第 7 章 组件与动态对象

6.1.3 节介绍自定义信号时，举了一个简单的例子，定义了一个颜色选择组件，当用户在组件内点击鼠标时，该组件会发出一个携带颜色值的信号，当时我使用 Connections 对象连接到组件的 colorPicked 信号，改变文本的颜色。当时用到的 Component、Loader 两个特性，一直没来得及介绍，可能很多人都还在雾里看花呢。这次呢，我们就来仔仔细细地把它们讲清楚。

7.1. Component（组件）

Component 是由 Qt 框架或开发者封装好的、只暴露了必要接口的 QML 类型，可以重复利用。一个 QML 组件就像一个黑盒子，它通过属性、信号、函数和外部世界交互。

一个 Component 既可以定义在独立的 QML 文件中，也可以嵌入到其他的 QML 文档中来定义。通常我们可以根据这个原则来选择将一个 Component 定义在哪里：如果一个 Component 比较小且只在某个 QML 文档中使用，或者一个 Component 从逻辑上看从属于某个 QML 文档，那么就可以采用嵌入的方式来定义该 Component。你也可以与 C++的嵌套类对比来理解。

7.1.1 嵌入式定义组件

6.1.3 节使用到 Component 的示例中的颜色选择组件代码如下：

```
Compo    nent {
    id: colorComponent;
    Rectangle {
        id: colorPicker;
        width: 50;
        height: 30;
        signal colorPicked(color clr);
        MouseArea {
            anchors.fill: parent
            onPressed: colorPicker.colorPicked(colorPicker.color);
        }
    }
}
```

如你所见，要在一个 QML 文档中嵌入 Component 的定义，需要使用 Component 对象。

定义一个 Component 与定义一个 QML 文档类似，Component 只能包含一个顶层 Item，而且在这个 Item 之外不能定义任何数据，除了 id。比如上面的代码中，顶层 Item 是 Rectangle 对象，在 Rectangle 之外我定义了 id 属性，其值为 colorComponent。而在顶层 Item 之内，则可以包含更多的子元素来协同工作，最终形成一个具有特定功能的组件。

Component 通常用来给一个 view 提供图形化组件，比如 ListView::delegate 属性就需要一个 Component 来指定如何显示列表的每一个项，又比如 ButtonStyle::background 属性也需要一个 Component 来指定如何绘制 Button 的背景。

Component 不是 Item 的派生类，而是从 QQmlComponent 继承而来的，虽然它通过自己的顶层 Item 为其他的 view 提供可视化组件，但它本身是不可见元素。你可以这么理解：你定义的组件是一个新的类型，它必须被实例化以后才可能显示。而要实例化一个嵌入在 QML 文档中定义的组件，则可以通过 Loader。后面我们详细讲述 Loader，这里先按下不表，我们要来看如何在一个文件中定义组件了。

7.1.2 在单独文件中定义组件

很多时候我们把一个 Component 单独定义在一个 QML 文档中，比如 Qt Quick 提供的 BusyIndicator 控件，其实就是在 BusyIndicator.qml 中定义的一个组件。下面是 BusyIndicator.qml 文件的内容：

```
Control {
    id: indicator

    property bool running: true

    Accessible.role: Accessible.Indicator
    Accessible.name: "busy"

    style: Qt.createComponent(Settings.style +
                "/BusyIndicatorStyle.qml", indicator)
}
```

第 4 章“Qt Quick 入门”中的显示网络图片实例（4.3.10 节），使用了 BusyIndicator 来提示用户图片正在加载中需要等候，你可以跳转到相关章节学习 BusyIndicator 的用法。

BusyIndicator 组件的代码非常简单，只是给 Control 元素（Qt Quick 定义的私有元素，用作其他控件的基类，如 ComboBox、BusyIndicator 等）增加了一个属性，设置了几个属性的值，仅此而已。

不知你是否注意到了，BusyIndicator.qml 文件中的顶层 Item 是 Control，而我们使用时却是以 BusyIndicator 为组件名（类名）。这是我们定义 Component 时要遵守的一个约定：组件名必须和 QML 文件名一致。好嘛，和 Java 一样啦，类名就是文件名。还有一点，组件名的第一个字母必须是大写的。对于在文件中定义一个组件，就这么简单，再没有其他的特殊要求了。Qt Quick 提供的多数基本元素和特性，你都可以在定义组件时使用。

一旦你在文件中定义了一个组件，就可以像使用标准的 Qt Quick 元素一样使用你的组件。比如我们给颜色选择组件起个名字叫 ColorPicker，对应的 QML 文件为 ColorPicker.qml，

那么你就可以在其他 QML 文档中使用 ColorPicker {...} 来定义 ColorPicker 的实例。

好了，现在让我们来看看在单独文件中定义的 ColorPicker 组件：

```
import QtQuick 2.2

Rectangle {
    id: colorPicker;
    width: 50;
    height: 30;
    signal colorPicked(color clr);

    function configureBorder(){
        colorPicker.border.width = colorPicker.focus ? 2 : 0;
        colorPicker.border.color = colorPicker.focus ?
                                        "#90D750" : "#808080";
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            colorPicker.colorPicked(colorPicker.color);
            mouse.accepted = true;
            colorPicker.focus = true;
        }
    }
    Keys.onReturnPressed: {
        colorPicker.colorPicked(colorPicker.color);
        event.accepted = true;
    }
    Keys.onSpacePressed: {
        colorPicker.colorPicked(colorPicker.color);
        event.accepted = true;
    }

    onFocusChanged: {
        configureBorder();
    }

    Component.onCompleted: {
        configureBorder();
    }
}
```

请注意上面的代码，它和嵌入式定义有明显的不同：Component 对象不见了！对，就是这样子的：在单独文件内定义组件，不需要 Component 对象，只有在其他 QML 文档中嵌入式定义组件时才需要 Component 对象。另外，为了能够让多个 ColorPicker 组件可以正常地显示焦点框，我还使用了 onClicked 信号处理器，新增了 onFocusChanged 信号处理器，在它们的实现中调用 configureBorder()函数来重新设置边框的宽度和颜色，新增 onReturnPressed 和 onSpacePressed 以便响应回车和空格两个按键。

你可以使用 Item 或其派生类作为组件的根 Item。ColorPicker 组件使用 Rectangle 作为根 Item。现在让我们看看如何在其他文件中使用新定义的 ColorPicker 组件。我修改了上节的示例，新的 QML 文件被我命名为 component_file.qml，内容如下：

```
import QtQuick 2.2

Rectangle {
    width: 320;
    height: 240;
    color: "#EEEEEE";
```

```
    Text {
        id: coloredText;
        anchors.horizontalCenter: parent.horizontalCenter;
        anchors.top: parent.top;
        anchors.topMargin: 4;
        text: "Hello World!";
        font.pixelSize: 32;
    }

    function setTextColor(clr){
        coloredText.color = clr;
    }

    ColorPicker {
        id: redColor;
        color: "red";
        focus: true;
        anchors.left: parent.left;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;

        KeyNavigation.right: blueColor;
        KeyNavigation.tab: blueColor;
        onColorPicked:{
            coloredText.color = clr;
        }
    }

    ColorPicker {
        id: blueColor;
        color: "blue";
        anchors.left: redColor.right;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;

        KeyNavigation.left: redColor;
        KeyNavigation.right: pinkColor;
        KeyNavigation.tab: pinkColor;
    }

    ColorPicker {
        id: pinkColor;
        color: "pink";
        anchors.left: blueColor.right;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;

        KeyNavigation.left: blueColor;
        KeyNavigation.tab: redColor;
    }

    Component.onCompleted:{
        blueColor.colorPicked.connect(setTextColor);
        pinkColor.colorPicked.connect(setTextColor);
    }
}
```

可以看到，component_file.qml 使用 ColorPicker 组件的方式与使用 Rectangle、Button、Text 等标准的 Qt Quick 组件完全一致：可以给组件指定唯一的 id，可以使用锚布局，可以使用 KeyNavigation 附加属性……总之，自定义的组件和 Qt Quick 组件并无本质不同。不过需要注意的是，组件实例的 id 和组成组件的顶层 Item 的 id 是各自独立的。从上面的例子来看，

redColor 和 colorPicker 是两个不同的 id，前者指代组件对象（虽然组件的定义没有使用 Component），后者指代 ColorPicker 的 Rectangle 对象。

上面的代码还演示了两种使用 QML 自定义信号的方式，redColor 使用了信号处理器，blueColor 和 pinkColor 则使用了 signal 对象的 connect()方法连接到 setTextColor()方法上。

我把 ColorPicker.qml 和 component_file.qml 放在同一个文件夹下面，否则可能会报错。图 7-1 是运行“qmlscene component_file.qml”命令的效果图。

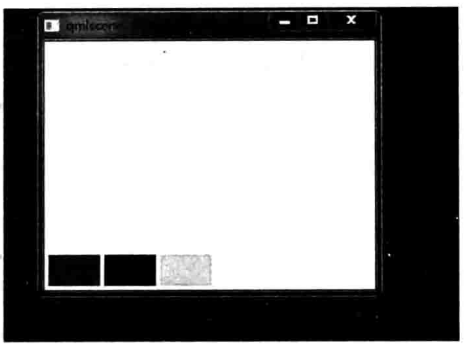

图 7-1　在文件中定义组件并使用

对于定义在单独文件中的 Component，除了可以像刚刚介绍的那样使用之外，也可以使用 Loader 来动态加载，根据需要再创建对象。下面我们就来看看 Loader 究竟是何方妖怪。

7.2　使用 Loader

Loader 用来动态加载 QML 组件。我们可以把 Loader 作为占位符使用，在需要显示某个元素时，才使用 Loader 把它加载进来。

7.2.1　Loader 详细介绍

Loader 可以使用其 source 属性加载一个 QML 文档，也可以通过其 sourceComponent 属性加载一个 Component 对象。当你需要延迟一些对象直到真正需要才创建它们时，Loader 非常有用。当 Loader 的 source 或 sourceComponent 属性发生变化时，它之前加载的 Component 会自动销毁，新对象会被加载。将 source 设置为一个空字符串或将 sourceComponent 设置为 undefined，将会销毁当前加载的对象，相关的资源也会被释放，而 Loader 对象则变成一个空对象。

Loader 的 item 属性指向它加载的组件的顶层 Item，比如 Loader 加载了颜色选择组件，其 item 属性就指向颜色选择组件的 Rectangle 对象。对于 Loader 加载的 Item，它暴露出来的接口，如属性、信号等，都可以通过 Loader 的 item 属性来访问。所以，我们才可以这么使用：

```
Loader{
    id: redLoader;
    anchors.left: parent.left;
    anchors.leftMargin: 4;
```

```
    anchors.bottom: parent.bottom;
    anchors.bottomMargin: 4;
    sourceComponent: colorComponent;
    onLoaded:{
        item.color = "red";
    }
}
```

上面的代码在 Loader 对象中使用 sourceComponent 属性来加载 id 为 colorComponent 的组件对象，然后在 onLoaded 信号处理器中使用 item 属性来设置颜色选择组件的颜色。对于信号的访问，我们则可以使用 Connections 对象，如下面的 QML 代码所示：

```
Connections {
    target: redLoader.item;
    onColorPicked:{
        coloredText.color = clr;
    }
}
```

我们创建的 Connections 对象，其 target 指向 redLoader.item，即指向颜色选择组件的顶层 Item——Rectangle，所以可以直接响应它的 colorPicked 信号。

虽然 Loader 本身是 Item 的派生类，但没有加载 Component 的 Loader 对象是不可见的，没什么实际的意义，只是个占位符号，而一旦你加载了一个 Component，Loader 的大小、位置等属性却可以影响它所加载的 Component。如果你没有显式指定 Loader 的大小，那么 Loader 会将自己的尺寸调整为与它加载的可见 Item 的尺寸一致；如果 Loader 的大小通过 width、height 或锚布局显式设置了，那么它加载的可见 Item 的尺寸会被调整以便适应 Loader 的大小。不管是哪种情况，Loader 和它所加载的 Item 具有相同的尺寸，这确保你使用锚来布局 Loader 就等同于布局它加载的 Item。

我们改变一下颜色选择器示例的代码，两个 Loader 对象，一个设置尺寸，一个不设置，看看是什么效果。新的 QML 文档我们命名为 loader_test.qml，内容如下：

```
import QtQuick 2.2

Rectangle {
    width: 320;
    height: 240;
    color: "#C0C0C0";

    Text {
        id: coloredText;
        anchors.horizontalCenter: parent.horizontalCenter;
        anchors.top: parent.top;
        anchors.topMargin: 4;
        text: "Hello World!";
        font.pixelSize: 32;
    }

    Component {
        id: colorComponent;
        Rectangle {
            id: colorPicker;
            width: 50;
            height: 30;
            signal colorPicked(color clr);
            MouseArea {
                anchors.fill: parent
                onPressed: colorPicker.colorPicked(colorPicker.color);
```

```
            }
        }
    }

    Loader{
        id: redLoader;
        width: 80; //[1]
        height: 60; //[2]
        anchors.left: parent.left;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        sourceComponent: colorComponent;
        onLoaded:{
            item.color = "red";
        }
    }

    Loader{
        id: blueLoader;
        anchors.left: redLoader.right;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        sourceComponent: colorComponent;
        onLoaded:{
            item.color = "blue";
        }
    }

    Connections {
        target: redLoader.item;
        onColorPicked:{
            coloredText.color = clr;
        }
    }

    Connections {
        target: blueLoader.item;
        onColorPicked:{
            coloredText.color = clr;
        }
    }
}
```

注意上面代码中的注释，方括号标注的两处修改，设置了红色 Loader 的尺寸，效果如图 7-2 所示。

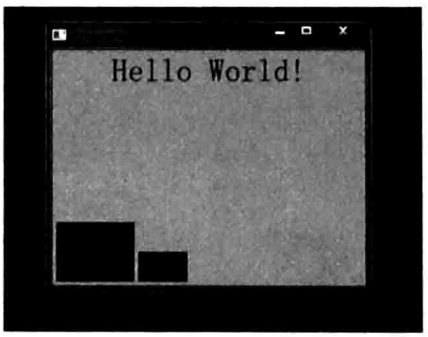

图 7-2　Loader 尺寸

如果 Loader 加载的 Item 想处理按键事件，那么必须将 Loader 对象的 focus 属性设置为

true。又因为 Loader 本身也是一个焦点敏感的对象，所以如果它加载的 Item 处理了按键事件，则应当将事件的 accepted 属性设置为 true，以免已经被吃掉的事件再传递给 Loader。我们来修改 loader_test.qml，加入对焦点的处理，当一个颜色组件拥有焦点时，绘制一个边框，此时如果你按下回车键或空格键，会触发其 colorPicked 信号。同时我们也处理左、右键，在不同的颜色选择组件之间切换焦点。将新代码命名为 loader_focus.qml，内容如下：

```
import QtQuick 2.2

Rectangle {
    width: 320;
    height: 240;
    color: "#EEEEEE";

    Text {
        id: coloredText;
        anchors.horizontalCenter: parent.horizontalCenter;
        anchors.top: parent.top;
        anchors.topMargin: 4;
        text: "Hello World!";
        font.pixelSize: 32;
    }

    Component {
        id: colorComponent;
        Rectangle {
            id: colorPicker;
            width: 50;
            height: 30;
            signal colorPicked(color clr);
            property Item loader;
            border.width: focus ? 2 : 0;
            border.color: focus ? "#90D750" : "#808080";
            MouseArea {
                anchors.fill: parent
                onClicked: {
                    colorPicker.colorPicked(colorPicker.color);
                    loader.focus = true;
                }
            }
            Keys.onReturnPressed: {
                colorPicker.colorPicked(colorPicker.color);
                event.accepted = true;
            }
            Keys.onSpacePressed: {
                colorPicker.colorPicked(colorPicker.color);
                event.accepted = true;
            }
        }
    }

    Loader{
        id: redLoader;
        width: 80;
        height: 60;
        focus: true;
        anchors.left: parent.left;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        sourceComponent: colorComponent;
        KeyNavigation.right: blueLoader;
        KeyNavigation.tab: blueLoader;
```

```
        onLoaded:{
            item.color = "red";
            item.focus = true;
            item.loader = redLoader;
        }
        onFocusChanged:{
            item.focus = focus;
        }
    }

    Loader{
        id: blueLoader;
        anchors.left: redLoader.right;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        sourceComponent: colorComponent;
        KeyNavigation.left: redLoader;
        KeyNavigation.tab: redLoader;

        onLoaded:{
            item.color = "blue";
            item.loader = blueLoader;
        }
        onFocusChanged:{
            item.focus = focus;
        }
    }

    Connections {
        target: redLoader.item;
        onColorPicked:{
            coloredText.color = clr;
        }
    }

    Connections {
        target: blueLoader.item;
        onColorPicked:{
            coloredText.color = clr;
        }
    }
}
```

首先我让颜色选择组件处理按键事件（如忘记请参看 6.3 节），收到回车键和空格键时发出 colorPicked 信号。我还给颜色选择组件定义了一个名为 loader 的属性，以便用鼠标点击颜色选择组件时可以改变 Loader 对象的焦点属性。我们在 Loader 的 onLoaded 信号处理器中给颜色选择组件的 loader 属性赋值。

颜色选择组件根据焦点状态决定是否绘制边框，当有焦点时绘制宽度为 2 的边框。

对于 Loader，我设置了 KeyNavigation 附加属性，指定左、右键和 Tab 键如何切换焦点，而当焦点变化时，同步改变颜色选择组件的焦点。最后我设置 redLoader 拥有初始焦点。

图 7-3 是运行效果图。

你可以运行“qmlscene loader_focus.qml”命令看看效果，用鼠标点击某个颜色选择组件，会触发焦点切换和边框变化，左、右键和 Tab 键也会触发焦点变化，而当一个颜色选择组件拥有焦点时，回车键、空格键都可以触发“Hello World!”改变颜色。

图 7-3　Loader 与按键、焦点

7.2.2　从文件加载组件

前面介绍 Loader 时，我们以嵌入式定义的 Component 为例说明了 Loader 的各种特性和用法，现在我们来看看如何从文件加载组件。

对于定义在一个独立文件中的Component，同样可以使用Loader来加载，只要指定Loader的 source 属性即可。现在再来修改一下我们的例子，使用 Loader 来加载 ColorPicker 组件。

新的 QML 文档我命名为 loader_component_file.qml，内容如下：

```
import QtQuick 2.2

Rectangle {
    width: 320;
    height: 240;
    color: "#EEEEEE";

    Text {
        id: coloredText;
        anchors.horizontalCenter: parent.horizontalCenter;
        anchors.top: parent.top;
        anchors.topMargin: 4;
        text: "Hello World!";
        font.pixelSize: 32;
    }

    Loader{
        id: redLoader;
        width: 80;
        height: 60;
        focus: true;
        anchors.left: parent.left;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        source: "ColorPicker.qml";
        KeyNavigation.right: blueLoader;
        KeyNavigation.tab: blueLoader;

        onLoaded:{
            item.color = "red";
            item.focus = true;
        }

        onFocusChanged:{
            item.focus = focus;
        }
```

```
    }

    Loader{
        id: blueLoader;
        anchors.left: redLoader.right;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        source: "ColorPicker.qml";
        KeyNavigation.left: redLoader;
        KeyNavigation.tab: redLoader;

        onLoaded:{
            item.color = "blue";
        }

        onFocusChanged:{
            item.focus = focus;
        }
    }

    Connections {
        target: redLoader.item;
        onColorPicked:{
            coloredText.color = clr;
            if(!redLoader.focus){
                redLoader.focus = true;
                blueLoader.focus = false;
            }
        }
    }

    Connections {
        target: blueLoader.item;
        onColorPicked:{
            coloredText.color = clr;
            if(!blueLoader.focus){
                blueLoader.focus = true;
                redLoader.focus = false;
            }
        }
    }
}
```

代码有几处改动：

- 一处是将 sourceComponent 修改为 source，其值为"ColorPicker.qml"。
- 一处是两个 Connections 对象，在 onColorPicked 信号处理器中，设置了 Loader 的焦点属性，因为只有 Loader 有焦点，它加载的 Item 才会有焦点，如果用鼠标点击某个颜色选择组件而加载它的 Loader 没有焦点，那么虽然颜色可以改变，但是焦点框出不来。

使用 Loader 加载定义在 QML 文档中的组件，比直接使用组件名构造对象要烦琐得多，但如果你的应用需要根据特定的情景来决定某些界面元素是否显示，这种方式也许可以满足你。

7.2.3　利用 Loader 动态创建与销毁组件

现在我们看看如何动态创建、销毁组件。下面是 dynamic_component.qml 的内容：

```
import QtQuick 2.2
import QtQuick.Controls 1.2
```

```
Rectangle {
    width: 320;
    height: 240;
    color: "#EEEEEE";
    id: rootItem;
    property var colorPickerShow : false;

    Text {
        id: coloredText;
        anchors.horizontalCenter: parent.horizontalCenter;
        anchors.top: parent.top;
        anchors.topMargin: 4;
        text: "Hello World!";
        font.pixelSize: 32;
    }

    Button {
        id: ctrlButton;
        text: "Show";
        anchors.left: parent.left;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;

        onClicked:{
           if(rootItem.colorPickerShow){
               redLoader.sourceComponent = undefined;
               blueLoader.source = "";
               rootItem.colorPickerShow = false;
               ctrlButton.text = "Show";
           }else{
               redLoader.source = "ColorPicker.qml";
               redLoader.item.colorPicked.connect(onPickedRed);
               blueLoader.source = "ColorPicker.qml";
               blueLoader.item.colorPicked.connect(onPickedBlue);
               redLoader.focus = true;
               rootItem.colorPickerShow = true;
               ctrlButton.text = "Hide";
           }
        }
    }

    Loader{
        id: redLoader;
        anchors.left: ctrlButton.right;
        anchors.leftMargin: 4;
        anchors.bottom: ctrlButton.bottom;

        KeyNavigation.right: blueLoader;
        KeyNavigation.tab: blueLoader;

        onLoaded:{
            if(item != null){
                item.color = "red";
                item.focus = true;
            }
        }

        onFocusChanged:{
            if(item != null){
                item.focus = focus;
            }
        }
```

```
    }

    Loader{
        id: blueLoader;
        anchors.left: redLoader.right;
        anchors.leftMargin: 4;
        anchors.bottom: redLoader.bottom;

        KeyNavigation.left: redLoader;
        KeyNavigation.tab: redLoader;

        onLoaded:{
            if(item != null){
                item.color = "blue";
            }
        }

        onFocusChanged:{
            if(item != null){
                item.focus = focus;
            }
        }
    }

    function onPickedBlue(clr){
        coloredText.color = clr;
        if(!blueLoader.focus){
          blueLoader.focus = true;
          redLoader.focus = false;
        }
    }

    function onPickedRed(clr){
        coloredText.color = clr;
        if(!redLoader.focus){
          redLoader.focus = true;
          blueLoader.focus = false;
        }
    }
}
```

这次我们在界面上放一个按钮，通过按钮来控制颜色选择组件的创建与销毁。启动应用时没有创建颜色选择组件，如图 7-4 所示。

图 7-4　动态创建组件初始效果

当你点击“Show”按钮时，代码通过设置 redLoader 和 blueLoader 的 source 来创建颜色选择组件，连接颜色组件的 colorPicked 信号到相应的方法，同时将改变按钮文字，也改变

rootItem 维护的颜色组件是否显示的标志位，以便下次再点击按钮时可以正常显示。图 7-5 是颜色选择组件显示后的效果图。

图 7-5　颜色组件创建后的效果

此时再点击“Hide”按钮，我通过将 redLoader 的 sourceComponent 属性设置为 undefined、将 blueLoader 的 source 属性设置为空串来卸载它们。这是卸载 Loader 所加载的组件的两种方式。

使用 Loader 控制组件的动态创建与销毁，只是 Qt Quick 提供的动态维护对象的两种方式中的一种。还有一种，就是在 ECMAScript 中动态创建 QML 对象。

7.3　在 ECMAScript 中动态创建对象

QML 支持在 ECMAScript 中动态创建对象。这对于延迟对象的创建、缩短应用的启动时间都是有帮助的。同时这种机制也使得我们可以根据用户的输入或者某些事件动态地将可见元素添加到应用场景中。

在 ECMAScript 中，有两种方式可以动态地创建对象：

- 使用 Qt.createComponent()动态地创建一个组件对象，然后使用 Component 的 createObject()方法创建对象。
- 使用 Qt.createQmlObject()从一个 QML 字符串直接创建一个对象。

如果你在一个 QML 文件中定义了一个组件（比如我们的 ColorPicker），而你想动态地创建它的实例，使用 Qt.createComponent()是比较好的方式；而如果你的 QML 对象本身是在应用运行时产生的，那么 Qt.createQmlObject()可能是比较好的选择。

7.3.1　从组件文件动态创建 Component

Qt 对象的 createComponent()方法可以根据 QML 文件动态地创建一个组件。函数原型如下：

```
object createComponent(url, mode, parent)
```

第一个参数 url 指向 QML 文档的本地路径或网络地址；第二个参数 mode 指定创建组件

的模式，可以是 Component.PreferSynchronous（优先使用同步模式）或 Component.Asynchronous（异步模式），忽略 mode 参数时，默认使用 PreferSynchronous 模式；parent 参数指定组件的父对象。

一旦你拥有了组件对象，就可以调用它的 createObject()方法创建一个组件的实例。下面是一个新示例，QML 文件是 qt_create_component.qml：

```
import QtQuick 2.2
import QtQuick.Controls 1.2

Rectangle {
    id: rootItem;
    width: 360;
    height: 300;
    property var count: 0;
    property Component component: null;

    Text {
        id: coloredText;
        text: "Hello World!";
        anchors.centerIn: parent;
        font.pixelSize: 24;
    }

    function changeTextColor(clr){
        coloredText.color = clr;
    }

    function createColorPicker(clr){
        if(rootItem.component == null){
            rootItem.component =
                    Qt.createComponent("ColorPicker.qml");
        }
        var colorPicker;
        if(rootItem.component.status == Component.Ready) {
            colorPicker = rootItem.component.createObject(rootItem,
                {"color" : clr, "x" : rootItem.count *55, "y" : 10});
            colorPicker.colorPicked.connect(rootItem.changeTextColor);
        }

        rootItem.count++;
    }

    Button {
        id: add;
        text: "add";
        anchors.left: parent.left;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        onClicked: {
            createColorPicker(Qt.rgba(Math.random(),
                        Math.random(), Math.random(), 1));
        }
    }
}
```

图 7-6 是示例启动后的界面。

图 7-7 是我点击了 6 次“add”按钮后的效果。

图 7-6　qt_create_component 初始效果

图 7-7　动态创建了颜色选择组件

好了，现在让我们来看看代码。

我在 qt_create_component.qml 中定义了 createColorPicker()函数，该函数的参数是颜色值，它根据颜色值来创建一个颜色选择组件实例。首先它判断 rootItem 的 component 属性如果为 null，就调用 Qt.createComponent()创建一个 ColorPicker 组件，然后调用 Component.createObject()创建一个颜色选择组件实例。createObject()方法有两个参数，第一个参数用来指定创建出来的 Item 的 parent，第二个参数用来传递初始化参数给待创建的 Item，这些参数以 key-value 的形式保存在一个对象中（对象的字面量表示法，参见 5.5.1 节）。我在创建颜色组件实例时，传递颜色、x、y 三个属性给待创建的 Item，于是你看到了，那些色块都在界面顶部。创建了颜色选择组件实例，我调用 colorPicked 信号的 connect()方法，连接 rootItem 的 changeTextColor 方法，以便用户点击色块时改变“Hello World!”文本的颜色。

再来看“add”按钮，它的 onClicked 信号处理器，调用 Math 对象的随机函数 random()和 Qt.rgba()，随机生成一个 Color 对象，传递给 createColorPicker()方法来创建指定颜色的颜色选择组件实例。

说明一下，对于嵌入在 QML 文档内定义的 Component，因为 Component 对象是现成的，可以略去 Qt.createComponent()调用，直接使用 createObject()方法创建组件实例。

代码就这么简单，解说到此为止。现在让我们看看怎么使用 Qt.createQmlObject()来创建对象。

7.3.2　从 QML 字符串动态创建 Component

如果你的软件，需要在运行过程中根据应用的状态适时地生成用于描述对象的 QML 字符串，进而根据这个 QML 字符串创建对象，那么可以像下面这样创建对象：

```
var newObject = Qt.createQmlObject('import QtQuick 2.2;
    Rectangle {color: "red"; width: 20; height: 20}',
    parentItem, "dynamicSnippet1");
```

createQmlObject 方法的第一个参数是要创建对象的 QML 字符串，就像一个 QML 文档一样，你需要导入你用到的所有类型和模块；第二个参数用于指定要创建的对象的父对象；

第三个参数用于给新创建的对象关联一个文件路径，主要用于报告错误。

对于动态创建的对象，该如何销毁呢？

7.3.3　销毁动态创建的对象

有些软件，在不需要一个动态创建的 QML 对象时，仅仅是把它的 visible 属性设置为 false 或者把 opacity 属性设置为 0，而不是删除这个对象。如果动态创建的对象很多，无用的对象都这么处理而不直接删除，那会给软件带来比较大的性能问题，比如内存占用增多，运行速度变慢等。所以呢，动态创建的对象，不再使用时，最好把它删除。

我们这里说的动态创建的对象，特指使用 Qt.createComponent()或 Qt.createQmlObject()方法创建的对象，而使用 Loader 创建的对象，应当通过将 source 设置为空串或将 sourceComponent 设置为 undefined 触发 Loader 销毁它们。

要删除一个对象，可以调用其 destroy()方法。destroy()方法有一个可选的参数，指定延迟多少毫秒再删除这个对象，其默认值为 0。destroy()方法有点儿像 Qt C++中 QObject 的 deleteLater()方法，即便你设定延迟为 0 去调用它，对象也并不会立即删除，QML 引擎会在当前代码块执行结束后的某个合适的时刻删除它们。所以，即便你在一个对象内部调用 destroy()方法也是安全的。

现在让我们再举一个实例，看看如何销毁对象。新的 QML 文档命名为 delete_dynamic_object. qml，从 qt_create_component.qml 拷贝而来，做了一点点修改。先看下面代码：

```
import QtQuick 2.2
import QtQuick.Controls 1.2

Rectangle {
    id: rootItem;
    width: 360;
    height: 300;
    property var count: 0;
    property Component component: null;

    Text {
        id: coloredText;
        text: "Hello World!";
        anchors.centerIn: parent;
        font.pixelSize: 24;
    }

    function changeTextColor(clr){
        coloredText.color = clr;
    }

    function createColorPicker(clr){
        if(rootItem.component == null){
            rootItem.component =
                    Qt.createComponent("ColorPicker.qml");
        }
        var colorPicker;
        if(rootItem.component.status == Component.Ready) {
            colorPicker = rootItem.component.createObject(rootItem,
                {"color" : clr, "x" : rootItem.count *55, "y" : 10});
         colorPicker.colorPicked.connect(rootItem.changeTextColor);
            //[1] add 3 lines to delete some obejcts
            if(rootItem.count % 2 == 1) {
```

```
                colorPicker.destroy(1000);
            }
        }

        rootItem.count++;
    }

    Button {
        id: add;
        text: "add";
        anchors.left: parent.left;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        onClicked: {
            createColorPicker(Qt.rgba(Math.random(),
                        Math.random(), Math.random(), 1));
        }
    }
}
```

修改的部分我用注释标注出来了：添加了三行代码，新创建的颜色选择组件实例，隔一个删一个，destroy(1000)调用指示对象在 1 秒后删除。

图 7-8 是运行后的效果图。

图 7-8　删除动态创建的对象

我还制作了一个演示删除动态创建的对象的示例，QML 文档是 delete_dynamic_object2.qml，我把点击“add”按钮创建的对象保存在一个数组中，当你点击“del”按钮时，删除最后添加的那个颜色选择组件实例。下面是代码：

```
import QtQuick 2.2
import QtQuick.Controls 1.2

Rectangle {
    id: rootItem;
    width: 360;
    height: 300;
    property var count: 0;
    property Component component: null;
    property var dynamicObjects: new Array();

    Text {
        id: coloredText;
        text: "Hello World!";
        anchors.centerIn: parent;
        font.pixelSize: 24;
```

```
    }

    function changeTextColor(clr){
        coloredText.color = clr;
    }

    function createColorPicker(clr){
        if(rootItem.component == null){
            rootItem.component =
                 Qt.createComponent("ColorPicker.qml");
        }
        var colorPicker;
        if(rootItem.component.status == Component.Ready) {
            colorPicker = rootItem.component.createObject(rootItem,
                  {
                        "color" : clr,
                      "x" : rootItem.dynamicObjects.length *55,
                      "y" : 10
                   });
         colorPicker.colorPicked.connect(rootItem.changeTextColor);
            rootItem.dynamicObjects[rootItem.dynamicObjects.length]
                                        = colorPicker;
        }
    }

    Button {
        id: add;
        text: "add";
        anchors.left: parent.left;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        onClicked: {
            createColorPicker(Qt.rgba(Math.random(),
                        Math.random(), Math.random(), 1));
        }
    }
    Button {
        id: del;
        text: "del";
        anchors.left: add.right;
        anchors.leftMargin: 4;
        anchors.bottom: add.bottom;
        onClicked: {
            if(rootItem.dynamicObjects.length > 0){
                var deleted = rootItem.dynamicObjects.splice(-1, 1);
                deleted[0].destroy();
            }
        }
    }
}
```

你可以自己使用 qmlscene 运行 delete_dynamic_object2.qml 看看效果。