Forum Nokia

Qt Quick

讲师：　　龚　宇
邮箱：yu.gong@digia.com

课程内容 – 第一天 - QML

- 介绍
  - 什么是 Qt Quick?
  - 开发工具
- QML 本质
  - 基础语法
  - 属性
  - 标准QML元素
  - 属性绑定
  - 附加属性
- QML中的布局管理
  - Grid, Row, 和 Column 布局
- 用户交互
  - 鼠标区域
  - 导航键
  - 键盘事件
- 状态, 过渡 和 动画
- 核心QML特性
  - QML 组件
  - 模块

课程内容 – 第二天

- **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
  - 脚本
  - QML 范围
  - QML 脚本 的限制
  - 启动脚本
- **在 Qt/C++ 应用中使用 QML**
  - 主要的类
  - 结构化数据
  - 动态结构化数据
  - 网络组件
- **通过Qt/C++扩展QML**
  - 添加新的类型
  - QML与Qt/C++之间的通信
  - QML扩展插件

Forum Nokia

Qt Quick

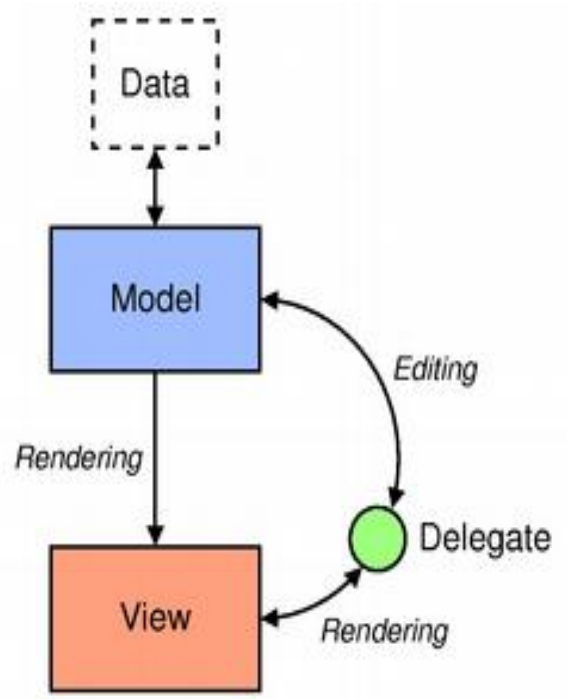
数据模型和视图

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
  - 模型可以是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`
  - 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 4.7

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

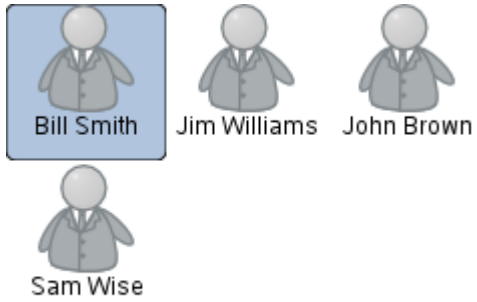
```
// Create a view to use the model e.g. in myList.qml
import Qt 4.7

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上的的items的的分布是由`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: delegate
    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元素类似
  - 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
  - 可以在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

- 让它的孩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
  - 比如：`ListView` 元素

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

Forum Nokia

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
- 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
    - 前面已经讲述

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类型都有对应的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   // empty argument list can be omitted
    signal performAction(string action, var actionArgument)
}
```

添加新的方法

- 可以为已有的类型添加新的方法
  - 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!);
```

Forum Nokia

Qt Quick

QML and Scripting

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.rgba(…), 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对象的创建和本地数据的访问

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文档本身的相对位置转化的。

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
  - 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 变量, 函数 以及 属性绑定
  - 附加属性 或者 枚举

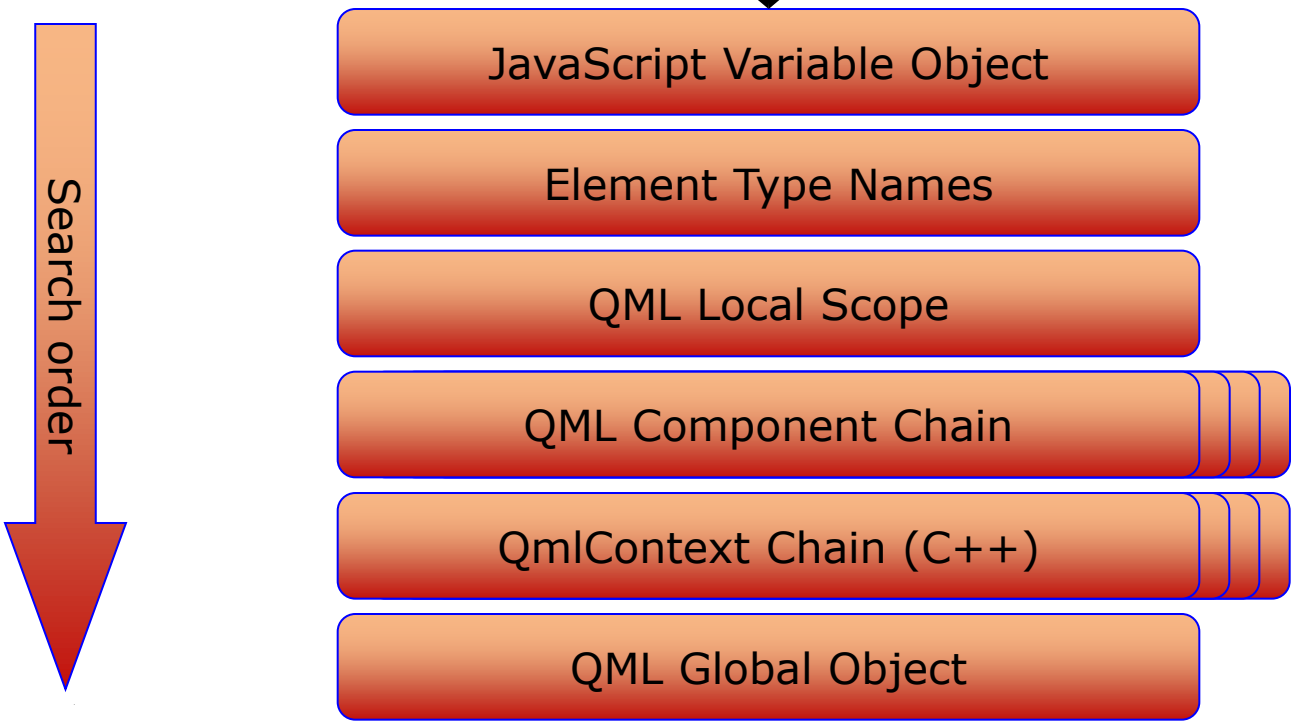
QML域(Scope) 2/2

Script block:

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

Property bindings:

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



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 4.7     // 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
  - 控件中的子控件也有自己的本地域
  - 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定的搜索顺序
  - IDs
  - Script methods
  - Scope object
  - Root object

QML域– 本地域 2/5

```
// main.qml
import Qt 4.7
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 4.7
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 4.7
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这个限制

```
// 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

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

- 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 这个附加属性
  - 它会在整个控件完全初始化后被调用到

Mega-练习6

- 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

Forum Nokia

Qt Quick

在 Qt/C++ 应用中使用QML

介绍

- 为了在C++中使用QML在QtDeclarative中几个主要的类
  - QDeclarativeView
  - QDeclarativeEngine
  - QDeclarativeComponent
  - QDeclarativeContext
- QML元素在Qt/C++端都有对应的类
  - Item <-> QDeclarativeItem
  - Scale <-> QGraphicsScale
  - Blur <-> QGraphicsBlurEffect
- 为了使用QtDeclarative，在工程文件中.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/QUrl>
#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是QML代码
  - URL可以是本地的文件或者`QNetworkAccessManager`支持的协议表示的网络文件
- 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 4.7
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

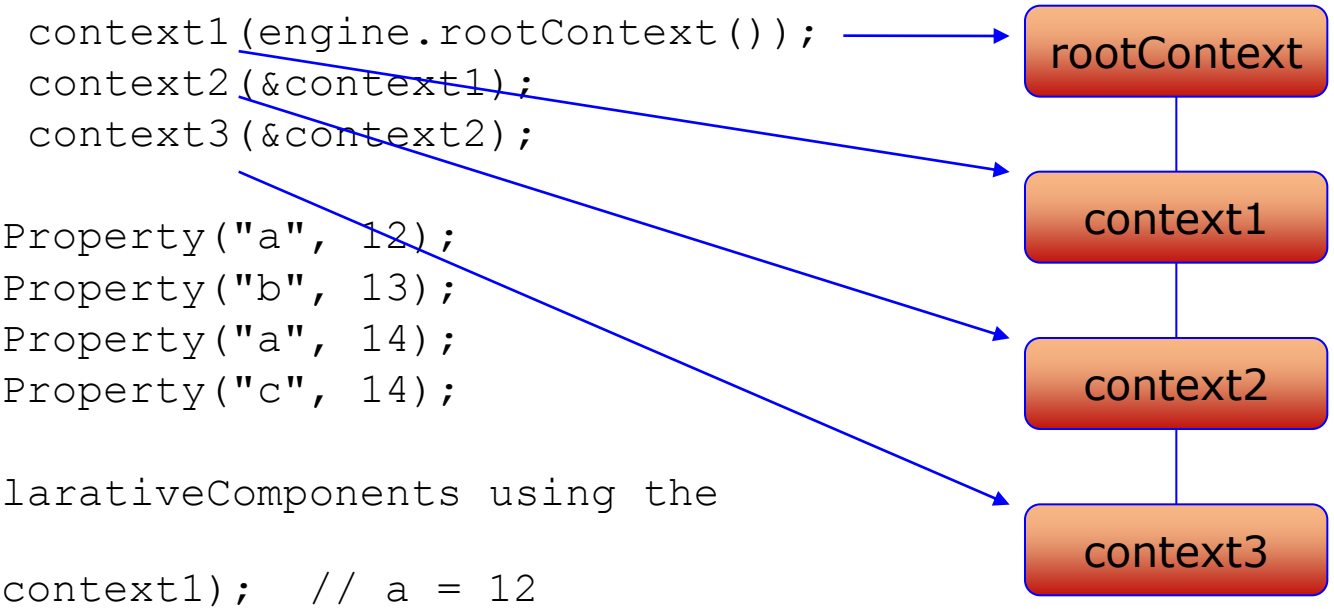
```
QDeclarativeEngine engine;

// 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 4.7 \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中的定义将会覆盖父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
// a = 14, b = 13, c = 14
component3.create(&context3);
```



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.setContextObject(new MyDataSet(...));
QDeclarativeComponent component(&engine, "ListView {
   model:myModel }");
component.create(&context);
```

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 4.7
Image { source: "images/background.png" }
```

Forum Nokia

Qt Quick

用 Qt/C++ 扩展QML

添加新的类型

- 前面讨论过，QML的可视元素的基类是Item
- Item在Qt/C++端对应的类是QDeclarativeItem
  - 因此，用Qt/C++扩展新的类型，我们应该从QDeclarativeItem继承
  - 当然，从QDeclarativeItem的子类继承也是可以的

添加新的类型

- 然后重写虚函数void QDeclarativeItem::paint (QPainter * painter, …)，完成我们需要的图形绘制
  - QDeclarativeItem是QGraphicsItem
  - 默认情况下是没有图形绘制的
    - 通过在构造函数中调用setFlag(QGraphicsItem::ItemHasNoContents, false)，支持绘图
  - 绘制的矩形区域由boundingRect()获取

添加新的类型

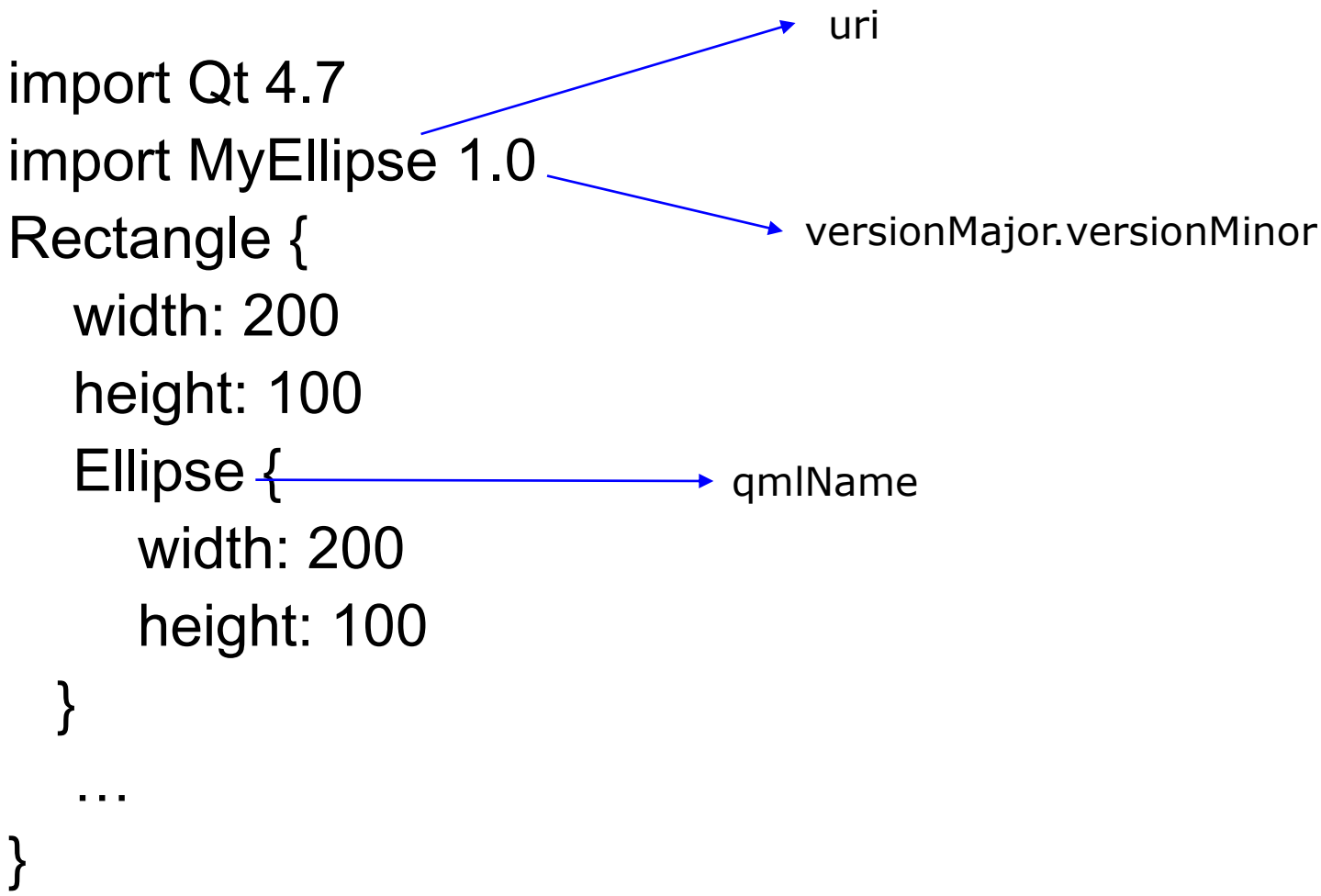
```
//declarativeellipse.h

class QDeclarativeEllipse : public QDeclarativeItem
{
    Q_OBJECT
    Q_PROPERTY(…)
public:
    explicit QDeclarativeEllipse(QDeclarativeItem *parent = 0);
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget = 0);
};

declarativeellipse.cpp
QDeclarativeEllipse::QDeclarativeEllipse(QDeclarativeItem *parent) : QDeclarativeItem(parent)
{
    setFlag(QGraphicsItem::ItemHasNoContents, false);
}
void QDeclarativeEllipse::paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget
    *widget)
{
    …
    painter->drawEllipse(boundingRect());
}
```

添加新的类型

- 要在QML文档中使用前面所定义的类型，首先需要调用qmlRegisterType (…)注册
  - int qmlRegisterType (const char * uri, int versionMajor, int versionMinor, const char * qmlName)
  - 其中uri和qmlName没有直接的关系，但是可以相同

```
qmlRegisterType<QDeclarativeEllipse>(“MyEllipse", 1, 0, "Ellipse");
QDeclarativeView view;
view.setSource(QUrl::fromLocalFile("app.qml"));
view.show();
```

添加新的类型


uri
import Qt 4.7
import MyEllipse 1.0
versionMajor.versionMinor
Rectangle {
width: 200
height: 100
Ellipse {
qmlName
width: 200
height: 100
}
…
}

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 4.7

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
  - 但是，我们通常都不会这样做。后面会提到其他方法。

mail.qml

```
Rectangle {
    signal signalFromQML()
    function slotFromQML(text) { text1.text = text }
    Text { id: text1; text: “text QML” }
}
```

在C++调用QML方法

```
class MyObject : public QObject
{
    Q_OBJECT
    …
signals:
    void signalFromQt(QVariant);
public slots:
    void slotFromQt();
    …
};
```

```
QDeclarativeView view;
view.setSource(QUrl::mail.qml));
QGraphicsObject *object = view.rootObject();
MyObject *myObj = new MyObject(…);
…
QObject::connect(myObj ,
SIGNAL(singalFromQt(QVariant)), object,
SLOT(slotFromQML(QVariant)));
…
QObject::connect(object,
SIGNAL(signalFromQML()), myObj,
SLOT(slotFromQt()));
```

信号与槽

- Qt/C++端的槽，我们在前面已经知道怎么调用了。
- Qt/C++端的信号可以通过元素Connections与QML端的槽建立连接
  - 假设LEDBlinker示例中LEDBlinker有一个信号stopped();

```
Connections {
    target: ledBlinker
    onStopped: { … }
}
```

QML扩展插件

- 通过QML扩展插件，我们就不必通过在main.cpp里面编写C++代码来支持新的QML类型
- 直接用qmlviewer就可以解析执行

- 有兴趣可以参看
  - Qt安装目录/examples/declarative/tutorials/extending/chapter6-plugins
  - 非常的详尽

Forum Nokia

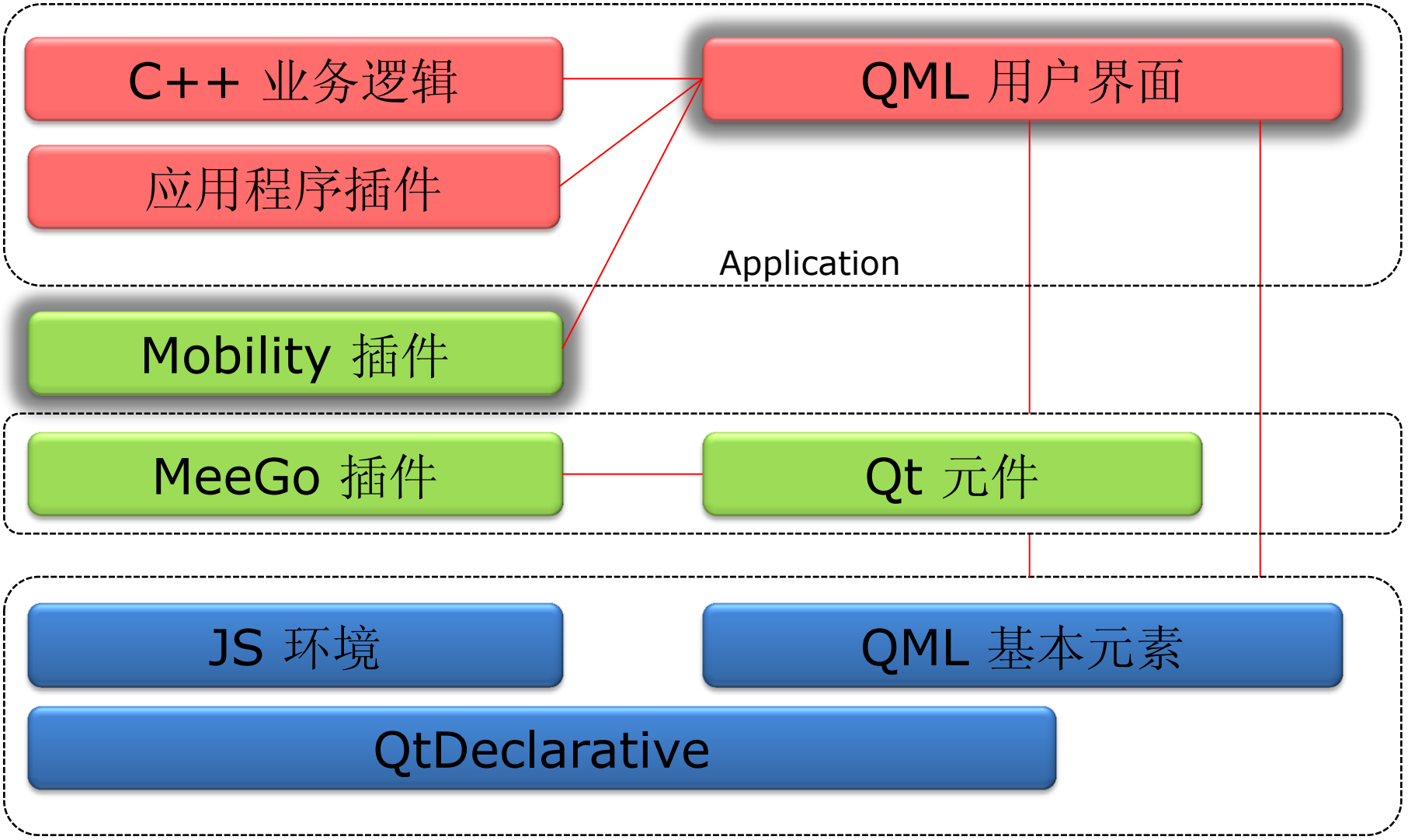
Qt Quick

在QML中使用Mobility插件

Mobility插件

- QML 与 Mobility APIs 绑定
- Mobility API 在 QML 中就是普通的元素
- Mobility API 1.2 会引入更多的 QML 绑定元素

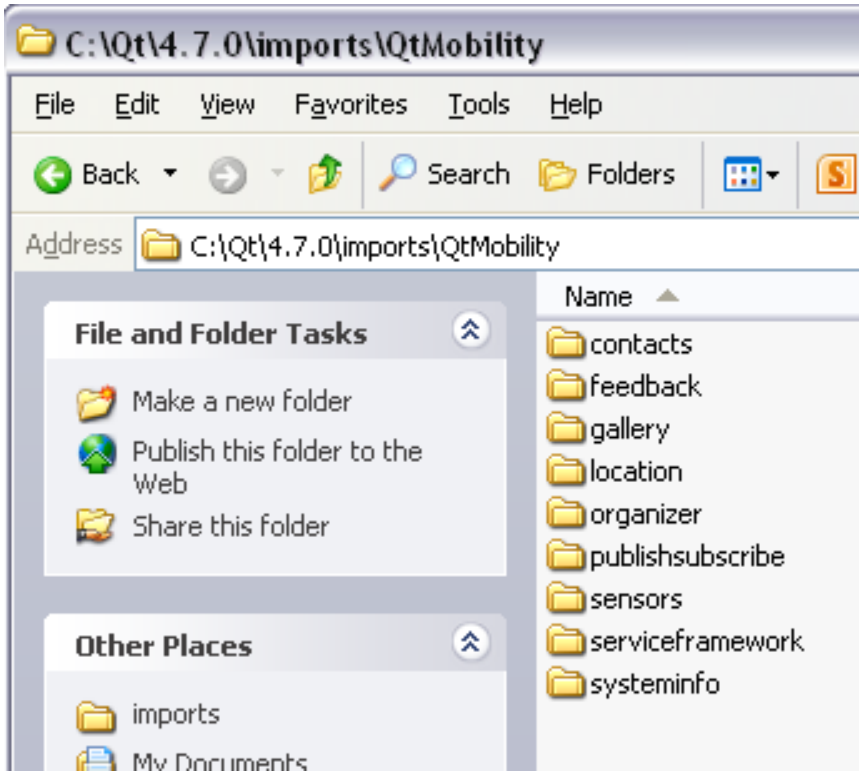
应用程序结构


C++ 业务逻辑
QML 用户界面
应用程序插件
Application
Mobility 插件
MeeGo 插件
Qt 元件
JS 环境
QML 基本元素
QtDeclarative

当前可用的 Mobility 插件

- Gallery
- Location
- Multimedia
- Service Framework
- Messaging

Mobility API v1.1



相关资源

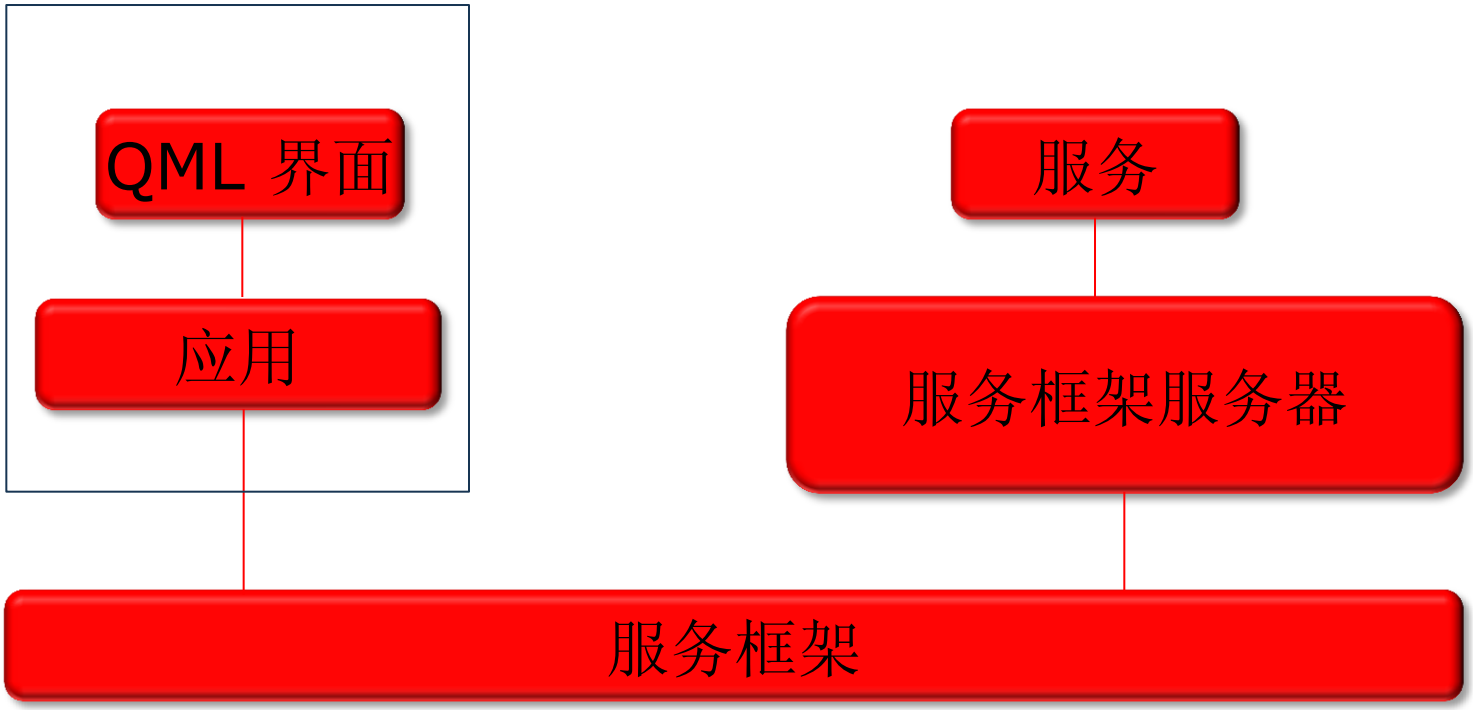
- http://doc.qt.nokia.com/qtmobility-1.1/qml-plugins.html

- Status:
  - http://doc.qt.nokia.com/qtmobility-1.1/index.html#platform-compatibility

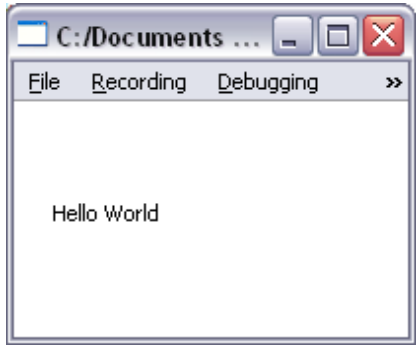
服务框架

- 在QML端可以使用被封装好的功能服务
  - 服务框架插件
  - 服务框架服务器
- 服务
  - 服务都有自己的标识
- 连接
  - 使用服务的槽函数去发出请求
  - 通过服务的信号得到所请求所返回的信息

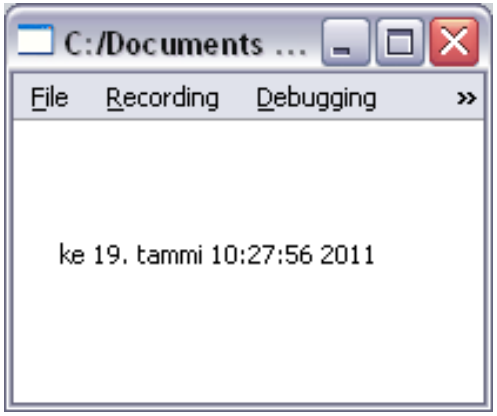
QML 应用访问服务



例子: 客户端




```
import Qt 4.7
import QtMobility.serviceframework 1.1

Rectangle {
    id: theWindow; width: 200; height: 120
    property variant clockService: 0

    Text {… MouseArea {…}}
    Service {…}
    Connections {…}
}
```



访问服务

```
Service {
  id: theService
  interfaceName: "com.digia.qt.example.clock_service"

  Component.onCompleted: {
    theWindow.clockService = theService.serviceObject
  }
}
```

服务的信号

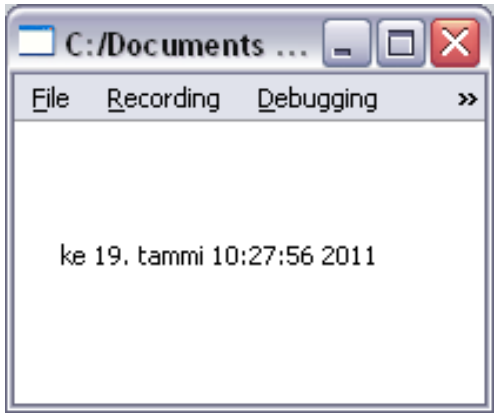
```
Connections {
        target: clockService
        ignoreUnknownSignals: true

        onHereItComes: {
            theText.text = time
        }
    }
```

ignoreUnknownSignals: true

- 在建立连接的时候，服务有可能还不能使用

- 连接会发出一条警告，但是仍会正常工作

服务的槽函数

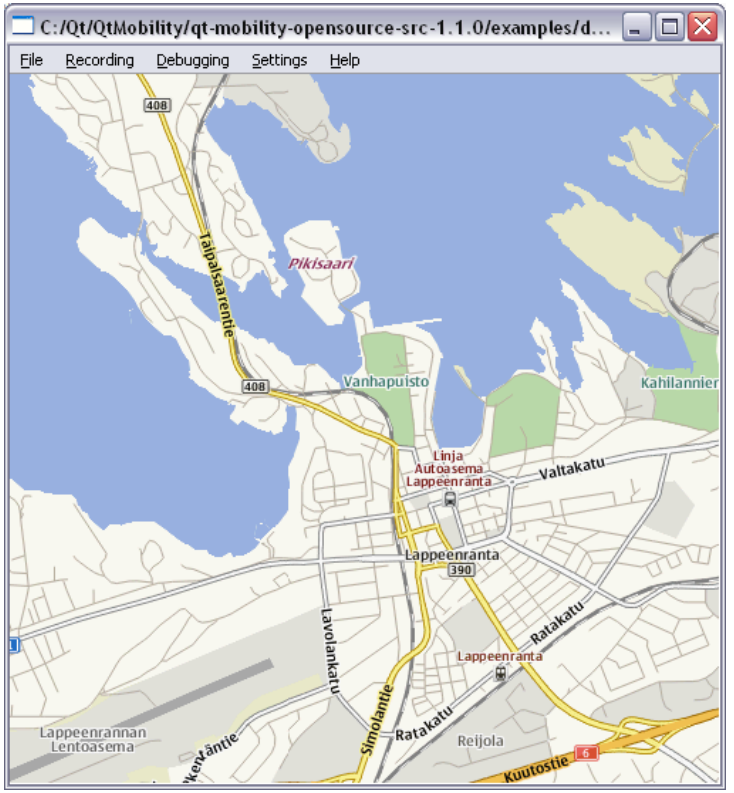



```
Text {
        id: theText; x: 20; y: 50
        text: "Hello World"

        MouseArea {
            anchors.fill: theText
            onClicked: {
                clockService.requestTime();
            }
        }
    }
```

QtMobility.sensors 1.1

```
Rectangle {
    width: 200; height: 200
    Text {
     id: theText; x: 66;  y: 93
     text: " -- not updated yet --"
    }
    OrientationSensor {
     active: true
     onReadingChanged: {
     theText.text = "orientation="+reading.orientation
    }}}
```

QtMobility.sensors 1.1




```
Map {
 id: map
 plugin: Plugin {
  name : "nokia"
 } …

MouseArea { …
onDoubleClicked: {
map.zoomLevel += 1
map.center = map.toCoordinate(Qt.point(mouse.x, mouse.y))
}
```

QtMobility.gallery 1.1

- DocumentGalleryItem
- DocumentGalleryModel
- DocumentGalleryType
- 另外，还有一些可以和model共同使用的过滤器元素
  - GalleryContainsFilter, GalleryEndsWithFilter, …

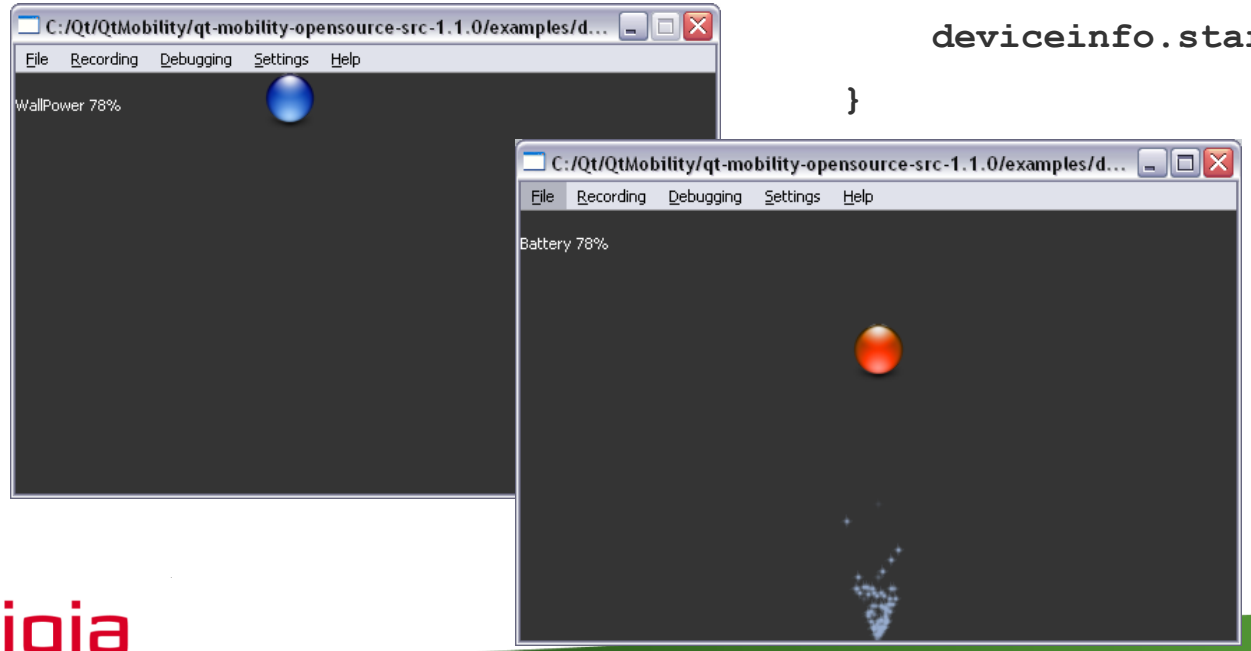
- 请参考 http://doc.qt.nokia.com/qtmobility-1.1/qml-gallery.html

QtMobility.systeminfo 1.1

- DeviceInfo
- GeneralInfo
- NetworkInfo
- ScreenSaver

- batteryLevelChanged
- batteryStatusChanged
- bluetoothStateChanged
- currentProfileChanged
- powerStateChanged

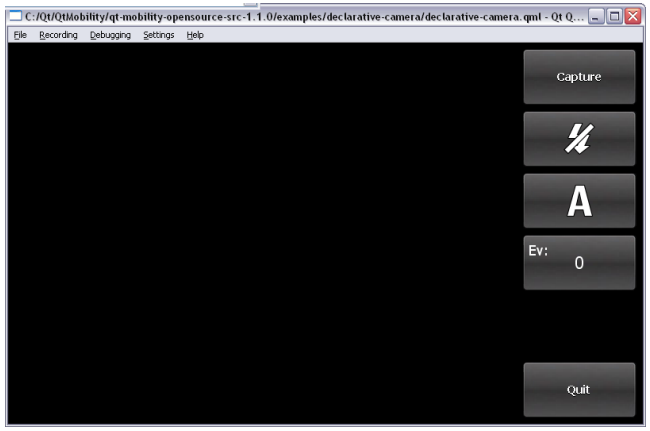
```
Component.onCompleted: {
    deviceinfo.startPowerStateChanged();
    deviceinfo.startBatteryLevelChanged();
}
```



QtMobility.messaging 1.1

- MessageModel
- MessageFilter
- MessageIntersectionFilter
- MessageUnionFilter

QtMobility.messaging 1.1




```
Camera {
    id: camera; x: 0; y: 0
    width: 640; height: 480
    focus: visible
    captureResolution : "640x480"

    flashMode: stillControls.flashMode
    whiteBalanceMode: stillControls.whiteBalance
    exposureCompensation: stillControls.exposureCompensation

    onImageCaptured: { …}}
```

相关资源

- http://doc.qt.nokia.com/qtmobility-1.1/qml-plugins.html
- http://doc.qt.nokia.com/qtmobility-1.2/qml-plugins.html

Forum Nokia

Qt Quick

总结

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
- Wiki链接
  - http://wiki.forum.nokia.com/index.php/Category:Qt

Forum Nokia

Thank You!