

Qt Quick

讲师： 杜 平
邮箱： pidu@digia.com

Forum **Nokia**

digia

NOKIA

Qt Quick

课程预览

Forum **Nokia**

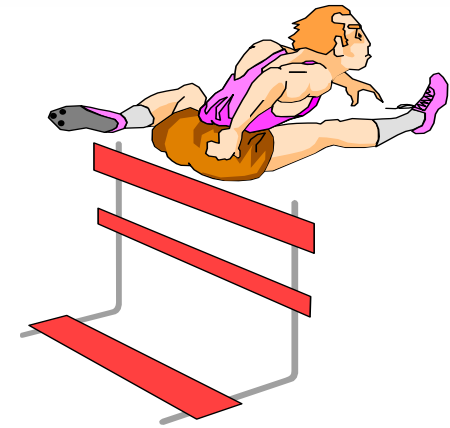
digia

NOKIA

课程要求

- 基础的JavaScript知识

Forum Nokia



digia

NOKIA

课程内容 – 第一天 - QML

- 介绍
 - 什么事 Qt Quick?
 - 开发工具
- QML 本质
 - 基础语法
 - 属性
 - 标准QML元素
 - 属性绑定
 - Attached 属性
- QML中的布局管理
 - Grid, Row, 和 Column 布局
- 用户交互
 - Mouse Area
 - KeyNavigation
 - Key 事件
- 状态, 过渡 和 动画



课程内容 – 第二天

- 核心**QML**特性

- QML Components
- Modules

- 数据模型和视图

- Model 类
- ListView, GridView, PathView
- Repeater
- Flickable

- 高级**QML**特性

- 在QML中扩展类型

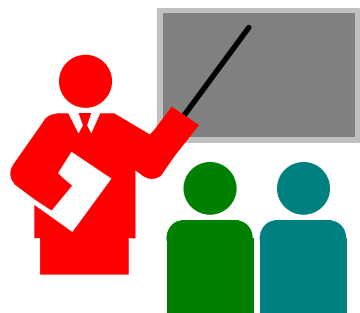
- **QML 和 Scripting**

- QML Global Object
- Script
- QML 范围
- QML Script 的限制
- 启动 Scripts

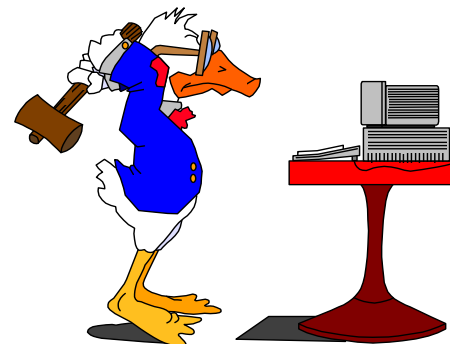
- 在 **Qt/C++** 应用中使用**QML**

- 主要的类
- Structured Data
- Dynamic Structured Data
- 网络 Components

Course Delivery



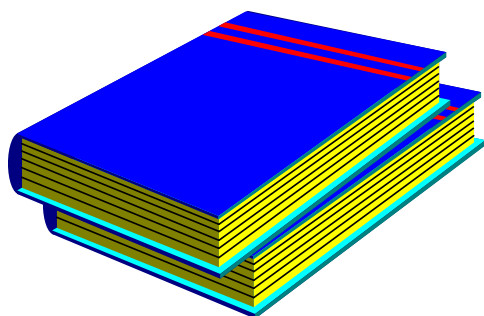
Presentations



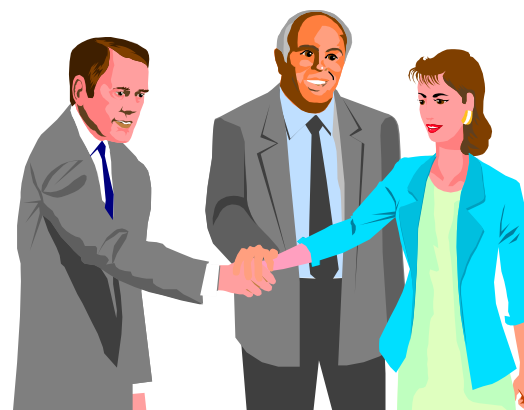
Hands-on Sessions



Questions



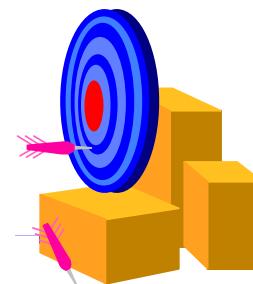
Course Manuals



Learn from others

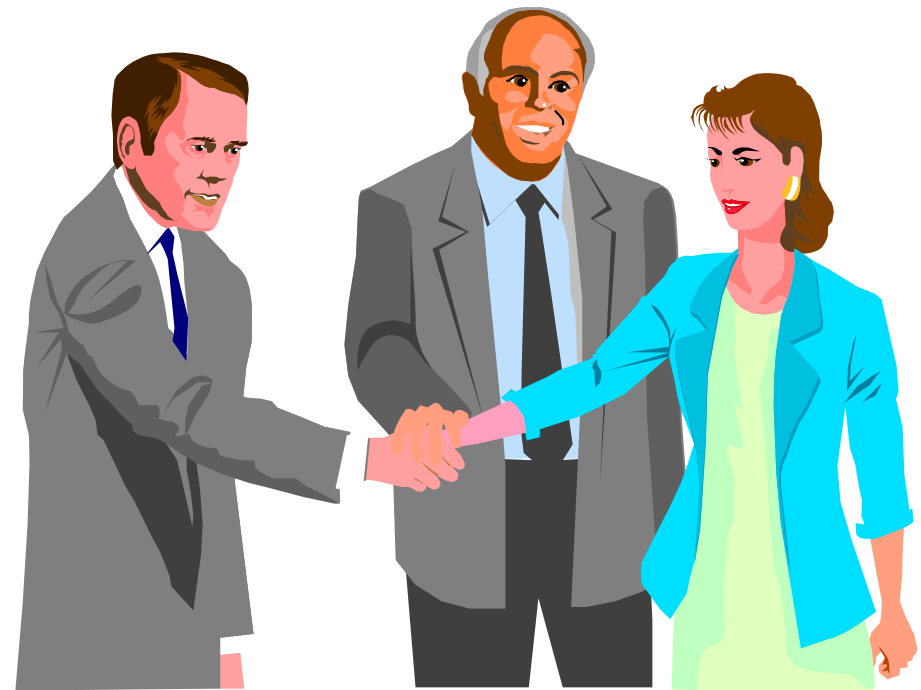
你的目标

你希望从这次课程中得到什么？



练习

- 介绍你自己
- Qt/C++ 经验
- JavaScript 经验
- 参加本次课程的目标



Qt Quick

介绍

Forum **Nokia**

digia

NOKIA

什么是Qt Quick? 1/4

- "Qt User Interface Creation Kit"
 - 带来了概念 Declarative UI 和 QML
- 一个为创建更好的界面的高级UI技术
 - No C++ skills needed, knowledge of JavaScript helps quite a bit
- 针对界面设计人员和开发者
 - Enables designers and developers to "speak the same language"!
 - Both parties can be involved in iterative development simultaneously
 - No need for separate Flash or PowerPoint UI prototypes
- 最新官方版本 Qt 4.7.1
 - Qt4.7引入Qt Quick

什么是Qt Quick? 2/4

- 技术包括:
 - Declarative markup language: QML
 - Qt提供了运行库支持
 - Qt Creator IDE 支持 QML language
 - 可视化设计工具
 - 提供了C++ API 将QML加入到Qt的应用中
 - QtDeclarative

什么是Qt Quick? 3/4 -QML

- "Qt Meta-Object Language"
- 就像脚本语言一样定义界面的元素
 - 是对ECMAScript标准的扩展 (cf. JavaScript)
 - 提供了建立一颗QML元素Object树的机制
 - 支持QML元素和基于Qt QObject的C++objects之间的交互
- QML 包含了一套QML元素
 - I.e. 图形的 和 行为的 building blocks
 - 可以通过QML文档来建立更加复杂的元件和QML应用
- 可以用来对现有程序的扩充或者是编译一个全新的程序
 - QML itself is also fully extensible with C++!

什么是Qt Quick? 4/4 -QtDeclarative

- QtDeclarative 在Qt中是新的模块
 - 提供对QML应用的运行库支持
 - 同时也提供了将QML内容嵌入到Qt/C++程序中的接口
 - 实现了QML端和C++端的绑定
- 包含了一个叫qmlview的工具(qmlviewer.exe)，这个工具用来运行单独的 QML/JavaScript程序
 - 让我们可以直接查看qml文件，而不必将其加载到Qt应用中
 - 主要是用于开发测试
 - 提供了一种“dummy”机制，动态的为QML提供数据
 - 最终的数据将是由Qt/C++代码所提供的

开发工具1/2

- Qt Creator 是 Qt 开发的首选工具
 - Also for Qt Quick, of course!
- 实际上，开始学习Qt Quick, 下面的你需要了解
 - Qt Creator已经能够支持Qt Quick
 - 提供了 *qmlviewer* (*qmlviewer.exe*)
- 为了把Qt Quick界面嵌入到Qt/C++应用程序中，需要使用 Qt 4.7以上的 SDK

开发工具2/2

- Qt Creator 2.1 快照
 - 与 Qt 4.7 SDK一起使用
 - 可能不太稳定
- Qt 4.7 SDK
 - 引入了新的 QtDeclarative 模块

Qt Quick

QML本质

Forum **Nokia**

digia

NOKIA

介绍

- QML是一种声明性的语言:
 - 定义应用的外观
 - 和引用的行为
- QML界面程序有一组树型结构的容器组合而成
- 需要了解JavaScript (+ HTML and CSS)是学习QML的前提条件
 - Not strictly required, though

QML 语法

```
/* woodenhead.qml starts here, with a  
multiline comment */  
import Qt 4.7  
//for 4.7.1 import QtQuick 1.0  
  
Rectangle {  
    width: 350 // Single line comment  
    height: 2 * 100  
    color: "lightblue"  
}
```



- 让我们从一个简单的实例开始吧：一块淡蓝色的矩形区域
- 很容易理解，对吧 😊?
 - Very "JavaScriptish", right?

QML语法-import

```
/* woodenhead.qml starts here, with a  
multiline comment */  
import Qt 4.7  
//for 4.7.1 import QtQuick 1.0  
  
Rectangle {  
    width: 350 // Single line comment  
    height: 2 * 100  
    color: "lightblue"  
}
```



- 为了使用Qt的特性需要引入Qt的模块
- 指明你要引入Qt模块的Qt版本
 - 仅仅导入对应版本的Qt所支持的特性
 - 不会使用后续版本的特性
 - 也不会退回到以前版本的特性
- 保证整个代码的行为不会因为Qt的版本不同而发生改变
 - 模块可以支持多个版本的特性
 - 更新的模块会保留对老版本的支持

QML语法-注释

```
/* woodenhead.qml starts here, with a  
multiline comment */  
import Qt 4.7  
  
Rectangle {  
    width: 350 // Single line comment  
    height: 2 * 100  
    color: "lightblue"  
}
```



- 使用 // 添加单行的注释
- 讲多行的注释放置 /* 和 */ 之间

QML语法 - Elements

```
/* woodenhead.qml starts here, with a  
multiline comment */  
import Qt 4.7  
  
Rectangle {  
    width: 350 // Single line comment  
    height: 2 * 100  
    color: "lightblue"  
}
```



- 声明你要使用的元素
 - 在文件中声明一个 根元素
- 每一个元素的内容需要在大括号里
- 在Qt模块中包含了一些默认的元素

标准QML元素

- QML提供了很多定义好的界面元素
 - Item, Rectangle, Image, Text, MouseArea, WebView, ListView, ...
 - 其中一些元素可以作为其他元素(children)的容器 (parent)
 - 被称为 *QML items*
 - 所有用于创建UI的元素都是从Item继承而来的
- 还有一些元素用来描述应用程序行为
 - State, PropertyAnimation, Transition, Timer, Connection, ...
 - 被称为 *QML declarative elements*

Item 元素

- 这些元素不会显示，但是使用上和一般的UI元素一样
 - As mentioned, all UI elements inherit the `Item` element
- 基本的属性:
 - x, y, z position
 - width and height
 - anchors (explained later)
 - opacity, rotation, scale
 - visibility (true/false)
 - parent and children
 - key event handling
 - ...

QML属性 1/2

```
/* woodenhead.qml starts here, with a  
multiline comment */  
import Qt 4.7  
  
Rectangle {  
    width: 350 // Single line comment  
    height: 2 * 100  
    color: "lightblue"  
}
```



- 元素包含了属性
- 每一个属性都包含了名称和值
 - name : value
 - value can also be a piece of JavaScript
- 在一行可以声明多个属性
 - Separate with semi-colons

```
Rectangle {  
    width: 350; height: 2 * 100  
    color: "lightblue"  
}
```


QML属性 2/2

- QML 支持多种数据类型
 - int, bool, real, color, string, list, ...
- 属性多是类型安全的
 - i.e. 将字符串付给整形的属性是不允许的

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

属性的例子

- **Standard properties** 标准属性可以直接用值初始化:

```
Text {  
    text: "Hello world"  
    height: 50  
}
```

- **Grouped properties** 分组属性将相关的属性放在一起:

```
Text {  
    font.family: "Helvetica"  
    font.pixelSize: 24  
}
```

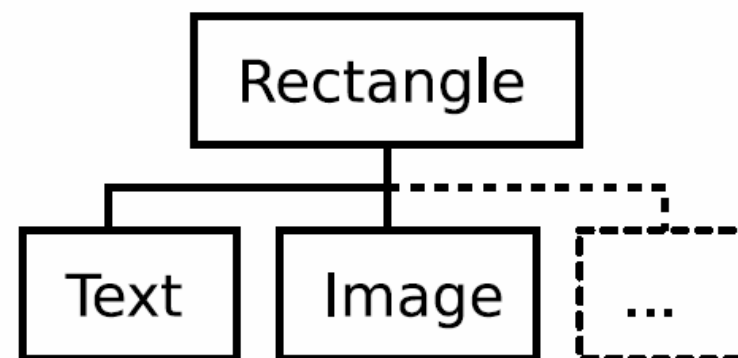
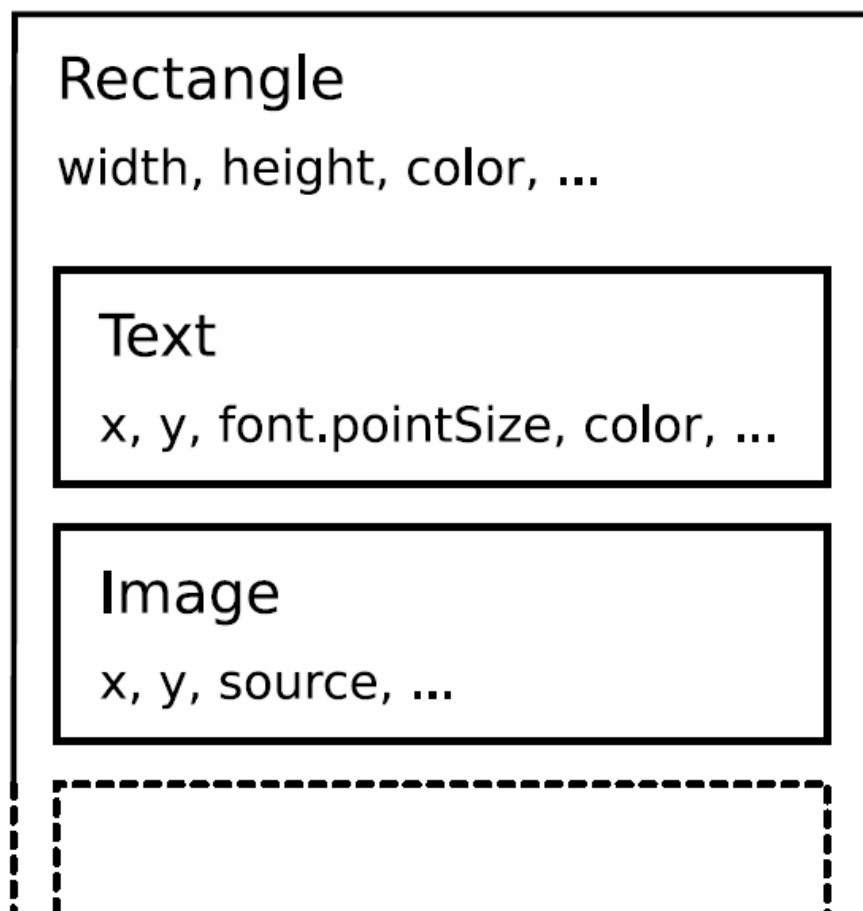
- **Identity property** 标识属性用于唯一的表示一个元素:

```
Text {  
    id: label  
    text: "Hello world"  
}
```

属性 - 颜色

- 元素的颜色定义可以有很多种方法:
- 用一个字符串来表示 (使用 SVG 颜色名字):
- "red", "green", "blue", ...
- 用一个字符串来表示颜色的组成:
- 红, 绿 和 蓝: #<rr><gg><bb>
- "#ff0000", "#008000", "#0000ff", ...
- 使用内建函数 (红, 绿, 蓝, 透明度):
- Qt.rgb(0,0.5,0,1)
- 使用 opacity 属性
- 从 0.0 (透明) 到 1.0 (不透明)

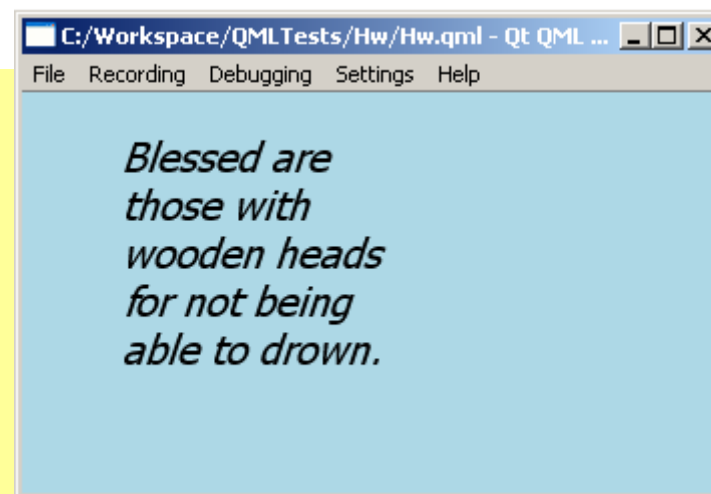
元素



第一个例子

```
/* woodenhead.qml starts here, with a
multiline comment */
import Qt 4.7

Rectangle {
    width: 350; height: 2 * 100
    color: "lightblue"
    Text {
        x: 50; y: 20; width: 150
        wrapMode: Text.WordWrap
        font.pixelSize: 20; font.italic: true
        text: "Blessed are those with wooden heads
              for not being able to drown."
    }
}
```



Rectangle

Text

字符元素

```
Text {  
    text: "Hello!"  
}
```

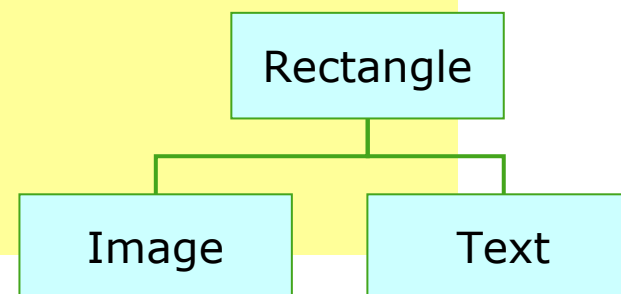
- 显示字符串
- 宽度和高度是由字体和文本所决定的
- 也可以是超级链接
- 也可以使用HTML标签: "<html>Qt Quick</html>"

```
TextInput {  
    width: 300  
    text: "Editable hello!"  
}
```

- 可编辑的字符串 TextInput
 - 它不是 QLineEdit
- 点击就可以获得焦点, 但是需要有可以点击的地方 (either text or a width)
- 当编辑的时候text属性会发生变化

第一个例子, 添加图片

```
...  
Rectangle {  
    width: 350; height: 2 * 100  
    color: "lightblue"  
  
    Image {  
        width: parent.width  
        source: "justiina.jpg"  
    }  
    // Text element as previously  
    Text {  
        ...  
    }  
}
```



图片

- source 指定了图片的相对路径
 - “../” 指代父文件夹
 - 可以是一个URL，本地文件或者资源中的文件
- 高度和宽度默认都是从图片文件获得
 - 如果直接设置，根据设置的值图片将会自动缩放
 - 使用属性 fillMode 来设置缩放时的长宽比例
- 设置 scale 缩放图片，设置 rotate 旋转图片 (度是旋转的单位)
 - 旋转式围绕图片的中心的
 - 通过 transformOrigin 属性来设置旋转的围绕点

```
Image {  
    width: parent.width  
    source: "justiina.jpg"  
}
```


绑定属性值 1/2

```
...  
Rectangle {  
  width: 350; height: 2 * 100  
  color: "lightblue"  
  
  Image {  
    width: parent.width  
    source: "justiina.jpg"  
  }  
}
```



- 属性值可以绑定到其他的值
 - 会自动的更新
- 可以用parent访问父元素，或者利用其他节点的id进行访问
 - 看下页的例子

绑定属性值 2/2

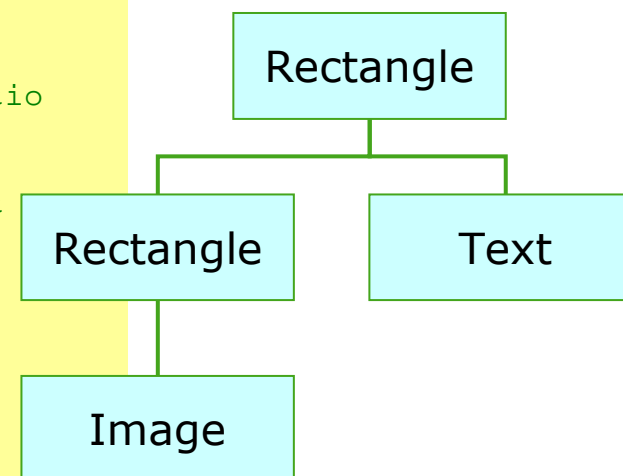
```

Rectangle {
    width: 350; height: 2 * 100
    color: "lightblue"
    Rectangle {
        width: 150
        anchors.right: parent.right
        height: justiina.height
        color: "blue"
        Image {
            id: justiina
            smooth: true // Smoothens scaling
            width: parent.width
            fillMode: Image.PreserveAspectRatio // Aspect ratio
            source: "justiina.jpg"
            opacity: 0.8 // 1.0 = opaque, 0.0 = transparent
        }
    }
    // Text element as previously
    Text {
        ...
    }
}

```

Bound properties

*Blessed are
those with
wooden heads
for not being
able to drown.*



例子的所有代码

```
/* woodenhead.qml starts here, with a
multiline comment */
import Qt 4.7

Rectangle {
    width: 350; height: 2 * 100
    color: "lightblue"
    Text {
        x: 50; y: 20; width: 150
        wrapMode: Text.WordWrap
        font.pixelSize: 20; font.italic: true
        text: "Blessed are those with wooden heads
              for not being able to drown."
    }
}
```

```
Rectangle {
    width: 150
    anchors.right: parent.right
    height: justiina.height
    color: "blue"
    Image {
        id: justiina
        smooth: true
        width: parent.width
        fillMode: Image.PreserveAspectFit
        source: "justiina.jpg"
        opacity: 0.8
    }
}
```

Qt Quick

布局管理

Forum **Nokia**

digia

NOKIA

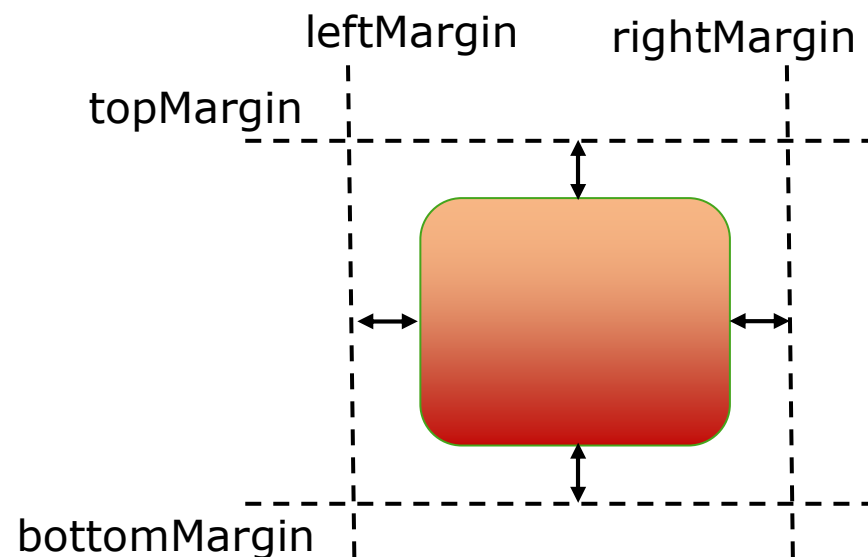
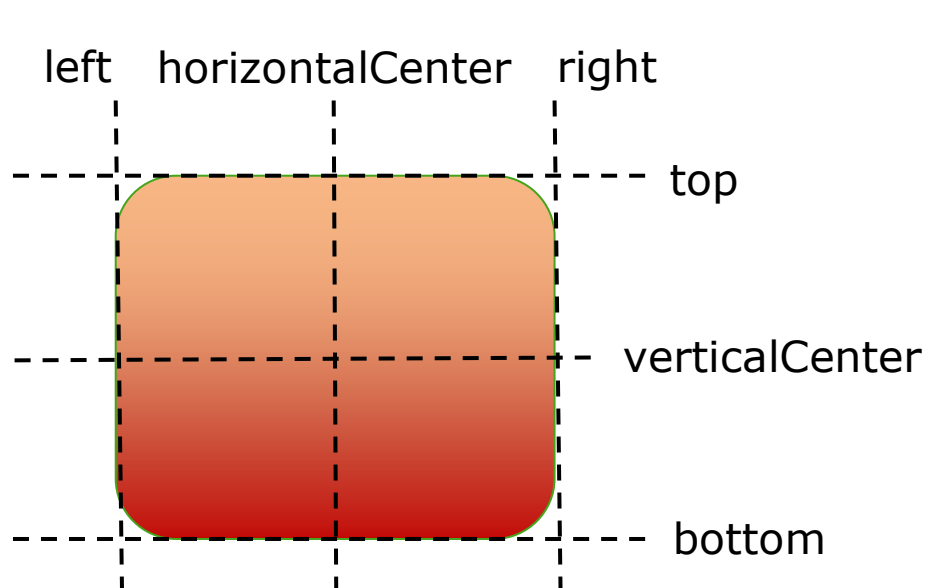
布局管理器介绍

- 硬编码图元的位置绝对不是一个好的方法
 - 提供UI的可测量性和很困难
 - 维护困难
- QML提供很多不同种类的布局工具代替硬编码的方法
 - Anchor 布局管理
 - 布局器
 - Grid, Row, Column

Anchor Layout 1/4

```
Rectangle {  
    anchors.right: parent.right  
    ...  
}
```

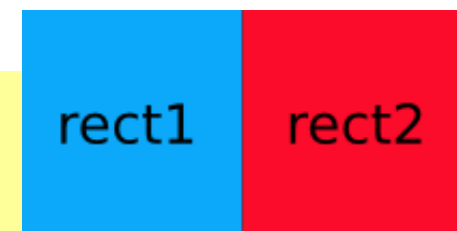
- 每个QML元素都可以认为有6个方位和4个边缘:



Anchor Layout 2/4

- 方位用来说明不同组件之间的位置信息

```
Rectangle { id: rect1; ... }  
Rectangle { id: rect2; anchors.left: rect1.right; ... }
```

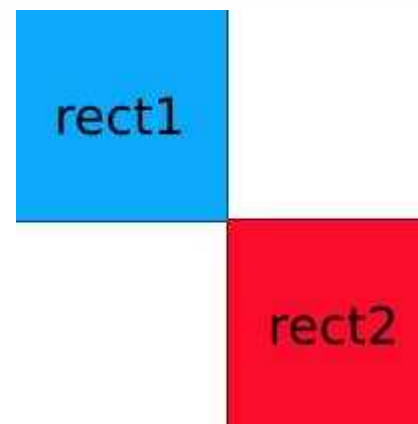


```
Rectangle { id: rect1; ... }  
Rectangle { id: rect2; anchors.left: rect1.right; anchors.leftMargin: 5; ... }
```



Anchor Layout 3/4

- 可以指定多个方位
 - 甚至可以用来控制元素的大小!



```
Rectangle { id: rect1; ... }  
Rectangle { id: rect2; anchors.left: rect1.right; anchors.top: rect1.bottom;  
    ... }  
  
Rectangle { id: rect1; x: 0; ... }  
Rectangle { id: rect2; anchors.left: rect1.right; anchors.right: Rect3.left;  
    ... }  
Rectangle { id: Rect3; x: 150; ... }
```



Anchor Layout 4/4

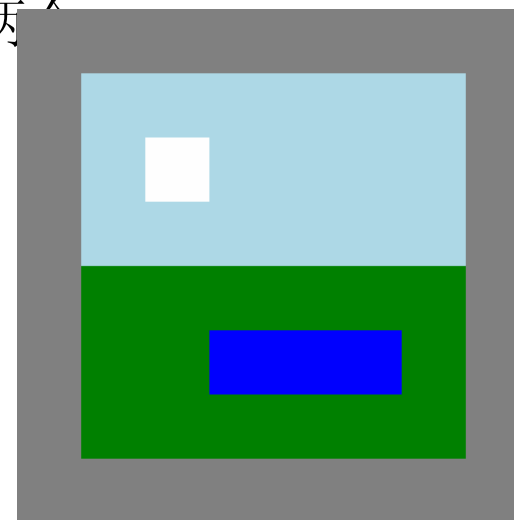
- 出于运行效率的考虑，只能在兄弟元素之间或者直接父子之间使用anchor

```
Item {  
    id: Group1  
    Rectangle { id: rect1; ... }  
}
```

```
Item {  
    id: Group2  
    Rectangle { id: rect2; anchors.left: rect1.right; ... } // Invalid anchor!  
}
```

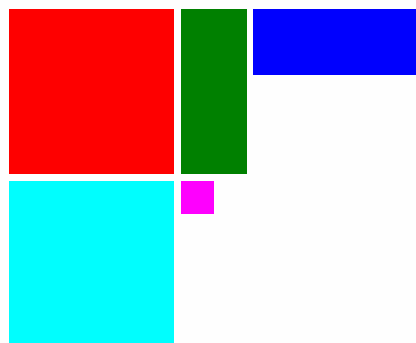
Exercise 1 - Items

- 右边的图片显示了一个400x400的正方形中有两个长方形，并且各自都包含了子对象。
 1. 使用Rectangle创建这个界面。
 - 2. items可以重叠吗？
试着移动浅蓝色或者绿色的矩形框。
 - 3. 孩子item可以显示在parent之外吗？
试着给白色或者蓝色的矩形框一个负的x值。



网格布局

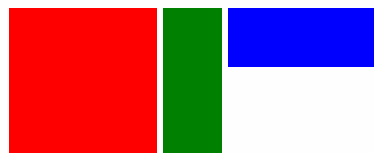
- 在QML中的关键字为Grid
 - 将孩子items以而为表格的形式展现
 - Provides for transition effects when items are added (shown), moved or removed (hidden)



```
Grid {  
    columns: 3  
    spacing: 2  
    Rectangle { color: "red"; width: 50; height: 50 }  
    Rectangle { color: "green"; width: 20; height: 50 }  
    Rectangle { color: "blue"; width: 50; height: 20 }  
    Rectangle { color: "cyan"; width: 50; height: 50 }  
    Rectangle { color: "magenta"; width: 10; height: 10 }  
}
```

纵向布局

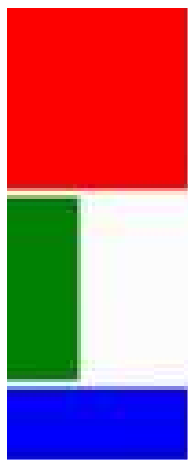
- 在QML中的关键字是Row
 - Positions child items in a row so that they do not overlap each other
 - Provides for transition effects when items are added (shown), moved or removed (hidden)



```
- Row {  
    spacing: 2  
    Rectangle { color: "red"; width: 50; height: 50 }  
    Rectangle { color: "green"; width: 20; height: 50 }  
    Rectangle { color: "blue"; width: 50; height: 20 }  
}
```

横向布局

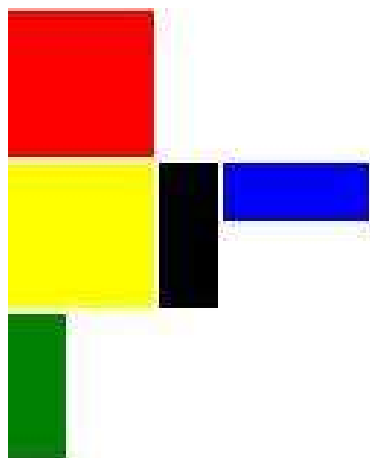
- 在QML的关键字为Column
 - Positions child items vertically so that they do not overlap each other
 - Provides for transition effects when items are added (shown), moved or removed (hidden)



```
Column {  
    spacing: 2  
    Rectangle { color: "red"; width: 50; height: 50 }  
    Rectangle { color: "green"; width: 20; height: 50 }  
    Rectangle { color: "blue"; width: 50; height: 20 }  
}
```

Combining Layouts

- Grid, Row and Column 可以被嵌套使用
- For example, a Row inside a Column:



```
Column {
  spacing: 2
  Rectangle { color: "red"; width: 50; height: 50 }
  Row {
    spacing: 2
    Rectangle { color: "yellow"; width: 50; height: 50 }
    Rectangle { color: "black"; width: 20; height: 50 }
    Rectangle { color: "blue"; width: 50; height: 20 }
  }
  Rectangle { color: "green"; width: 20; height: 50 }
}
```

Qt Quick

用户交互

Forum **Nokia**

digia

NOKIA

处理用户输入

- 用户与QML交互:
 - 鼠标移动, 点击和拖拽
 - 键盘输入

事件处理

- Qt使用信号槽的基础处理大部分（非所有）的事件响应问题
 - 信号槽使用了观察者模式
- 在QML，类似的当有事件发生的时候，一个与事件相关的信号会被发出
- 所以，要处理事件，需要定义一个槽
 - 这个槽仅仅只是一个属性（property）
 - 这个属性的名字与事件的类型是相关的（鼠标点击，计时，键盘按键，...）

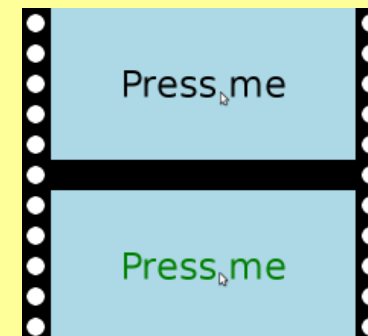
鼠标区域

- Mouse areas 用于定义屏幕的某区域接收鼠标事件
- 位置和大小与普通的items是一样使用的
 - 可以对Mouse Area使用anchors（锚）
- 两种方法处理鼠标输入：
 - 处理信号
 - 动态属性绑定

点击鼠标区域

```
import Qt 4.7

Rectangle {
    width: 400; height: 200; color: "lightblue"
    Text {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        text: "Press me"; font.pixelSize: 48
        MouseArea {
            anchors.fill: parent
            onPressed: parent.color = "green"
            onReleased: parent.color = "black"
        }
    }
}
```



放置鼠标区域

```
Text {  
    ...  
    MouseArea {  
        anchors.fill: parent  
        onPressed: parent.color = "green"  
        onReleased: parent.color = "black"  
    }  
}
```

- 定义一个Text元素
- 定义一个MouseArea元素，让其为Text的子元素
- 使用anchor让MouseArea的区域为Text的区域

鼠标区域的信号

```
Text {  
    ...  
    MouseArea {  
        anchors.fill: parent  
        onPressed: parent.color = "green"  
        onReleased: parent.color = "black"  
    }  
}
```

- 定义响应信号的槽 `onPressed` 和 `onReleased`
 - 默认情况，只接收左键
 - 通过设置 `acceptedButtons` 来改变接收的鼠标按键
- 简单的改变**Text**的颜色
- 可以对其他的属性进行改变...

不同的鼠标事件

- 与鼠标事件相关的信号
 - `onClicked`, `onDoubleClicked`, `onPressAndHold`, `onReleased`, ...
 - 信号携带类型为 `MouseEvent` 的参数 *mouse*

```
Rectangle {  
    width: 100; height: 100; color: "green"  
    MouseArea {  
        anchors.fill: parent  
        // See Qt::MouseButtons for a list of available buttons  
        acceptedButtons: Qt.LeftButton | Qt.RightButton  
        onClicked: {  
            if (mouse.button == Qt.RightButton)  
                parent.color = 'blue';  
            else  
                parent.color = 'red';  
        }  
    }  
}
```

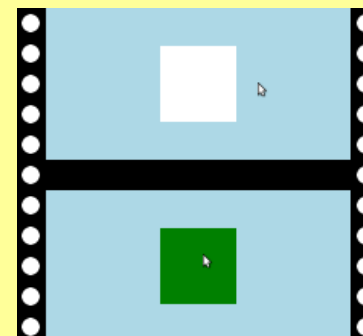
拖拽元素

- 通过设置MouseArea的属性drag，可以让某个元素可以被拖拽

```
Rectangle {  
    id: opacitytest; width: 600; height: 200; color: "white"  
    Image {  
        id: pic; source: "qtlogo-64.png"  
        anchors.verticalCenter: parent.verticalCenter  
        opacity: (600.0-pic.x) / 600;  
        MouseArea {  
            anchors.fill: parent  
            drag.target: pic  
            drag.axis: "XAxis"  
            drag.minimumX: 0  
            drag.maximumX: opacitytest.width-pic.width  
        }  
    }  
}
```

鼠标悬停与属性

```
import Qt 4.7
Rectangle {
    width: 400; height: 200; color: "lightblue"
    Rectangle {
        x: 150; y: 50; width: 100; height: 100
        color: mouse_area.containsMouse ? "green" : "white"
        MouseArea {
            id: mouse_area
            anchors.fill: parent
            hoverEnabled: true
        }
    }
}
```



允许鼠标悬停支持

```
Rectangle {  
    ...  
    color: mouse_area.containsMouse ? "green" : "white"  
    MouseArea {  
        id: mouse_area  
        anchors.fill: parent  
        hoverEnabled: true  
    }  
}
```

- 确定MouseArea覆盖了parent的范围
- 设置属性 hoverEnabled 为true

标识 Mouse Area

```
Rectangle {  
    ...  
    color: mouse_area.containsMouse ? "green" : "white"  
    MouseArea {  
        id: mouse_area  
        anchors.fill: parent  
        hoverEnabled: true  
    }  
}
```

- 设置 MouseArea 的 id 属性
- 这个id就是MouseArea的标识
 - 通过id就可以访问到MouseArea的属性
- 当 mouse_area.containsMouse 改变 color 也会相应改变
 - 使用一段简单的javascript代码

鼠标区域的一些注意点

- 鼠标区域只会相应 `acceptedButtons` 所定义的鼠标按键
 - 槽不会接受到其他的鼠标按键事件
 - 只有指定的鼠标按键才会被相应
 - 多个鼠标按键被按下时 `pressedButtons` 属性会记录所有的按键
 - 如果有 `acceptedButtons` 指定的按钮被按下，没有被指定的按钮也会被记录
- 当 `hoverEnabled` 为 **false** 的时候
 - 当鼠标按下时 `containsMouse` 属性为 `true`

信号 vs. 属性绑定

- 使用哪种方法?
- 在某些情况下信号更容易使用
 - 当一个信号只影响到某个元素时
- 属性绑定只能针对有id的元素使用
 - 多个元素可以对同一个鼠标事件作出响应
- 使用对你来说最直观的方式
- 使用更简单的方式，避免大量的javascript代码

键盘输入

- 键盘输入:
 1. 接受文本输入
 - TextInput and TextEdit (difference: TextEdit is multi-line)
 2. 在元素之间导航
 - 改变元素的焦点
 - 导航键 (arrow keys), tab and backtab
 3. 按键输入
 - 响应任意的按键, 比如: 游戏中的快捷键...

改变焦点

- UIs 只有一个 `TextInput` 的时候
 - 焦点自动在 `TextInput` 上
- 如果有多个 `TextInput`
 - 需要通过点击鼠标改变焦点呢
- 当一个 `TextInput` 没有文字的时候怎么办？
 - 鼠标不能点到它
 - 除非它有 `width` 属性或者通过 `anchors` 布置了
- 通过设置 `focus` 属性来该表焦点



Field 1
Field 2...|

使用 TextInputs

```
import Qt 4.7
Rectangle {
    width: 200; height: 112; color: "lightblue"
    TextInput {
        anchors.left: parent.left; y: 16
        anchors.right: parent.right
        text: "Field 1"; font.pixelSize: 32
        color: focus ? "black" : "gray"
        focus: true
    }
    TextInput {
        anchors.left: parent.left; y: 64
        anchors.right: parent.right
        text: "Field 2"; font.pixelSize: 32
        color: focus ? "black" : "gray"
    }
}
```



Field 1
Field 2...|

焦点 导航

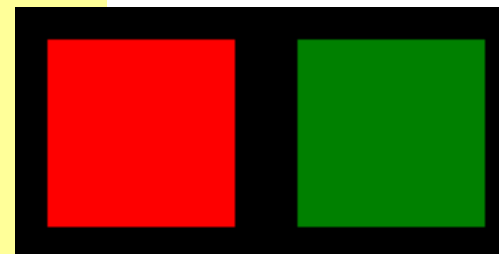
```
TextInput {  
  id: name_field  
  ...  
  focus: true  
  KeyNavigation.tab: address_field  
}  
TextInput {  
  id: address_field  
  ...  
  KeyNavigation.backtab: name_field  
}
```



- id为 name_field 的元素定义了 KeyNavigation.tab
 - 当按 **Tab** 键的时候焦点就移到了address_field 上
- id为 address_field的元素定义了 KeyNavigation.backtab
 - 当按 **Shift+Tab**键的时候焦点就移到了name_field上

按键导航

```
Rectangle {  
  width: 400; height: 200; color: "black"  
  Rectangle {  
    id: leftRect  
    x: 25; y: 25; width: 150; height: 150  
    color: focus ? "red" : "darkred"  
    KeyNavigation.right: rightRect  
    focus: true }  
  Rectangle {  
    id: rightRect  
    x: 225; y: 25; width: 150; height: 150  
    color: focus ? "#00ff00" : "green"  
    KeyNavigation.left: leftRect }  
}
```

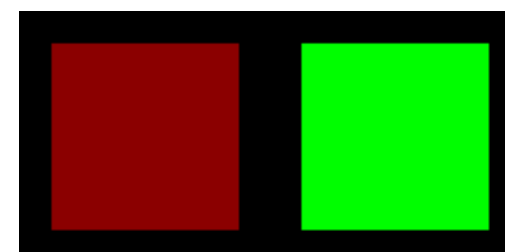
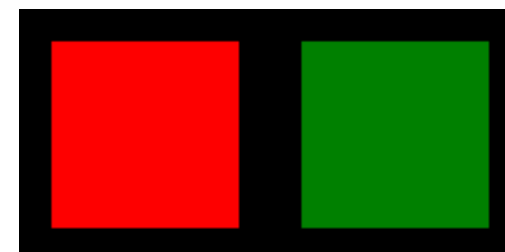


- 为 non-text 元素使用导航键
- Non-text 元素也可以有焦点

按键导航

```
Rectangle {  
  width: 400; height: 200; color: "black"  
  Rectangle {  
    id: leftRect  
    x: 25; y: 25; width: 150; height: 150  
    color: focus ? "red" : "darkred"  
    KeyNavigation.right: rightRect  
    focus: true }  
  Rectangle {  
    id: rightRect  
    x: 225; y: 25; width: 150; height: 150  
    color: focus ? "#00ff00" : "green"  
    KeyNavigation.left: leftRect }  
}
```

左边的矩形具有最初的焦点



- 为 non-text 元素使用导航键
- Non-text 元素也可以有焦点

键盘按键输入 1/2

- 所有可视的元素都可以通过 **Keys** 的 **attached** 属性支持键盘事件的处理
- 支持非常多的键盘事件
 - 通用事件: `onPressed`, `onReleased`
 - 专用事件: `onReturnPressed`, `onSelectPressed`, `onVolumeUpPressed`, ...
 - 他们都有个类型为 `KeyEvent` 的参数 *event*
- 处理通用信号时
 - 需要显示的告知事件被处理了
`event.accepted = true;`
 - 否则, 这个事件将会传递
- 专用事件默认就将事件处理了

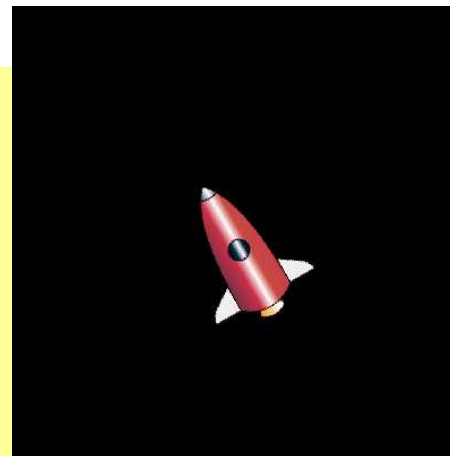
键盘按键输入 2/2

```
Item {    // Handle a key event with a generic handler
    focus: true
    Keys.onPressed: {
        if (event.key == Qt.Key_Left) {    // See Qt::Key for codes
            console.log("move left");
            event.accepted = true;    // Must accept explicitly
        }
    }
}

Item { // Handle a key event with a specialized handler
    focus: true
    Keys.onLeftPressed:                // Accepts the event by default
        console.log("move left")
}
```

另一个例子

```
import Qt 4.7
Rectangle {
    width: 400; height: 400; color: "black"
    Image {
        id: rocket
        x: 150; y: 150
        source: "../images/rocket.svg"
        transformOrigin: Item.Center
    }
    Keys.onLeftPressed:
        rocket.rotation = (rocket.rotation - 10) % 360
    Keys.onRightPressed:
        rocket.rotation = (rocket.rotation + 10) % 360
    focus: true
}
```



练习 2



- 制作一个用户界面，并支持以下特性：
 - 当 **items** 获取焦点的时候，颜色发生改变
 - 点击一个 **item**，就让它获取焦点
 - 焦点可以通过方向键改变

Qt Quick

状态, 过渡 和 动画

Forum **Nokia**

digia

NOKIA

目的

- 通过设置状态和过渡来定义用户界面的行为：
 - 提供一种方法来描述用户界面
 - 有效的组织应用程序的逻辑
 - 可以帮助我们判断是否所有的功能都被覆盖到了
 - 可以应用动画和视觉效果来扩展过渡变化

States (状态)

- 状态用于管理有id的元素
- 它是由多个 state 元素构成的
- 每个元素都可以定义多个不同的状态
 - 使用 states 属性定义状态列表
 - 当前的状态由 state 属性指定
- 当元素进入某个状态时，状态所对应的属性将被设置
- 我们可以
 - 修改 anchors 对齐方式
 - 修改item的parent
 - 执行一段javascript代码

状态例子 1/3

```
import Qt 4.7
Rectangle {
    width: 150; height: 250
    Rectangle {
        id: stop_light
        x: 25; y: 15; width: 100; height: 100
    }
    Rectangle {
        id: go_light
        x: 25; y: 135; width: 100; height: 100
    }
    ...
}
```



- 为所有的 item 命名一个id
- 设置好不会被状态改变的属性值

状态例子2/3

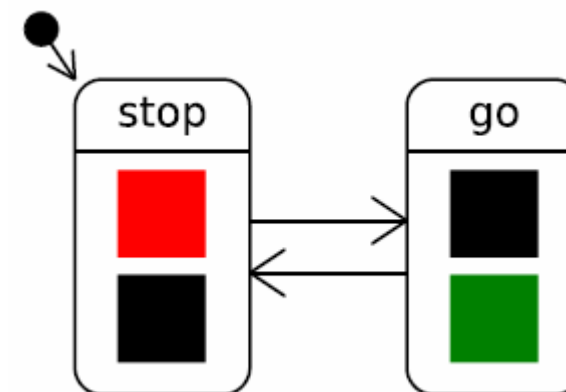
```
states: [  
  State {  
    name: "stop"  
    PropertyChanges { target: stop_light; color: "red" }  
    PropertyChanges { target: go_light; color: "black" }  
  },  
  State {  
    name: "go"  
    PropertyChanges { target: stop_light; color: "black" }  
    PropertyChanges { target: go_light; color: "green" }  
  }  
]
```

...

- 定义两个状态，名字分别为“stop”和“go”
- 使用 PropertyChanges 为每个状态设置目标和其对应的属性值

状态例子3/3

```
...  
state: "stop" // Define initial state  
  
MouseArea {  
    anchors.fill: parent  
    onClicked: parent.state == "stop" ?  
        parent.state = "go" : parent.state = "stop"  
}
```



- 使用 **MouseArea** 的事件响应来完成不同状态间的切换
 - 响应用户界面的单击事件
 - 让parent的状态在“stop”和“go”直接切换

修改属性

- States 通过PropertyChanges 来修改属性:

```
State {  
    name: "stop"  
    PropertyChanges { target: stop_light; color: "red" }  
    PropertyChanges { target: go_light; color: "black" }  
}
```

- 指定修改的目标元素的id给target属性
- 定义要修改的目标元素的属性值
 - 一个 PropertyChanges 元素可以修改多个属性
- 当进入相应状态时，对应的属性设置就会生效

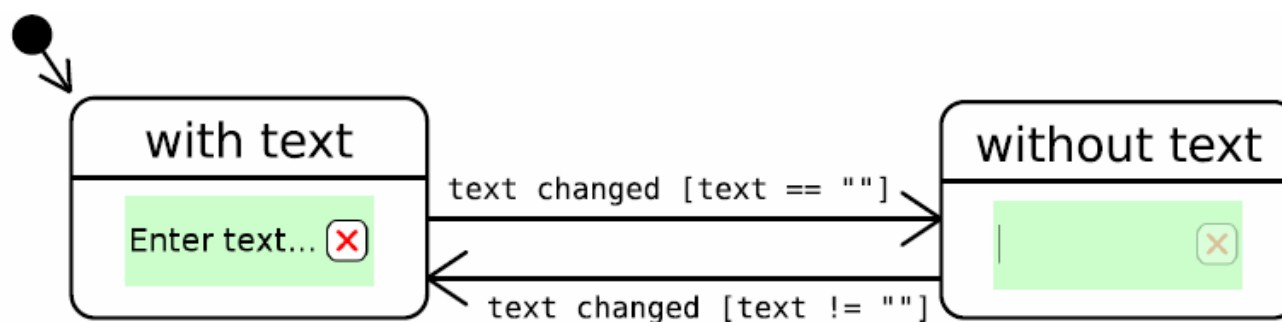
默认属性

- 每个元素都可以指定一个默认属性
 - 设置属性时，属性的名字标签可以被省略
 - **State**元素的默认属性是changes

```
State {  
    changes: [  
        PropertyChanges {},  
        PropertyChanges {}  
    ]  
}  
  
// ... can be simplified to:  
State {  
    PropertyChanges {}  
    PropertyChanges {}  
}
```

状态条件


- 使用状态的另一种方法:
- 让 **State** 决定何时被激活
 - 使用条件判断来决定是否激活一个 **state**
- 使用 **when** 属性
 - 用表达式来判断条件，并返回**true**或者**false**
- **states list** 中只能有一个 **state** 是被激活的
 - 确保一个时间只有一个条件为真



状态条件的例子

```
Rectangle {
    width: 250; height: 50; color: "#ccffcc"
    TextInput {
        id: text_field
        text: "Enter text..."
        ...
    }
    Image {
        id: clear_button
        source: "../images/clear.svg"
        ...
        MouseArea {
            anchors.fill: parent
            onClicked: text_field.text = ""
        }
    }
}
```

```
states: [
    State {
        name: "with text"
        when: text_field.text != ""
        PropertyChanges { target: clear_button;
                           opacity: 1.0 }
    },
    State {
        name: "without text"
        when: text_field.text == ""
        PropertyChanges { target: clear_button;
                           opacity: 0.25 }
        PropertyChanges { target: text_field;
                           focus: true }
    }
]
```

Enter text... 

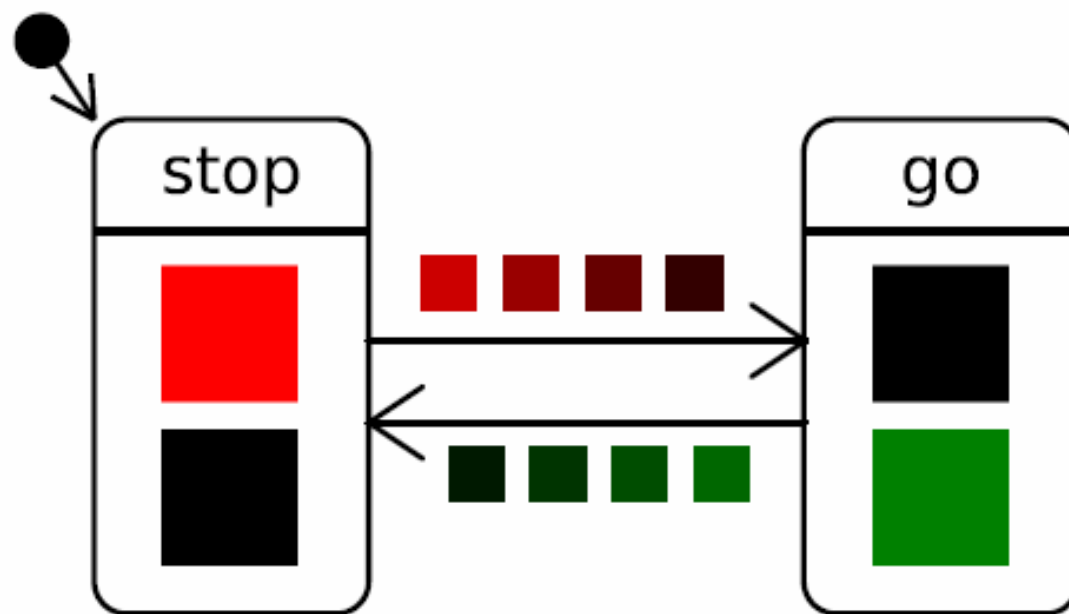


过渡

- Transition 元素用于为状态之间的切换提供动画支持
 - 过渡只能被状态切换激活
 - 过渡中的动画可以以串行或者并行的方式执行
- 通过设置 **to** 和 **from** 属性，我们可以指定与特定状态绑定的动画
 - 它们默认被设置为 `"*"`, i.e. 任意状态
- 过渡可以被设置为 `reversible` (默认 `false`)
 - 当条件满足时，自动切换到以前的状态
 - 例子: 状态切换 1 -> 2 然后 2 -> 1

过渡例子

- 为前面的例子添加过渡...



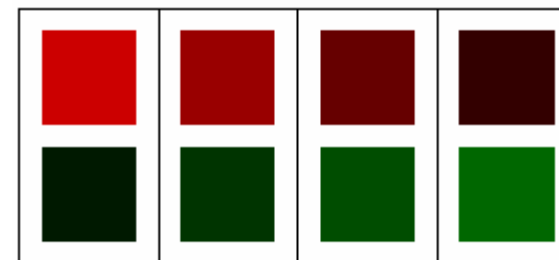
过渡例子

```
transitions: [  
  Transition {  
    from: "stop"; to: "go"  
    PropertyAnimation {  
      target: stop_light  
      properties: "color"; duration: 1000  
    }  
  },  
  Transition {  
    from: "go"; to: "stop"  
    PropertyAnimation {  
      target: go_light  
      properties: "color"; duration: 1000  
    }  
  }  
]
```

- 通过transitions属性来定义多个Transition
- 设置“stop”和“go”之间的状态切换

通用过渡


```
transitions: [  
  Transition {  
    from: "*"; to: "*"  
    PropertyAnimation {  
      target: stop_light  
      properties: "color"; duration: 1000  
    }  
    PropertyAnimation {  
      target: go_light  
      properties: "color"; duration: 1000  
    }  
  }  
]
```



- 使用 “*” 匹配所有状态 (默认值为 “*”)
- 只要有状态改变，过渡就会被执行
- 两个目标的过渡时同时进行的

可逆过渡

```
transitions: [  
  Transition {  
    from: "with text"; to: "without text"  
    reversible: true  
    PropertyAnimation {  
      target: clear_button  
      properties: "opacity"; duration: 1000  
    }  
  }  
]
```

Enter text... 

- 状态如果使用 **when** 属性，就可以使用可逆过渡
- 当两个过渡应用到相同的属性时，可以采用
- 从状态“with text”到“without text”的时候，过渡生效
 - 然后满足条件时，再从“without text”到“with text”
- 没必要定义两个过渡

动画 1/2

- 可以对元素的属性变化加入动画
 - **Types:** `real`, `int`, `color`, `rect`, `point`, `size`
- 有三种使用动画的方法
 - 基本的属性动画, 过渡, 属性行为
 - 后面将会讲到
- 动画可以分组, **i.e.** 串行或并行的执行
 - `SequentialAnimation`, `ParallelAnimation`, `PauseAnimation`
- 预定义的 **easing curve**
 - `OutQuad`, `InElastic`, `OutBounce`, ...
 - 了解更详细的信息, 查看 `PropertyAnimation` 文档

动画 2/2

- 使用属性动画, 我们使用
 - PropertyAnimation, NumberAnimation, or ColorAnimation
 - 它们都有一个公共的基类 Animation
- 对于属性行为, 使用 Behavior
- 对于过渡, 使用 Transition
 - 前面已经学习过了

```
PropertyAnimation {           // A basic property animation
    target: theObject          // The object (id) to be
    animated
    property: "size"           // The property to be animated
    to: "20x20"; duration: 200 // End value & duration
}
```

动画示例 1/2

```
Rectangle {    // Example of a drop-and-bounce effect on an image
  id: rect
  width: 120; height: 200;
  Image {
    id: img
    source: "qt-logo.png"
    x: 60-img.width/2
    y: 0

    SequentialAnimation on y {
      running: true; loops: Animation.Infinite
      NumberAnimation {
        to: 200-img.height; easing.type: "OutBounce"; duration: 2000
      }
      PauseAnimation { duration: 1000 }
      NumberAnimation {
        to: 0; easing.type: "OutQuad"; duration: 1000
      }
    }
  }
}
```


动画示例 2/2

```
PropertyAnimation {           // Animation as a separate element,
    id: animation              // referred to by its id
    target: image
    property: "scale"
    from: 1; to: .5
}

Image {
    id: image
    source: "image.png"
    MouseArea {                // The animation is started upon mouse press
        anchors.fill: parent
        onClicked: animation.start()
    }
}
```

属性行为

- 设置一个默认的动画在属性发生改变的时候执行
 - 无论什么造成的属性改变都会执行!
- 下面这段动画在redRect的 x 发生改变时执行

```
Rectangle {  
    id: redRect  
    color: "red"  
    width: 100; height: 100  
    x: ...  
    Behavior on x {  
        NumberAnimation { duration: 300; easing.type: "InOutQuad" }  
    }  
}
```

过渡示例2, 1/2

```
transitions: [ Transition {  
  // Apply for state changes from any state to MyState and back (optional)  
  from: "*"; to: "MyState"; reversible: true  
  SequentialAnimation {  
    ColorAnimation { duration: 1000 }  
    PauseAnimation { duration: 1000 }  
    ParallelAnimation {  
      // Animate x and y of box1 and box2 simultaneously.  
      // How do we know the start and end values of x and y?  
      NumberAnimation {  
        duration: 1000; easing.type: "OutBounce"  
        target: box1  
        properties: "x,y"  
      }  
      NumberAnimation {  
        duration: 1000  
        target: box2  
        properties: "x,y"  
      }  
    }  
  }  
} ] // End list of Transition elements
```

过渡示例2, 2/2

```
// Example of an explicit transition animation
transitions: [ Transition {
  from: "*"; to: "MyState"; reversible: true
  SequentialAnimation {
    NumberAnimation {
      duration: 1000 easing.type: "OutBounce"
      // Animate myItem's x and y if they have changed in the state
      target: myItem
      properties: "x,y"
    }
    NumberAnimation {
      duration: 1000
      // Animate myItem2's y to 200, regardless of what happens in
      // the state - i.e. run an explicit animation on myItem2
      target: myItem2
      property: "y"
      to: 200 // Must give an end value
    }
  }
} ]
```

使用状态和过渡

- 避免定义过于复杂的状态机
 - 不要用一个状态机来管理所有的UI部分
 - 针对不同的控件使用独立的状态机
 - 然后通过状态把控件的状态联系起来
- 使用script代码设置状态
 - 简单，但是很难管理
 - 没有可逆的过渡
- 使用状态条件来控制状态
 - 更符合声明式的风格
 - 但是状态条件可能会很复杂

总结 - 状态

- 使用状态来管理其他元素的属性:
- 定义状态是用元素的 **states** 属性
 - 每个状态都必须有独立的名称
- 给元素设置一个 **id** 是很有用的
 - 使用 `PropertyChanges` 来修改元素的属性
- 元素的 **state** 属性, 保存了当前的状态
 - 可以使用 `javascript` 来修改它的值
 - 也可以使用 **when** 属性来设置状态成立条件

总结 - 过渡

- 过渡用来描述状态之间的切换过程
- 使用transitions 属性来定义元素的过渡
- 过渡需要描述始末的两个状态
 - 使用from 和to 这个两个属性
 - 使用通配符"*"来表示所有状态
- 过渡是可逆的
 - 相当于把from 和to 属性的值交换

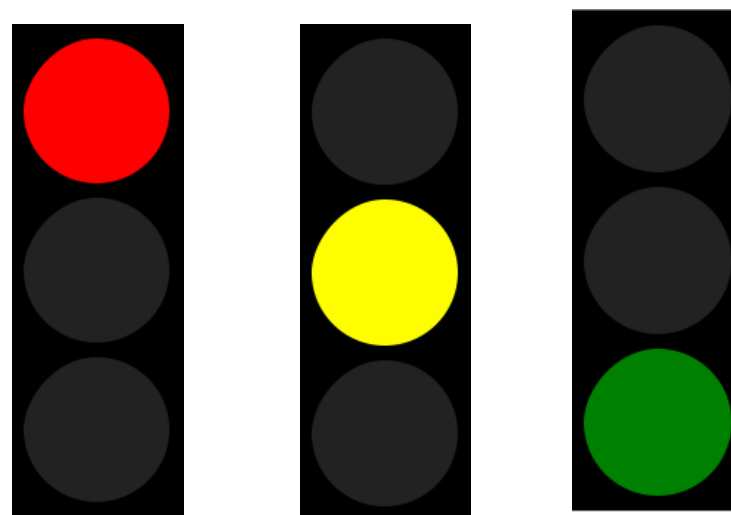
计时器

- 计时器是用Timer 元素表示的
 - 只提供了一个简单的信号: onTriggered
 - 可是单次的或者重复的计时器

```
Timer {  
    interval: 500;  
    running: true;  
    repeat: true  
    onTriggered: time.text = Date().toString()  
}  
  
Text {  
    id: time  
}
```


练习 3 – 交通信号灯

- 使用下面的元素，构造一个模拟的交通信号灯
 - States
 - Transitions
 - Timer



- 不同的状态持续不同的时间

Qt Quick

QML核心特征

Forum **Nokia**

digia

NOKIA

QML文件1/2

- 简单的QML代码包含了QML元素
 - .qml后缀名的文件，或者文本
 - 编码方式为 UTF-8
 - 总是以至少一个 `import` 语句开始
 - Nothing is imported by default
 - 并没有包含什么代码，仅仅只是为在运行时解释器寻找元素的定义
- 定义唯一的顶层QML组件
- Self-contained
 - 在执行之前并没有预处理机制
 - 在运行时解释执行!

QML文件2/2

- HelloWorld这个程序就是存放到QML文件里面的，比如存放到 **HelloWorld.qml** 文件

```
import Qt 4.6 // Import existing QML types to be used in this
               // application, such as Rectangle and Text

Rectangle {
    id: page
    width: 500; height: 200
    color: "lightgray"
    Text {
        id: helloText
        text: "Hello world!"
        font.pointSize: 24; font.bold: true
        y: 30; anchors.horizontalCenter: page.horizontalCenter
    }
}
```

QML组件

- 如前所述，QML文档定义了一个单一的，顶级QML组件
 - 元素对象是根据对应的某个c++的原型而创建的 – i.e. `component` 是在运行时创建的
 - 例如，一个 `"button"` `component` 会用不同的按键字符串被初始化多次
- 组件是由QML基本元素构成的
 - 创建自己的组件是非常容易的
 - 组件的名字要以大写字母开始 (`"MyButton.qml"`)
- 一个组件可以声明为内联组件
 - 用关键字 `Component` 声明
 - 可以使用`parent`的属性或者是导入列表中的元素
 - 需要多次在一个QML文件中根据需要使用（从逻辑上这个组件也只属于这个QML文件）

顶级 QML 控件

```
// Definition in MyButton.qml
// (Notice the capital "M" in the file name above)
import Qt 4.7
Rectangle {
    property alias text: textElement.text
    width: 100; height: 30
    source: "images/toolbutton.sci"
    Text {
        id: textElement
        anchors.centerIn: parent
        font.pointSize: 20
        style: Text.Raised; color: "white"
    }
}

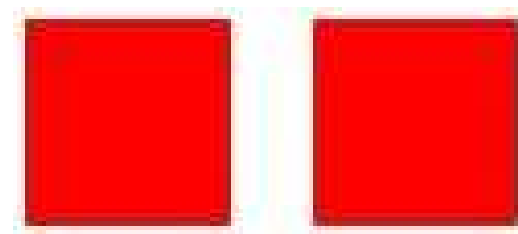
// Usage e.g. in main.qml
// (Just an entry point file, thus lower-case "m")
import Qt 4.7
Rectangle {
    MyButton {
        anchors.horizontalCenter: parent.horizontalCenter
        text: "Orange"
    }
...}
```



Inline QML Component

```
// In MyComponent.qml
import Qt 4.6

Item {
    Component { // The inline component
        id: redSquare
        Rectangle {
            color: "red"
            width: 50
            height: 50
        }
    }
    Loader { sourceComponent: redSquare }
    Loader { sourceComponent: redSquare; x: 70 }
}
```



QML 模块1/2

- 多个QML组件可以组合到一个QML模块中
 - 最简单的方法就是创建一个子文件夹，然后把所有的控件放置进去
 - 这些控件可以使用 `import` 语句导入到QML的文档中:
`import "path_to_mymodule"`
 - 导入路径是和导入控件的QML文件之间的相对路径
- 也可以使用命令的导入
- 用于区分不同的模块，也可以使代码更具有可读性

```
import Qt 4.6 as TheQtLibrary // Into a namespace called TheQtLibrary
TheQtLibrary.Rectangle { ... }

// Multiple imports into the same namespace are also allowed:
import Qt 4.6 as Nokia
import Ovi 1.0 as Nokia
```


QML 模块2/2

- QML 组件也可以在工程外的文件定义

- 这种情况下会用到 URI :

```
import com.nokia.SomeStuff 1.0
```

- 通过上面的导入，就可以访问到放在系统其他地方的 **com/nokia/SomeStuff/** 下的控件了

- E.g. 放在路径 **c:/mycomponents/** 下

- 这个路径可以通过下面的方法设置

- 在C++中，使用 `QmlEngine::addImportPath()`,
 - 在执行 `qml.exe` 的时候使用选项 `-L` 指定路径

- 另外的导入方法:

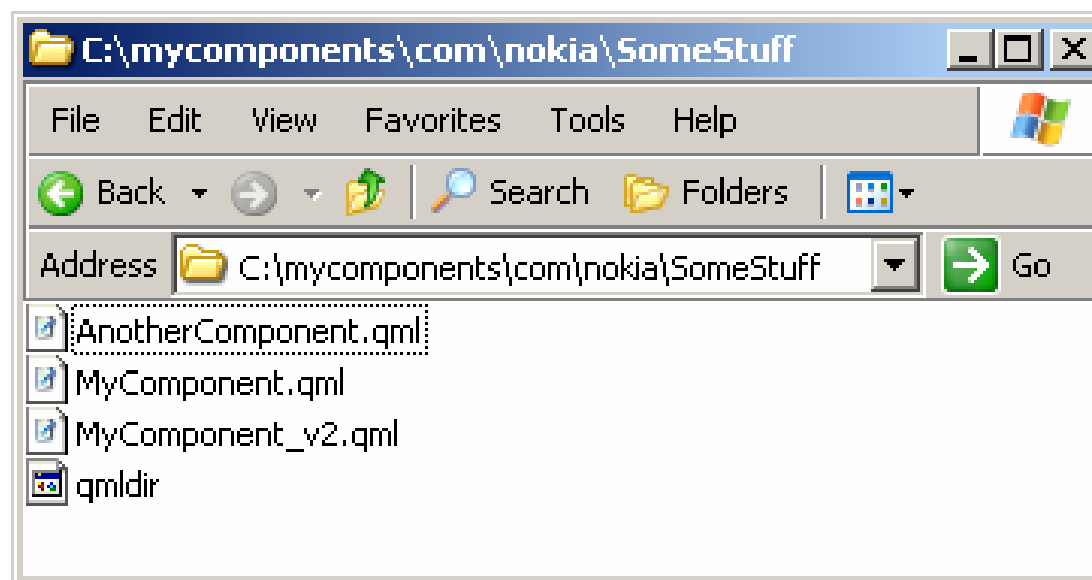
```
import "http://myserver.com/.../..." 1.0
```

- 这种情况下，一个叫 `qmlidir` 的文件描述了目录的内容:

```
# <Comment>
```

```
<TypeName> <InitialVersion> <File>
```

QML 模块- 例子



```
// qmldir contents:  
# This is a comment  
MyComponent 2.0 MyComponent_v2.qml  
MyComponent 1.0 MyComponent.qml  
AnotherComponent 1.0 AnotherComponent.qml
```

网络透明1/2

- 简单的说，就是QML代码中使用的其他内容都用URLs表示
 - 对于本地的或者网络上的内容都适用，支持相对或者绝对路径

```
// Test1.qml containing a reference to an absolute URL
Image { source: "http://www.example.com/images/logo.png" }

// Test2.qml with a relative URL
Image { source: "images/logo.png" }
```

网络透明2/2

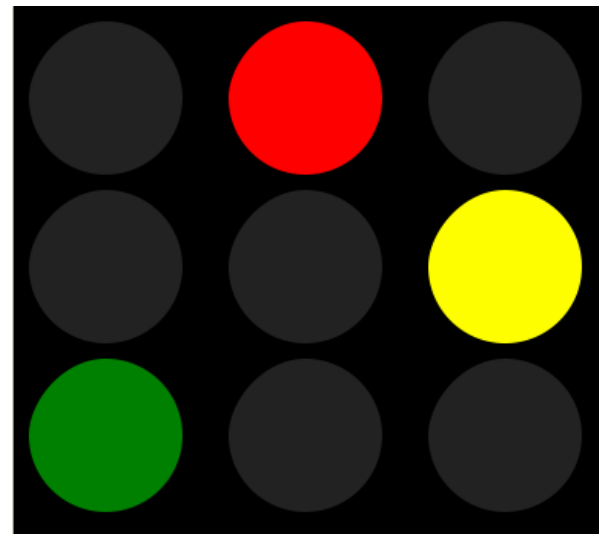
- 相对的URLs会自动转换成绝对路径
 - Absolute URLs always stay as they are
- 例 1: **Test2.qml** 是从下面的网址加载的
`http://www.example.com/mystuff/Test2.qml`
 - 那么 `iamge` 的路径就自动被解析为
 - **`http://www.example.com/mystuff/images/logo.png`**
- 例2: **Test2.qml** 从本地文件加载
`C:/temp/mystuff/Test2.qml`
 - 那么 `iamge` 的路径就自动被解析为
`C:/temp/mystuff/images/logo.png`

渐进载入

- 当QML的某个对象引用网络资源的时候，那么它将会提供整个加载过程的进度状态
 - 访问网络是异步的操作
- 比如, Image有一些的像下面的特殊属性:
 - `status` (Null, Ready, Loading, Error)
 - `progress` (0.0 - 1.0)
 - `width` and `height` also change as the image is loaded
- 应用程序能够绑定这些属性，比如：在合适的时候显示一个进度条
- 对于本地图片状态已经准备好了
 - In future versions this might change
 - If you wish to remain network transparent, do not rely on this!

练习4 – 组件

- 将你前面的所做的练习TrafficLight保存为一个单独的QML控件，并另外写一个mail.qml文件来实例化三个这种控件
- 让这三个控件在同一时间不会处在同一个状态（亮的灯的颜色不同）



Qt Quick

数据模型和视图

Forum **Nokia**

digia

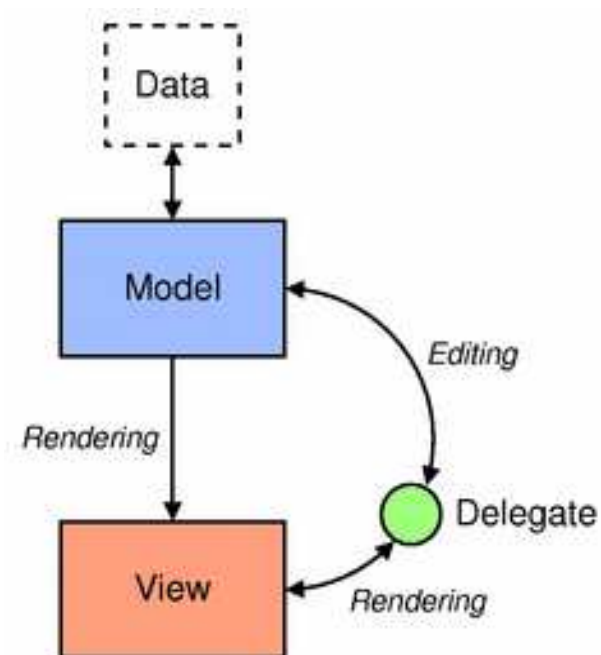
NOKIA

数据模型和视图

- QML使用了与Qt中Model-View类似的结构
- 模型类提供了数据
 - 模型可以使QML的简单数据，或者复杂的C++数据
 - **QML:** `ListModel`, `XmlListModel`, `VisualItemModel`
 - **C++:** `QAbstractItemModel`, `QStringList`, `QList<QObject*>`
- 视图显示模型提供的数据
 - `ListView`, `GridView`, `PathView`, `Repeater` (all QML)
 - 都自动支持滚动
- 代理为视图创建模型中数据的实例
- *Highlight* 控件 用来高亮视图里面的选中item

For Reference: Model-View in Qt

- Model为其他部件提供数据的接口
 - QAbstractItemModel
- View获取model的indices
 - Indices是对数据的引用
- 代理用来定制View中的数据显示方式
 - 当用户编辑时，代理直接与Model交互



我们需要什么？它们是什么？

- Model
 - 你的数据
- Delegate
 - 一个描述model中每条数据的显示方式的控件
- View
 - 可视的元素，使用delegate来显示model中的数据

例子- 列表1/3

```
// Define the data in MyModel.qml - data is static in this simple case
import Qt 4.6

ListModel {
    id: contactModel
    ListElement {
        name: "Bill Smith"
        number: "555 3264"
    }
    ListElement {
        name: "John Brown"
        number: "555 8426"
    }
    ListElement {
        name: "Sam Wise"
        number: "555 0473"
    }
}
```

例子- 列表List 2/3

```
// Create a view to use the model e.g. in myList.qml
import Qt 4.6

Rectangle {
    width: 180; height: 200; color: "green"

    // Define a delegate component. A delegate will be
    // instantiated for each visible item in the list.
    Component {
        id: delegate
        Item {
            id: wrapper
            width: 180; height: 40
            Column {
                x: 5; y: 5
                Text { text: '<b>Name:</b> ' + name }
                Text { text: '<b>Number:</b> ' + number }
            }
        }
    }
} // Rectangle continues on the next slide...
```

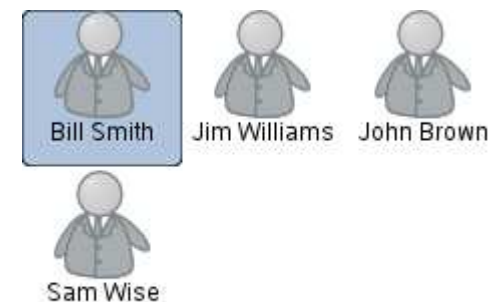
例子- 列表3/3

```
// ...Rectangle continued...
// Define a highlight component. Just one of these will be
// instantiated by each ListView and placed behind the current item.
Component {
    id: highlight
    Rectangle {
        color: "lightsteelblue"
        radius: 5
    }
}

// The actual list
ListView {
    width: parent.width; height: parent.height
    model: MyModel{} // Refers to MyModel.qml
    delegate: delegate // Refers to the delegate component
    highlight: highlight // Refers to the highlight component
    focus: true
}
} // End of Rectangle element started on previous slide
```

网格视图

- GridView
 - 以网格的形式显示数据
 - 与ListView的使用方式一致



```
GridView {  
    width: parent.width; height: parent.height  
    model: MyModel  
    delegate: delegate  
    highlight: highlight  
    cellWidth: 80; cellHeight: 80  
    focus: true  
}
```

路径视图1/3

- PathView
 - 通过一个独立的Path object格式化数据的显示方式
 - 一些预定义的元素可以用于初始化Path
 - PathLine, PathQuad, PathCubic
 - 在path上的items的分布是由PathPercent元素定义的
 - items的显示方式是通过PathAttribute元素来控制的



路径视图2/3

```
PathView {      // With equal distribution of dots
    anchors.fill: parent; model: MyModel; delegate: delegate
    path: Path {
        startX: 20; startY: 0
        PathQuad { x: 50; y: 80; controlX: 0; controlY: 80 }
        PathLine { x: 150; y: 80 }
        PathQuad { x: 180; y: 0; controlX: 200; controlY: 80 }
    }
}
```



```
PathView {      // With 50% of the dots in the bottom part
    anchors.fill: parent; model: MyModel; delegate: delegate
    path: Path {
        startX: 20; startY: 0
        PathQuad { x: 50; y: 80; controlX: 0; controlY: 80 }
        PathPercent { value: 0.25 }
        PathLine { x: 150; y: 80 }
        PathPercent { value: 0.75 }
        PathQuad { x: 180; y: 0; controlX: 200; controlY: 80 }
        PathPercent { value: 1 }
    }
}
```



路径视图3/3

```

Component {
    id: delegate
    Item {
        id: wrapper; width: 80; height: 80
        scale: PathView.scale
        opacity: PathView.opacity
        Column {
            Image { ... }
            Text { ... }
        }
    }
}

PathView {
    anchors.fill: parent; model: MyModel; delegate: delega
    path: Path {
        startX: 120; startY: 100
        PathAttribute { name: "scale"; value: 1.0 }
        PathAttribute { name: "opacity"; value: 1.0 }
        PathQuad { x: 120; y: 25; controlX: 260; controlY: 75 }
        PathAttribute { name: "scale"; value: 0.3 }
        PathAttribute { name: "opacity"; value: 0.5 }
        PathQuad { x: 120; y: 100; controlX: -20; controlY: 75 }
    }
}

```



Repeater 1/2

- 用于创建大量其他items实例的元素
- model的使用和前面的View元素很像
 - model的数据类型可以是object list, a string list, **a number**, or a Qt/C++ model
 - 当前的 model index 可以通过 index 属性访问

```
Column {  
    Repeater {  
        model: 10 // The model is just a number here!  
        Text { text: "I'm item " + index }  
    }  
}
```

I'm item 0
I'm item 1
I'm item 2
I'm item 3
I'm item 4
I'm item 5
I'm item 6
I'm item 7
I'm item 8
I'm item 9

Repeater 2/2

- Repeater所创建的items是按照顺序插入到这个Repeater 的parent中的
 - 可以再layout里面使用Repeater
 - 比如：在Row里面使用Repeater:

```
Row {  
    Rectangle { width: 10; height: 20; color: "red" }  
    Repeater {  
        model: 10  
        Rectangle { width: 20; height: 20; radius: 10; color: "green" }  
    }  
    Rectangle { width: 10; height: 20; color: "blue" }  
}
```



Flickable

- 可以让它的子元素可以被拖拽和滚动
 - 没有必要创建一个MouseArea或者处理鼠标事件
- Flickable界面很容易通过属性配置
 - flickDirection, flickDeceleration, horizontalVelocity, verticalVelocity, boundsBehavior, ...
- 很多QML元素默认是flickable
 - The ListView element, for example

```
Flickable {  
    width: 200; height: 200  
    contentWidth: image.width; contentHeight: image.height  
    Image { id: image; source: "bigimage.png" }  
}
```

Qt Quick

QML高级特性

扩展QML的类型

- QML很多核心的类型和元素都由C++实现的
- 然而, 用纯的QML对这些类型进行扩展也是可能的
- 用QML开发者可以
 - 添加新的属性 `properties`,
 - 添加新的信号 `signals`,
 - 添加新的方法 `methods`,
 - 定义新的QML控件
 - We covered this already

添加新的属性1/4

- 每个属性都必须有一个类型
 - QML有很多已经定义好的类型
 - 所有的QML类型都与对应的C++类型

```
// Syntax of adding a new property to an element  
[default] property <type> <name>[: defaultValue]
```

```
// Example:
```

```
Rectangle {  
    property color innerColor: "black"  
    color: "red"; width: 100; height: 100  
    Rectangle {  
        anchors.centerIn: parent  
        width: parent.width - 10  
        height: parent.height - 10  
        color: innerColor  
    }  
}
```

QML Type	C++ Type
int	int
bool	bool
double	double
real	double
string	QString
url	QUrl
color	QColor
date	QDate
var	QVariant
variant	QVariant

添加新的属性2/4

- 新的属性也可以是现有的属性的别名
 - 新的属性不会被分配新的存储空间
 - 它的类型是由 **aliased** 属性决定的

```
// Syntax of creating a property alias
[default] property alias <name>: <alias reference>

// The previous example using a property alias:
Rectangle {
    property alias innerColor: innerRect.color
    color: "red"; width: 100; height: 100
    Rectangle {
        id: innerRect; anchors.centerIn: parent
        width: parent.width - 10; height: parent.height - 10
        color: "black"
    }
}
```


添加新的属性3/4

- 在定义新的组件时，属性的别名是非常有用的
- 然而，对于别名有一些限制
 - 只有在控件完全实例化的时候才能使用别名
 - 在这个控件里面不能使用别名
 - 在同一个控件中，不能在别名上再建立别名

```
// Does NOT work:  
property alias innerColor: innerRect.color  
innerColor: "black"  
  
// ...and neither does this:  
id: root  
property alias innerColor: innerRect.color  
property alias innerColor2: root.innerColor
```

添加新的属性4/4

- 除了上面的限制，别名提供了很多的灵活性
 - 可以重定义已经存在的属性的行为
 - 并在控件内部仍然使用这个属性
- 比如下面的例子:
 - 定义color别名属性
 - 外面的这个矩形总是红色的，并且用户只能修改里面的矩形的颜色

```
Rectangle {  
    property alias color: innerRect.color  
    color: "red"; width: 100; height: 100  
    Rectangle { id: innerRect; ...; color: "black" }  
}
```

添加新的信号

- 在前面的例子中用到很多QML元素的信号
 - `MouseArea.onClicked`, `Timer.onTriggered`, ...
- 也可以定义自己的信号
 - 在QML中可以直接使用
 - 在C++端，它是普通的Qt信号
 - 信号可以有参数（前面我们所看到的QML类型）

```
Item {  
    signal hovered() // A signal without arguments  
    signal clicked    // The same as above, empty argument list can be omitted  
    signal performAction(string action, var actionArgument)  
}
```

添加新的方法

- 已有的类型也可以添加新的方法
 - 使用JavaScript实现
 - 在QML端可以直接使用，在C++端是槽函数
 - 使用没有类型的参数
 - JavaScript本身是弱类型的
 - 在C++端，它的类型为QVariant

```
// Define a method
Item {
    id: myItem
    function say(text) {
        console.log("You said " + text);
    }
}

// Use the method
myItem.say("HelloWorld!");
```

Qt Quick

QML and Scripting

Forum **Nokia**

digia

NOKIA

介绍

- 前面已经介绍在扩展QML元素时如何添加新的函数
 - 是采用JavaScript编写的，并只属于定义它的元素
- 然而，应用程序的逻辑都是和界面程序分开的
- 为了能够使用这些函数，需要将他们导入到新的QML文档中
 - JavaScript可以直接被写在qml文件中，或者
 - 保存在一个独立的js文件里面
 - 这是个更好的选择
- 应用程序也可以使用QML全局对象提供的服务

QML全局对象

- QML提供了全局的JavaScript对象Qt
 - 在QML的任意部分都可以使用
 - 在前面的例子中我们已经见过全局对象的使用了，MouseArea例子：
acceptedButtons: Qt.LeftButton | Qt.RightButton
- 提供了大量的函数：
 - 创建QML类型：
 - Qt.rect(...), Qt.rgb(...), Qt.point(...)
 - 做一些其他的常用操作：
 - Qt.playSound(...), Qt.openUrlExternally(...), Qt.md5(...)
- 也提供了动态QML对象的创建，AJAX和本地数据访问的接口

在QML中使用JavaScript

- 在QML中使用JavaScript有如下一些限制和特点：
 - JavaScript不能用于为全局对象添加新的成员
 - 在声明变量时，可以省略“var”关键字
- 两种方法使用JavaScript
 - Inline JavaScript
 - 独立的javascript文件

Inline JavaScript

```
Item {  
    function factorial(a) {  
        a = parseInt(a);  
        if (a <= 0)  
            return 1;  
        else  
            return a * factorial(a - 1);  
    }  
  
    MouseArea {  
        anchors.fill: parent  
        onClicked: console.log(factorial(10))  
    }  
}
```

独立 JavaScript 文件

```
import "factorial.js" as MathFunctions
Item {
    MouseArea {
        anchors.fill: parent
        onClicked: console.log(MathFunctions.factorial(10))
    }
}
```

- 如果有很多的JavaScript代码建议就把JS代码写到单独的文件中
- 相对或者绝对的路径的javascript的URLs都是可以被加载的
 - 对于相对路径来讲，是根据与QML文档本省的相对位置转化的。

QML域(Scope)1/2

- 当创建了QML组件实例，QML自动会为他生成一个域(chain scope)用于
 - JavaScript 的执行 (cf. *JavaScript context* in WWW), 和
 - 属性的绑定
- 注意，同一个组件的不同实例可以有不同的域
- 当系统解析某个引用的时候，作用域的搜索是按照特定的顺序的
 - JavaScript variables, functions and property bindings
 - Attached properties or enumerations

QML域(Scope) 2/2

Script block:

```
Script {  
    Function myFunction() {...}  
}
```

Property bindings:

```
anchors.fill: parent  
color: SystemPalette.background
```

Search order

JavaScript Variable Object

Element Type Names

QML Local Scope

QML Component Chain

QmlContext Chain (C++)

QML Global Object

QML域- 元素的类型

- 当访问属性和枚举值的时候使用
- 导入定义好的元素类型的列表
 - 如果所需要的类型没有找，那么将会有一条警告消息发出

```
import Qt 4.6      // Imports PathView and Text type names
Text {
    id: root; scale: root.PathView.scale      // Attached property access
    horizontalAlignment: Text.AlignLeft      // Enumeration access
}
```

QML域- 本地域 1/5

- 每个QML组件都有一个本地域
 - 控件中的子空间也有自己的本地域
 - 绝大多数的变量都是从本地域进行解析的
- 即使在本地域用也有一搜索顺序
 - IDs
 - Script methods
 - Scope object
 - Root object

QML域- 本地域 2/5

```
// main.qml
import Qt 4.6
Rectangle { // Local scope component for binding 1
    id: root
    property string text
    Button { text: root.text // binding 1 }
    ListView {
        delegate: Component { // Local scope component for binding 2
            Rectangle {
                width: ListView.view.width // binding 2
            }
        }
    }
}

// Button.qml
import Qt 4.6
Rectangle { // Local scope component for binding 3
    id: root
    property string text
    Text { text: root.text // binding 3 }
}
```

QML域- 本地域 3/5

- 在组件内部的脚本中，搜索的顺序与属性类似
 - 比如：javascript的函数调用一定是调用最近定义的那个函数

```
Item {  
  
    function getValue() { return 10; } // Method 1  
  
    Rectangle {  
  
        function getValue() { return 11; } // Method 2  
        function getValue2() { return parent.getValue(); } // Method  
3  
  
        x: getValue() // Resolves to Method 2, set to 11  
        y: getValue2() // Resolves to Method 3, set to 10  
    }  
}
```


QML域– 本地域 4/5

- 域object就是包含某段代码或者绑定的块

```
Item {
    Rectangle {          // Scope object for Binding 1
        width: height * 2 // Binding 1 - height is a property of Rectangle
    }
    Text {               // Scope object for Binding 2
        font.pixelSize: parent.height * 0.7 // Binding 2 - parent is a property of Text
    }
}

ListView {
    delegate: Rectangle {
        id: root
        width: ListView.view.width          // Binding 1
        Text {
            width: ListView.view.width // Binding 2 - possibly not the same value as in Binding
1!
        }
        // Should probably be: root.ListView.view.width
    }
}
```

QML域- 本地域 5/5

- 在本地域中最后搜索的是 *root object*
 - 使用 *root object* 可以让数据（属性）传递给子控件
 - *root object* 可能就是与 *scope object* 相同的

```
import Qt 4.6
Item {
    property string description    // Properties of the root object
    property int fontSize
    Text {
        text: description
        font.pixelSize: fontSize
    }
}
```

QML脚本限制1/2

- 在JavaScript不能添加新的成员到QML全局对象中去
 - 由于javascript处理未定义变量的方法，在无意间很可能就违背了某个限制

```
// Assuming that "a" has not been declared anywhere before, this code  
// is illegal - JavaScript would implicitly try to create "a" as  
// a member of the global object, which is not allowed.
```

```
a = 1;  
for (var ii = 1; ii < 10; ++ii) { a = a * ii; }  
console.log("Result: " + a);
```

```
// To make it legal, simply declare "a" properly first:
```

```
var a = 1;  
for (var ii = 1; ii < 10; ++ii) { a = a * ii; }  
console.log("Result: " + a);
```

QML脚本限制2/2

- 在加载的时候，如有**QML**引用了一段外部的脚本文件，这个文件里面有一段全局的代码，那么这段代码的执行的域将会是受限的
 - 执行的域只包含全局对象和引入的脚本文件
- 这个时候不能保证所有的**QML**对象都已经被正确初始化了
 - 所以全局的代码不能像平时那样正常的访问到**QML**对象及其属性

```
// Global code outside a function - works, because there are no
// references to any QML objects or properties
var colors = [ "red", "blue", "green", "orange", "purple" ];

// Invalid global code - the "rootObject" variable is undefined
var initialPosition = { rootObject.x, rootObject.y }
```

启动脚本

- 某些时候我们需要在应用开始执行时，运行一段初始化的代码
 - 或者当一个控件初始化时运行
- 将这段代码放在外部的脚本文件中，并不是一个好的解决方案
 - 当这段代码执行时，并非所有的QML的域都已经被完全初始化了
 - 参考 "QML脚本限制" 小节
- 最好的解决方案是采用Component 元素的onCompleted 这个 attached 属性
 - 它会在整个控件完全初始化后被调用到

```
Rectangle {  
  
    function startupFunction() { // ... startup code }  
  
    Component.onCompleted: startupFunction();  
}
```

Mega-练习

- 联系人
- 将联系人在GridView中列出
- 把Model单独放在另外一个文件中
- 点击联系人打开详细的信息，并提供关闭详细信息的方法
- 可以从最简单的文本开始，一步一步的扩展



Qt Quick

在 Qt/C++ 应用中使用QML

Forum **Nokia**

digia

NOKIA

介绍

- 为了在C++中使用QML在QtDeclarative中有三个主要的类
 - QDeclarativeEngine
 - QDeclarativeComponent
 - QDeclarativeContext
- 很多QML元素也有对应方法获取用C++创建好的元素实例
 - Item <-> QDeclarativeItem
 - Scale <-> QGraphicsScale
 - Blur <-> QGraphicsBlurEffect
- 为了使用QtDeclarative 在的工程文件中.pro加入下面的内容:
 - QT += declarative

QDeclarativeView

- 一个简单易用的显示类
 - QDeclarativeView (继承自 QGraphicsView)
 - 主要是用于快速的建立应用原型

```
#include <QtGui/QApplication>
#include <QtCore/QString>
#include <QtDeclarative/QDeclarativeView>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QDeclarativeView canvas;
    canvas.setSource(QUrl("main.qml"));
    canvas.show();
    return app.exec();
}
```

QDeclarativeEngine

- 要在Qt/C++中访问QML都必须有一个QDeclarativeEngine实例
 - 提供在C++中初始化QML控件的环境
 - 可以通过它来配置全局的QML设置
 - 如果要提供不同的QML设置，需要实例化多个QDeclarativeEngine

QDeclarativeComponent

- 用来加载QML文件
 - QDeclarativeComponent 对应一个QML文档的实例
- 加载的内容可以是路径也可以试QML代码
 - URL可以是本地的文件或者QNetworkAccessManager支持的协议访问的网络文件
- 包含了QML文件的状态信息
 - Null, Ready, Loading, Error

实例 – 初始化组件

```
// Create the engine (root context created automatically as well)
QDeclarativeEngine engine;

// Create a QML component associated with the engine
// (Alternatively you could create an empty component and then set
// its contents with setData().)
QDeclarativeComponent component(&engine, QUrl("main.qml"));

// Instantiate the component (as no context is given to create(),
// the root context is used by default)
QDeclarativeItem *item =
    qobject_cast<QDeclarativeItem *>(component.create());

// Add item to a view, etc ...
```

QDeclarativeContext 1/5

- 每个QML组件初始化都会对应一个QDeclarativeContext
 - engine会自动建立root context
- 子context可以根据需要而创建
 - 子context是有继承关系的
 - 根context是子context的父亲
 - 这个继承关系是由QDeclarativeEngine管理维护的
- QML组件实例的数据都应该加入到engine的root环境中
- QML子组件的数据也应该加入到子环境中(sub-context)

QDeclarativeContext 2/5

- 使用context可以把C++的数据和对象暴露给QML

```
// main.qml
import Qt 4.6
Rectangle {
    color: myBackgroundColor
    Text {
        anchors.centerIn: parent
        text: "Hello Light Steel Blue World!"
    }
}

// main.cpp
QDeclarativeEngine engine;
// engine.rootContext() returns a QDeclarativeContext*
(engine.rootContext())->setContextProperty("myBackgroundColor",
                                           QColor(Qt::lightsteelblue));
QDeclarativeComponent component(&engine, "main.qml");
QObject *window = component.create(); // Create using the root context
```

QDeclarativeContext 3/5

- 这种机制可以被用来为QML中的View提供C++端的model

```
QDeclarativeEngine engine;  
  
// Expose modelData (e.g. of type QAbstractItemModel) by the name  
// myModel to QML  
(engine.rootContext())->setContextProperty("myModel", modelData);  
  
// Create a QML component  
QDeclarativeComponent component(&engine,  
    "import Qt 4.6 \n ListView { model=myModel }");  
  
// Instantiate component  
component.create();
```

QDeclarativeContext 4/5

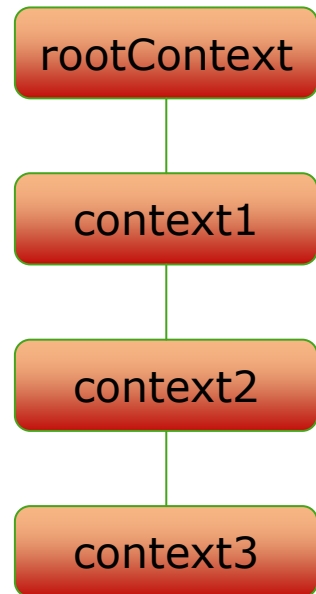
- 前面提到过，`context`是具有继承关系的
 - 控件初始化的时候，可以使用对应的`context`里的数据，也可以访问到祖先`context`的数据
- 对于重复定义的数据，子`context`中的定义将会覆盖`context`中的定义

QDeclarativeContext 5/5

```
QDeclarativeEngine engine;
QDeclarativeContext context1(engine.rootContext());
QDeclarativeContext context2(&context1);
QDeclarativeContext context3(&context2);

context1.setContextProperty("a", 12);
context2.setContextProperty("b", 13);
context3.setContextProperty("a", 14);
context3.setContextProperty("c", 14);

// Instantiate QDeclarativeComponents using the sub-contexts
component1.create(&context1); // a = 12
component2.create(&context2); // a = 12, b = 13
component3.create(&context3); // a = 14, b = 13, c = 14
```



结构化数据

- 如果你有很多数据需要暴露给QML可以使用默认对象（default object）代替
 - 所有默认对象中定义的属性都可以在QML控件中通过名字访问到
 - 通过这种方式暴露的数据，可以在QML端被修改
 - 使用默认对象的速度比多次调用setContextProperty（）快一些
- 多个默认的对象可以加到同一个QML组件实例中
 - 先添加的默认对象是不会被后面添加的覆盖
 - 与此不同的是，使用setContextProperty（）设置的属性，将会被新的属性覆盖

结构化数据

- 如果你有很多数据需要暴露给QML可以使用默认对象（default object）代替
 - 所有默认对象中定义的属性都可以在QML控件中通过名字访问到
 - 通过这种方式暴露的数据，可以在QML端被修改
 - 使用默认对象的速度比多次调用setContextProperty（）快一些
- 多个默认的对象可以加到同一个QML组件实例中
 - 先添加的默认对象是不会被后面添加的覆盖
 - 与此不同的是，使用setContextProperty（）设置的属性，将会被新的属性覆盖

结构化数据

- model数据通常都是由C++端代码动态提供的，而不是一个静态QML的数据model
 - 在delegate里面通过model属性可以访问到数据模型
 - 默认属性
- 我们可以使用的C++端的数据模型有
 - QList<QObject*> <-> model.modelData.xxx (xxx是属性)
 - QAbstractDataModel <-> model.display (decoration)
 - QStringList <-> model.modelData

结构化数据 – 例子

```
// MyDataSet.h
class MyDataSet : ... {
    ...
    // The NOTIFY signal informs about changes in the property's value
    Q_PROPERTY(QAbstractItemModel *myModel READ model NOTIFY modelChanged)
    Q_PROPERTY(QString text READ text NOTIFY textChanged)
    ...
};

// SomeOtherPieceOfCode.cpp exposes the QObject using e.g. a sub-context
QDeclarativeEngine engine;
QDeclarativeContext context(engine.rootContext());
context.addContextObject(new MyDataSet(...));
QDeclarativeComponent component(&engine, "ListView { model=myModel }");
component.create(&context);
```

QML调用C++方法

- 所有Qobject对象的public的槽方法都可以在QML中调用
- 如果你不想你的方法是槽方法，可以使用 `Q_INVOKABLE`
 - `Q_INVOKABLE void myMethod();`
- 这些方法可以有参数和返回值
- 目前支持下面的类型:
 - `bool`
 - `unsigned int, int, float, double, real`
 - `QString, QUrl, QColor`
 - `QDate, QTime, QDateTime`
 - `QPoint, QPointF, QSize, QSizeF, QRect, QRectF`
 - `QVariant`

示例 1/2

```
// In C++:
class LEDBlinker : public QObject {
    Q_OBJECT
    // ...
public slots:
    bool isRunning();
    void start();
    void stop();
};

int main(int argc, char **argv) {
    // ...
    QDeclarativeContext *context =
        engine->rootContext();
    context->setContextProperty("ledBlinker",
        new LEDBlinker);
    // ...
}
```

```
// In QML:
import Qt 4.6

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

示例 2/2

- 需要注意的是，我们完全可以通过声明一个“running”属性来达到同样的效果
 - 代码更加优雅
 - 需要实现这里省略掉的 `isRunning()` 和 `setRunning()` 两个方法

```
// In C++:
class LEDBlinker : public QObject {
    Q_OBJECT
    Q_PROPERTY(bool running READ isRunning WRITE setRunning)
    // ...
};

// In QML:
Rectangle {
    MouseArea {
        anchors.fill: parent
        onClicked: ledBlinker.running = !ledBlinker.running
    }
}
```


在C++调用QML方法

- 很明显反过来在C++中调用QML的方法也是可以的
 - 在QML中定义的方法在C++中都是一个槽函数
 - 前面也提到了，在QML中定义的信号可以与C++中定义的槽函数连接

网络组件1/2

- 前面讨论过，QML组件可以通过网络加载
- 这种方式，组件的实例化可能会花些时间
 - 由于网络有一定延迟
- 在C++中初始化网络上QML控件的时候
 - 需要观察控件的加载状态
 - 只有当状态为Ready后，才能调用create（）创建控件

网络组件2/2

```
MyObject::MyObject() {
    component = new QDeclarativeComponent(engine,
        QUrl("http://www.example.com/main.qml"));
    // Check for status before creating the object - notice that this kind of
    // code could (should?) be used regardless of where the component is located!
    if (component->isLoading())
        connect(component, SIGNAL(statusChanged(QDeclarativeComponent::Status)),
            this, SLOT(continueLoading()));
    else
        continueLoading(); // Not a network-based resource, load straight away
}

// A slot that omits the Status parameter of the signal and uses the isXxxx()
// functions instead to check the status - both approaches work the same way
void MyObject::continueLoading() {
    if (component->isError()) {
        qWarning() << component->errors();
    } else if (component->isReady()) {
        QObject *myObject = component->create();
    } // The other status checks here ...
}
```

QML Components in Resource File 1/2

- 在Qt工程中最方便的方法还是把QML组件添加到资源文件中
 - 所有的javascript文件也可以被放在资源文件中
- 更加容易访问文件
 - 没有必要知道文件的路径
 - 只需要使用一个指向资源文件中的文件URL就行了
- 资源文件可以编译到二进制程序中
 - 这样资源文件就与二进制文件一起分发了，非常的方便

QML Components in Resource File 2/2

```
// MyApp.qrc
<!DOCTYPE RCC>
<RCC version="1.0">
    <qresource> <file>qml/main.qml</file> </qresource>
</RCC>

// MyObject.cpp
MyObject::MyObject() {
    component = new QDeclarativeComponent(engine,
        QUrl("qrc:/qml/main.qml"));
    if (!component->isError()) {
        QObject *myObject = component->create();
    }
}

// main.qml
import Qt 4.6
Image { source: "images/background.png" }
```

Qt Quick

总结

Forum **Nokia**

digia

NOKIA

Summary

- Qt Quick 用来为界面设计人员和开发人员设计Qt应用程序的可视化界面
- QML为开发人员预定义了一套类型
 - 可以很容易的使用QML进行扩展
 - 可以使用C++进行扩展
- Qt的meta-object系统提供了QML和C++关联的机制
 - Qt 属性机制
 - 信号与槽机制

Thank You!