、实数型、布尔、字符串、颜色以及列表。

```
Item {
    x: 10.5          // a 'real' property
    ...
    state: "details"    // a 'string' property
    focus: true         // a 'bool' property
}
```

QML 属性是有类型安全检测的。也就是说，它们只允许你指定一个与属性类型相匹配的值。例如，项目 x 属性类型是实数，如果你赋值一个字符串；那么将会得到错误的信息。

```
Item {
    x: "hello"  // illegal!
}
```

id 属性

每个对象可给予一个特定唯一的属性称之为 id。在同一个 QML 文件中不可能拥有与其它对象同名的 id 值。指定一个 id 可以允许该对象可以被其它的对象与脚本引用。

下面的示例中，第一个矩形元素的 id 名为” myRect”。第二个矩形元素的宽度是引用的 myRect.widtch，这意味着它将与第一矩形具有相同的 width 值。

```
Item {
    Rectangle {
        id: myRect
        width: 100
        height: 100
    }
    Rectangle {
        width: myRect.width
        height: 200
    }
}
```

请注意，一个 id 首字符必须是小写字母或下划线并且不能包含字母，数字和下划线以外的字符。

列表属性

列表属性类似于下面这样：

```
Item {
    children: [
        Image {},
        Text {}
    ]
}
```

列表是包含在方括号内，以逗号分隔的列表元素。在你只分配单一项目列表的情况下，是可以省略方括号：

```
Image {
    children: Rectangle {}
}
```

默认属性

每个对象类型可以指定列表或对象属性之一作为其默认属性。如果一属性已被声明为默认属性，该属性标记可以被省略。例如该代码：

```
State {
    changes: [
        PropertyChanges {},
        PropertyChanges {}
    ]
}
```

可以简化成这样：

```
State {
        PropertyChanges {},
        PropertyChanges {}
}
```

因为 changes 是 State 类型的默认属性。

分组属性

在某些情况下使用一个'.'符号或分组符号形成一个逻辑组。

分组属性可写以下这样：

```
Text {
    font.pixelSize: 12
    font.bold: true
}
```

或者这样：

```
Text {
    font { pixelSize: 12; bold: true }
}
```

在元素文件分组属性使用'.'符号显示。

附加属性

有些对象的属性附加到另一个对象。附加属性的形式为 *Type.property* 其中 *Type* 是附加 *property* 元素的类型。

例如：

```
Component {
    id: myDelegate
    Text {
        text: "Hello"
        color: ListView.isCurrentItem ? "red" : "blue"
    }
}
ListView {
    delegate: myDelegate
}
```

`ListView` 元素附加 *ListView.isCurrentItem* 属性到每个它创建的代理上。

另一个附加属性的例子就是 `Keys` 元素，它用于处理任意可视项目上的按键，例如：

```
Item {
    focus: true
    Keys.onSelectPressed: console.log("Selected")
}
```

信号处理器

信号处理器允许响应事件时处理动作。例如，`MouseArea` 元素有信号处理器来处理鼠标按下，释放

以及单击：

```
MouseArea {
    onPressed: console.log("mouse button pressed")
}
```

所有信号处理器开始都是启用的。

有一些信号处理器包含一个可选的参数，例如 `MouseArea onPressed` 信号处理程序有鼠标参数：

```
MouseArea {
    acceptedButtons: Qt.LeftButton | Qt.RightButton
    onPressed: if (mouse.button == Qt.RightButton) console.log("Right mouse button pressed")
}
```

QML 教程

这节内容是通过 `QML` 入门教程来熟悉 `Qt Quick` 语言。当然它不会面面俱到；只是重点让大家熟悉主要语法与特性。

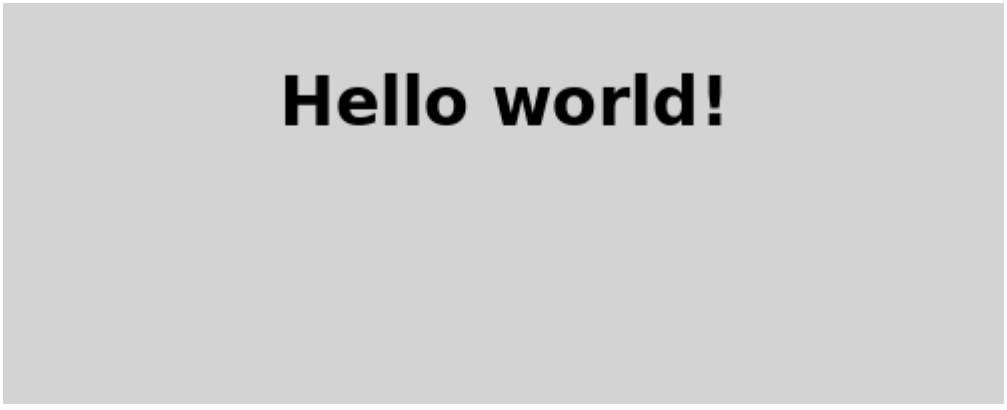
通过这节几个不同的步骤我们学习 `QML` 基本类型，使用属性与信号来建立我们自己的 `QML` 组件；而且还要使用状态与变换的帮助来建立一个简单的动画。

这节从迷你的“`Hello World`”程序出发，在随后的章节中将引入新的概念。

这个教程的源代码位于$QTDIR/examples/declarative/tutorials/helloworld 文件夹下。

教程 1——基本类型

这是一个非常简单的” `Hello World`”程序，它将介绍一些基本的 `QML` 概念。下图是该程序运行的结果。



这是该程序的源代码：

```
import Qt 4.7

Rectangle {
    id: page
    width: 500; height: 200
    color: "lightgray"

    Text {
        id: helloText
        text: "Hello world!"
        y: 30
        anchors.horizontalCenter: page.horizontalCenter
        font.pointSize: 24; font.bold: true
    }
}
```

逐步讲解

导入

首先，我们必须导入我们这个范例需要的类型。大多数 QML 文件都将导入内置的 QML 类型（类似于 Rectangle,Image……）这都包含在 Qt 内，使用：

```
import Qt 4.7
```

Rectangle 元素

```
Rectangle {
    id: page
    width: 500; height: 200
    color: "lightgray"
```

我们声明类型 Rectangle 为根元素。你可以使用它在应用程序中创建一个基本块。我们这里给予它一个 id 作为标签以便稍后引用。在这种情况里，我们命名它为 “page”。同样也设置 width,height 与 color 属性。Rectangle 元素还包含许多其它的属性（例如 x 与 y）；但这些我们保留它们默认的值。

Text 元素

```
Text {
    id: helloText
```

```
        text: "Hello world!"
        y: 30
        anchors.horizontalCenter: page.horizontalCenter
        font.pointSize: 24; font.bold: true
    }
```

这里添加一个 Text 元素作为根元素 Rectangle 的子级来显示文本“Hello World!”。

这里的 y 属性是用于定位文本从它父级顶部垂直距离 30 像素的位置。

anchors.horizontalCenter 属性将参考一个元素的水平中心位置。在这里我们指定文本元素位于 page 元素的水平中心位置。

font.poingSize 与 font.bold 属性是与字体设置相关。

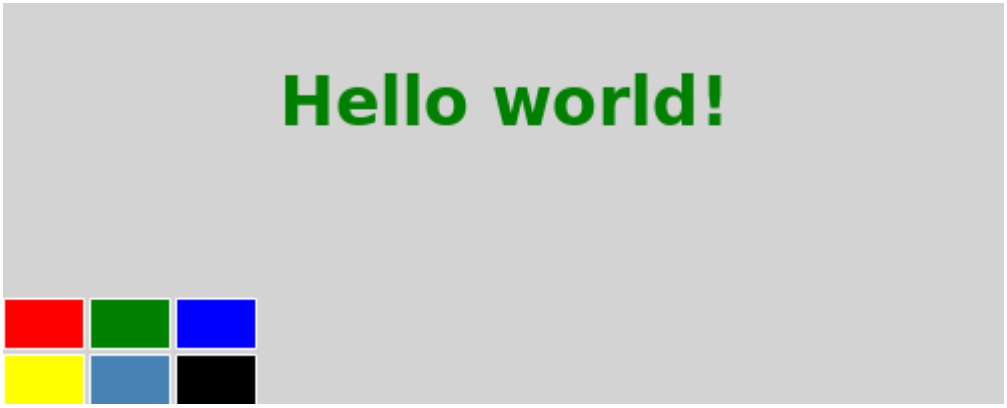
查看该范例

为了查看你编写好的程序，请运行 QML Viewr 工具（位于 bin 文件夹下），用你程序文件名作为第一个参数。例如运行这里提供的教程 1 范例，你需要这样输入命令行：

```
bin/qmlviewer $QTDIR/examples/declarative/tutorials/helloworld/tutorial1.qml
```

教程 2——QML 组件

这节增加了一个颜色拾取器用于颜色该文本。



我们的颜色拾取器是六格不同颜色的单元格。为了避免每个单元格重复多次相同的代码，我们将创建一个新的 cell 组件。组件是定义一个新类型的方法，可以在其它的 QML 文件重复使用。QML 组件类似于黑匣子通过属性、信号以及函数与外部世界相互影响，通常是在自己的 QML 文件中定义的。（详情请参见 定义新组件）。该组件的文件名必须以一个大写字母开头。

Cell.qml 的 QML 代码

```
import Qt 4.7

Item {
    id: container
    property alias cellColor: rectangle.color
    signal clicked(color cellColor)

    width: 40; height: 25

    Rectangle {
        id: rectangle
        border.color: "white"
        anchors.fill: parent
    }

    MouseArea {
        anchors.fill: parent
        onClicked: container.clicked(container.cellColor)
    }
}
```

逐步讲解

Cell 组件

```
Item {
    id: container
    property alias cellColor: rectangle.color
    signal clicked(color cellColor)

    width: 40; height: 25
```

我们组件的根元素是一个 `id` 名为 *`container`* 的 `Item`。`Item` 是一个最基本的可视元素并经常用作其它元素的容器。

```
property alias cellColor: rectangle.color
```

声明一个 `cellColor` 属性。这个属性是从我们组件外访问的，允许我们使用不同的颜色来实例化单元格。该属性只是对现存的属性使用了一个别名。

```
signal clicked(color cellColor)
```

我们需要这个组件 cellColor 属性带有 color 类型的单击信号，这将在稍后的主 QML 文件中使用到这个信号来改变文本的颜色。

```
    Rectangle {
        id: rectangle
        border.color: "white"
        anchors.fill: parent
    }
```

我们单元格组件是 id 名为 *rectangle* 的颜色化矩形。

anchors.fill 属性是设置元素尺寸的简便办法。在这种情况下矩形将拥有父级同样尺寸的大小。

```
    MouseArea {
        anchors.fill: parent
        onClicked: container.clicked(container.cellColor)
    }
```

为了单击一个单元格来改变文本的颜色，我们创建一个 MouseArea 元素，它是与父级有同样尺寸的。

MouseArea 定义了一个 *Clicked* 信号。当这个信号被触发时；我们发射使用颜色作为参数的单击信号。

主 QML 文件

在我们的主 QML 文件中，我们使用 Cell 组件来创建颜色拾取器：

```
import Qt 4.7

Rectangle {
    id: page
    width: 500; height: 200
    color: "lightgray"

    Text {
        id: helloText
        text: "Hello world!"
        y: 30
        anchors.horizontalCenter: page.horizontalCenter
        font.pointSize: 24; font.bold: true
    }

    Grid {
        id: colorPicker
        x: 4; anchors.bottom: page.bottom; anchors.bottomMargin: 4
        rows: 2; columns: 3; spacing: 3

        Cell { cellColor: "red"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "green"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "blue"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "yellow"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "steelblue"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "black"; onClicked: helloText.color = cellColor }
    }
}
```

我们在网格中放置六个单元格，使用不同的颜色来创建一个颜色拾取器。

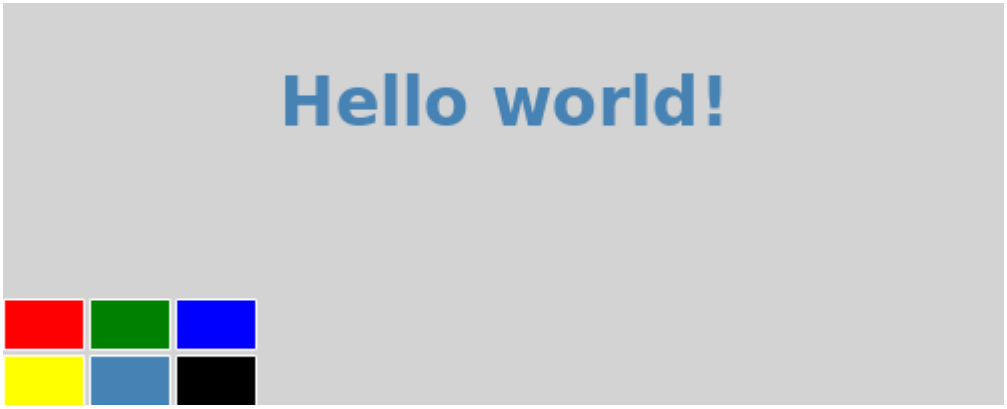
```
Cell { cellColor: "red"; onClicked: helloText.color = cellColor }
```

当我们单元格的 *clicked* 信号触发时，需要传递 *cellColor* 参数来设置文本的颜色 。通过一个属性的 *'onSignalName'* 的命名方式来应付组件的任何信号。

教程 3——状态与变换

在这一节中，我们使用这个例子有更多的动态，这里介绍状态与变换。

我们可以使用文本移动到屏幕的底部，当单击时旋转并变成红色。



QML 代码：

```
import Qt 4.7

Rectangle {
    id: page
    width: 500; height: 200
    color: "lightgray"

    Text {
        id: helloText
        text: "Hello world!"
        y: 30
        anchors.horizontalCenter: page.horizontalCenter
        font.pointSize: 24; font.bold: true

        MouseArea { id: mouseArea; anchors.fill: parent }

        states: State {
            name: "down"; when: mouseArea.pressed == true
            PropertyChanges { target: helloText; y: 160; rotation: 180; color: "red" }
        }

        transitions: Transition {
            from: ""; to: "down"; reversible: true
            ParallelAnimation {
                NumberAnimation { properties: "y,rotation"; duration: 500; easing.type: Easing.InOutQuad }
                ColorAnimation { duration: 500 }
            }
        }
    }

    Grid {
```

```
        id: colorPicker
        x: 4; anchors.bottom: page.bottom; anchors.bottomMargin: 4
        rows: 2; columns: 3; spacing: 3

        Cell { cellColor: "red"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "green"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "blue"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "yellow"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "steelblue"; onClicked: helloText.color = cellColor }
        Cell { cellColor: "black"; onClicked: helloText.color = cellColor }
    }
}
```

逐步讲解

```
states: State {
        name: "down"; when: mouseArea.pressed == true
        PropertyChanges { target: helloText; y: 160; rotation: 180; color: "red" }
    }
```

首先我们对文本元素创建名为 *down* 的状态。当 *MouseArea* 被按下时，这个状态将被激活；当鼠标键释放时，将处于非激活状态。

down 状态包含一套从默认状态中的属性改变（这在 QML 中已经初始化定义过）。这里是设置文本的 *y* 属性为 *160,*旋转 *180* 度并且设置为 *color* 为红色。

```
transitions: Transition {
        from: ""; to: "down"; reversible: true
        ParallelAnimation {
            NumberAnimation { properties: "y,rotation"; duration: 500; easing.type: Easing.InOutQuad }
            ColorAnimation { duration: 500 }
        }
    }
```

因为我们不希望文本运动时出现不平滑的现象，这里在两个状态之间添加一个过渡。

from 与 *to* 定义两状态间的变换。在这里是从默认状态到 *down* 状态的变换。

因为我们需要从 *down* 状态返回到默认状态有同样的变换，还需要设置 *reversible* 为 *ture*。这等于分别写了两种变换。

ParallelAnimation 元素确保动画的两种类型（数值与颜色的变化）同时开始。我们同样可以使用 *SequentialAnimation* 来替代，它是序列进行的，当一个动画完成后，再接着另一个动画。\

利用 *QML* 开发 *Qt* 程序

概要

QML 虽然并不需要了解 *Qt*，但你已经非常熟悉 *Qt*；那么学习并使用 *QML* 是非常有必要的。因为一个应用程序可以使用 *QML* 做用户界面，非界面逻辑部分可以使用 *Qt* 完成。

熟悉概念

QML 可以直接访问 *Qt* 如下概念：

- *QAction* 动作类型
- *QObject* 信号与槽——在 *JavaScript* 里可作为函数
- *QWidget——QDeclarativeView* 是一个 *QML* 显示组件
- *Qt* 模型——直接用于数据绑定（*QAbstractItemModel*）

QML 项目与 Qt 组件的比较

QML 项目与 *Qt* 组件非常类似：它们定义的用户界面外观。

有三种不同类型的 *QWidget* 结构：

- 简单组件是不能作为父级的（如：*QLabel,QCheckBox,QToolButton* 等等）
- 父级组件是可以作为其它组件的父级（如：*QGroupBox,QStackedWidget,QTabWidget* 等等）
- 在内部的子级组件的组合组件（如 *QComboxBox,QSpinBox,QFileDialog,QTabWidget* 等等）

QML 项目同样也可以这样分类。

简单组件

有个非常重要的规则要记住，在 C++实现的 QDeclarativeItem 是不包含任何界面外观策略的，这是保留给该项目的 QML 用法。

举例说明，假设你需要一个可重复使用的按钮项目。如果你决定声明一个 QDeclarativeItem 子类来实现一个按钮，就像 QoToolButton 是 QWidget 的子样一样。如上面所述的规则那样，QDeclarativeButton 的没有任何界面外观，只是启用概念，触发等等。

但在 Qt 里已经有一个对象做这件事：QAction。

QAction 是 QPushButton,QCheckBox,QMenu 项目，QoToolButton 以及其它可视组件的 UI 本质无关，通常绑定一个 QAction。

因此，在 QML 已经完成了实现一个抽象——这就是 QAction。一个按钮界面行为的外观，状态间的转变以及如何响应鼠标、键盘或触摸输入都应留在 QML 里定义。

QDeclarativeTextEdit 是基于 QTextControl，QDeclarativeWebView 是基于 QWebPage and ListView uses QAbstractItemModel, just as QTextEdit, QWebView and QListView are built upon those same UI-agnostic components.

QWidget 封装了界面外观是非常重要的，对 QML 概念也服务同一目的。如果你开发一个完整的应用程序，应用有一致的界面外观。你应该构建可以重用的组件集。

所以你要实现一个可以重复使用的按钮，只需简单的构建一个 QML 组件即可。

父级组件

有时候需要把界面分成几大类，每个类放任意个其它的部分。QaIb Widget 就提供一个多“页面“的界面，在任何时间只能切换到其中一个页面。一个 QScollAred 提供一个滚动条来扩大界面有效空间。

几乎所有的组件都可以直接在 QML 创建。只有少数情况下，需要非常特殊的事情处理，例如 Filckable 需要 C++完成。

QML 相较于 Qt 组件的父级概念是有显著差异的，子级项目是定位相对于它们的父级，在这里没有需要它们被整个包含到父级。这种差异影响是深远的，例如：

- 围绕组件的阴影或高亮可能是该组件的子级。
- 粒子效果可以流到对象范围以外。
- 变换动画可以通过移动项目超出屏幕外来达到”隐藏“项目的目的。

组合组件

有些组件通过包含其它组件提供”实现细节“的功能。例如 QSpinB ox 是一个行编辑与两个增加/减少值的按钮。QFileDialog 使用了一大堆组件来为用户提供一个寻找与选择一个文件名的方式。

在开发可重复使用的 QML 项目时；你可以选择做同样的事情：构建你已经定义的其它项目组成一个项目。

唯一要考虑是你构建的复合组件是用户期望的那样动画与变换。例如一个微调框可能需要从任意文本或字符项目平常的过渡，因此微调框项目需要足够的灵活性，允许这样的动画。

QML 项目与 QGraphicsWidgets 比较

QML 项目与 QGraphicsWidgets 之间主要区别是如何使用。技术实现细节大致相同；但实际上它们是不同的，因为 QML 项目需要声明与合成使用，QGraphicsWidgets 是更为整合的使用。QML 项目与 QGraphicsWidgets 都继续自 QGraphicsObject，可以并存。差别是布局系统与其它组件的接口。如 QGraphicsWidgets 趋向于将所有功能集为一身，一个 QGraphicsWidgets 可能有多个独立的 QML 文件交叉许多 QML 项目组合而成的；但它仍然被 C++载入与使用作为单个 QGraphicsObject。

QGraphicsWidgets 是设计为被 QGraphicsLayouts 布局的。QML 则不使用 QGraphicsLayouts，由于 Qt 的布局器不能混合动画与流体界面；所以几何体界面是主要差别之一。当编写 QML 元素时；可以使设计人员能够使用绝对的几何体，绑定或锚来放置它们的边界矩形并不使用布局或尺寸伸缩器。如果尺寸适当的提示；那么在 QML 文件里放置它们，从而使设计者知道如何使用项目最好；但仍具有完全的控制界面外观。

其它区别在于 QGraphicsWidgets 往往遵循组件模式，它们是一个用户界面与逻辑的自我包含绑定。与之相反的是 QML 通常是单一项目，就其本身并不满足使用的情况。所以当编写 QML 项目，尽量避免做 UI 逻辑或项目内部合成可视元素；转为编写更为通用的基元，使外观（如涉及 UI 逻辑）被编写在 QML 中。

两者间的差别是由于不同的互动方式。QGraphicsWidget 是 QGraphicsObject 子类使 C++更容易开发流体用户界面。QDeclarativeItem 是 QGraphicsObject 子级使 QML 开发流体用户界面。因此公开界面是主要区别之一以及使用它来设计项目（声明元素用于 QML 而并 QGraphicsWidget，因为你需要编写自己的界面逻辑到子类）。

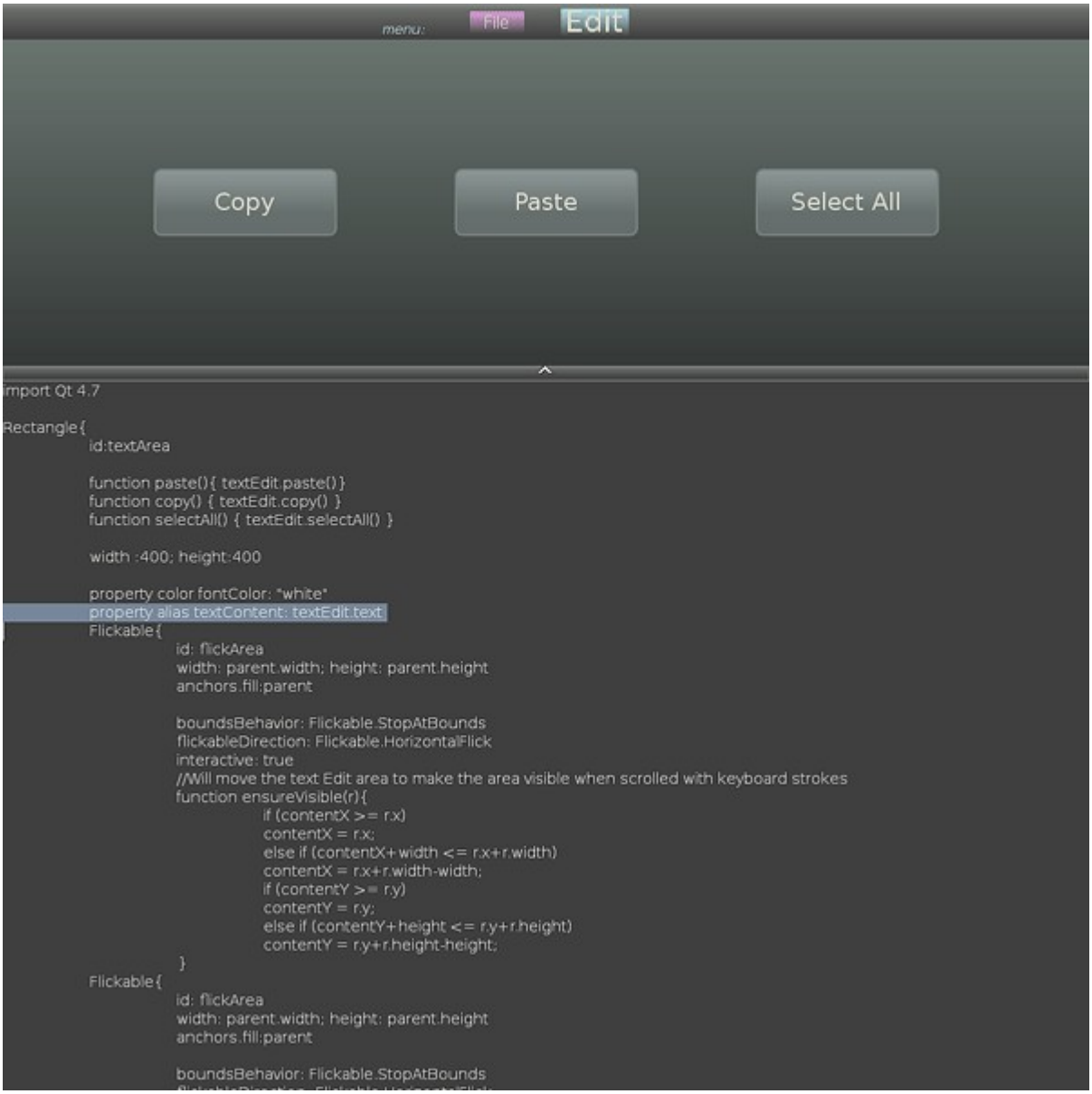
如果你想同时使用 QML 与 C++来编写用户界面，例如：缓进过渡期，建议使用 QDeclarativeItem 子类（尽管你同样可以使用 QGraphicsWidgets）。以允许更容易使用从 c++使根项目的每一个 c++组成一个布局器项目、载入单个的 QML 组件(可能是由多个 QML 文件,并包含一个独立的用户界面和逻辑）到你的场景来替换单独的 QGraphicsWidgets。

使用 QML 程序设计入门

欢迎来到 QML，描述性 UI 语言的世界。在本入门指南中，我们将使用 QML 创建一个简单的文本编辑器应用程序。看完本指南后，你应该可以使用 QML 与 Qt C++开发自己的应用程序了。

QML 构建用户界面

这个应用程序是一个简单的文本编辑器，有载入、保存以及一些文本处理操作。本指南包括两在部分。第一部分将涉及使用 QML 描述性语言设计应用程序布局与行为。第二部分使用 Qt C++完成文件载入与保存功能部分。使用 Qt 的元对象系统，我们导出 C++函数作为 QML 元素属性使用。利用 QML 与 Qt C++我们可以有效的从应用程序逻辑中分离出界面逻辑。



要运行 QML 范例代码，使用 qmlviewer 工具，以 QML 文件作为参数来查看。C++部分本教程是假设读者具有 Qt 编程经验的。

定义一个按钮与菜单

基本组件——按钮

我们开始构建文本编辑器上的一个按钮。功能上，一个按钮有鼠标敏感区与标签，当用户按下按钮时，按钮执行操作。

在 QML 里，基本可视元素是矩形元素。矩形元素的属性来控制元素的外观与位置。

```
import Qt 4.7
```

```
Rectangle {
    id: simplebutton
    color: "grey"
    width: 150; height: 75

    Text{
        id: buttonLabel
        anchors.centerIn: parent
        text: "button label"
    }
}
```

首先，import Qt 4.7 允许 qmlviewer 工具导入 QML 元素，这将稍后使用。这一行是每个 QML 文件必须存在的。注意在 import语句里 Qt 模块的版本号是被包括的。

这个简单的矩形拥有独立唯一的标识符 simplebutton，它是绑定的 id 属性。矩形元素的属性由列表列出的属性在冒号后面对应值。在示例代码中，矩形的颜色属性为灰色，同时还决定的矩形的宽度与高度。

文本元素是一个不可编辑的文本域。我们命名这个文本元素为 buttonLable。要设置文本域的字符串内容，将绑定一个值到该文本属性。该标签在矩形内的中心位置，指定文本元素的锚到父级（simplebutton）。锚可以绑定到其它项目的锚，允许布局简单分配。

我们保存这段代码为 SimpleButton.qml。运行 qmlviewer 来查看代码运行的效果。

为了实现按钮的单击功能，我们可以使用 QML 的事件处理。QML 事件处理是非常类似于 Qt 的信号与槽机制的。信号发射并链接到槽调用。

```
Rectangle{
    id:simplebutton
    ...

    MouseArea{
        id: buttonMouseArea

        anchors.fill: parent //anchor all sides of the mouse area to the rectangle's anchors
                //onClicked handles valid mouse button clicks
        onClicked: console.log(buttonLabel.text + " clicked" )

    }
}
```

我们包含一个鼠标区域（MouseArea）元素在我们的 simplebutton 中。鼠标区域元素描述着互动区，检测鼠标运动。对于我们的按钮，我们锚定的整个鼠标区是它的父级（simplebutton）区域。anchors.fill 语法是填充指定的区域内部。QML 使用锚定来进行项目布局。

鼠标区域 有很多的信号处理，就是鼠标在指定的鼠标区域边界内移动。它们中有一个是 onClicked，它只接受鼠标按钮的单击。默认为左击。我们可以绑定动作到 onClicked 句柄上。在我们的例子中，执

行 console.log()函数，当鼠标单击该区域时将输出控制台文本，它是用于调试与输出文本的有用工具。

在 simpleButton.qml 代码里只要鼠标单击就会在控制台输出文本。

```
Rectangle {
    id:Button
    ...

    property color buttonColor: "lightblue"
    property color onHoverColor: "gold"
    property color borderColor: "white"

    signal buttonClick()
    onButtonClick: {
        console.log(buttonLabel.text + " clicked" )
    }

    MouseArea{
        onClicked: buttonClick()
        hoverEnabled: true
        onEntered: parent.border.color = onHoverColor
        onExited:  parent.border.color = borderColor
    }

    //determines the color of the button by using the conditional operator
    color: buttonMouseArea.pressed ? Qt.darker(buttonColor, 1.5) : buttonColor
}
```

在 Button.qml 里是一个完整功能的按钮。在本文中有些代码被省略，由省略号表示了。因为它们要么在前面章节有介绍；要么有些是现在不用去涉及的。

自定义属性是使用 property 类型命名语法声明的。在该段代码里，property buttonColor 是声明颜色类型并绑定" lightblue"。buttonColor 是稍后用在条件操作中来决定按钮的填充颜色。注意属性值使用=等号赋值是有可能的，另外值绑定是使用冒号。自定义属性允许内部项目访问矩形范围外的。在这里是基本的 QML 类型，例如：整型、字符串、实数型以及变体类型。

通过绑定 onEntered 与 onExited 信号句柄到颜色，当鼠标悬停在按钮上面的时候该按钮的边框将转变为黄色以及当鼠标退出鼠标区域的时候颜色恢复。

在 Button.qml 里在 buttonClick()信号名称前面加一个 signal 关键字来声明。所有信号是它们自动创建的处理程序，它们的名称以 on 开头。像 `onButtonClick` 是按钮单击的事件处理。然后在 `onButtonClick` 内指定执行的动作。在我们按钮的例子中，`onClicked` 鼠标处理程序只是简单的调用 `onButtonClick` 来显示一个文本。`onButtonClick` 启用外面的对象来访问该按钮的鼠标区域。例如项目可能声明了多个 `MouseArea` 以及一个 `buttonClick` 信号可以使几个 `MouseArea` 信号处理程序很好的区分。

我们现在基本知道在 `QML` 里实现项目处理基本的鼠标动作。我们在一个矩形内部创建一个文本标签，自定义它的属性以及实现响应鼠标动作的行为。这种在元素内创建元素的想法是整个文本编辑器应用程序中贯穿的。

这个按钮是没有什么用的，除非作为一个组件来执行一个动作。在下一节中，我们将创建一个菜单来包含几个这样的按钮。

下面的代码是我完善好的，书的例子代码并不完整。

```
import Qt 4.7

Rectangle {
    id:button
    width: 150
    height: 75
    radius: 10
    border.width: 2

    property color buttonColor: "lightblue"
    property color onHoverColor: "gold"
    property color borderColor: "white"
    property string label:""



    signal buttonClick()
    onButtonClick: {
        console.log(buttonLabel.text + " clicked" )
    }

    Text{
          id: buttonLabel
          anchors.centerIn: parent
          text: label
      }

    MouseArea{
        id:buttonMouseArea
        anchors.fill: parent
        onClicked: buttonClick()
        hoverEnabled: true
        onEntered: parent.border.color = onHoverColor
        onExited:  parent.border.color = borderColor
    }

    //determines the color of the button by using the conditional operator
    color: buttonMouseArea.pressed ? Qt.darker(buttonColor, 1.5) : buttonColor
}
```

创建一个菜单页面

到这一个阶段，我们介绍了在一个 QML 文件中如何创建元素以及指定行为。在本节中，我们将介绍如何导入 QML 元素以及如何重新使用一些创建的组件来构建其它的组件。

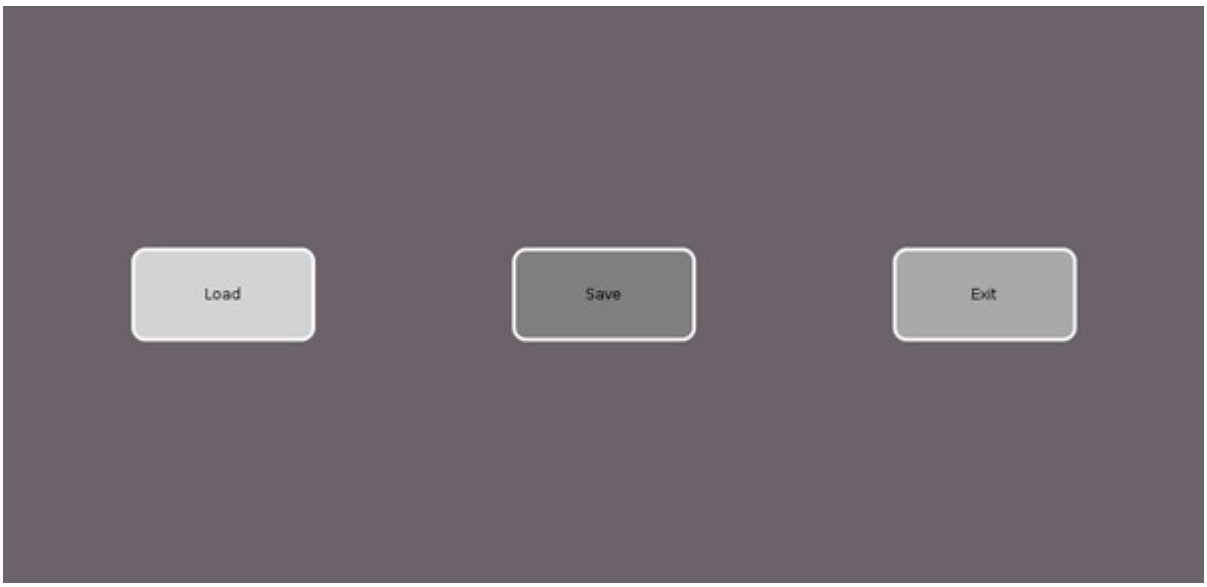
菜单显示了一个列表的内容，每个项目有能力执行一个动作。在 QML 里，我们有几种方式来创建菜单。首先我们将创建一个菜单来包含几个执行不同动作的按钮。该菜单代码是 FileMenu.qml 文件里。

```
import Qt 4.7              //import the main Qt QML module
import "folderName"          //import the contents of the folder
import "script.js" as Script     //import a Javascript file and name it as Script
```

上面的语句显示了如何使用 impor t关键字。这里需要使用 Ja vaScript 文件或 QML 文件，这些文件没有在同一目录内。Button.qml 与 FIleMenu.qml 在同一目录内，我们不需要导入 Button.qml 文件来使用它。我们可以直接通过声明 Button{}来直接创建一个按钮。类似于 Re ctange{}那样声明。

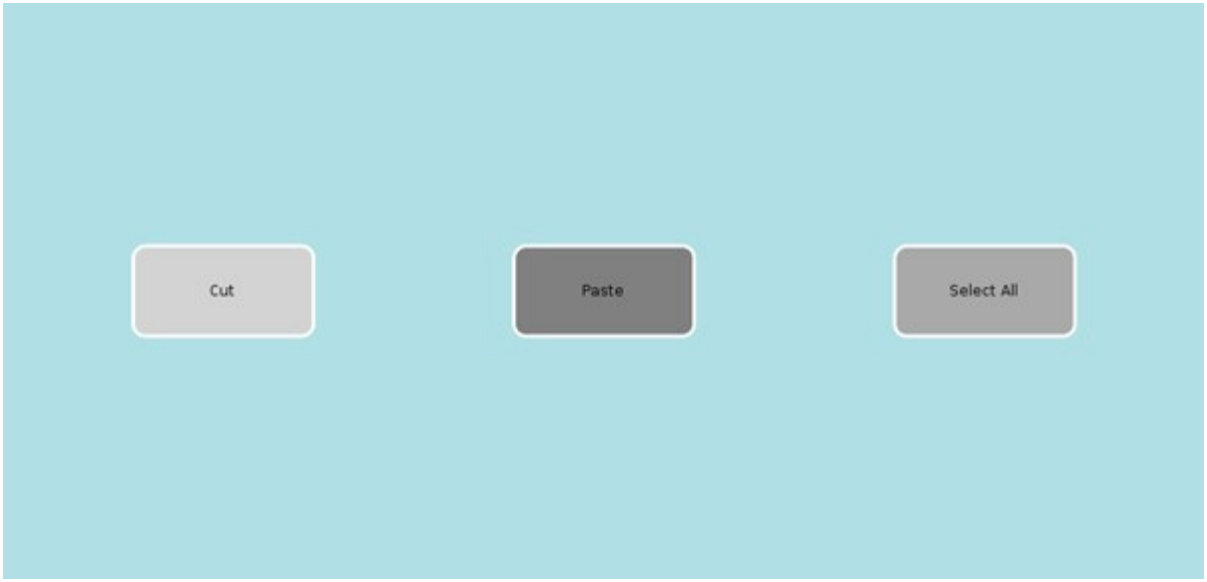
```
In FileMenu.qml:

    Row{
        anchors.centerIn: parent
        spacing: parent.width/6

        Button{
            id: loadButton
            buttonColor: "lightgrey"
            label: "Load"
        }
        Button{
            buttonColor: "grey"
            id: saveButton
            label: "Save"
        }
        Button{
            id: exitButton
            label: "Exit"
            buttonColor: "darkgrey"

            onButtonClick: Qt.quit()
        }
    }
```

在 FileMenu.qml 里，我们声明了三个 Button 元素。他们是声明在一个 Ro w 元素内，一个定位器将沿着垂直行来定位它的子级。该按钮声明属于 Button.qml，它是与上节的 Button.qml 一样。新属性绑定可以被新创建的按钮声明，有效地覆盖 Button.qml 的属性设置。称为 exitButton 按钮单击时将退出并关闭该窗口。注意在 Button.qml 内的信号句柄 onButtonClick 将调用 exitButton 里的 onButtonClick 处理程序。



Ro w 是在一个矩形内声明的，创建一个矩形包含一行按钮。这个矩形用来组织一行按钮来替代菜单。

编辑菜单的声明也非常类似于上一步。菜单按钮有：Copy，Paste 以及 Select All。



用我们先前制作组件的知识，可以结合这些菜单页面来创建一个菜单栏，按钮组成来选择菜单并看看使用 QML 结构数据。

实现一个菜单栏

我们的文本编辑器应用程序将需要一个菜单栏来显示菜单。菜单栏在不同的菜单间切换并且用户可以选择其中一个菜单来显示。菜单切换实现需要更复杂的结构。QML 使用模型与视图来结构化数据并显示结构数据。

使用数据模型与视图

QML 有不同的数据视图来显示数据模型。我们的菜单栏将以列表形式来显示菜单，一个标头显示一行菜单名。

菜单列表是声明在 VisualItemModel 内。`VisualItemModel` 元素已经包含项目有 `Rectangle` 元素与导入 `UI` 元素。其它的模型类型例如 `ListModel` 元素需要一个代理来显示它们的数据。

我们在 `menuListModel` 内声明两个可视项目：`FileMenu` 与 `EditMenu`。使用 `ListView` 自定义两个菜单并显示它们。`MenuBar.qml` 文件包含 `QML` 声明以及在 `EditMenu.qml` 内的一个简单的编辑菜单。

```
VisualItemModel{
    id: menuListModel
    FileMenu{
        width: menuListView.width
        height: menuBar.height
        color: fileColor
    }
    EditMenu{
```

```
        color: editColor
        width:  menuListView.width
        height: menuBar.height
    }
}
```

`ListView` 元素将以代理显示一个模型。代理可以声明模型项目来显示在 `Row` 元素或显示项目在一个网格里。我们的 `menuListModel` 已经是可视项目，因此我们并不需要声明一个代理。

```
ListView{
    id: menuListView

    //Anchors are set to react to window anchors
    anchors.fill:parent
    anchors.bottom: parent.bottom
    width:parent.width
    height: parent.height

    //the model contains the data
    model: menuListModel

    //control the movement of the menu switching
    snapMode: ListView.SnapOneItem
    orientation: ListView.Horizontal
    boundsBehavior: Flickable.StopAtBounds
    flickDeceleration: 5000
    highlightFollowsCurrentItem: true
    highlightMoveDuration:240
    highlightRangeMode: ListView.StrictlyEnforceRange
}
```

另外 ListView 继承了 Flickable，使列表响应鼠标拖拽与其它手势动作。上面最后一部分代码是设置 Flickable 属性来创建期望的手按动作。特别是 highlightMoveDuration 属性改变手按转换的持续时间。高值将非常慢速的切换菜单。

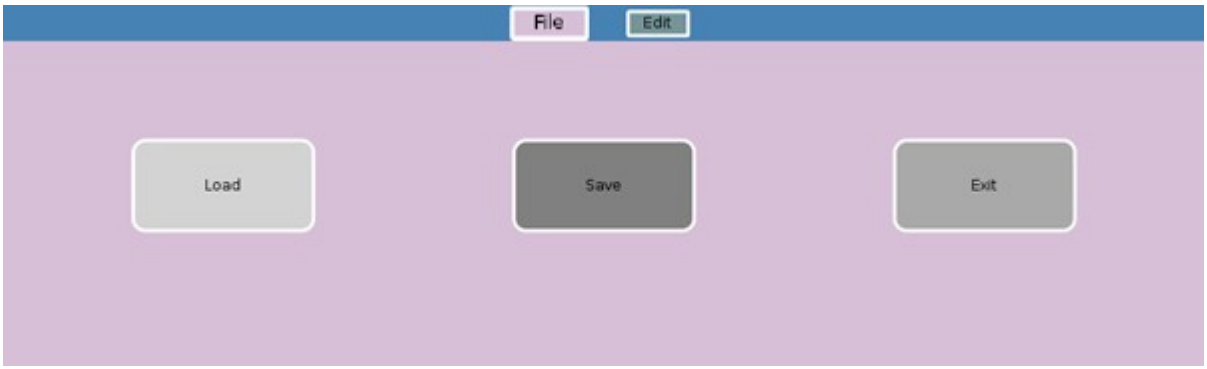
ListView 通过索引来维护模型项目并且通过索引来访问模型里的每个可视项目（以声明的顺序）。改变 currentIndex 将会影响 ListView 内的高亮项目。我们菜单栏标题将体现这种效果。当单击时，这里一行有两个按钮，当前菜单全部改变。fileButton 单击时改变当前菜单为文件菜单，索引为 0，因为 FileMenu 是在 menuListModel 中第一个声明的。同样 editButton 单击时将改变当前菜单为编辑菜单。

labelList 矩形的 Z 值为 1，表示它在菜单栏的前面显示。高值时项目都显示在低值的前面，Z 值默认时为 0。

```
Rectangle{
    id: labelList
    ...
    z: 1
    Row{
        anchors.centerIn: parent
        spacing:40
        Button{
            label: "File"
            id: fileButton
            ...
            onButtonClick: menuListView.currentIndex = 0
        }
        Button{
            id: editButton
            label: "Edit"
```

```
            ...
            onButtonClick:    menuListView.currentIndex = 1
        }
    }
}
```

菜单栏我们刚刚创建可被触摸或单击来访问。切换菜单屏幕有直观的感受与反应。



构建文本编辑器

声明文本区

我们的文本编辑器如果不包含一个可编辑文本区；那么它就不是一个文本编辑器。QML 的 eTxtEdit 元素允许多行可编辑文本区的声明。eTxt 与eTxtEdit 元素是有区别的，它并不允许用户直接编辑文本。

```
TextEdit{
    id: textEditor
    anchors.fill:parent
    width:parent.width; height:parent.height
    color:"midnightblue"
    focus: true

    wrapMode: TextEdit.Wrap

    onCursorRectangleChanged: flickArea.ensureVisible(cursorRectangle)
}
```

□ □ 器有其字体和□ 色属性的□ 置，有文本折行属性□ 置。文字□ □ 区是在一个可触摸区域内，如果文本光□ 在可□ 区域之外□ 会□ □ 文本。□ ensureVisible()函数将□ □ 光□ 如果在可□ 矩形□ 界区域外；那么将移□ 相□ 的文字。□ QML 使用脚本 Javascript □ 法，如先前提到的那□，JavaScript 文件是可以□ 入并在 QML 文件中使用。

```
function ensureVisible(r){
    if (contentX >= r.x)
        contentX = r.x;
    else if (contentX+width <= r.x+r.width)
        contentX = r.x+r.width-width;
    if (contentY >= r.y)
        contentY = r.y;
```

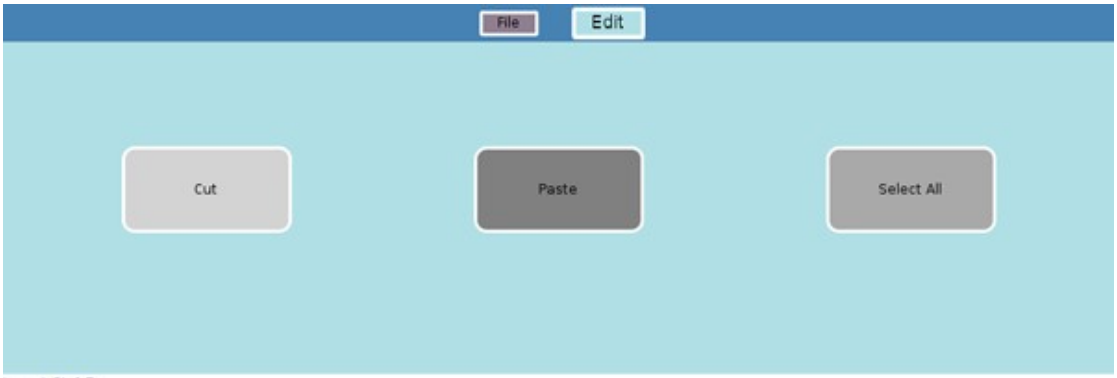
```
        else if (contentY+height <= r.y+r.height)
            contentY = r.y+r.height-height;
    }
```

结合组件到文本编辑器

我们现在正准备使用 QML 创建文本编辑器的布局。文本编辑器有两个组件：菜单栏与文本区。QML 允许我们重用组件，因此使代码更为简单，需要时导入组件并自定义。我们文本编辑器分隔成两个窗口，屏幕的三分之一，是菜单栏，三分之二是为文本区域准备。菜单栏是显示在任何元素的前面。

```
Rectangle{

        id: screen
        width: 1000; height: 1000

        //the screen is partitioned into the MenuBar and TextArea. 1/3 of the screen is assigned to the MenuBar
        property int partition: height/3

        MenuBar{
            id:menuBar
            height: partition
            width:parent.width
            z: 1
        }

        TextArea{
            id:textArea
            anchors.bottom:parent.bottom
            y: partition
            color: "white"
            height: partition*2
            width:parent.width
        }
    }
```

通过导入可重用组件，我们的文本编辑代码看起来简单得多。然后我们可以自定义主程序，不用担心已经有定义行为的属性。使用这种方法，应用程序布局与 UI 组件可以很容易建立。




```
import Qt 4.7

Rectangle{
        id:textArea

        function paste(){ textEdit.paste()}
        function copy() { textEdit.copy() }
        function selectAll() { textEdit.selectAll() }

        width :400; height:400

        property color fontColor: "white"
        property alias textContent: textEdit.text
        Flickable{
                id: flickArea
                width: parent.width; height: parent.height
                anchors.fill:parent

                boundsBehavior: Flickable.StopAtBounds
                flickableDirection: Flickable.HorizontalFlick
                interactive: true
                //Will move the text Edit area to make the area visible when scrolled with keyboard strokes
                function ensureVisible(r){
                        if (contentX >= r.x)
                        contentX = r.x;
                        else if (contentX+width <= r.x+r.width)
                        contentX = r.x+r.width-width;
                        if (contentY >= r.y)
                        contentY = r.y;
                        else if (contentY+height <= r.y+r.height)
                        contentY = r.y+r.height-height;
                }
        Flickable{
                id: flickArea
                width: parent.width; height: parent.height
                anchors.fill:parent

                boundsBehavior: Flickable.StopAtBounds
                flickableDirection: Flickable.HorizontalFlick
                interactive: true
                //Will move the text Edit area to make the area visible when scrolled with keyboard strokes
                function ensureVisible(r){
                        if (contentX >= r.x)
                        contentX = r.x;
```

装饰文本编辑器

实际一个抽屉界面

我们的文本编辑器看起来有些简单，我们需要来装饰它。使用 QML 我们可以声明文本编辑器的转换与动画。我们的菜单栏是占用屏幕的三分之一，我们需要时才会显示，这将是不错的。

我们可以添加一个抽屉式的界面，当单击菜单栏时，将收缩或放大时。我们实现一条细小的矩形作为鼠标单击响应区。抽屉像应用程序那样有两种状态：打开的抽屉 与 关闭的抽屉。drawer 项目是一个小高度的矩形带。这里内嵌了 Image 元素，声明箭头图标在抽屉的中心。抽屉为整个应用程序指定一个状态，识别符是 `screen`，只要用户单击鼠标区域。

```
Rectangle{
        id:drawer
        height:15
```

```
    Image{
        id: arrowIcon
        source: "images/arrow.png"
        anchors.horizontalCenter: parent.horizontalCenter
    }

    MouseArea{
        id: drawerMouseArea
        anchors.fill:parent
        onClicked:{
            if (screen.state == "DRAWER_CLOSED"){
                screen.state = "DRAWER_OPEN"
            }
            else if (screen.state == "DRAWER_OPEN"){
                screen.state = "DRAWER_CLOSED"
            }
        }
        ...
    }
}
```

一个状态仅仅是一个配置集合，它是以 State 元素声明。状态列表可以被列出并绑定到 states 属性。在我们的应用程序里，这两个状态被称为 DRAWER_CLOSED 与 DRAWER_OPEN。项目配置是声明在 PropertyChanges 元素里。在 DRAWER_OPEN 状态有四个项目将接受属性的变化。第一个目标是菜单栏，将其 y 属性设置为 0。同样当为 DRAWER_OPEN 状态时对 TextArea 将降低到一个新的位置。 文本区，抽屉以及抽屉的图标将发生属性变化以满足当前的状态。

```
states:[
    State{
        name: "DRAWER_OPEN"
        PropertyChanges { target: menuBar; y:0}
        PropertyChanges { target: textArea; y: partition + drawer.height}
        PropertyChanges { target: drawer; y: partition}
        PropertyChanges { target: arrowIcon; rotation: 180}
    },
    State{
        name: "DRAWER_CLOSED"
        PropertyChanges { target: menuBar; y:-partition}
        PropertyChanges { target: textArea; y: drawer.height; height: screen.height - drawer.height}
        PropertyChanges { target: drawer; y: 0}
        PropertyChanges { target: arrowIcon; rotation: 0}
    }
```

状态的改变是突然性的还是平滑过渡。使用 Transition 元素定义状态间的过渡，它可以绑定到项目的 `transitions` 属性。我们的文本编辑器在 DRAWER_OPEN 或 DRAWER_CLOSED 状态变化时有一个状态要转换。重要的是，转变需要需要 `from` 与 `to` 状态，我们可以使用通配符*符号来表示，过渡适用于所有状态的变化。

过渡期间，我们可以指定动画到属性的变化。我们 menuBar 从 y:0 到 y:-partition 位置切换并且使用 NumberAnimation 元素进行过渡。我们声明这个目标的属性将动画为一定的时间期限并使用一定的缓和曲线。缓和曲线控制状态过渡期间的动画率与插值的行为。缓和曲线我们选择的是 Easing.OutQuint，在动画接近结束的时候减慢动作。请阅读 QML 的动画文章。

```
transitions: [
        Transition{
            to: "*"
            NumberAnimation { target: textArea; properties: "y, height"; duration: 100; easing.type: Easing.OutQuint }
            NumberAnimation { target: menuBar; properties: "y"; duration: 100;easing.type: Easing.OutQuint }
            NumberAnimation { target: drawer; properties: "y"; duration: 100;easing.type: Easing.OutQuint }
        }
]
```

动画属性发生变化的另一个方法是声明一个行为的元素。仅在状态变化期间过渡并且常规属性变化的行为可以设置的动画。在文本编辑器，箭头有 NumberAnimation 动画旋转属性的变化。

```
In TextEditor.qml:

    Behavior{
        NumberAnimation{property: "rotation";easing.type: Easing.OutExpo }
    }
```

回到我们的组件，已经了解状态动画，我们可以改善组件的外观。在 Button.qml 里，我们可以当该按钮单击时，按钮 `color` 与 `scale` 属性的改变。颜色类型使用 ColorAnimation 动画的数值是使用 NumberAnimation 动画的。关于 on propertyName 的语法如下所示，当目标是单一属性是非常有帮助的。

```
In Button.qml:
    ...

    color: buttonMouseArea.pressed ? Qt.darker(buttonColor, 1.5) : buttonColor
    Behavior on color { ColorAnimation{ duration: 55} }

    scale: buttonMouseArea.pressed ? 1.1 : 1.00
    Behavior on scale { NumberAnimation{ duration: 55} }
```

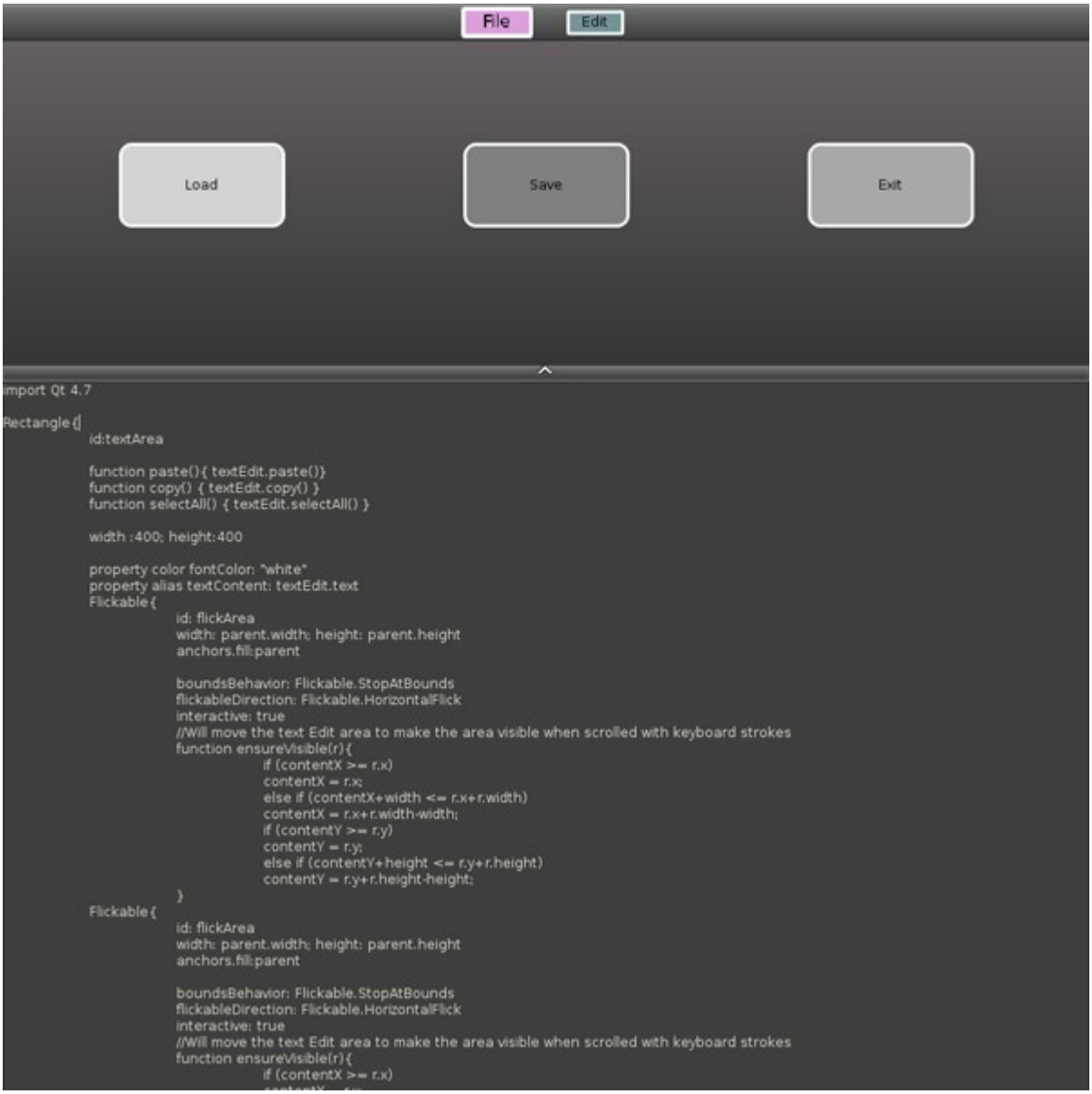
此外，我们可以通过添加颜色效果（例如渐变颜色和不透明度效果）来增强 QML 组件的外观。声明一个 Gradien 元素将覆盖元素的 `color` 属性。你可以在渐变里使用 GradientStop 元素声明颜色。渐变是使用 0.0 和 1.0 之间值。

```
In MenuBar.qml
    gradient: Gradient {
        GradientStop { position: 0.0; color: "#8C8F8C" }
        GradientStop { position: 0.17; color: "#6A6D6A" }
        GradientStop { position: 0.98;color: "#3F3F3F" }
        GradientStop { position: 1.0; color: "#0e1B20" }
```

该菜单栏使用了一个渐变来模拟深度。第一个颜色起始于 0.0 并且以 1.0 颜色结束。

到哪里去

我们完成了一个非常简单的文本编辑器用户界面。这样看来用户界面是完整的，我们可以实现使用的 Qt 应用程序逻辑和 C + +。QML 是非常好的原型工具，用户界面设计分离了应用程序逻辑。



使用 Qt C++来扩展 QML

现在我们有文本编辑器的布局，现在可以以 C + +实现文本编辑器的功能。带有 C++的 QML 允许我们能够使用 Qt 创建应用程序逻辑。我们可以创建在 C ++应用程序使用 Qt 的声明类并使用图形场景显示 QML 元素。另外我们可以导出 C + +代码为插件，qmlviewer 工具可以阅读。对于我们这个应用程序以 C++实现载入与保存功能并导出它作为一个插件。这样我们只需要加载 QML 文件而不直接运行一个可执行文件。

公开 C + +类到 QML

我们使用 Qt 和 C++实现文件的载入与保存。 C++类与函数通过注册它们可以用在 QML 里。该类还需要被编译成一个 Qt 插件并且 QML 文件需要知道该插件的位置。

对于我们的应用程序，我们需要创建以下项目：

1. Directory 类将处理文件夹相关的操作
2. File 类是一个 QObject，模拟在文件夹中的文件列表
3. plugin 类注册该类到 QML
4. Qt 工程文件将编译成插件
5. qmldir 文件告诉 qmlviewer 工具在哪里可以找到该插件

建设一个 Qt 插件

要建设一个插件，我们需要设置一个 Qt 工程文件。首先需要源文件、头文件和 Qt 模块加入到我们的工程文件中。所有的 C++代码和工程文件在 fileDialog 的目录内。

```
In cppPlugins.pro:

    TEMPLATE = lib
    CONFIG += qt plugin
    QT += declarative

    DESTDIR +=  ../plugins
    OBJECTS_DIR = tmp
    MOC_DIR = tmp

    TARGET = FileDialog

    HEADERS +=      directory.h \
            file.h \
            dialogPlugin.h

    SOURCES +=     directory.cpp \
            file.cpp \
            dialogPlugin.cpp
```

特别是我们使用 `declarative` 模块编译 Qt 并配置它作为一个插件，需要一个 lib 模板。我们应把编译插件放到该父级的 `plugins` 目录内。

注册一个类到 QML

```
In dialogPlugin.h:

    #include <QtDeclarative/QDeclarativeExtensionPlugin>

    class DialogPlugin : public QDeclarativeExtensionPlugin
    {
        Q_OBJECT

        public:
        void registerTypes(const char *uri);

    };
```

我们的插件类，DialogPlugin 是 QDeclarativeExtensionPlugin 的子类。我们需要实现继承函数 registerTypes()。dialogPlugin.cpp 文件看起来类似于下面那样：

DialogPlugin.cpp:

```
#include "dialogPlugin.h"
#include "directory.h"
#include "file.h"
#include <QtDeclarative/qdeclarative.h>

void DialogPlugin::registerTypes(const char *uri){

    qmlRegisterType<Directory>(uri, 1, 0, "Directory");
    qmlRegisterType<File>(uri, 1, 0,"File");

}

Q_EXPORT_PLUGIN2(FileDialog, DialogPlugin);
```

registerTypes()函数注册我们的文件和目录类到 QML。此函数需要它的模板类名、主要版本号、次版本号以及我们的类名。

我们需要使用 Q_EXPORT_PLUGIN2 宏输出插件。请注意在我们的 dialogPlugin.h 文件中已经有 Q_OBJECT 宏在类的顶部。同样需要在工程文件里运行 qmake 的生成所必需的元对象代码。

以 C++类创建 QML 属性

我们可以使用 C ++和 Qt 的元对象系统创建 QML 元素与属性。我们可以使用槽和信号实现属性，使 Qt 认识这些属性。然后这些属性可以被用在 QML。

对于这个文本编辑器，我们需要能够加载和保存文件。通常情况下，这些功能都包含在一个文件对话框中。幸运的是，我们可以使用的 QDir、QFile 以及 QTextStream 来实现目录读取以及输入/输出流。

```
class Directory : public QObject{

    Q_OBJECT

    Q_PROPERTY(int filesCount READ filesCount CONSTANT)
    Q_PROPERTY(QString filename READ filename WRITE setFilename NOTIFY filenameChanged)
    Q_PROPERTY(QString fileContent READ fileContent WRITE setFileContent NOTIFY fileContentChanged)
    Q_PROPERTY(QDeclarativeListProperty<File> files READ files CONSTANT )

    ...
```

Directory 类使用 Qt 的元对象系统来注册它需要完成的文件处理。 Directory 类是输出为一个插件并可用在 QML 里作为 Directory 元素。每个列出的属性都使用了 Q_PROPERTY 宏作为一个 QML 属性。

该 Q_PROPERTY 声明一个属性以及它读取与写入到 Qt 的元对象系统。例如 fileName 属性，QString 类型的，是使用 filename 函数()可读以及使用 setFilename()函数可写的。此外这里有个与文件名属性相关的信号为 filenameChanged()，它的属性改变时发射。在头文件里读写函数声明为公共。

同样，我们有其它属性声明给它们使用。filesCount 属性表示一个目录中的文件数量。FileName 属性是设置当前选定的文件名称，fileContent 属性是加载/保存文件的内容。

```
Q_PROPERTY(QDeclarativeListProperty<File> files READ files CONSTANT )
```

文件列表属性是列出目录中的所有文件。 Directory 类可以实现过滤文本文件以外的无效文件；只有 txt 扩展名是有效的。此外通过以 C++形式 QDeclarativeListPropertyQLists 声明它们可用于 QML 文件。模板化的对象需要从一个 QObject 继承；因此 File 类必须继承自 QObject。在 Directory 类里文件对象的列表是存储在一个名为 m_fileList 的 QList。

```
class File : public QObject{

    Q_OBJECT
    Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)

    ...
};
```

这些属性可以被用来作为在 QML 第 Directory 元素属性的一部分。请注意，我们在 C + +代码没有创造一个标识符的 ID 属性。

```
Directory{
    id: directory

    filesCount
    filename
    fileContent
    files

    files[0].name
}
```

由于QML 使用 JavaScript 的语法和结构，我们可以遍历文件列表和检索其属性。要检索的第一个文件名的属性，我们可以调用 files[0].name。

常规 C++的函数 QML 也能访问。以 C++实现的文件加载和保存功能是使用 Q_INVOKABLE 宏声明的。另外我们可以声明该函数作为一个槽并且该函数是可以被 QML 访问的。

```
In Directory.h:

    Q_INVOKABLE void saveFile();
    Q_INVOKABLE void loadFile();
```

Directory 类在目录内容改变时还必须通知其他对象。此功能是通过使用一个信号实现的。如前所述 QML 信号有一个自己的名字前面加上相应的 on 处理程序。该信号为 directoryChanged 并且每当目录刷新时就发射。刷新只是简单的重新加载该目录的内容并更新目录中的有效文件列表。QML 项目可以通过 onDirectoryChanged 信号处理程序来作出动作。

list 属性需要进一步研究。这是因为列表属性使用了回调访问并修改列表内容。 list 属性是 QDeclarativeListProperty <File>类型。每当列表访问，访问函数需要返回一个 QDeclarativeListProperty <File>类型。该模板类型 File 需要一个 QObject 的衍生物。此外为了创

建 QDeclarativeListProperty，该列表的访问和修改需要传递给该构造函数作为函数指针。该列表，QList 在我们的这种情况下还需要一个 File 指针列表。

对 QDeclarativeListProperty 构造函数和该 `Directory` 实现：

```
QDeclarativeListProperty  ( QObject * object, void * data, AppendFunction append, CountFunction count = 0, AtFunction at = 0, ClearFunction clear = 0 )
    QDeclarativeListProperty<File>( this, &m_fileList, &appendFiles, &filesSize, &fileAt,  &clearFilesPtr );
```

该构造函数传递指针将追加列表、列表计算、使用一个索引检索项目并且清空列表。只有附加函数是强制性的。请注意该函数指针必须符合 AppendFunction、CountFunction、AtFunction 或 ClearFunction 定义

```
void appendFiles(QDeclarativeListProperty<File> * property, File * file)
    File* fileAt(QDeclarativeListProperty<File> * property, int index)
    int filesSize(QDeclarativeListProperty<File> * property)
    void clearFilesPtr(QDeclarativeListProperty<File> *property)
```

为了简化我们的文件对话框，`Directory` 类过滤无效的文本文件，这些文件不具有 txt 扩展名。如果文件名称不具有 txt 扩展名；那么就不会出现在我们的文件对话框里。此外确保保存文件时文件名带有 txt 扩展名。 Directory 使用 QTextStream 来读取文件和输出文件内容到一个文件。

使用我们的 `Directory` 元素，我们可以检索文件作为一个列表，知道有多少文本文件在应用程序目录中，获取文件的名称与内容作为字符串并且每当有目录内容有变化时将发出通知。

要构建插件，在 cppPlugins.pro 工程文件中运行 qmake；然后运行 make 来构建和移动插件到 `plugins` 目录。

QML 导入一个插件

qmlviewer 工具导入与应用程序相同目录的文件。我们也可以创建一个 qmldir 文件，其中包含 QML 文件希望导入的位置。Qmldir 文件同样也可在储存插件和其他资源的位置。

```
In qmldir:

    Button ./Button.qml
    FileDialog ./FileDialog.qml
    TextArea ./TextArea.qml
    TextEditor ./TextEditor.qml
    EditMenu ./EditMenu.qml

    plugin FileDialog plugins
```

我们刚刚创建的插件称为 FileDialog，通过工程文件中的 TARGET 字段表示。编译的插件是在 `plugins` 目录中。

整合文件对话框到文件菜单中

我们的 `FileMenu` 需要显示 FileDialog 元素，包含目录里的文本文件列表，从而允许用户通过列表选择点击的文本文件。我们还需要指定保存、加载以及其它动作的新按钮。`FileMenu` 包含一个可编辑的文本输入栏，允许用户使用键盘键入一个文件名称。

该 `Directory` 元素用在 FileMenu.qml 文件，它通知 FileDialog 元素刷新该目录的内容。以信号处理程序 `onDirectoryChanged` 通知执行。

```
In FileMenu.qml:

    Directory{
       id:directory
       filename: textInput.text
       onDirectoryChanged: fileDialog.notifyRefresh()
    }
```

与我们的应用程序保持一致，文件对话框将始终可视并且不会显示无效的文本文件（不具有 txt 扩展名的文件）。

```
In FileDialog.qml:

    signal notifyRefresh()
    onNotifyRefresh: dirView.model = directo
```

FileDialog 元素将显示目录内的内容。这些文件被用来作为一个 GridView 元素的模型，它显示数据项目在网格布局到一代理。代理处理模型的外观以及我们的文件对话框将简单地创建一个居中的文本格的外观。单击一文件名称即出现一个矩形框以高亮显示文件名。每当重新加载目录时，将通知的 FileDialog 的 `notifyRefresh` 信号发射。

```
In FileMenu.qml:

    Button{
       id: newButton
       label: "New"
       onButtonClick:{
          textArea.textContent = ""
       }
    }
    Button{
       id: loadButton
       label: "Load"
       onButtonClick:{
          directory.filename = textInput.text
          directory.loadFile()
          textArea.textContent = directory.fileContent
       }
    }
    Button{
       id: saveButton
       label: "Save"
       onButtonClick:{
          directory.fileContent = textArea.textContent
          directory.filename = textInput.text
          directory.saveFile()
```

```
        }
    }
    Button{
        id: exitButton
        label: "Exit"
        onButtonClick:{
            Qt.quit()
        }
    }
```

我们的 `FileMenu` 现在可以连接到它们各自的动作。saveButton 将从 `TextEdit` 传输文本到目录的 `f`ileContent 属性，然后从可编辑的文本栏复制文件名。最后该按钮调用 saveFile()函数保存文件。该 loadButton 也有类似的动作。此外，new 的行动将是清空 `TextEdit` 的内容。

此外，EditMenu 按钮连接到 `TextEdit` 功能，复制、粘贴、以及选择所有文本。

完成文本编辑器

这个应用程序可以作为一个简单的文本编辑器，能够接受文本并保存到一个文件。文本编辑器还可以加载文件以及一些文本操作。

QML 与 Qt 程序

Qt Declarative UI 运行环境

QML 文档是通过 QML 运行环境载入并执行的。这包含 Declarative UI 引擎连同内置的 QML 元素与插件模块一起，它还提供了访问第三方 QML 元素与模块。

应用程序使用 QML，需要 QML 运行环境来执行 QML 文档。这可以通过创建 QDeclarativeView 或 QDeclarativeEngine 来完成，这在下面有描述。此外 Declarative UI 开发包包括了 Qt QML 查看工具，它可以载入.qml 文件。这个工具对于开发与测试 QML 代码；而不需要写 C++应用程序来载入 QML 运行环境是非常方便的。

部署基于 QML 的应用程序

要部署 QML 开发的应用程序，QML 运行环境必须被应用程序调用。这是通过编写 Qt C++应用程序加载 QDeclarativeEngine 完成：

- 通过 QDeclarativeView 实例加载 QML 文件
- 创建一个 QDeclarativeEngine 实例并使用 QDeclarativeComponent 加载 QML 文件。

使用 QDeclarativeView 部署

QdeclarativeView 是一个 QWidget 基类，它是能够加载 QML 文件。例如，如果有一个 QML 文件 application.qml 类似这样：

```
import Qt 4.7

Rectangle { width: 100; height: 100; color: "red" }
```

它在 Qt 应用程序的 main.cpp 文件载入类似于这样：

```
#include <QApplication>
#include <QDeclarativeView>

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

    QDeclarativeView view;
```

```
    view.setSource(QUrl::fromLocalFile("application.qml"));
    view.show();

    return app.exec();
}
```

这创建了一个 QWidget 基类，来查看 application.qml 的内容。

该应用程序的.pro 工程文件必须给 QT 变量指定 declarative 模块。例如：

```
TEMPLATE += app
QT += gui declarative
SOURCES += main.cpp
```

直接创建一个 QDeclarativeEngine

如果 application.qml 没有任何图形化的组件或是有其它的原因避免使用 QDeclarativeView，就可以直接用 QDeclarativeEngine 来代替。在这种情况下,application.qml 加载作为 QDeclarativeComponent 实例,而不是放入到一个视窗内:

```
#include <QCoreApplication>
#include <QDeclarativeEngine>

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

    QDeclarativeEngine engine;
    QDeclarativeContext *windowContext = new QDeclarativeContext(engine.rootContext());

    QDeclarativeComponent component(&engine, "application.qml");
    QObject *window = component.create(windowContext);

    // ... delete window and windowContext when necessary

    return app.exec();
}
```

查阅 在 C++程序中使用 QML 可以获取更多关于 QDeclarativeEngine,QdeclarativeContext 与 QDeclarativeComponent 详细使用信息，还可以参阅 Qt 的资源系统 获取一些 QML 文件的信息。

使用 QML 查看器部署与原型测试

Declarative UI 开发包包括了一个 QML 运行环境工具,Qt QML 查看器,它可以载入与显示 QML 文件。这是在应用程序开发阶段为原型测试基于 QML 应用程序；而不自己写 c++程序调用 QML 运行环境是非常有用的。

参阅 QML 查看器 文档获取更多细节。

在 C++程序中使用 QML

QML API 是分为三个主类——QDeclarativeEngine， QdeclarativeComponent 与 QDeclarativeContext。QDeclarativeEngine 提供 QML 运行的环境，
Q
d

QML 还包含了 API 的一个方便，通过 QDeclarativeView 应用程序只需要简单嵌入 QML 组件到一个新的 QGraphicsView 就可以了。这有许多细节将在下面讨论。QDeclarativeView 主要是用于快速成型的应用程序里。

如果你是重新改进使用 QML 的 Qt 应用程序，请参阅 整合 QML 到现有的 Qt UI 代码。

基本用法

每个应用程序至少需求一个 QDeclarativeEngine。QDeclarativeEngine 允许配置全局设置应用到所有的 QML 组件实例中，例如 QNetworkAccessManager 是用于网络通信以及永久储存的路径。如果应用程序需求在 QML 组件实例间需求不同的设置只需要多个 QDeclarativeEngine。

使用 QDeclarativeComponent 类载入 QML Documents。每个 QDeclarativeComponent 实例呈现单一 QML 文档。QDeclarativeComponent 可以传递一个文档的地址或文档的原始文本内容。该文档的 URL 可以是本地文件系统的地址或通过 QNetworkAccessManager 支持的网络地址。

QML 组件实例通过调用 QDeclarativeComponent::create()模式来创建。在这里载入一个 QML 文档的示例并且从它这里创建一个对象。

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

导出数据

QML 组件是以 QDeclarativeContext 实例化的。context 允许应用程序导出数据到该 QML 组件实例中。单个 QDeclarativeContext 可用于一应用程序的所有实例对象或针对每个实例使用 QDeclarativeContext 可以创建更为精确的控制导出数据。如果不传递一个 context 给 QDeclarativeComponent::create()模式；那么将使用 QDeclarativeEngine 的 root context。数据导出通过该 root context 对所有对象实例是有效的。

简单数据

为了导出数据到一个 QML 组件实例，应用程序设置 Context 属性；然后由 QML 属性绑定的名称与 JavaScrip 访问。下面的例子显示通过 QGraphicsView 如何导出一个背景颜色到 QML 文件中：

```
//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));

    view.setSource(QUrl::fromLocalFile("main.qml"));
    view.show();

    return app.exec();
 }
```

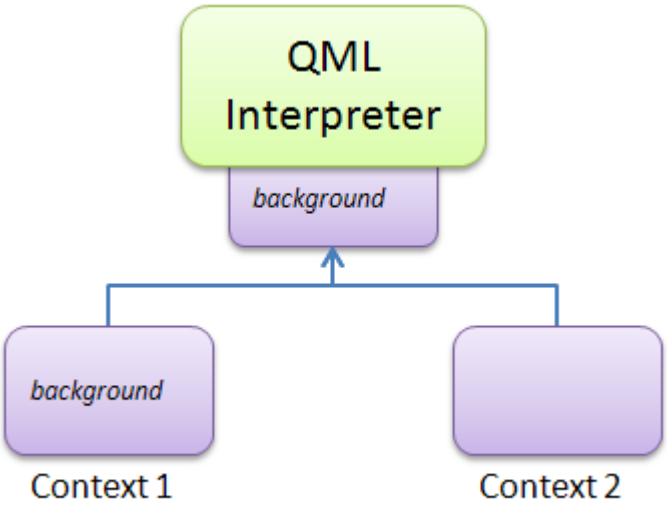
```
//main.qml
import Qt 4.7

 Rectangle {
    width: 300
    height: 300

    color: backgroundColor

    Text {
       anchors.centerIn: parent
       text: "Hello Yellow World!"
    }
 }
```

或者，如果你需要 main.cpp 不需要在 QDeclarativeView 显示创建的组件，你就需要使用 QDeclarativeEngine::rootContext()替代创建 QDeclarativeContext 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);
```

Context 属性的操作像 QML 绑定的标准属性那样——在这个例子中的 backgroundColor Context 属性改变为红色；那么该组件对象实例将自动更新。注意：删除任意 QDeclarativeContext 的构造是创建者的事情。当 window 组件实例撤消时不再需要 windowContext 时，windowContext 必须被消毁。最简单的方法是确保它设置 window 作为 windowContext 的父级。

QDeclarativeContexts 是树形结构——除了 root context 每个 QDeclarativeContexts 都有一个父级。子级 QDeclarativeContexts 有效的继承它们父级的 context 属性。这使应用程序分隔不同数据导出到不同的 QML 对象实例有更多自由性。如果 QDeclarativeContext 设置一 context 属性，同样它父级也被影响，新的 context 属性是父级的影子。如下例子中，background context 属性是 Context 1，也是 root context 里 background context 属性的影子。



结构化数据

context 属性同样可用于输出结构化与写数据到 QML 对象。除了 QVariant 支持所有已经存在的类型外，QObject 派生类型可以分配给 context 属性。 QObject context 属性允许数据结构化输出并允许 QML 来设置值。

下例创建 `CustomPalette` 对象并设置它作为 `palette` context 属性。

```
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();
}
```

QML 引用 palette 对象以及它的属性，为了设置背景与文本的颜色，这里是当单击窗口时，面板的文本颜色将改变成蓝色。

```
import Qt 4.7

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";
        }
    }
}
```

可以检测一个 C++属性值——这种情况下的 CustomPalette 的文本属性改变，该属性必须有相应的 NOTIFY 信息。NOTIFY 信号是属性值改变时将指定一个信号发射。

实现者应该注意的是，只有值改变时才发射信号，以防止发生死循环。访问一个绑定的属性，没有 NOTIFY 信号的话，将导致 QML 在运行时发出警告信息。

动态结构化数据

如果应用程序对结构化过于动态编译 QObject 类型；那么对动态结构化数据可在运行时使用 QDeclarativePropertyMap 类构造。

从 QML 调用 C++

通过 public slots 输出模式或 Q_INVOKABLE 标记模式使它可以调用 QObject 派生出的类型。

C++模式同样可以有参数并且可以返回值。QML 支持如下类型：

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

下面例子演示了，当 MouseArea 单击时控制 “Stopwatch”对象的开关。

```
//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();
 }
```

```
//main.qml
import Qt 4.7

 Rectangle {
    width: 300
```

```
    height: 300

    MouseArea {
        anchors.fill: parent
        onClicked: {
            if (stopwatch.isRunning())
                stopwatch.stop()
            else
                stopwatch.start();
        }
    }
}
```

值得注意的是，在这个特殊的例子里有更好的方法来达到同样的效果，在 main.qml 有" running"属性，这将会是一个非常优秀的 QML 代码：

```
// main.qml
import Qt 4.7

Rectangle {
    MouseArea {
        anchors.fill: parent
        onClicked: stopwatch.running = !stopwatch.running
    }
}
```

当然，它同样可以调用 functions declared in QML from C++。

网络组件

如果 URL 传递给 QDeclarativeComponent 是一网络资源 或者 QML 文档引用一网络资源，QDeclarativeComponent 要先获取网络数据；然后才可以创建对象。在这种情况下 QDeclarativeComponent 将有 Loading status。直到组件调用 QDeclarativeComponent::create
(

下面的例子显示如何从一个网络资源载入 QML 文件。在创建 QDeclarativeComponent 之后，它测试组件是否加载。如果是，它连接 QDeclarativeComponent::statusChanged()信号，否则直接调用 `continueLoading()`。这个测试是必要的，甚至 URL 都可以是远程的，只是在这种情况下要防组件是被缓存的。

```
MyApplication::MyApplication()
{
    // ...
    component = new QDeclarativeComponent(engine, QUrl("http://www.example.com/main.qml"));
    if (component->isLoading())
        QObject::connect(component, SIGNAL(statusChanged(QDeclarativeComponent::Status)),
                         this, SLOT(continueLoading()));
    else
        continueLoading();
}

void MyApplication::continueLoading()
{
```

```
    if (component->isError()) {
        qWarning() << component->errors();
    } else {
        QObject *myObject = component->create();
    }
}
```

Qt 资源

QML 的内容可以使用 qrc：URL 方案从 Qt 资源系统载入。例如：

[project/example.qrc]

```
<!DOCTYPE RCC>
 <RCC version="1.0">

 <qresource prefix="/">
    <file>main.qml</file>
    <file>images/background.png</file>
 </qresource>

 </RCC>
```

[project/project.pro]

```
QT += declarative

SOURCES += main.cpp
RESOURCES += example.qrc
```

[project/main.cpp]

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

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

    return app.exec();
}
```

[project/main.qml]

```
import Qt 4.7

Image {
    source: "images/background.png"
}
```

请注意，资源系统是不能从 QML 直接访问的。如果主 QML 文件被加载作为资源，所有的文件指定在 QML 中做为相对路径从资源系统载入。在 QML 层使用资源系统是完全透明的。这也意味着，如果主 QML 文件没有被加载作为资源，那么从 QML 不能访问资源系统。

与现有的 Qt 用户界面代码整合 QML

有很多方法整合 QML 到基于 QWidget 用户界面的应用程序，取决于现有用户界面代码的特征。

整合基于 QWidget 用户界面

如果你有一个基于 QWidget 用户界面的程序，QML 部件可以使用 QDeclarativeView 整合进去。QDeclarativeView 是 QWidget 的子类，你添加它就像你的用户界面添加 QWidget 任何其它的部件一样。使用 QDeclarativeView::setSource()来载入一个 QML 文件到视图里；然后添加该视窗到你的用户界面：

```
QDeclarativeView *qmlView = new QDeclarativeView;
qmlView->setSource(QUrl::fromLocalFile("myqml.qml"));

QWidget *widget = myExistingWidget();
QVBoxLayout *layout = new QVBoxLayout(widget);
widget->addWidget(qmlView);
```

这种方法有一个缺点就是 QDeclarativeView 初始化慢；比使用 QWidget 占用更多内存并且创建大量的 QDeclarativeView 对象也会导致性能下降。如果是这种情况，用 QML 重写你的用户界面部件并且从主 QML 部件载入该部件来替代 QDeclarativeView。

请记住，QWidgets 是不同于 QML 类型用户界面设计的；所以把基于 QWidgets 的应用程序导向 QML 并不是一个好主意。如果你的用户界面是一个复杂和少数静态元素组成的；那么 QWidgets 是最好的选择。如果的用户界面是一个简单和大量动态元素组成的；那么 QML 是最好的选择。

整合基于 QGraphicsView 用户界面

添加 QML 部件到一个 QGraphicsScene

如果你已有用户界面是基于 Graphics View Framework，你可以直接整合 QML 部件到你的 QGraphicsScene。从一个 QML 文件里使用 QDeclarativeComponent 来创建一个 QGraphicsObject，并且使用 QGraphicsScene::addItem()放置图形对象到你的场景中或者重新父级它到 QGraphicsScene 里已经存在的一个项目上。

例如：

```
QGraphicsScene* scene = myExistingGraphicsScene();
QDeclarativeEngine *engine = new QDeclarativeEngine;
QDeclarativeComponent component(engine, QUrl::fromLocalFile("myqml.qml"));
QGraphicsObject *object =
    qobject_cast<QGraphicsObject *>(component.create());
scene->addItem(object);
```

以下 QDeclarativeView 选项是适合 QML 用户界面最佳表现的：

- QGraphicsView::setOptimizationFlags(QGraphicsView::DontSavePainterState)
- QGraphicsView::setViewportUpdateMode(QGraphicsView::BoundingRectViewportUpdate)
- QGraphicsScene::setItemIndexMethod(QGraphicsScene::NoIndex)

在 QML 里载入 QGraphicsWidget 对象

另一种方法是将你现有的 QGraphicsWidget 对象输出到 QML 并以 QML 构造你的场景。参阅 graphics layouts example 例子，它展示了如何使用 QGraphicsWidget 输出 Qt 图形布局类到 QML。

为了输出你现有 QGraphicsWidget 类到 QML，使用 qmlRegisterType()。参阅稍后介绍的 Extending QML in C++。

教程：用 C++写 QML 扩展

QtDeclarative 模块提供了一套 API 给 C++来扩展 QML。你可以写扩展来添加自己的 QML 类型，扩展现有 Qt QML 类型或者调用 C/C++函数，它不能由普通 QML 代码访问。

这个教程显示如何使用 C++写 QML 扩展，包括 QML 核心特征、属性、信号以及绑定。它同样显示如何通过插件来部署扩展。

该教程的源代码在 Qt 的 examples/declarative/tutorials/extending 目录下。

第一节：创建新类型

文件：

- declarative/tutorials/extending/chapter1-basics/app.qml
- declarative/tutorials/extending/chapter1-basics/piechart.cpp
- declarative/tutorials/extending/chapter1-basics/piechart.h
- declarative/tutorials/extending/chapter1-basics/main.cpp
- declarative/tutorials/extending/chapter1-basics/chapter1-basics.pro

有些自定义功能超出内置 QML Elements；这需要扩展 QML 来提供一个新的 QML 类型支持。例如：可以实现特定的数据模型或提供用于自定义绘画与绘图功能的元素，或者像网络编程访问系统功能元素，这是通过内置的 QML 特性无法完成的。

在这节教程里，我们将显示如何在 QtDeclarative 模块里使用 C++类来扩展 QML。这个程序运行的最终结果是通过几个自定义 QML 类型链接 QML 特性（像绑定、信号以及通过插件 QML 运行环境）显示一个简单的饼图。

我们建一个名为“PieChart”的新 QML 类型，它有两个属性：名称 与颜色。我们使它在一“Charts”模块有效，模块的版本为 1.0。

我们需要 PieChart 类型，类似于下面代码：

```
import Charts 1.0

PieChart {
    width: 100; height: 100
    name: "A simple pie chart"
    color: "red"
}
```

需要一 C++类封装这个 PieChart 类型以及它的两个属性。由于 QML 大量使用 Qt 元对象系统，这个新类必须：

- 继承 QObject
- 使用 Q_PROPERTY 宏声明它的属性

在 piechart.h 里是我们的 PieChart 类：

```
#include <QDeclarativeItem>
#include <QColor>

class PieChart : public QDeclarativeItem
{
    Q_OBJECT
    Q_PROPERTY(QString name READ name WRITE setName)
    Q_PROPERTY(QColor color READ color WRITE setColor)

public:
    PieChart(QDeclarativeItem *parent = 0);

    QString name() const;
    void setName(const QString &name);

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

    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget = 0);

private:
    QString m_name;
    QColor m_color;
};
```

该类继承 QDeclarativeItem，为了重新定义绘制功能，我们需要重载 QDeclarativeItem::paint()函数。如果该类只描述了一些数据类型并不是真正需要显示一项目，它可以只简单的继承 QObject。或者如果我们需要扩充一个基于 QObject 类的功能，它可以直接继承那个类。

PieChart 类用 Q_PROPERTY 定义两个属性：name 与 color 并重载 QDeclarativeItem::paint()。这个类的实现是在 piechart.cpp 里，只需简单的设置并返回 m_name 与 m_color 值以及实现 paint() 来绘制一个简单的饼图。它还关闭 QGraphicsItem::ItemHasNoContents 标志来启用绘画：

```
PieChart::PieChart(QDeclarativeItem *parent)
    : QDeclarativeItem(parent)
{
    // need to disable this flag to draw inside a QDeclarativeItem
    setFlag(QGraphicsItem::ItemHasNoContents, false);
}
...
void PieChart::paint(QPainter *painter, const QStyleOptionGraphicsItem *, QWidget *)
{
    QPen pen(m_color, 2);
    painter->setPen(pen);
    painter->setRenderHints(QPainter::Antialiasing, true);
    painter->drawPie(boundingRect(), 90 * 16, 290 * 16);
}
```

现在我们已经定义了 PieChart 类型，我们将用在 QML 里。这个 app.qml 文件使用一个标准的 QML Text 项目来创建一个 PieChart 项目并显示饼图的细节：

import Charts 1.0

```
import Qt 4.7

Item {
    width: 300; height: 200

    PieChart {
        id: aPieChart
        anchors.centerIn: parent
        width: 100; height: 100
        name: "A simple pie chart"
        color: "red"
    }

    Text {
        anchors { bottom: parent.bottom; horizontalCenter: parent.horizontalCenter; bottomMargin: 20 }
        text: aPieChart.name
    }
}
```

注意：尽管颜色在 QML 里被指定为一个字符串，它是自动转换为一个用于 PieChart 颜色属性的 QColor 对象。自动转换是用于其它各种不同的基本类型的转换；例如一个字符串” 640*480”可以自动被转换成 QSize 类型的值。

我们的 C++程序同样可以使用 QDeclarativeView 来运行并显示 app.qml。这个应用程序必须使用 qmlRegisterType()函数注册 PieChart 类型，这样才能被允许用在 QML 里。如果你没有注册这个类型，app.qml 将不允许创建一个饼图。

这个应用程序的 main.cpp 代码：

```
#include "piechart.h"
 #include <qdeclarative.h>
```

```
#include <QDeclarativeView>
#include <QApplication>

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

    qmlRegisterType<PieChart>("Charts", 1, 0, "PieChart");

    QDeclarativeView view;
    view.setSource(QUrl::fromLocalFile("app.qml"));
    view.show();
    return app.exec();
}
```

这里调用 qmlRegisterType()函数注册 PieChart 类型，在名为” Charts”模块里，模块的版本号为 1.0。

最后，在工程文件里包含

```
QT += declarative

HEADERS += piechart.h
SOURCES += piechart.cpp \
           main.cpp
```

现在我们构建并运行该应用程序如下图：

A simple pie chart

自己试着运行一下这个代码看看效果。

目前是在 C++应用程序里运行 app.qml。这可能看上去很奇怪。如果你用 QML 查看器来查看 QML 文件，后面我们将展示如何创建一个插件让你的程序可以运行在 QML 查看器里。

第二节：连接到 C++方法与信号

文件：

- declarative/tutorials/extending/chapter2-methods/app.qml
- declarative/tutorials/extending/chapter2-methods/piechart.cpp

- declarative/tutorials/extending/chapter2-methods/piechart.h
- declarative/tutorials/extending/chapter2-methods/main.cpp
- declarative/tutorials/extending/chapter2-methods/chapter2-methods.pro

假设我们需要 PieChart
有”clearChart

```
import Charts 1.0
import Qt 4.7

Item {
    width: 300; height: 200

    PieChart {
        id: aPieChart
        anchors.centerIn: parent
        width: 100; height: 100
        color: "red"

        onChartCleared: console.log("The chart has been cleared")
    }

    MouseArea {
        anchors.fill: parent
        onClicked: aPieChart.clearChart()
    }

    Text {
        anchors { bottom: parent.bottom; horizontalCenter: parent.horizontalCenter; bottomMargin: 20 }
        text: "Click anywhere to clear the chart"
    }
}
```

为了做到这一步，添加一个 clearChart()方法与 `chartCleared()` 信号到我们的 C++类里：

```
class PieChart : public QDeclarativeItem
{
    ...
public:
    ...
    Q_INVOKABLE void clearChart();

signals:
    void chartCleared();
    ...
};
```

Q_INVOKABLE 的用途，使 clearChart()方法可用于 Qt 元对象系统并对 QML 也有用。注意它也可以被声明为一个 Qt 槽来替代 Q_INVOKABLE。作为槽同样也能被 QML 调用。这两种方法都是有效的。

clearChart()方法只是简单的改变颜色为 Qt::transparent，重绘图表；然后发射 `chartCleared()` 信号。

```
void PieChart::clearChart()
{
    setColor(QColor(Qt::transparent));
    update();

    emit chartCleared();
}
```

当运行该应用程序后，单击该窗口，这个饼图将被禁止显示，然后输出如下信息：

```
The chart has been cleared
```

尝试更新一下你的代码，或者去 QT 范例文件夹编译运行看看效果。

第三节：添加属性绑定

文件：

- declarative/tutorials/extending/chapter3-bindings/app.qml
- declarative/tutorials/extending/chapter3-bindings/piechart.cpp
- declarative/tutorials/extending/chapter3-bindings/piechart.h
- declarative/tutorials/extending/chapter3-bindings/main.cpp
- declarative/tutorials/extending/chapter3-bindings/chapter3-bindings.pro

属性绑定是 QML 非常强的一个特性，这允许不同元素的值自动同步。当属性值改变时它使用信号来通知并更新其它的元素值。

记我们启用 color 属性的属性绑定。代码如下：

```
import Charts 1.0
 import Qt 4.7

 Item {
    width: 300; height: 200

    Row {
       anchors.centerIn: parent
       spacing: 20

       PieChart {
          id: chartA
```

```
            width: 100; height: 100
            color: "red"
        }

        PieChart {
            id: chartB
            width: 100; height: 100
            color: chartA.color
        }
    }

    MouseArea {
        anchors.fill: parent
        onClicked: { chartA.color = "blue" }
    }

    Text {
        anchors { bottom: parent.bottom; horizontalCenter: parent.horizontalCenter; bottomMargin: 20 }
        text: "Click anywhere to change the chart color"
    }
}
```

Click anywhere to change the chart color

"color: chartA.color" 语句绑定 chartB 的 color 值到 chartA 的 color。每当 chartA 的 color 值改变时；chartB 的 color 值将自动更新为同样的值。当该窗口单击时；在 MouseArea 里的 `onClicked` 处理程序将改变 `chartA` 的颜色。因此所有的饼图的颜色为蓝色。

`color` 属性开启属性绑定是非常容易的。我们添加 NOTIFY 特性到它的 Q_PROPERTY()声明，当值改变时将发射" `colorChanged`"信号。

```
class PieChart : public QDeclarativeItem
{
    ...
    Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
public:
    ...
signals:
    void colorChanged();
    ...
};
```

然后我们在

```
void PieChart::setColor(const QColor &color)
{
    if (color != m_color) {
```

```
        m_color = color;
        update();   // repaint with the new color
        emit colorChanged();
    }
}
```

在发射
colorChanged
(

绑定的使用是 QML 的基本操作。你应该随时为属性添加 NOTIFY 信号；以此你的属性是可以被绑定的。属性没有绑定，就不能自动更新，这在 QML 使用也就没有弹性了。一旦绑定了就调用频繁，也被 QML 信赖。如果绑定没有被执行；那么你自定义的 QML 类型可能会出现意外的行为。

第四节：使用自定义属性类型

文件：

- declarative/tutorials/extending/chapter4-customPropertyTypes/app.qml
- declarative/tutorials/extending/chapter4-customPropertyTypes/piechart.cpp
- declarative/tutorials/extending/chapter4-customPropertyTypes/piechart.h
- declarative/tutorials/extending/chapter4-customPropertyTypes/pieslice.cpp
- declarative/tutorials/extending/chapter4-customPropertyTypes/pieslice.h
- declarative/tutorials/extending/chapter4-customPropertyTypes/main.cpp
- declarative/tutorials/extending/chapter4-customPropertyTypes/chapter4-customPropertyTypes.pro

PieChart 类型目前有一个字符串类型属性与颜色类型属性。它可以有许多其它类型的属性。例如，它可以有一个枚举类型的属性来保存每个图表的显示模式：

```
// C++
class PieChart : public QDeclarativeItem
{
    Q_ENUMS(DisplayMode)
    Q_PROPERTY(DisplayMode displayMode READ displayMode WRITE setDisplayMode)
    ...

public:
    enum DisplayMode {
        MultiLevel,
        Exploded,
        ThreeDimensional
    };

    void setDisplayMode(DisplayMode mode);
    DisplayMode displayMode() const;
    ...

};
```

```
// QML
PieChart {
    ...
    displayMode: PieChart.Exploded
}
```

我们可以使用各种其它属性类型。QML 内置支持如何类型：

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

我们新建的一个属性，默认情况下 QML 并不支持；这需要注册该类型。

例如，让我们使用一个称为 PieSlice 属性类型来替代一个颜色属性。而不是指定一个颜色，我们指定个 PieSlice 值，它本身包含一个颜色。

```
import Charts 1.0
 import Qt 4.7

 Item {
    width: 300; height: 200

    PieChart {
        id: chart
        anchors.centerIn: parent
        width: 100; height: 100

        pieSlice: PieSlice {
            anchors.fill: parent
            color: "red"
        }
    }

    Component.onCompleted: console.log("The pie is colored " + chart.pieSlice.color)
}
```

像 `pieChart`，这个新的 `PieChart` 类型继承 QDeclarativeItem 并使用 Q_PROPERTY()声明它的属性：

```
class  PieSlice : public QDeclarativeItem
 {
```

```
    Q_OBJECT
    Q_PROPERTY(QColor color READ color WRITE setColor)

public:
    PieSlice(QDeclarativeItem *parent = 0);

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

    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget = 0);

private:
    QColor m_color;
};
```

为了在 PieChart 中使用它，我们修改 color 属性声明与关联方法：

```
class PieChart : public QDeclarativeItem
{
    Q_OBJECT
    Q_PROPERTY(PieSlice* pieSlice READ pieSlice WRITE setPieSlice)
    ...
public:
    ...
    PieSlice *pieSlice() const;
    void setPieSlice(PieSlice *pieSlice);
    ...
};
```

在这里实现 setPieSlice() 有件事要做。PieSlice 是一个可视项目；所以必须被设置作为该 PieChart 的子级，使用

```
void PieChart::setPieSlice(PieSlice *pieSlice)
{
    m_pieSlice = pieSlice;
    pieSlice->setParentItem(this);
}
```

像 PieChart 类型一样，PieSlice 类型必须使用 qmlRegisterType()

```
int main(int argc, char *argv[])
{
    ...
    qmlRegisterType<PieSlice>("Charts", 1, 0, "PieSlice");
    ...
}
```

试着编译这个例子。

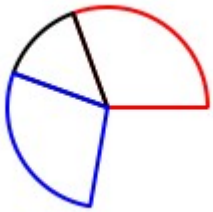
第五节：使用列表属性类型

文件：

- declarative/tutorials/extending/chapter5-listproperties/app.qml
- declarative/tutorials/extending/chapter5-listproperties/piechart.cpp
- declarative/tutorials/extending/chapter5-listproperties/piechart.h
- declarative/tutorials/extending/chapter5-listproperties/pieslice.cpp
- declarative/tutorials/extending/chapter5-listproperties/pieslice.h
- declarative/tutorials/extending/chapter5-listproperties/main.cpp
- declarative/tutorials/extending/chapter5-listproperties/chapter5-listproperties.pro

现在，一个 PieChart 仅只一个 PieSlice。理想的情况下一个图表应有多个片，它有不同颜色与大小。要做到这一点；我们可以有一个片的属性，PieSlice 项目列表：

```
import Charts 1.0
 import Qt 4.7

 Item {
    width: 300; height: 200

    PieChart {
       anchors.centerIn: parent
       width: 100; height: 100

       slices: [
          PieSlice {
             anchors.fill: parent
             color: "red"
             fromAngle: 0; angleSpan: 110
          },
          PieSlice {
             anchors.fill: parent
             color: "black"
             fromAngle: 110; angleSpan: 50
          },
          PieSlice {
             anchors.fill: parent
             color: "blue"
             fromAngle: 160; angleSpan: 100
          }
       ]
    }
 }
```



要做到这一点，我们替换一个切片属性 PieChart 里 pieSlice 属性，声明为 QDeclarativeListProperty 类型。QDeclarativeListProperty 类允许在 QML 扩展里创建一个列表属性。我们用
slices
(

```
class PieChart : public QDeclarativeItem
{
    Q_OBJECT
    Q_PROPERTY(QDeclarativeListProperty<PieSlice> slices READ slices)
    ...
public:
    ...
    QDeclarativeListProperty<PieSlice> slices();

private:
    static void append_slice(QDeclarativeListProperty<PieSlice> *list, PieSlice *slice);

    QString m_name;
    QList<PieSlice *> m_slices;
};
```

尽管切片属性并没有关联 WRITE 功能；但由于 QDeclarativeListProperty 的原因它仍然可以被修改。执行 PieChart ，我们实现
PieChart::slices
(

```
QDeclarativeListProperty<PieSlice> PieChart::slices()
{
    return QDeclarativeListProperty<PieSlice>(this, 0, &PieChart::append_slice);
}

void PieChart::append_slice(QDeclarativeListProperty<PieSlice> *list, PieSlice *slice)
{
    PieChart *chart = qobject_cast<PieChart *>(list->object);
    if (chart) {
        slice->setParentItem(chart);
        chart->m_slices.append(slice);
    }
}
```

append_slice
(
) 函数简单设置它父级项目；添加新项目到 m_slices 列表。正如你所看到的，用于

PieSlice 类也同样被修改，包括 fromAngle 与 angleSpan 属性，通过这些值来绘制切片。这只是一个简单的修改，如果你是教程第一节看过来的，那么知道该如何修改代码，这里不能显示完整代码，

当然你可以到范例文件夹查看完整代码并试着运行。

第六节：写一个扩展插件

文件：

- declarative/tutorials/extending/chapter6-plugins/app.qml
- declarative/tutorials/extending/chapter6-plugins/chartsplugin.cpp
- declarative/tutorials/extending/chapter6-plugins/chartsplugin.h
- declarative/tutorials/extending/chapter6-plugins/piechart.cpp
- declarative/tutorials/extending/chapter6-plugins/piechart.h
- declarative/tutorials/extending/chapter6-plugins/pieslice.cpp
- declarative/tutorials/extending/chapter6-plugins/pieslice.h
- declarative/tutorials/extending/chapter6-plugins/chapter6-plugins.pro
- declarative/tutorials/extending/chapter6-plugins/qmldir

目前 app.qml 都用了 PieChart 与 PieSlice 两种类型。这是在一个 C++应用程序使用 QDeclarativeView 来显示的。另一种方法就是使用我们的 QML 扩展来创建一个插件库来使它对 QML 引擎有效。这允许 app.qml 被 QML Viewer（或一些其它的 QML 运行环境应用程序）载入；而不是用 C++应用程序中编写 main.cpp 来载入。

为了创建一个插件库，我们需要：

- 一个插件类来注册我们的 QML 类型。
- 一个工程文件来描述该插件。
- 一个 qmldir 文件来告诉 QML 引擎来载入该插件。

首先，我们创建一个插件类，命名为 ChartsPlugin。它是 QDeclarativeExtensionPlugin 子类并以继承 registerTypes()方法来注册我们 QML 类型。它还需要调用 Q_EXPORT_PLUGIN2 用于 Qt 的插件系统。

`chartsplugin.h` 里这样定义 `ChartsPlugin`：

```
#include <QDeclarativeExtensionPlugin>

class ChartsPlugin : public QDeclarativeExtensionPlugin
{
    Q_OBJECT
public:
    void registerTypes(const char *uri);
};
```

chartsplugin.cpp 里实现：

```
#include "piechart.h"
 #include "pieslice.h"
 #include <qdeclarative.h>

 void ChartsPlugin::registerTypes(const char *uri)
 {
    qmlRegisterType<PieChart>(uri, 1, 0, "PieChart");
    qmlRegisterType<PieSlice>(uri, 1, 0, "PieSlice");
 }

 Q_EXPORT_PLUGIN2(chartsplugin, ChartsPlugin);
```

然后我们写一个.pro 工程文件，这样定义工程为一个插件库并指定 DESTDIR，它的库文件将编译生成到” lib”子目录里：

```
TEMPLATE = lib
 CONFIG += qt plugin
 QT += declarative

 DESTDIR = lib
 OBJECTS_DIR = tmp
 MOC_DIR = tmp

 HEADERS += piechart.h \
            pieslice.h \
            chartsplugin.h

 SOURCES += piechart.cpp \
            pieslice.cpp \
            chartsplugin.cpp

 symbian {
    include($$QT_SOURCE_TREE/examples/symbianpkgrules.pri)
    TARGET.EPOCALLOWDLLDATA = 1
 }
```

最后，我们添加一个 qmldir 文件，它是自动被 QML 引擎解析的。在这个文件里，我们指定插件名为 “chapter6-plugin”（这个名是范例工程名）可以在” lib”子目录找到它。

```
plugin chapter6-plugins lib
```

现在我们有一个插件，这样不用 main.cpp 也能执行了，我们可以通过 QML Viewer 直接载入该 QML 文件了。

```
qmlviewer app.qml
```

注意该"import Charts 1.0"语句是禁止在 app.qml 出现的。这是因为 qmldir 文件也在 app.qml 所在目录里，这就等于有 piechart.qml 与 pieslice.qml 文件在这个工程目录里。

第七节：总结

在这个教程里，我们显示了创建一个 QML 扩展的基本步骤：

- 通过 QObject 子类定义新的 QML 类型并使用 qmlRegisterType()注册它们。
- 使用 Q_INVOKABLE 或 Qt 槽添加可以调用模式并使用 `onSignal` 语法链接到 Qt 信号。
- 通过定义 NOTIFY 信号添加属性绑定。
- 如果内置类型没有的可以定义自定义属性类型。
- 使用 QDeclarativeListProperty 定义列表属性类型。
- 通过定义一个 Qt 插件并编写一个 `qmldir` 文件创建一个插件库。

Extending QML in C++参考文档显示其它有用的特性，它可添加到 QML 扩展。例如：我们可以使用 default properties 来允许不使用 `slices` 属性来添加切片：

```
PieChart {
    PieSlice { ... }
    PieSlice { ... }
    PieSlice { ... }
}
```

或使用 property value sources 随意添加与移除切片：

```
PieChart {
    PieSliceRandomizer on slices {}
}
```

参阅 reference documentation 用于更详细的信息。

Qt Quick 针对 C++开发者入门

今天的消费者与企业用户都是很难搞定的。他们是看游戏与电影里炫丽的用户界面长大的；那些中规中矩的界面不再是卖点了。视觉上的冲击力才是今天的卖点。现在的消费者可以用上强大的笔记本电脑、机顶盒或移动设备。

这需要设计师与开发人员脱颖而出，Qt Quick 是建立在今天产品团队的工作方式。核心逻辑代码是由开发者进行性能优化；界面则由设计师使用可视化工具完成，这样大家可以分工完成。

Qt Quck 的快速性能，是因为它提供了 Qt 应用程序与用户界面框架的构建。Qt 框架是运行性能高，占内存小；非常适合开发移动、嵌入式与上网本的应用程序。

Qt Quck 的扩展是使用 QML 的 Qt 框架，一个描述性语言代码，非常适合设计师编写代码。一个故事板画面是被声明为元素树的一个分支；每个画面的可视比率被声明为分支上元素的属性；在画面间过渡是可以用各种效果的动画。

Qt Quck 包括 Qt Creator 工具，一个集成开发环境；包括界面与代码编写。设计师是工作在可视环境里，开发者工作完全特性的

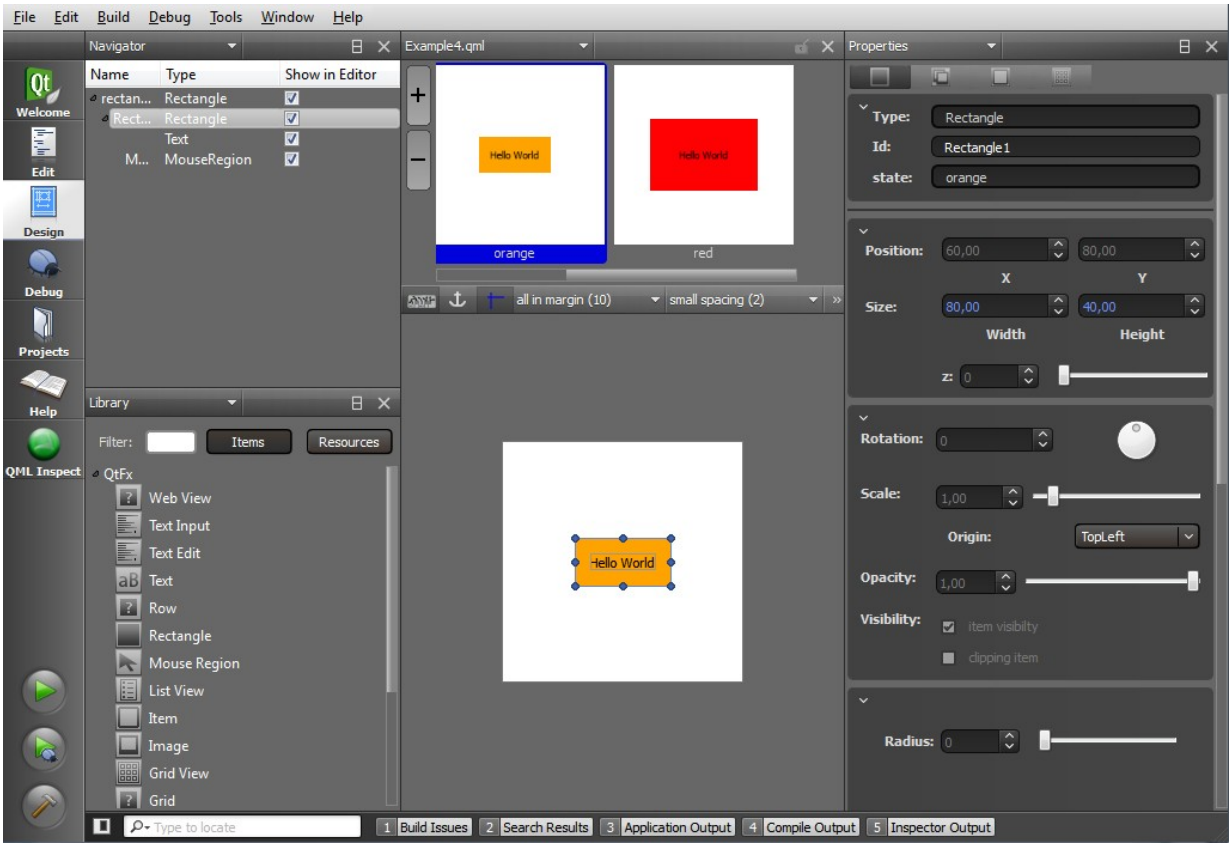
Qt Quic 概况

Qt Quick 是由 QML 语言、QtDeclarative C++模块（整合了 C++对象的 QML 语言）以及现在支持 QML 环境的 Qt Creator 工具组成。Qt Quick 帮助开发者与设计师合作来构建流畅的用户界面，可以为便携式消费设备（如：移动电话、媒体播放器、机顶盒以及上网本）开发炫的应用程序。使用 QtDeclarative C++模块，你可以从 Qt 应用程序载入 QML 文件进行互动。

QML 提供机制来使用 QML 元素声明构建一个对象树。QML 提高了 JavaScript 与基于 Qt 已存在的 QObject 类型系统间的整合，增加了自动属性绑定的支持并提供了语言层次的网络透明度。

QML 元素是一套先进的图形与行为构建块的集合。这些不同的元素组合在一起，在 QML 文件里从简单的按钮与滑块到复杂完整的网络功能的应用程序，类似于 Flickr 图片浏览器。

Qt Quick 是建立于 Qt 固有优势上。QML 可以用于逐步扩展已有的应用程序或构建全新的应用程序。由 C++通过 QtDeclarative 模块来完全扩展 QML 的功能。



用于 Qt Quck 组件的 Qt Creator 的用户界面

QML 简介

QML 是一个丰富的语言，完全介绍已经走出本章的范围。本章将介绍 QML 如何结合 C++编写应用程序：高效的逻辑由 C++完成，高动态用户界面由 QML 完成。QML 完全处理可以通过官方的在线文档获取。

理解 QML，先得从元素概念开始。一个元素是 QML 程序的基本组成部分。QML 支持如 Rectangle 与 Text 类型的可视元素，MouseArea 与 Flipable 类型的交互式元素，RotationAnimation 与 Transition 类型的动画元素。也有复杂的元素类型，允许开发人员处理数据，实现以模型视图架构来显示数据，还有其它内部元素类，这里只是补充一下。

所有的 QML 元素包括一个或更多的属性（例如：颜色）；可以由开发人员控制。许多元素包含信号（例如 onClicked），可用于对事件作出反应或状态变化。

可视元素：你好世界

这个例子是必须要了解的。在这里的代码是要求在一个简单背景的矩形框的顶部放置 Hello World 文本。

```
import Qt 4.7
Rectangle {
    width: 300
    height: 200
    Text {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        text: ”Hello World”
    }
}
```

让我们剖析这个简单的代码。这个 Hello World 例子是一个 QML 文件，这意味着它是一个完整的 QML 代码。是可以运行的，QML 文档一般是硬盘上的纯文本文件或网络资源，也可以是直接构建的文本数据。

QML 文档总是有一个或多个导入语句。在这里你看到 Qt 4.7 的导入。为了避免后继版本引入的元素影响现在的 QML 文档，文档内只用导入 QML 模块里有效的元素类型。这也就说 QML 是一个版本化的语言。

接下来，你看到的 Rectangle 元素模板用于创建一个活动对象。对象是可以包含其它的对象，建立父子关系。在上述代码中 Rectangle 对象是 Text 对象的父级对象。Rectangle 元素还定义了用于管理全用户界面的可视边框与焦点分割的顶级窗口。

技术说明：QML 元素的子级属性包含该元素所有可视子级的列表，资源属性包含所有非可视对象的列表。这两个列表都是默认隐性的填充，或者你可以明确指定它们。第三个属性，数据是一个列表包含了整个子级或资源列表中的对象。你可能没有明确填充该数据属性；如果你需要遍历可视与非可视对象的列表是非常有用的，所以你可以这样写：

```
Item {
    Text {}
    Rectangle {}
    Timer {}
}
```

代替：

```
Item {
    children: [ //default property and implicitly assigned
        Text {},
        Rectangle {}
    ]
    resources: [ //default property and implicitly assigned
        Timer {}
    ]
}
```

在对象里，属性是使用 property 绑定值的：表达式语句。在这里有两个方面：

首先，表达式是一个 JavaScript 表达式，这意味着你可以基于一个计算、一个条件或其它复杂的 JavaScript 操作来设置属性。例如：你可以基于一个变量定向值来设置该矩形的长宽比。

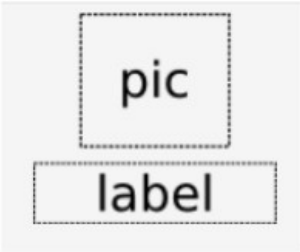
其次，是绑定不同的赋值语句。对于一个赋值语句中属性的值是赋值语句执行来设置的。对于绑定一个属性是绑定语句第一次执行来设置；但会改变，如果表达式的结果用于设置属性改变。（如果需要，你可以指定一个值到属性，使用 JavaScrip 语法 属性 = 表达式）。

当方向变化（从纵向到横向，有些移动设备有传感器）时要考虑会发生什么。因为属性绑定，父级矩形的长宽比将改变并且文本元素会生变化，重新定位在矩形的中心。

技术说明：属性绑定是由一个 NOTIFY 信号在 Qt C++里使用 Q
PROPERTY
(
)宏声明的对象来实现的。如果你不知道这是什么意思，不要担心。如果你计划纯粹使用 QML 构建应用

anchors.horizontalCenter: parent.horizontalCenter 语句文本元素的中心对齐父级矩形的中心。Anchors 提供了一种通过指定父级或同级项目的关系来定位该项目。（注意：如果通过在线文档查看 Rectangle 元素你将不会看到 anchors.horizontalCenter 属性。因为 Rectangle 元素继承了 QML 项目元素的属性；Item 元素提供了 anchors.horizontalCenter 属性。）

目前有十七个锚定属性，允许你来对齐、中心以及相对于其它来设置边距与偏移填充元素。例如：下图显示文本元素固定在一个图像元素水平居中以及带有边距的垂直下方。




```
Text {
    id: label
    anchors.horizontalCenter: pic.horizontalCenter
    anchors.top: pic.bottom
    anchors.topMargin: 5
    ...
}
```

图层化可视元素

QML 可视元素可以被分层在其它元素的顶部，使用 opacity :real（在这里实数型从 0（透明）到 1（不透明）来控制透明度来显示下面的元素。出于性能的考虑，这应该谨慎使用，特别是在场景中的需要动画每个图层；这将在运行时动画每帧都需要被渲染；会大大降低性能，因此在最终部署之前，最好为尽可能多的场景做预渲染；然后在运行时只需要简单的载入像素。

下图两个位移与重叠的矩形，一红一蓝，都具有透明性；在重叠的区域将会产生紫色。请注意子级矩形（蓝色）继承父级（红色）矩形的 5 0 %透明度。

```
Rectangle {
    opacity: 0.5
    color: "red"
    width: 100; height: 100
    Rectangle {
        color: "blue"
        x: 50; y: 50; width: 100; height: 100
    }
}
```

技术说明：继承元素总是继承它们基元素属性；即，一个矩形是一个项目；所以项目里的所有属性全都在矩形元素里出现。子对象在有些时间继承它们父级属性应归于
QGrap
h
icsView 的工作方式。这意味着你在一个元素里嵌入另一个元素时，某些属性对它的子级是有影响的。

互动元素：鼠标与触摸

要添加鼠标与触摸互动，你需要添加一个 MouseArea 对象。MouseArea 对象让你使用鼠标的单击与拖动（或触摸点）。其它有效的互动元素包含：Flickable、Flipable 与 FocusScope。

请注意 MouseArea 对象可以从任何视觉对象分离，提供设计师的灵活性。例如：为了创建一个用于用户单击可视按钮，它围绕该按钮可视面积更大的鼠标区域允许用户来丢失几个像素的可视元素。

我们使用 Hello World 范例来介绍鼠标区域，该矩形包含一个子级文本，一个新的矩形子级用来定义为鼠标区域。

MouseArea 元素包含信号处理，允许你编写 JavaScript 表达式将被特定的事件调用或状态的改变。有效的事件处理包括 onClicked、onEntered、onPressed 以及 onReleased。在上面的范例中，onClicked 信号处理用来切换矩形的颜色。

这个例子是矩形响应任何有效的单击来改变颜色。单击一般是在 MouseArea 里按下然后释放的动作（按下、移出 MouseArea 外；然后移回来并释放鼠标按钮同样也称为单击）。该处理的完整语法是 MouseArea::onClicked (mouse)，在这里的 mouse 参数提供了关于单击的信息，包括单击释放鼠标的 x 与 y 位置以及单击是否被保持。在我们的例子里并不注意在哪里进行单击。

鼠标触摸互动的范例展示了一个通过响应一个事件改变一个值可视化状态的简单情况。如果你是尝试响应多个状态改变多个值 onClicked 描述将是非常糟糕的。

状态声明

QML 状态声明定义一个属性值由基本状态的改变。基本状态是属性值的初始化声明并且是使用一个空字符串作为状态名称表示。状态改变后你可以总是可以通过指定一个空字符串到 state 属性来恢复到基本状态。

在下面的例子中，状态有两个颜色。在红色矩形的定义中，id 属性是设置的。命名的对象可以被同级或子级引用。两种状态是：红色与橙色。state 属性是指定该元素为初化状态。

状态元素包含包含了一个条件用于决定。在这里当在 MouseArea 按下时，你可以看见红色状态被应用。

```
import Qt 4.7
Rectangle {
    color: ”#ff0000”
    width: 310
    height: 210
    MouseArea {
        anchors.fill: parent
        onClicked: {
            if (parent.color == ”#ff0000”) {
                parent.color = ”#ff9900”;
            } else {
                parent.color = ”#ff0000”;
            }
        }
    }
    Rectangle {
        width: 300
        height: 200
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        Text {
            anchors.horizontalCenter: parent.horizontalCenter
            anchors.verticalCenter: parent.verticalCenter
            text: ”Hello World”
                }
    }
}
```

定义的状态不仅仅只是设置每个状态的颜色；还可以设置矩形的高宽度。橙色状态提供一个大的按钮。为了利用这个状态，鼠标区域的 onClicked JavaScript 是被更新。

它可以使用代码定义状态集，像这个例子中或使用图像化的 Qt Creator 里 Qt Quick Desinger 组件。

要建立状态间的动画，就要 Transition 元素。Transition 元素可使用从基本状态到目标状态使用动画元素插值属性的改变。动画元素可以使用不同参数值曲线与群组技术，给予开发者与设计者在状态过渡期间高度控制属性改变。这在稍后有详细的说明。

```
id: buttonRect;
state: ”red”
      states: [
   State {
      name: ”red”
            when: mouseArea.pressed == true
      PropertyChanges {
         target: buttonRect;
         color: ”red”;
         width: 80; height: 40
      },
   }
   State {
      name: ”orange”
            when: mouseArea.pressed == false
      PropertyChanges {
         target: buttonRect;
         color: ”#ff9900”;
         width: 120; height: 80
      }
   }
]
```

定义状态

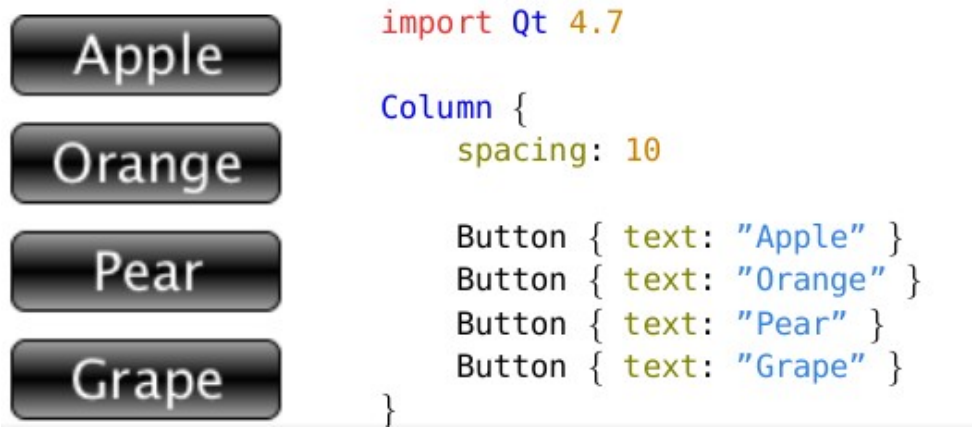
QML 组件

在 Hello World 例子中讨论描述了一个 QML 文档内容。QML 文档命名也是很重要的。QML 文件名称必须是以大写字母开头。QML 组件是 QML 运行时使用一些预定义行为创建一个对象的模板。由于它是一个模板，单个 QML 组件可以被运行多次来生成几个对象，每个对象都是组件的实例。

一旦创建后，实例是不依赖该组件的，这样它们可以各自独立操作数据。在这里有一个简单按钮组件的例子（在一个 Button.qml 文件里定义），它在 Application.qml 里实例化四次。每个实例是使用不同的 text 属性值来创建的。

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        if (parent.state == ”red”) {
            parent.state = ”orange”
                    } else {
            parent.state = ”red”;
        }
    }
}
```

简单状态过渡



使用四次不同文本属性的按钮

```
import Qt 4.7
Rectangle {
    property alias text: textItem.text
    width: 100; height: 30
    border.width: 1
    radius: 5
    smooth: true
    gradient: Gradient
    GradientStop {
        GradientStop {
            GradientStop {
```

```
        }
        {
            position: 0.0; color: ”darkGray” }
        position: 0.5; color: ”black” }
    position: 1.0; color: ”darkGray” }
  Text {
    id: textItem
    anchors.centerIn: parent
    font.pointSize: 20
    color: ”white”
        }
}
```

Button.qml 文件创建一个按钮组件

注意 QML 文档也可以使用 Component[组件]元素来创建内联组件。

动画元素：流畅过渡

动画效果的关键是流畅的用户界面。在 QML 里，通过应用对象到对象动画属性值，随着时间的推移来逐渐的改变它们来制作动画。动画对象是由内置的动画元素制作的，它可用来制作不同类型属性值的动画，此外动画对象可以以不同方式应用中，这取决于它们上下文不同的内容。

在线联机文档中关于 QML 动画有着更详细的信息。这里只是一个介绍而已。

下例代码显示一个矩形运动的动画。创建带有两种状态的矩形（默认状态与附加的移动状态）。在移动状态里，矩形的位置改变为（50，50）。变换对象是指矩形在默认状态与移动状态间的过渡，任何 x 与 y 属性的改变将被动画，使用 Easing.InOutQuad。

```
import Qt 4.7
Rectangle {
  id: rect
  width: 100; height: 100
  color: ”red”
        states: State {
    name: ”moved”
            PropertyChanges { target: rect; x: 50; y: 50 }
  }
  transitions: Transition {
    PropertyAnimation {
```

```
        properties: "x,y";
        easing.type: Easing.InOutQuad
    }
  }
```

状态动画过渡

你可以应用多个过渡到一项目里，像下面的例子。默认情况下过渡是适用于所有的状态改变。为了更好的控制你可以设置属性从给定的状态，到给定的状态或两个给定的状态间应用过渡。

```
Item {
    ...
    transitions: [
        Transition { ... }
        Transition { ... }
  ]
}
```

多个过渡

QML 里的模型—视图模式

在模型—视图里使用 QML 设计模式是非常自然的。QML 可以创建流行、悦目的方式来显示数据模型，这些模型是可以以 C++或直接用 QML 实现的。

QML 目前只提供了用于三个元素来显示模型的视图。ListView 与 GridView 元素创建列表与网格视图显示。PathView 元素勾画出一个路径来显示模型，例如循环路径允许你列表项目循环流动传送。

让我们对同一模型创建两个不同的视图——一个基本的联系人列表。

你可以使用 ListMo d el 元素直接在 QML 里构建模型。下例中显示了如何创建一个联系人模型，其中每个联系人记录包括了姓名、电话号码以及相片。每个元素在列表是通过 ListElement 元素定义的，每条项目都包含两个数据

```
import Qt 4.7
ListModel {
    ListElement {
        name: "Bill Jones"
            number: "+1 800 555 1212"
            icon: "pics/qtlogo.png"
        }
    ListElement {
```

```
        name: ”Jane Doe”
              number: ”+1 800 555 3434”
              icon: ”pics/qtlogo.png”
          }
    ListElement {
        name: ”John Smith”
              number: ”+1 800 555 5656”
              icon: ”pics/qtlogo.png”
          }
}
```

在 QML 里定义列表模型

下例使用 ListView 元素进行水平或垂直布局项目。下例设置模型属性为已经创建的 ContactMo
d
el 组件。代理属性提供了一个模板定义每个项目的视图实例。在这种情况下，模板使用内置的 Text 组件

技术说明：当使用 C++模型时，名称用来指向模型的不同角色，使用 QAbstractItemMo
d

```
import Qt 4.7
ListView {
    width: 180; height: 200
    model: ContactModel {}
    delegate: Text {
        text: name + ”: ” + number
    }
}
```

联系模型，列表视图

现在让我们为联系人模型构建一个视图，就像 3D 的旋转木马那样，允许用户通过触摸列表中的项目。如下例所示。注意以 PathView 元素使用代理属性创建一个内联组件。

```
import Qt 4.7

Rectangle {
    width: 240; height: 200

    Component {
        id: delegate
        Column {
            Image { anchors.horizontalCenter:
                    name.horizontalCenter;
                    width: 64; height: 64;
                    source: icon
            }
            Text { text: name; font.pointSize: 16 }
        }
    }

    PathView {
        anchors.fill: parent
        model: ContactModel {}
        delegate: delegate
        path: Path {
           startX: 120; startY: 100
        PathQuad { x: 120; y: 25; controlX: 260; controlY: 75 }
        PathQuad { x: 120; y: 100; controlX: -20; controlY: 75 }
        }
    }
}
```

联系人模型以旋转视图查看

在 C++应用程序中使用 Qt Quick

Qt Quick 带有它自己的运行环境以及通过模块载入新功能，使用 QML 开发应用程序完全成为可能。不管用何种方法，Qt Quick 真正强大是能够整合它成为一个 C++应用程序。

对于最基本的需求，例如：整合 QML 视图到一个 C++工程里，用 QDeclarativeView 部件即可。它由
QGrap
h

对于更复杂的情况，你可能需要用 C++代码构建完成。

如果你着手基于 C++部件的应用程序，你可以重复使用所有的图形资源，重构整个 QWidgets 到 QML。一旦你已经有充分的交互式以及设计所做的工作；那么开发编写 QML 代码相对容易些。

如果你不是开发一个基于
QGrap
h
icsView 应用程序；那么转换的过程更容易些,就可以进一步分为几个阶段来做。整个 QML 应用程序可

分割符

QML 核心特性

QML 文档

一个 QML 文档是指 QML 源代码的一个块。QML 文档一般相对应于磁盘与网络资源上储存的一个文件；但也可以直接从文本数据中构造。

在这里有一个简单的 QML 文档：

```
import Qt 4.7

Rectangle {
    width: 240; height: 320;

    resources: [
        Component {
            id: contactDelegate
            Text {
                text: modelData.firstName + " " + modelData.lastName
            }
        }
    ]

    ListView {
        anchors.fill: parent
        model: contactModel
        delegate: contactDelegate
    }
}
```

QML 文档总是 UTF-8 编码格式。

QML 文档

假设我们需要