1.介绍

QML 是一种描述语言，主要是对界面效果等的一种描述，它可以结合 javaScript 来进行更复杂的效果及逻辑实现。比如做个游戏，实现一些更有趣的功能等

2.简单的例子

```
import Qt 4.7
        Rectangle{
        width:200
        height:200
        color:"blue"
        }
```

代码是绘制一个蓝色的矩形，宽200高200，import 包含一个 qt4.7的包

3.基本元素的介绍（自己翻译意思会有出入，敬请见谅）

基本可视化项

Item 基本的项元素 在 QML 中所有可视化的向都继承他

Rectangle 基本的可视化矩形元素

Gradient 定义一个两种颜色的渐变过程

GradientStop 定义个颜色，被 Gradient 使用

Image 在场景中使用位图

BorderImage(特殊的项)定义一张图片并当做边界

AnimatedImage 为播放动画存储一系列的帧

Text 在场景中使用文本

TextInput 显示可编辑为文本

IntValidator int 验证器

DoubleValidator double 验证器

RegExpValidator 验证字符串正则表达式

TextEdit 显示多行可编辑文本

基本的交互项

MouseArea 鼠标句柄交互

FocusScope 键盘焦点句柄

Flickable 提供一种浏览整张图片的一部分的效果,具体看例子

Flipable 提供一个平面，可以进行翻转看他的前面或后面,具体看例子

状态

State 定义一个配置对象和属性的集合

PropertyChanges 使用一个 State 描述属性的改变

StateGroup 包含一个状态集合和状态变换

ParentChange 重新定义父集，也就是换个父节点

AnchorChanges 在一个状态中改变 anchors

动画和变换

Behavior 默认的属性变换动画

SequentialAnimation 对定义的动画串行播放

ParallelAnimation 对定义的动画并行播放

PropertyAnimation 属性变换动画

NumberAnimation 对实数类型属性进行的动画

Vector3dAnimation 对 QVector3d 进行的属性

ColorAnimation 颜色进行的变换动画

RotationAnimation 对旋转进行的变换动画

ParentAnimation 对父节点进行变换的动画，改变绑定的父节点

AnchorAnimation 对 anchor 进行改变的动画

PauseAnimation 延迟处理

SmoothedAnimation 允许属性平滑的过度

SpringAnimation 一种加速的效果

PropertyAction 允许在动画过程中对属性的直接改变

ScriptAction 允许动画过程中调用脚本

Transition 在状态变换中加入动作变化

工作中的数据

Binding 在创建的时候绑定一些数据到一些属性

ListModel 定义链表数据

ListElement 定义 ListModel 的一个数据项

VisualItemModel 包含可视化项(visual items)到一个 view 中，相当是一个容器

VisualDataModel 包含一个 model 和一个 delegate，model 包含需要的数据，delegate 设计显示的项的信息，具体的去看例子

Package 他的目的是把 VisualDataModel 共享给多个 view，具体还要学习

XmlListModel 特殊的一个模式使用 XPath 表达式，使用 xml 来设置元素，参考例子

XmlRole XmlListModel 的一个特殊的角色

试图

ListView 提供一个链表显示模型试图

GridView 提供一个网格显示模型试图

PathView 提供一个内容沿着路径来显示的模型

Path 定义一个 PathView 使用的轨迹

PathLine 定义一个线性的轨迹

PathQuad 定义一个二次贝塞尔曲线的轨迹

PathCubic 定义一个三次贝塞尔曲线的轨迹

PathAttribute 允许绑定一个属性上，具体看例子

PathPercent 修改 item 分配的轨迹不是很明了其中的意思

WebView 允许添加网页内容到一个 canvas 上

定位器

Column 整理它的子列(纵)

Row 整理它的子行(横)

Grid 设置它的子到一个网格上

Flow 目的是不让他的子项重叠在一起

实用

Connections 明确连接信号和信号句柄

Component 封装 QML items 想一个组件一样

Timer 提供时间触发器

QtObject 基本的元素只包含 objectName 属性

Qt qml 全局 Qt object 提供使用的枚举和函数

WorkerScript 允许在 QML 使用线程

Loader 控制载入 item 或组件

Repeater 使用一个模型创建多个组件

SystemPalette 为 Qt palettes 提供一个通道

FontLoader 载入字体根据名字或 URL

LayoutItem 允许声明 UI 元素插入到 qtGraphicsView 布局中

变换

Scale 分派 item 缩放行为

Rotation 分派 item 旋转行为

Translate 分派 item 移动行为

4.基本元素的使用例子

1. 位置是0,0宽高分别是200

```
Item {
    x: 0; y: 0;
    width: 200; height: 200;
    }
```

2. 位置是：0,0宽高分别是200，颜色是红色

```
Rectangle {
x: 0; y: 0;
```

```
width: 200; height: 200;
color: "red"
}
```

3.分别在总高度的0颜色红色 总高度的1/3黄色 总高度的1是绿色

```
Rectangle{
width:100;height:100
gradient:Gradient{
GradientStop{position:0.0;color:"red"}
GradientStop{position:0.33;color:"yellow"}
GradientStop{position:1.0;color:"green"}
}
}
```

4.设置一张图片

```
Image {
source: "../Images/button1.png"
}
```

5.他将一张图片分成9部分

当图片进行缩放的时候

A. 1 3 7 9位置的都不会进行缩放

B. 2 8将根据属性 horzontalTileMode 进行缩放

C. 4 6将根据属性 verticalTileMode 进行缩放

D. 5将根据属性 horzontalTileMode 和 verticalTileMode 进行缩放

```
BorderImage {
width: 180; height: 180
// 分割1~9块的4个点是根据 border 设置的坐标来实现的
// 本别是距离坐标 上边 右边 下边的距离
border { left: 30; top: 30; right: 30; bottom: 30 }
horizontalTileMode: BorderImage.Stretch
```

```
verticalTileMode: BorderImage.Stretch
source: "../Images/button1.png"
}
```

6.主要用于播放 gif 图片

```
Rectangle {
width: animation.width; height: animation.height + 8
AnimatedImage { id: animation; source: "animation.gif" }
Rectangle {
property int frames: animation.frameCount
width: 4; height: 8
x: (animation.width - width) * animation.currentFrame / frames
y: animation.height
color: "red"
    }
}
```

7.显示文本(具体的其他设置请看文档)

```
Text{
text:"text"
}
```

8.下面是设置一个输入文本框，框中的字符串是 Text,并设置鼠标可以选择文本

```
TextInput{
text:"Text"
selectByMouse:true;//鼠标可以选择
}
```

9. int 型验证器，和输入框结合后就是只能输入整型数据

```
TextInput{
```

```
//最高可以输入100，最低输入10
IntValidator{id:intval;bottom:10;top:100;}
width:100;height:20;
text:"";
//使用校验器
validator:intval;
}
```

10.只能输入浮点数

```
TextInput{
//最高可以输入100，最低输入10decimals 最多有多少位小数
//notation 表示是使用科学计数法还是(默认),还是直接的小数当获取里面的数据
DoubleValidator{id:intval;decimals:4;bottom:10;top:100;notation:DoubleValidator.StandardNotation}
width:100;height:20;
text:"";
//使用校验器
validator:intval;
}
```

11.使用正则表达式

```
TextInput{
//使用一个正则表达式来控制输入的字符串
///^[a-zA-Z]{1}[0-1]{0,2}[a-z]{1,3}$/表示开始位置必须是一个大写或小写字母
//接下来是0~2个的数字而且是0或1，在接下来是1~3个的小写字母
RegExpValidator{id:intval;regExp:/^[a-zA-Z]{1}[0-1]{0,2}[a-z]{1,3}$/;}
width:100;height:20;
text:"";
```

```
//使用校验器
validator:intval;
        }
```

12.

显示一段 hello world 的 html 文本和相同

```
TextEdit {
width: 240
text: "<b>Hello</b> <i>World!</i>"
font.family: "Helvetica"
font.pointSize: 20
color: "blue"
focus: true
}
```

13.

主要是用来判断鼠标事件的区域

```
Rectangle{
x: 0; y: 0;
width: 100; height:100;
Rectangle{
id: mousrect
x: 20; y: 20;
width: 20; height: 20;
```

```
color: "blue"
MouseArea{
// 使用父的区域作为鼠标判断的区域及 x: 20; y: 20; width: 20; height: 20;
anchors.fill: parent;
// 但鼠标按下后 mousrect 变成红色，当鼠标松开后变成蓝色
onPressed: { mousrect.color = "red";}
onReleased: { mousrect.color = "blue";}
}
}
}
```

14.

不是很清楚说的什么，好像是说同一个时刻只有一个 item 有焦点

15.

显示一个200x200的框，框中显示图片上200x200的部分

```
Flickable {
width: 200; height: 200
// 设置使用图片的宽 高，而现实的是 200x200的现实框
contentWidth: image.width; contentHeight: image.height
Image { id: image; source: "../Images/need.png" }
}
```

16.

包含两个面，一个前面，一个后面，实现一个控件前后的翻转效果，并且在后面

可以添加一些控制

```
Flipable {
id: flipable
width: 240
height: 240
property int angle: 0
property bool flipped: false
front: Image { source: "front.png" } //  前面
back: Image { source: "back.png" } //  后面
//  旋转动画  前后面交换
transform: Rotation {
origin.x: flipable.width/2; origin.y: flipable.height/2
axis.x: 0; axis.y: 1; axis.z: 0 // rotate around y-axis
angle: flipable.angle
}
states: State {
name: "back"
PropertyChanges { target: flipable; angle: 180 }
when: flipable.flipped
}
transitions: Transition {
NumberAnimation { properties: "angle"; duration: 1000 }
}
MouseArea {
anchors.fill: parent
onClicked: flipable.flipped = !flipable.flipped
```

```
}
}
```

17.

```
// 当鼠标按下后改变 myRect 的颜色
Rectangle {
id: myRect
width: 100; height: 100
color: "black"
MouseArea {
id: mouseArea
anchors.fill: parent
onClicked: myRect.state == 'clicked' ? myRect.state = "" : myRect.state =
'clicked';
}
// 设置状态
states: [
State {
name: "clicked"
PropertyChanges { target: myRect; color: "red" }
}
```

```
]
}
```

18.

```
// 当鼠标按下后改变状态
// 状态里面的属性改变包含了文本和颜色的改变
Text {
id: myText
width: 100; height: 100
text: "Hello"
color: "blue"
states: State {
name: "myState"
// 当这个状态被设置的时候，将改变 myText 的文本和颜色
PropertyChanges {
target: myText
text: "Goodbye"
color: "red"
}
}
MouseArea { anchors.fill: parent; onClicked: myText.state = 'myState' }
}
```

19.

一个状态组中可以包含很多的状态和变化，而状态也可以和变换绑定.

20.

在状态中可以对脚本中的函数进行调用

```
// Sc.js
function changeColor() //返回蓝色
{
return "blue";
}

// test.qml
import "Sc.js" as Code
Rectangle{
id: rect
```

```
width: 50; height: 50
color: "red"
MouseArea {
anchors.fill: parent
onClicked: rect.state = "first" //鼠标按下改变状态
}
states: State {name: "first";
StateChangeScript{
name: "myScript";
script: rect.color = Code.changeColor();
}
}
}
```

21.

把指定的 item 换一个 item 父节点

```
Item {
width: 200; height: 100
Rectangle {
id: redRect
width: 100; height: 100
color: "red"
}
```

```
// 本来 blueRect 的父节点是 Item 当鼠标按下后他被设置到 redRect 上
Rectangle {
id: blueRect
x: redRect.width
width: 50; height: 50
color: "blue"
states: State {
name: "reparented"
// 改变父节点
ParentChange { target: blueRect; parent: redRect; x: 10; y: 10 }
}
MouseArea { anchors.fill: parent; onClicked: blueRect.state = "reparented" }
}
}
```

22.

```
Rectangle {
id: window
width: 120; height: 120
color: "black"
Rectangle { id: myRect; width: 50; height: 50; color: "red" }
states: State {
name: "reanchored"
```

```
AnchorChanges { // 改变 myRect 的 anchors 属性
target: myRect
anchors.top: window.top
anchors.bottom: window.bottom
}
PropertyChanges {
target: myRect
anchors.topMargin: 10
anchors.bottomMargin: 10
}
}
// 鼠标事件
MouseArea { anchors.fill: parent; onClicked: window.state = "reanchored" }
}
```

23.

```
Rectangle {
id: rect
width: 100; height: 100
color: "red"
// 针对宽度的动画
Behavior on width {
NumberAnimation { duration: 1000 }
```

```
}
MouseArea {
anchors.fill: parent
onClicked: rect.width = 50
}
}
```

24.

串行播放多个动画

```
Rectangle {
id: rect1
width: 500; height: 500
Rectangle{
id: rect;
color: "red"
width: 100; height: 100
// 串行播放多个动画，先横向移动，在纵向移动
SequentialAnimation{
running: true;
NumberAnimation {target:rect; properties:"x"; to: 50; duration: 1000 }
NumberAnimation {target:rect; properties:"y"; to: 50; duration: 1000 }
}
}
}
```

25.

```
Rectangle {
id: rect1
width: 500; height: 500
Rectangle{
id: rect;
color: "red"
width: 100; height: 100
// 并行播放动画，同时横向和纵向移动
ParallelAnimation{
running: true;
NumberAnimation {target:rect; properties:"x"; to: 50; duration: 1000 }
NumberAnimation {target:rect; properties:"y"; to: 50; duration: 1000 }
}
}
}
```

26.

```
Rectangle {
id: rect
```

```
width: 100; height: 100
color: "red"
states: State {
name: "moved"
PropertyChanges { target: rect; x: 50 }
}
transitions: Transition {
// 属性动画 这里是当属性 x 或 y 发生变化的时候，就播放这样一个动画
PropertyAnimation { properties: "x,y"; easing.type: Easing.InOutQuad }
}
MouseArea{
anchors.fill: parent;
onClicked: rect.state = "moved";
}
}
```

27.

```
Rectangle {
width: 100; height: 100
color: "red"
// 对当前 item 的 x 进行移动，目标移动到 x = 50
NumberAnimation on x { to: 50; duration: 1000 }
}
```

28.

29.

颜色的过度

```
Rectangle {
width: 100; height: 100
color: "red"
ColorAnimation on color { to: "yellow"; duration: 1000 }
}
```

30.

默认是绕 z 轴进行的旋转

```
Item {
width: 300; height: 300
Rectangle {
id: rect
width: 150; height: 100; anchors.centerIn: parent
color: "red"
smooth: true
states: State {
name: "rotated"; PropertyChanges { target: rect; rotation: 180 }
}
transitions: Transition {
RotationAnimation { duration: 1000; direction:
RotationAnimation.Counterclockwise }
}
}
MouseArea { anchors.fill: parent; onClicked: rect.state = "rotated" }
}
```

31.

一个切换父节点的动画,平滑的过度

```
Item{
width:200;height:100
Rectangle{
```

```
id:redRect
width:100;height:100
color:"red"
}
Rectangle{
id:blueRect
x:redRect.width
width:50;height:50
color:"blue"
states:State{
name:"reparented"
ParentChange{target:blueRect;parent:redRect;x:10;y:10}
}
transitions:Transition{
ParentAnimation{
NumberAnimation{properties:"x,y";duration:1000}
}
}
MouseArea{anchors.fill:parent;onClicked:blueRect.state="reparented"}
}
}
```

32.

```
Item{
id:container
```

```
width:200;height:200
Rectangle{
id:myRect
width:100;height:100
color:"red"
}
states:State{
name:"reanchored"
AnchorChanges{target:myRect;anchors.right:container.right}
}
transitions:Transition{
//smoothlyreanchormyRectandmoveintonewposition
AnchorAnimation{duration:1000}
}
//当控件加载完成后
Component.onCompleted:container.state="reanchored"
}
```

33.

延迟效果

```
Item {
id: container
width: 200; height: 200
Rectangle {
id: myRect
```

```
width: 100; height: 100
color: "red"
SequentialAnimation {
running: true;
NumberAnimation {target: myRect;to: 50; duration: 1000; properties: "x"; }
PauseAnimation { duration: 5000 } //  延迟100毫秒
NumberAnimation {target: myRect; to: 50; duration: 1000; properties: "y"; }
}
}
}
```

34.

平滑过度

```
Rectangle {
width: 800; height: 600
color: "blue"
Rectangle {
width: 60; height: 60
x: rect1.x - 5; y: rect1.y - 5
color: "green"
Behavior on x { SmoothedAnimation { velocity: 200 } }
Behavior on y { SmoothedAnimation { velocity: 200 } }
}
```

```
Rectangle {
id: rect1
width: 50; height: 50
color: "red"
}
focus: true
Keys.onRightPressed: rect1.x = rect1.x + 100
Keys.onLeftPressed: rect1.x = rect1.x - 100
Keys.onUpPressed: rect1.y = rect1.y - 100
Keys.onDownPressed: rect1.y = rect1.y + 100
}
```

35.

平滑的过度过程，在动画结束的时候有种弹性的效果

```
Item {
width: 300; height: 300
Rectangle {
id: rect
width: 50; height: 50
color: "red"
Behavior on x { SpringAnimation { spring: 2; damping: 0.2 } }
```

```
Behavior on y { SpringAnimation { spring: 2; damping: 0.2 } }
}
MouseArea {
anchors.fill: parent
onClicked: {
rect.x = mouse.x - rect.width/2
rect.y = mouse.y - rect.height/2
}
}
}
```

36.

主要是在动画过程中直接的改变一个属性

```
transitions:Transition{
...
PropertyAction{target:theImage;property:"smooth";value:true}
...
}
```

38.

在动画过程中嵌入脚本的调用

```
SequentialAnimation {
NumberAnimation { ... }
ScriptAction { script: doSomething(); }
NumberAnimation { ... }
}
```

39.

```
Rectangle {
id: rect
width: 100; height: 100
color: "red"
MouseArea {
id: mouseArea
anchors.fill: parent
}
states: State {
name: "moved"; when: mouseArea.pressed
PropertyChanges { target: rect; x: 50; y: 50 }
}
transitions: Transition {
NumberAnimation { properties: "x,y"; easing.type: Easing.InOutQuad }
}
```

```
}
```

40.

```
Item {
width: 300; height: 300
Text {id: app; text: "xxxfa"}
TextEdit { id: myTextField; text: "Please type here..." }
// 把 myTextField 和 app 的 enteredText 属性进行绑定
Binding { target: app; property: "enteredText"; value: myTextField.text }
}
```

41.

```
直接看效果
Rectangle{
width:200;height:200
ListModel{
id:fruitModel
ListElement{
```

```
name:"Apple"
cost:2.45
}
ListElement{
name:"Orange"
cost:3.25
}
ListElement{
name:"Banana"
cost:1.95
}
}
Component{
id:fruitDelegate
Row{
spacing:10
Text{text:name}
Text{text:'$'+cost}
}
}
ListView{
anchors.fill:parent
model:fruitModel
delegate:fruitDelegate
}
}
```

42.

请参照

43.

把可视化图元添加到链表试图

```
Rectangle {
width: 100; height: 100;
VisualItemModel {
id: itemModel
Rectangle { height: 30; width: 80; color: "red" }
Rectangle { height: 30; width: 80; color: "green" }
Rectangle { height: 30; width: 80; color: "blue" }
}
ListView {
anchors.fill: parent
model: itemModel
}
}
```

44.

看下面效果

```
Rectangle {
width: 200; height: 100
VisualDataModel {
id: visualModel
model: ListModel {
ListElement { name: "Apple" }
ListElement { name: "Orange" }
}
delegate: Rectangle {
height: 25
width: 100
Text { text: "Name: " + name}
}
}
ListView {
anchors.fill: parent
model: visualModel
}
}
```

45.

具体请参考

declarative/modelviews/package

46.

从网络获取 xml，暂时没有测试成功

47.

参考

48.

看效果

Rectangle {

```
width: 200; height: 400;
ListModel {
id: fruitModel
ListElement {
name: "Apple"
cost: 2.45
}
ListElement {
name: "Orange"
cost: 3.25
}
ListElement {
name: "Banana"
cost: 1.95
}
}
GridView {
anchors.fill: parent
model: fruitModel
delegate: Column{
Text {text:"name" + name}
Text {text:"cost"+ cost}
}
}
}
```

49.

看例子

```
Rectangle{
width:200;height:400;
ListModel{
id:fruitModel
ListElement{
name:"Apple"
cost:2.45
}
ListElement{
name:"Orange"
cost:3.25
}
ListElement{
name:"Banana"
cost:1.95
}
}
PathView{
anchors.fill:parent
```

```
model:fruitModel
delegate:Column{
Text{text:"name"+name}
Text{text:"cost"+cost}
}
path:Path{
startX:120;startY:100
PathQuad{x:120;y:25;controlX:260;controlY:75}
PathQuad{x:120;y:100;controlX:-20;controlY:75}
}
}
}
```

50.

具体的看运行的例子

```
Rectangle{
width:200;height:400;
ListModel{
id:fruitModel
```

```
ListElement{
name:"Apple"
cost:2.45
}
ListElement{
name:"Orange"
cost:3.25
}
ListElement{
name:"Banana"
cost:1.95
}
}
PathView{
anchors.fill:parent
model:fruitModel
delegate:Column{
Text{text:"name"+name}
Text{text:"cost"+cost}
}
path:Path{
startX:150;startY:120
PathLine{x:200;y:80;}
PathLine{x:100;y:80;}
PathLine{x:150;y:120;}
}
}
}
```

51.

参考

52.

还要看

53.

可以直接针对一些属性进行改变

```
Rectangle{
width:200;height:400;
ListModel{
id:fruitModel
ListElement{
name:"Apple"
cost:2.45
}
```

```
ListElement{
name:"Orange"
cost:3.25
}
ListElement{
name:"Banana"
cost:1.95
}
}
PathView{
anchors.fill:parent
model:fruitModel
delegate:
Item{
id:delitem;
width:80;height:80;
Column{
//这里使用图片试试
Rectangle{
width:40;height:40;
scale:delitem.scale;
color:"red"
}
Text{text:"name"+name}
Text{text:"cost"+cost}
}
}
//
```

```
path:Path{
startX:120;startY:100
PathAttribute{name:"Scale";value:1.0}
PathQuad{x:120;y:25;controlX:260;controlY:75}
PathAttribute{name:"Scale";value:0.3}
PathQuad{x:120;y:100;controlX:-20;controlY:75}
}
}
}
```

54.

具体请看 QML 文档

55.

```
import QtWebKit 1.0
WebView {
url: "http://www.nokia.com"
preferredWidth: 490
preferredHeight: 400
scale: 0.5
```

```
smooth: false
}
```

56

横向排列

```
Rectangle{
width: 100; height: 100;
// 纵向排列
Column {
spacing: 2
Rectangle { color: "red"; width: 50; height: 50 }
Rectangle { color: "green"; width: 20; height: 50 }
Rectangle { color: "blue"; width: 50; height: 20 }
}
}
```

57

```
Rectangle{
width: 100; height: 100;
// 横向排列
```

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

58

```
Rectangle{
width:100;height:100;
//网格排列
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}
}
}
```

59

```
Rectangle{
width: 100; height: 100;
// 网格排列
Flow {
spacing: 2
width: 100; height: 100;
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 }
}
}
```

60

下面是3中情况下会使用的，具体的不好翻译

Multiple connections to the same signal are required

有多个连接要连接到相同的信号时

Creating connections outside the scope of the signal sender

创建的连接在范围之外

Connecting to targets not defined in QML

创建的连接没有在 QML 中定义的

```
Rectangle{
width: 100; height: 100;
MouseArea {
id: area
anchors.fill: parent;
}
Connections {
target: area
onClicked: { console.log(" ok");}
}
}
```

61

组件是可以重用的 QML 元素，具体还是看 QML 的文档翻译不是很好

```
Item {
width: 100; height: 100
// 定义一个组件他包含一个10x10的红色矩形
Component {
```

```
id: redSquare
Rectangle {
color: "red"
width: 10
height: 10
}
}
// 动态的载入一个组件
Loader { sourceComponent: redSquare }
Loader { sourceComponent: redSquare; x: 20 }
}
```



```
Item {
width: 200; height: 40;
// 和 QTimer 差不多
Timer {
interval: 500; running: true; repeat: true
onTriggered: time.text = Date().toString() // 使用 javascript 获取系统时间
}
Text { id: time }
}
```

63

他是不可见的只有 objectName 一个属性

通过这个属性我们可以在 c++中找到我们想要的对象

```
//MyRect.qml
importQt4.7
Item{
width:200;height:200
Rectangle{
anchors.fill:parent
color:"red"
objectName:"myRect"
}
}

//main.cpp
QDeclarativeViewview;
view.setSource(QUrl::fromLocalFile("MyRect.qml"));
view.show();
QDeclarativeItem*item=view.rootObject()->findChild<QDeclarativeItem*>("my
Rect");
```

```
if(item)
item->setProperty("color",QColor(Qt::yellow));
```

64

提供全局有用的函数和枚举，具体的看 QML 文档

65.

使用它可以把操作放到一个新的线程中，使得它在后台运行而不影响主 GUI

具体可以看 QML 中它的文档

66.

可以参考

还有 QML 中的文档

67

他可以创建很多相似的组件，QML 中还有几个例子

```
Row {
Repeater {
model: 3
Rectangle {
width: 100; height: 40
border.width: 1
color: "yellow"
}
}
}
```

68

具体看效果和文档

```
Rectangle{
SystemPalette{id:myPalette;colorGroup:SystemPalette.Active}
```

```
width:640;height:480
color:myPalette.window
Text{
anchors.fill:parent
text:"Hello!";color:myPalette.windowText
}
}
```

69.

```
载入一种字体，可以是网络上的，也可以是本地的
Column {
FontLoader { id: fixedFont; name: "Courier" }
FontLoader { id: webFont; source: "http://www.mysite.com/myfont.ttf" }
Text { text: "Fixed-size font"; font.family: fixedFont.name }
Text { text: "Fancy font"; font.family: webFont.name }
}
```

70

不清楚

71

对缩放的控制

```
Rectangle {
width: 100; height: 100
color: "blue"
Rectangle{
x: 50; y: 50;
width: 20; height: 20;
color: "red"
// 这里是在当前矩形的中间位置沿 x 轴进行3倍缩放
transform: Scale { origin.x: 10; origin.y: 10; xScale: 3}
}
}
```

72

```
Rectangle {
width: 100; height: 100
color: "blue"
// 绕位置25,25 旋转45度
transform: Rotation { origin.x: 25; origin.y: 25; angle: 45}
```

```
}
```

73

```
Row {
Rectangle {
width: 100; height: 100
color: "blue"
// 沿 y 轴正方向移动20个像素
transform: Translate { y: 20 }
}
Rectangle {
width: 100; height: 100
color: "red"
// 沿 y 轴负方向移动20个像素
transform: Translate { y: -20 }
}
}
```