中文版《Qt5 Cadaques》

Y.C Cheng

Published with GitBook

Table of Contents

Qt5 Cadaques 正體中文版

雖然 C++ 的複雜會讓人確步，但如果加上跨平台昨為條件，選擇還真的是很少。

Qt 無疑的是在這樣的的選擇下，一個強大穩定且悠久的平台。

對岸的朋友 cwc1987 將 Qt5 Cadaques 一書翻譯為簡體中文，我在此共襄盛舉， 轉譯為正體中文，並將用詞順為台灣用語，以利學人閱讀。

正體中文版WIP

正體中文版於GitHub工作區

GitHub程式碼

簡體中文版

Meet Qt 5

這本書展示了通過Qt 5.x 版本開發應用的各方面內容。主要介紹Qt Quick的技術，但也提供了如何設計C++後端和擴充Qt Quick的方法。

這一章是在一個較高的層次對Qt5的一個概述，它展示了開發者可以使用的不同開發模式，並通過一個Qt5範例說明本書將要介紹的內容。本章也提供了對Qt5的廣泛介紹與如何與Qt開發者聯系的方法。

這章的源代碼能夠在**assetts folder**找到。

序（Preface）

歷史

Qt4自2005年發布以來向成千上萬的應用程序提供了開發框架，甚至是完整的桌面與移動系統。在最近幾年計算機的使用模式發生了改變。從PC機向可攜式設備和移動電腦發展。傳統的桌面設備被越來越多的基於觸摸屏的手機設備取代。桌面用戶的體驗模式也在發生改變。在過去，Windows UI佔據了我們的世界，但現在我們會花更多的時間在其它的UI上。

Qt4設計用於滿足在大多數主流平台的桌面上有一個可以使用的UI視窗。如今Qt的開發者面臨新的問題，它將提供更多的觸摸驅動的使用界面，並且適用於大多數主流桌面與移動系統。Qt4.7開始引進了QtQuick技術，允許用戶設計一個滿足客戶需求的，從簡單的元素來實現一個完整的新的用戶界面。

1.1.1 Qt5 焦點（Qt5 Focus）

Qt5是Qt4版本完整的更新，到Qt4.8版本，Qt4已經發布了7年。是時候讓這個令人驚奇的工具更加驚奇了。

Qt5主要焦點如下：

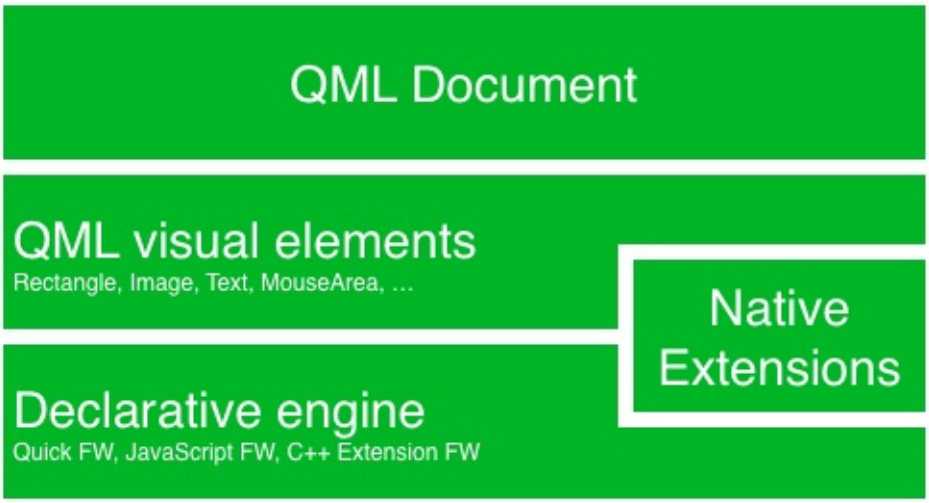
- 傑出的圖形繪制：Qt Quick2是基於OpenGL(ES)場景的實現。重新設計的圖形引擎可以得到更加好的圖形效果與更加簡單的使用方法，在這一領域是之前是從未實現的。
- 開發者生產率：QML和JavaScript語言是主要用於設計UI的方法。後端將有C++來完成繪制。將JavaScript與C++分開能夠快速的開發，讓前端的開發人員專注於建立漂亮的用戶界面，後端的C++開發人員專注於穩定，性能和擴展。
- 跨平台移植性：基於Qt平台的統一抽象概念，現在可以更加容易和快速的將Qt移植到更多的平台上。Qt5是一個Qt核心元件和附加元件的概念，系統開發者只需要專注於必要模塊的實現，可以使程序更加效率的運行。
- 開放的開發：Qt是由Qt-Porject(qt-project.org)主持，它的開發是開放的，由Qt社區驅動的。

Qt5介紹（Qt5 Introduction）

1.2.1 Qt Quick

Qt Quick是Qt5的使用界面技術。Qt Quick自身包含了以下幾種技術：

- QML-使用於用戶界面的標示語言
- JavaScript-動態腳本語言
- Qt C++-具有高度可移植性的C++程式庫.



類似HTML語言，QML是一個標識語言。它由QtQuick封裝在Item {}的元素的標識組成。它從頭設計了用戶界面的設計，並且可以讓開發人員快速，簡單的理解。用戶界面可以使用JavaScript代碼來提供和加強更多的功能。Qt Quick可以使用你自己本地已有的Qt C++輕鬆快速的擴展它的能力。簡單宣告式的UI作為前端，本地部分被稱作後端。這樣你可以將程序的計算密集部分與來自應用程序用戶界面操作部分分開。

在典型的項目中前端開發使用QML/JaveScript，後端代碼開發使用Qt C++來完成系統接口和繁重的計算工作。這樣就很自然的將設計界面的開發者和功能開發者分開了。後端開發測試使用Qt自有的單元測試框架後，輸出給前端開發者使用。

1.2.2 一個用戶界面（Digesting an User Interface）

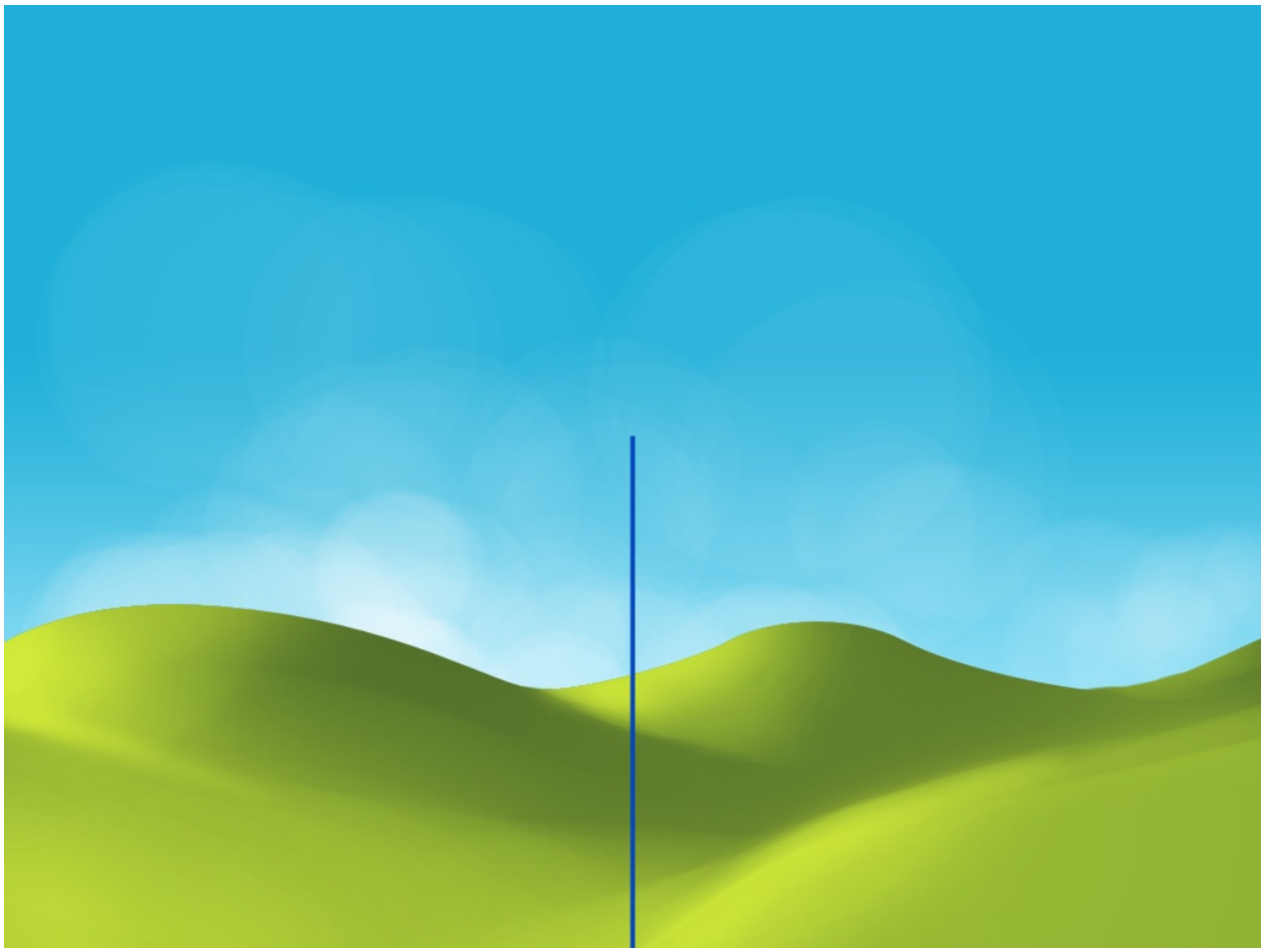
讓我們來使用QtQuick來創建一個簡單的用戶界面，範例QML語言某些方面的特性。最後我們將獲得一個旋轉的風車。

我們開始創建一個空的main.qml檔案。所有的QML文件都已.qml作為後綴。作為一個標識語言（類似HTML）一個QML文檔需要並且只有一個根元素，在我們的案例中是一個基於background的圖像高度與寬度的幾何圖形元素：

```
import QtQuick 2.0

Image {
    id: root
    source: "images/background.png"
```

```
}
```

QML不會對最上層元素設置任何限制，我們使用一個backgournd圖像作為資源的圖像元素來作為我們的根元素。



注意

每一個元素都有屬性，比如一個圖像有寬度，高度但是也有一些其它的屬性例如資源。圖像元素的大小能夠自動的從圖像大小上得出。否則我們應該設置寬度和高度屬性來顯示有效的像素。

大多數典型的元素都放置在**QtQuick2.0**模塊中，我們首先應該在第一行作這個重要的聲明。

id是這個特殊的屬性是可選的，包含了一個標識符，在檔案後面的地方可以直接引用。

重要提示：一個**id**屬性無法在它被設置後改變，並且在程序執行期間無法被設置。使用**root**作為根元素**id**僅僅是作者的習慣，可以在比較大的**QML**檔案中方便的引用最頂層元素。

風車作為前景元素使用圖像的方式放置在我們的用戶界面上。

正常情況下你的用戶界面應該有不同類型的元素構成，而不是像我們的例子一樣只有圖像元素。

```
Image {
    id: root
    ...
    Image {
        id: wheel
        anchors.centerIn: parent
        source: "images/pinwheel.png"
    }
    ...
}
```

為了把風車放在中間的位置，我們使用了一個復雜的屬性，稱之為錨。錨定允許你指定幾何對象與父對象或者同級對象之間的位置關系。比如放置我在另一個元素中間（anchors.centerIn:parent）.有左邊（left），右邊（right），頂部（top），底部（bottom），中央（centerIn），填充（fill），垂直中央（verticalCenter）和水平中央（horizontalCenter）來表示元素之間的關系。確保他們能夠匹配，錨定一個對象的左側頂部的一個元素這樣的做法是沒有意義的。所以我們設置風車在父對象background的中央。

注意

有時你需要進行一些微小的調整。使用**anchors.horizontalCenterOffset**或者**anchors.verticalCenterOffset**可以幫你實現這個功能。類似的調整屬性也可以用於其他所有的錨。查閱**Qt**的幫助檔案可以知道完整的錨屬性列表。

注意

將一個圖像作為根矩形元素的子元素放置範例了一種聲明式語言的重要概念。你描述了用戶界面的層和分組的順序，最頂部的一層（根矩形框）先繪制，然後子層按照包含它的元素局部坐標繪制在包含它的元素上。

為了讓我們的範例更加有趣一點，我們應該讓程序有一些交互功能。當用戶點擊場景上某個位置時，讓我們的風車轉動起來。

我們使用mouseArea元素，並且讓它與我們的根元素大小一樣。

```
Image {
    id: root
    ...
    MouseArea {
        anchors.fill: parent
        onClicked: wheel.rotation += 90
    }
    ...
}
```

當用戶點擊覆蓋區域時，鼠標區域會發出一個信號。你可以重寫onClicked函數來鏈接這個信號。在這個案例中引用了風車的圖像並且讓他旋轉增加90度。

注意

對於每個工作的信號，命名方式都是**on + SignalName**的標題。當屬性的值發生改變時也會發出一個信號。它們的命名方式是：**on + PropertyName + Chagned**。 如果一個寬度（**width**）屬性改變了，你可以使用**onWidthChanged: print(width)**來得到這個監控這個新的寬度值。

現在風車將會旋轉，但是還不夠流暢。風車的旋轉角度屬性被直接改變了。我們應該怎樣讓90度的旋轉可以持續一段時間呢。現在是動畫效果發揮作用的時候了。一個動畫定義了一個屬性的在一段時間內的變化過程。為了實現這個效果，我們使用一個動畫類型叫做屬性行為。這個行為指定了一個動畫來定義屬性的每一次改變並賦值給屬性。每次屬性改變，動畫都會運行。這是QML中聲明動畫的幾種方式中的一種方式。

```
Image {
    id: root
    Image {
        id: wheel
        Behavior on rotation {
            NumberAnimation {
                duration: 250
            }
        }
    }
}
```

現在每當風車旋轉角度發生改變時都會使用NumberAnimation來實現250毫秒的旋轉動畫效果。每一次90度的轉變都需要花費250ms。

現在風車看起來好多了，我希望以上這些能夠讓你能夠對Qt Quick編程有一些了解。

Qt構建模組（Qt Building Blocks）

Qt5是由大量的模組組成的。一個模組通常情況下是一個程式庫，提供給開發者使用。一些模組是強制性用來支持Qt平台的，它們分成一組叫做Qt基礎模組。許多模組是可選的，它們分成一組叫做Qt附加模組，預計大多數得到開發人員將不會使用它們，但是最好知道它們可以對一些通用的問題提供非常有價值的解決方案。

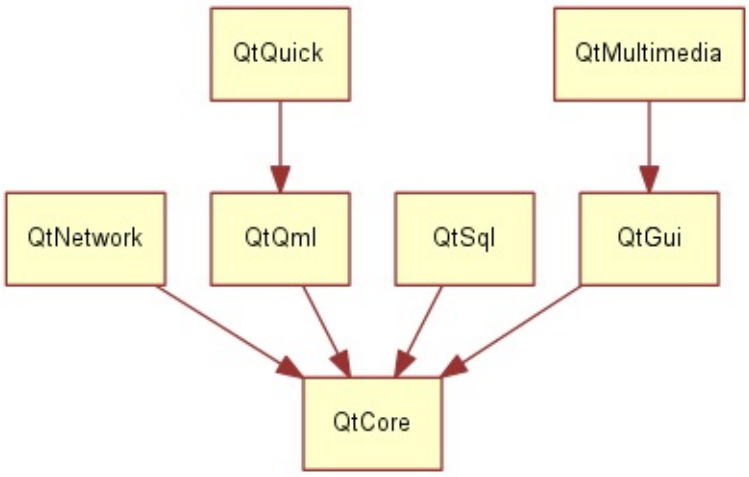
1.3.1 Qt模組（Qt Modules）

Qt基礎模組是對Qt平台的必要支持。它們使用Qt Quick 2開發Qt 5應用程序的基礎。

核心基礎模組

以下這些是啟動QML程序最小的模組集合。

模組名	描述
Qt Core	核心的非圖形類，供其它模組使用。
Qt GUI	圖形用戶界面（GUI）組件的基類，包括OpenGL。
Qt Multimedia	聲音，影片，電台，照相的功能類。
Qt Network	簡化方便的網路編程的類。
Qt QML	QML類與JavaScript語言的支持。
Qt Quick	可高度動態構建的自定義應用程序用戶界面框架。
Qt SQL	集成SQL資料程式庫類。
Qt Test	Qt應用程序與程式庫的單元測試類。
Qt WebKit	集成WebKit2的基礎實現並且提供了新的QML應用程序介面。在附件模組中查看Qt WebKit Widgets可以獲取更多的資訊。
Qt WebKit Widgets	Widgets 來自Qt4中集成WebKit1的窗口基礎類別。
Qt Widgets	擴展Qt GUI模組的C++窗口類別。



Qt附加模組

除了必不可少的基礎模組，Qt提供了附加模組供軟件開發者使用，這部分不一定包含在發布的版本中。以下簡短的列出了一些可用的附加模組列表。

- Qt 3D - 一組使3D編程更加方便的應用程序介面和聲明。
- Qt Bluetooth - 在多平台上使用無線藍牙技術的C++和QML應用程序介面。
- Qt Contacts - 提供存取聯系人與聯系人資料程式庫的C++和QML應用程序介面。
- Qt Location - 提供了定位，地圖，導航和位置搜索的C++與QML介面。使用NMEA在後端進行定位。（NMEA縮寫，同時也是數據傳輸標準工業協會，在這裡，實際上應為NMEA 0183。它是一套定義接收機輸出的標準資訊，有幾種不同的格式，每種都是獨立相關的ASCII格式，逗點隔開數據流，數據流長度從30-100字符不等，通常以每秒間隔選擇輸出，最常用的格式為"GGA"，它包含了定位時間，緯度，經度，高度，定位所用的衛星數，DOP值,差分狀態和校正時段等，其他的有速度，跟蹤，日期等。NMEA實際上已成為所有的GPS接收機和最通用的數據輸出格式，同時它也被用於與GPS接收機介面的大多數的軟件包裡。）
- Qt Organizer - 提供了組織事件（任務清單，事件等等）的C++和QML應用程序介面。
- Qt Publish and SubScribe - Qt發布與訂閱
- Qt Sensors - 存取傳感器的QML與C++介面。
- Qt Service Framework - 允許應用程序讀取，操縱和訂閱來改變通知資訊。
- Qt System Info - 發布系統相關的資訊和功能。
- Qt Versit - 支持電子名片與日歷數據格式（iCalendar）。（iCalendar是“日歷數據交換”的標準（RFC 2445）。 此標準有時指的是“iCal”，即蘋果公司的出品的一款同名日歷軟件，這個軟件也是此標準的一種實現方式。）
- Qt Wayland - 只用於Linux系統。包含了Qt合成器應用程序介面（server），和Wayland平台插件（clients）。
- Qt Feedback - 回饋用戶的觸摸和聲音操作。
- Qt JSON DB - 對於Qt的一個不使用SQL的資料庫。

注意

這些模組一部分還沒有發布，這依賴於有多少貢獻者，並且它們能夠獲得更好的測試。

1.3.2 支持的平台（Supported Platforms）

Qt支持各種不同的平台。大多數主流的桌面與嵌入式平台都能夠支持。通過Qt應用程序抽象，現在可以更容易的將Qt移植到你自己的平台上。在一個平台上測試Qt5是非常花費時間的。選擇測試的平台子集可以參考qt-project元件的平台。這些平台需要完全通過系統的測試才能確保最好的品質。友情提醒：任何代碼都可能會有Bug的。

Qt項目（Qt Project）

來自qt-project百科：Qt-Project是由Qt社區上對Qt感興趣的人達成共識的地方。任何人都可以在社區上分享它感興趣的東西，參與它的開發，並且向Qt的開發做出貢獻。

Qt-Project是一個為Qt未來開發開源部分的組織。它基於使用者的貢獻。最大的貢獻者是DIGIA，它可以提供Qt的商業授權。 Qt對於公司分為開源授權和商業授權。使用商業授權的公司不需要遵守開源協議。沒有商業授權的許可的公司不能使用Qt，並且它也不允許DIGIA向Qt項目貢獻太多的代碼。

在全球有很多公司，他們在不同的平台上使用Qt開發產品，提供咨詢。同樣也有很多開源專案和開源開發者，它們使用Qt作為它們的開發程式庫。成為這樣開發活潑的社區的一部分，並且使用這個很棒的程式庫讓人感覺很好。它能讓你成為一個更好的人嗎？也許:-)。

Get Start

這一章介紹了如何使用Qt5進行開發。我們將告訴你如何安裝Qt軟件開發工具包（Qt SDK）和如何使用Qt Creator集成開發環境（Qt Creator IDE）創建並運行一個簡單的hello word應用程序。

注意

這章的源代碼能夠在**assetts folder**找到。

安裝**Qt5**軟件工具包（**Installing Qt 5 SDK**）

Qt軟件工具包包含了編譯桌面或者嵌入式應用程序的工具。最新的版本可以從Qt-Project下載。我們將使用這種方法開始。

軟件工具包自身包含了一個維護工具允許你更新到最新版本的軟件工具包。

Qt軟件工具包非常容易安裝，並且附帶了一個它自身的快速集成開發環境叫做Qt Creator。這個集成開發環境可以讓你高效的使用Qt進行開發，我們推薦給所有的讀者使用。在任何情況下Qt都可以通過命令的方式來編譯，你可以自由的選擇你的代碼編輯器。

當你安裝軟件工具包時，你最好選擇默認的選項確保Qt 5.x可以被使用。然後一切準備就緒。

你好世界（**Hello World**）

為了測試你的安裝，我們創建一個簡單的應用程序hello world.打開Qt Creator並且創建一個Qt Quick UI Project（File->New File 或者 Project-> Qt Quick Project -> Qt Quick UI）並且給項目取名 HelloWorld。

注意

Qt Creator集成開發環境允許你創建不同類型的應用程序。如果沒有另外**說**明，我們都創建**Qt Quick UI Project**。

提示

一個典型的**Qt Quick**應用程序在運行時解釋，與本地插件或者本地代碼在運行時解釋代碼一樣。對于才開始的我們不需要關注本地端的解釋開發，只需要把注意力集中在**Qt5**運行時的方面。

Qt Creator將會為我們創建幾個文件。HellWorld.qmlproject文件是項目文件，保存了項目的配置信息。這個文件由Qt Creator管理，我們不需要編輯它。

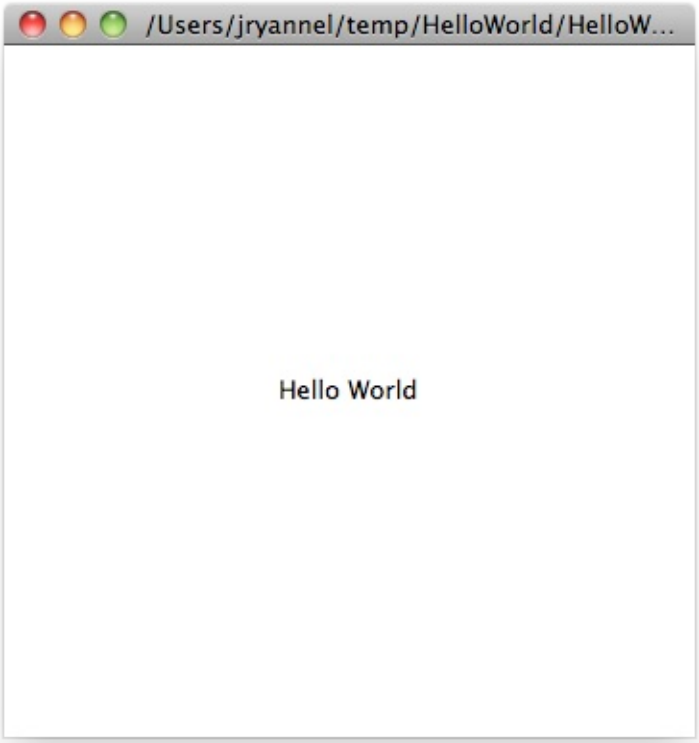
另一個文件HelloWorld.qml保存我們應用程序的代碼。打開它，並且嘗試想想這個應用程序要做什麼，然後再繼續讀下去。

```
// HelloWorld.qml

import QtQuick 2.0

Rectangle {
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: "Hello World"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }
}
```

HelloWorld.qml使用QML語言來編寫。我們將在下一章更深入的討論QML語言，現在只需要知道它描述了一系列有層次的用戶界面。這個代碼指定了顯示一個360乘以360像素的一個矩形，矩形中間有一個“Hello World"的文本。鼠標區域覆蓋了整個矩形，當用戶點擊它時，程序就會退出。

你自己可以運行這個應用程序，點擊左邊的運行或者從菜單選擇select Bulid->Run。

如果一切順利，你將看到下面這個窗口：



Qt 5似乎已經可以工作了，我們接著繼續。

建議

如果你是一個名系統集成人員，你會想要安裝最新穩定的**Qt**版本，將這個**Qt**版本的源代碼編譯到你特定的目標機器上。

從頭開始構建

如果你想使用命令行的方式構建Qt5，你首先需要拷貝一個代碼庫並構建他。

```
git clone git://gitorious.org/qt/qt5.git
cd qt5
./init-repository
./configure -prefix $PWD/qtbase -opensource
make -j4
```

等待兩杯咖啡的時間編譯完成後，qtbase文件夾中將會出現可以使用的Qt5。任何飲料都好，不過我喜歡喝著咖啡等待最好的結果。

如果你想測試你的編譯，只需簡單的啟動qtbase/bin/qmlscene並且選擇一個QtQuick的例子運行，或者跟著我們進入下一章。

為了測試你的安裝，我們創建了一個簡單的hello world應用程序。創建一個空的qml文件example1.qml，使用你最喜愛的文本編輯器並且粘貼一下內容：

```
// HelloWorld.qml

import QtQuick 2.0
```

```
Rectangle {
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: "Greetings from Qt5"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }
}
```

你現在使用來自Qt5的默認運行環境來可以運行這個例子：

```
$ qtbase/bin/qmlscene
```

應用程序類型（Application Types）

這一節貫穿了可能使用Qt5編寫的不同類型的應用程序。沒有任何建議的選擇，只是想告訴讀者Qt5通常情況下能做些什麼。

2.3.1 控制台應用程序

一個控制台應用程序不需要提供任何人機交互圖形界面通常被稱作系統服務，或者通過命令行來運行。Qt5附帶了一系列現成的組件來幫助你非常有效的創建跨平台的控制台應用程序。例如網絡應用程序編程接口或者文件應用程序編程接口，字符串的處理，自Qt5.1發布的高效的命令解析器。由于Qt是基于C++的高級應用程序接口，你能夠快速的編程並且程序擁有快速的執行速度。不要認為Qt僅僅只是用戶界面工具，它也提供了許多其它的功能。

字符串處理

在第一個例子中我們展示了怎樣簡單的增加兩個字符串常量。這不是一個有用的應用程序，但能讓你了解本地端C++應用程序沒有事件循環時是什麼樣的。

```
// module or class includes
#include <QtCore>

// text stream is text-codec aware
QTextStream cout(stdout, QIODevice::WriteOnly);

int main(int argc, char** argv)
{
    // avoid compiler warnings
    Q_UNUSED(argc)
    Q_UNUSED(argv)
    QString s1("Paris");
    QString s2("London");
    // string concatenation
    QString s = s1 + " " + s2 + "!";
    cout << s << endl;
}
```

容器類

這個例子在應用程序中增加了一個鏈表和一個鏈表迭代器。Qt自帶大量方便使用的容器類，並且其中的元素使用相同的應用程序接口模式。

```
QString s1("Hello");
QString s2("Qt");
QList<QString> list;
// stream into containers
list <<  s1 << s2;
// Java and STL like iterators
QListIterator<QString> iter(list);
while(iter.hasNext()) {
    cout << iter.next();
    if(iter.hasNext()) {
        cout << " ";
    }
```

```
}
cout << "!" << endl;
```

這裡我們展示了一些高級的鏈表函數，允許你在一個字符串中加入一個鏈表的字符串。當你需要持續的文本輸入時非常的方便。使用QString::split()函數可以將這個操作逆向（將字符串轉換為字符串鏈表）。

```
QString s1("Hello");
QString s2("Qt");
// convenient container classes
QStringList list;
list <<  s1 << s2;
// join strings
QString s = list.join(" ") + "!";
cout << s << endl;
```

文件**IO**

下一個代碼片段我們從本地讀取了一個CSV文件並且遍歷提取每一行的每一個單元的數據。我們從CSV文件中獲取大約20行的編碼。文件讀取僅僅給了我們一個比特流，為了有效的將它轉換為可以使用的Unicode文本，我們需要使用這個文件作為文本流的底層流數據。編寫CSV文件，你只需要以寫入的方式打開一個文件並且一行一行的輸入到文件流中。

```
QList<QStringList> data;
// file operations
QFile file("sample.csv");
if(file.open(QIODevice::ReadOnly)) {
    QTextStream stream(&file);
    // loop forever macro
    forever {
        QString line = stream.readLine();
        // test for empty string 'QString("")'
        if(line.isEmpty()) {
            continue;
        }
        // test for null string 'String()'
        if(line.isNull()) {
            break;
        }
        QStringList row;
        // for each loop to iterate over containers
        foreach(const QString& cell, line.split(",")) {
            row.append(cell.trimmed());
        }
        data.append(row);
    }
}
// No cleanup necessary.
```

現在我們結束Qt關于基于控制台應用程序小節。

2.3.2 窗口應用程序

基于控制台的應用程序非常方便，但是有時候你需要有一些用戶界面。但是基于用戶界面的應用程序需要後端來寫入/讀取文件，使用網絡進行通訊或者保存數據到一個容器中。

在第一個基于窗口的應用程序代碼片段，我們僅僅只創建了一個窗口並顯示它。沒有父對象的窗口部件是Qt世界中的一個窗口。我們使用智能指針來確保當智能指針指向範圍外時窗口會被刪除掉。

這個應用程序對象封裝了Qt的運行，調用exec開始我們的事件循環。從這裡開始我們的應用程序只響應由鼠標或者鍵盤或者其它的例如網絡或者文件IO的事件觸發。應用程序也只有在事件循環退出時才退出，在應用程序中調用"quit()"或者關掉窗口來退出。 當你運行這段代碼的時候你可以看到一個240乘以120像素的窗口。

```
#include <QtGui>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QScopedPointer<QWidget> widget(new CustomWidget());
    widget->resize(240, 120);
    widget->show();
    return app.exec();
}
```

自定義窗口部件

當你使用用戶界面時你需要創建一個自定義的窗口部件。典型的窗口是一個窗口部件區域的繪制調用。附加一些窗口部件內部如何處理外部觸發的鍵盤或者鼠標輸入。為此我們需要繼承QWidget並且重寫幾個函數來繪制和處理事件。

```
#ifndef CUSTOMWIDGET_H
#define CUSTOMWIDGET_H

#include <QtWidgets>

class CustomWidget : public QWidget
{
    Q_OBJECT
public:
    explicit CustomWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
private:
    QPoint m_lastPos;
};

#endif // CUSTOMWIDGET_H
```

在實現中我們繪制了窗口的邊界並在鼠標最後的位置上繪制了一個小的矩形框。這是一個非常典型的低層次的自定義窗口部件。鼠標或者鍵盤事件會改變窗口的內部狀態並觸發重新繪制。我們不需要更加詳細的分析這個代碼，你應該有能力分析它。Qt自帶了大量現成的桌面窗口部件，你有很大的幾率不需要再做這些工作。

```
#include "customwidget.h"

CustomWidget::CustomWidget(QWidget *parent) :
    QWidget(parent)
{
```

```
}

void CustomWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QRect r1 = rect().adjusted(10,10,-10,-10);
    painter.setPen(QColor("#33B5E5"));
    painter.drawRect(r1);

    QRect r2(QPoint(0,0),QSize(40,40));
    if(m_lastPos.isNull()) {
        r2.moveCenter(r1.center());
    } else {
        r2.moveCenter(m_lastPos);
    }
    painter.fillRect(r2, QColor("#FFBB33"));
}

void CustomWidget::mousePressEvent(QMouseEvent *event)
{
    m_lastPos = event->pos();
    update();
}

void CustomWidget::mouseMoveEvent(QMouseEvent *event)
{
    m_lastPos = event->pos();
    update();
}
```

桌面窗口

Qt的開發者們已經為你做好大量現成的桌面窗口部件，在不同的操作系統中他們看起來都像是本地的窗口部件。你的工作只需要在一個打的窗口容器中安排不同的的窗口部件。在Qt中一個窗口部件能夠包含其它的窗口部件。這個操作由分配父子關系來完成。這意味著我們需要準備類似按鈕（button），復選框（check box），單選按鈕（radio button）的窗口部件並且對它們進行布局。下面展示了一種完成的方法。

這裡有一個頭文件就是所謂的窗口部件容器。

```
class CustomWidget : public QWidget
{
    Q_OBJECT
public:
    explicit CustomWidgetQWidget *parent = 0);
private slots:
    void itemClicked(QListWidgetItem* item);
    void updateItem();
private:
    QListWidget *m_widget;
    QLineEdit *m_edit;
    QPushButton *m_button;
};
```

在實現中我們使用布局來更好的安排我們的窗口部件。當容器窗口部件大小被改變後它會按照窗口部件的大小策略進行重新布局。在這個例子中我們有一個鏈表窗口部件，行編輯器與按鈕垂直排列來編輯一個城市的鏈表。我們使用Qt的信號與槽來連接發送和接收對象。

```
CustomWidget::CustomWidget(QWidget *parent) :
    QWidget(parent)
{
    QVBoxLayout *layout = new QVBoxLayout(this);
    m_widget = new QListWidget(this);
    layout->addWidget(m_widget);

    m_edit = new QLineEdit(this);
    layout->addWidget(m_edit);

    m_button = new QPushButton("Quit", this);
    layout->addWidget(m_button);
    setLayout(layout);

    QStringList cities;
    cities << "Paris" << "London" << "Munich";
    foreach(const QString& city, cities) {
        m_widget->addItem(city);
    }

    connect(m_widget, SIGNAL(itemClicked(QListWidgetItem*)), this, SLOT(itemClicked(QList
    connect(m_edit, SIGNAL(editingFinished()), this, SLOT(updateItem()));
    connect(m_button, SIGNAL(clicked()), qApp, SLOT(quit()));
}

void CustomWidget::itemClicked(QListWidgetItem *item)
{
    Q_ASSERT(item);
    m_edit->setText(item->text());
}

void CustomWidget::updateItem()
{
    QListWidgetItem* item = m_widget->currentItem();
    if(item) {
        item->setText(m_edit->text());
    }
}
```

繪制圖形

有一些問題最好用可視化的方式表達。如果手邊的問題看起來有點像幾何對象，qt graphics view是一個很好的選擇。一個圖形視窗（graphics view）能夠在一個場景（scene）排列簡單的幾何圖形。用戶可以與這些圖形交互，它們使用一定的算法放置在場景（scene）上。填充一個圖形視圖你需要一個圖形窗口（graphics view）和一個圖形場景（graphics scene）。一個圖形場景（scene）連接在一個圖形窗口（view）上，圖形對象（graphics item）是被放在圖形場景（scene）上的。這裡有一個簡單的例子，首先頭文件定義了圖形窗口（view）與圖形場景（scene）。

```
class CustomWidgetV2 : public QWidget
{
    Q_OBJECT
public:
    explicit CustomWidgetV2(QWidget *parent = 0);
private:
    QGraphicsView *m_view;
    QGraphicsScene *m_scene;

};
```

在實現中首先將圖形場景（scene）與圖形窗口（view）連接。圖形窗口（view）是一個窗口部件，能夠被我們的窗口部件容器包含。最後我們添加一個小的矩形框在圖形場景（scene）中。然後它會被渲染到我們的圖形窗口（view）上。

```
#include "customwidgetv2.h"

CustomWidget::CustomWidget(QWidget *parent) :
    QWidget(parent)
{
    m_view = new QGraphicsView(this);
    m_scene = new QGraphicsScene(this);
    m_view->setScene(m_scene);

    QVBoxLayout *layout = new QVBoxLayout(this);
    layout->setMargin(0);
    layout->addWidget(m_view);
    setLayout(layout);

    QGraphicsItem* rect1 = m_scene->addRect(0,0, 40, 40, Qt::NoPen, QColor("#FFBB33"));
    rect1->setFlags(QGraphicsItem::ItemIsFocusable|QGraphicsItem::ItemIsMovable);
}
```

2.3.3 數據適配

到現在我們已經知道了大多數的基本數據類型，並且知道如何使用窗口部件和圖形視圖（graphics views）。通常在應用程序中你需要處理大量的結構體數據，也可能需要不停的儲存它們，或者這些數據需要被用來顯示。對于這些Qt使用了模型的概念。下面一個簡單的模型是字符串鏈表模型，它被一大堆字符串填滿然後與一個鏈表視圖（list view）連接。

```
m_view = new QListView(this);
m_model = new QStringListModel(this);
view->setModel(m_model);

QList<QString> cities;
cities << "Munich" << "Paris" << "London";
model->setStringList(cities);
```

另一個比較普遍的用法是使用SQL（結構化數據查詢語言）來存儲和讀取數據。Qt自身附帶了嵌入式版的SQLLite並且也支持其它的數據引擎（比如MySQL，PostgresSQL，等等）。首先你需要使用一個模式來創建你的數據庫，比如像這樣：

```
CREATE TABLE city (name TEXT, country TEXT);
INSERT INTO city value ("Munich", "Germany");
INSERT INTO city value ("Paris", "France");
INSERT INTO city value ("London", "United Kingdom");
```

為了能夠在使用sql，我們需要在我們的項目文件（*.pro）中加入sql模塊。

```
QT += sql
```

然後我們需要c++來打開我們的數據庫。首先我們需要獲取一個指定的數據庫引擎的數據對象。使用這個數據庫對象我們可以打開數據庫。對于SQLLite這樣的數據庫我們可以指定一個數據庫文件的路徑。Qt提供了一些高級的數據庫模型，其中有一種表格模型（table model）使用表格標示符和一個選項分支語句（where clause）來選擇數據。這個模型的結果能夠與一個鏈表視圖連接，就像之前連接其它數據模型一樣。

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName('cities.db');
if(!db.open()) {
    qFatal("unable to open database");
}

m_model = QSqlTableModel(this);
m_model->setTable("city");
m_model->setHeaderData(0, Qt::Horizontal, "City");
m_model->setHeaderData(1, Qt::Horizontal, "Country");

view->setModel(m_model);
m_model->select();
```

對高級的模型操作，Qt提供了一種分類文件代理模型，允許你使用基礎的分類排序和數據過濾來操作其它的模型。

```
QSortFilterProxyModel* proxy = new QSortFilterProxyModel(this);
proxy->setSourceModel(m_model);
view->setModel(proxy);
view->setSortingEnabled(true);
```

數據過濾基于列號與一個字符串參數完成。

```
proxy->setFilterKeyColumn(0);
proxy->setFilterCaseSensitive(Qt::CaseInsensitive);
proxy->setFilterFixedString(QString)
```

過濾代理模型比這裡演示的要強大的多，現在我們只需要知道有它的存在就夠了。

注意

這裡是綜述了你可以在**Qt5**中開發的不同類型的經典應用程序。桌面應用程序正在發生著改變，不久之後移動設備將會為佔據我們的世界。移動設備的用**戶**界面設計非常不同。它們相對于桌面應用程序更加簡潔，只需要專注的做一件事情。動畫效果是一個非常重要的部分，用**戶**界面需要生動活潑。傳統的**Qt**技術已經不適于這些市場了。

接下來：**Qt Quick**將會解決這個問題。

2.3.4 Qt Quick應用程序

在現代的軟件開發中有一個內在的衝突，用戶界面的改變速度遠遠高于我們的後端服務。在傳統的技術中我們開發的前端需要與後端保持相同的步調。當一個項目在開發時用戶想要改變用戶界面，或者在一個項目中開發一個用戶界面的想法就會引發這個衝突。敏捷項目需要敏捷的方法。

Qt Quick 提供了一個類似HTML聲明語言的環境應用程序作為你的用戶界面前端（the front-end），在你的後端使用本地的c++代碼。這樣允許你在兩端都遊刃有餘。

下面是一個簡單的Qt Quick UI的例子。

```
import QtQuick 2.0

Rectangle {
    width: 240; height: 1230
    Rectangle {
        width: 40; height: 40
        anchors.centerIn: parent
        color: '#FFBB33'
    }
}
```

這種聲明語言被稱作QML，它需要在運行時啟動。Qt提供了一個典型的運行環境叫做qmlscene，但是想要寫一個自定義的允許環境也不是很困難，我們需要一個快速視圖（quick view）並且將QML文檔作為它的資源。剩下的事情就只是展示我們的用戶界面了。

```
QQuickView* view = new QQuickView();
QUrl source = Qurl::fromLocalUrl("main.qml");
view->setSource(source);
view.show();
```

回到我們之前的例子，在一個例子中我們使用了一個c++的城市數據模型。如果我們能夠在QML代碼中使用它將會更加的好。

為了實現它我們首先要編寫前端代碼怎樣展示我們需要使用的城市數據模型。在這一個例子中前端指定了一個對象叫做cityModel，我們可以在鏈表視圖（list view）中使用它。

```
import QtQuick 2.0

Rectangle {
    width: 240; height: 120
    ListView {
        width: 180; height: 120
        anchors.centerIn: parent
        model: cityModel
        delegate: Text { text: model.city }
    }
}
```

為了使用cityModel，我們通常需要重復使用我們以前的數據模型，給我們的根環境（root context）加上一個內容屬性（context property）。（root context是在另一個文檔的根元素中）。

```
m_model = QSqlTableModel(this);
... // some magic code
QHash<int, QByteArray> roles;
roles[Qt::UserRole+1] = "city";
roles[Qt::UserRole+2] = "country";
m_model->setRoleNames(roles);
view->rootContext()->setContextProperty("cityModel", m_model);
```

警告

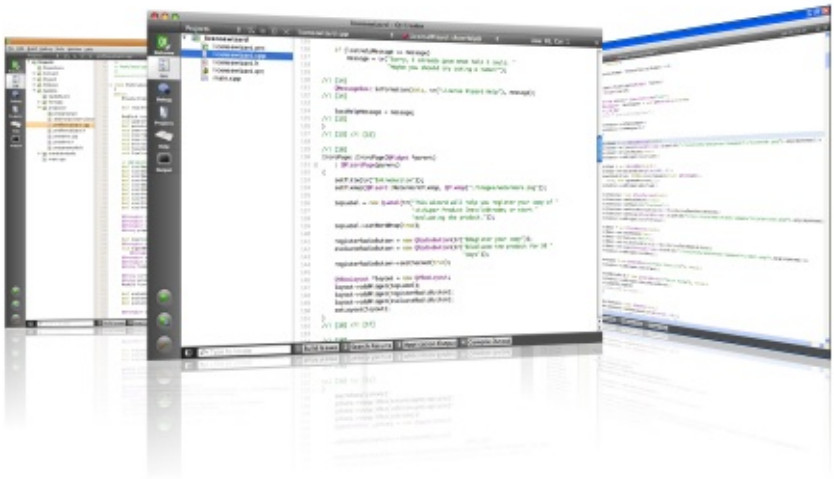
這不是完全正確的用法，作為包含在**SQL**表格模型列中的數據，一個**QML**模型的任務是來表達這些數據。所以需要做一個在列和任務之間的映射關系。請查看來**QML and QSqlTableModel**獲得更多的信息。

總結（ **Summary**）

我們已經知道了如何安裝Qt軟件開發工具包，並且知道如何創建我們的應用。我們向你展示和概述了使用Qt開發不同類型的應用程序。展示Qt可以給你的應用程序開發提供的一些功能。我希望你對Qt留下一個好的印象，Qt是一個非常好的用戶界面開發工具並且盡可能的提供了一個應用開發者期望的東西。當前你也不必一直鎖定使用Qt，你也可以使用其它的庫或者自己來擴展Qt。Qt對于不同類型的應用程序開發支持非常豐富：包括控制台程序，經典的桌面用戶界面程序和觸摸式用戶界面程序。

Qt Creator IDE

Qt Creator是Qt默認的集成開發環境。它由Qt的開發者們編寫提供的。這個集成開發環境能夠在大多數的桌面開發平台上使用，例如 Windows/Mac/Linux。我們也已經看到有些用戶在嵌入式設備上使用Qt Creator。Qt Creator有著精簡的用戶界面，可以幫助開發者們高效的完成開發生產。Qt Creator 能夠啟動你的QtQuick用戶界面，也可以用來編譯c++代碼到你的主機系統或者使用交叉編譯到你的設備系統上。



注意

這章的源代碼能夠在**assetts folder**找到。

用戶界面（The User Interface）

當你啟動Qt Creator時，你可以看到一個歡迎畫面。在這裡你可以找到怎樣在Qt Creator中繼續的重要提示，或者你最近使用的項目。你可以看到一個會話列表，你可以看到是一個空的。一個會話是供你參考使用的一堆項目的集合。當你在同時擁有幾個客戶的大項目時，這個功能非常有用。

你可以在左邊看到模式選擇。模式選擇包含了你典型的工作步驟。

- 歡迎模式：你目前所在的位置。
- 編輯模式：專注于編碼。
- 設計模式：專注于用戶界面設計。
- 調試模式：獲取當前運行程序的相關信息。
- 項目模式：修改你的項目編譯運行配置。
- 分析模式：檢查內存洩露並剖析。
- 幫助模式：閱讀Qt的幫助文檔。

在模式選擇下面你可以找到項目配置選擇與執行/調試。

你應該大多數時間都處于編輯模式的中央面板中的代碼編輯器編輯你的代碼。當你需要配置你的項目時，你將不時的訪問項目模式。當你點擊Run（運行）。Qt Creator會先確保充分的構建你的項目後再運行它。

在最下面的輸出窗是錯誤信息，應用程序信息，編譯信息和其它的信息。

注冊你的**Qt**工具箱（**Registering your Qt Kit**）

最開始使用Qt Creator時最困難的部分可能是Qt Kit。一個Qt Kit由Qt的版本，編譯系統和設備等等其它設置來配置它。它使用唯一標識的工具組合來構建你的項目。一個典型的桌面kit（工具箱）可能包含一個GCC編譯程序，一個Qt版本庫（比如Qt5.1.1）和一個設備（”桌面“）。在你創建好你的項目後你需要為項目指定一個kit（工具箱）來構建項目。在你創建一個kit（工具箱）之前你需要先安裝一個編譯程序並注冊一個Qt版本。Qt版本的注冊由指定qmake的執行路徑完成。Qt Creator通過查詢qmake的信息來獲取Qt的版本標識。

添加kit（工具箱）與注冊Qt版本在Settings->Bulild & Run entry中完成，在這裡你也可以查看有哪些編譯程序已經被注冊了的。

注意

請首先確保你的**Qt Creator**中已經注冊了正確的**Qt**版本，並且確保一個**Kit**（工具箱）指定了一個編譯程序與**Qt**版本和設備的組合。你無法離開**Kit**（工具箱）來構建一個項目。

項目管理（**Managing Projects**）

Qt Creator在項目中管理你的源代碼。你可以使用File->New File或者Project來創建一個新項目。當你創建一個項目時，你可以選擇多種應用程序模板。Qt Creator 能夠創建桌面，手機應用程序。這些應用程序使用窗口部件（Widgets）或者QtQuick或者控制台，甚至可以是更加簡單的項目。當然也支持HTML5與python的項目。對于一個新手是很難選擇的，所以我們為你選擇了三種類型的項目。

- 應用程序/QtQuick2.0用戶界面：這將會為你創建一個QML/JS的項目，不需要使用任何的C++代碼。使用這個你可以迅速的創建一個新的用戶界面或者計劃創建一個基于本地插件的現代的用戶界面應用程序。

- 庫/Qt Quick2.0擴展插件：使用這個安裝引導能夠創建一個你自己的Qt Quick用戶界面插件。這個插件被用來擴展Qt Quick的本地元素。

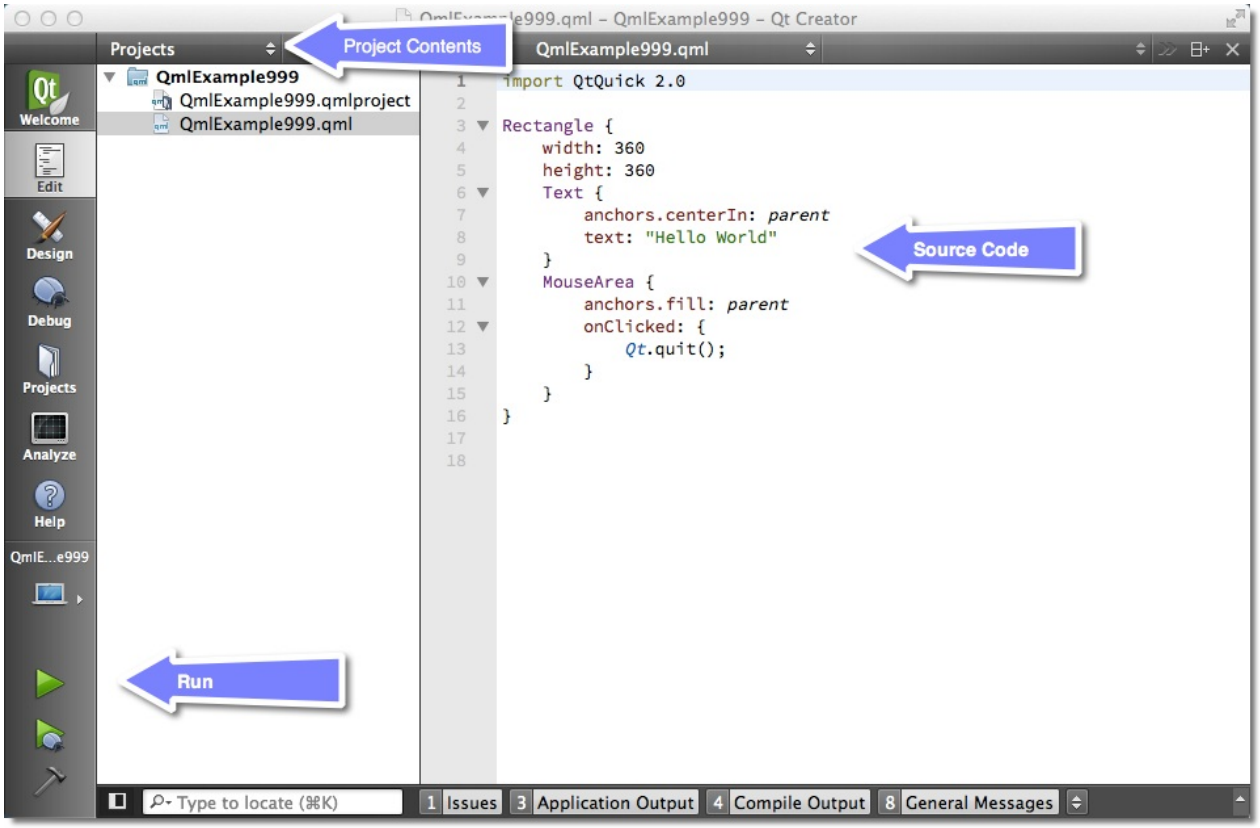
- 其它項目/空的Qt項目：只是一個項目的骨架。如果你想從頭使用C++來編寫你的應用程序，你可以使用這種方式。你需要知道你在這裡能做什麼。

注意

在這本書的前面部分我們主要使用**QtQuick 2.0**用**戶**界面項目。在後面我們會使用空的**Qt**項目或者類似的項目描述一些**C++**方面的使用。為了使用我們自己的本地插件來擴展**QtQuick**，我們將會使用**Qt Quick2.0**擴展插件安裝引導項目。

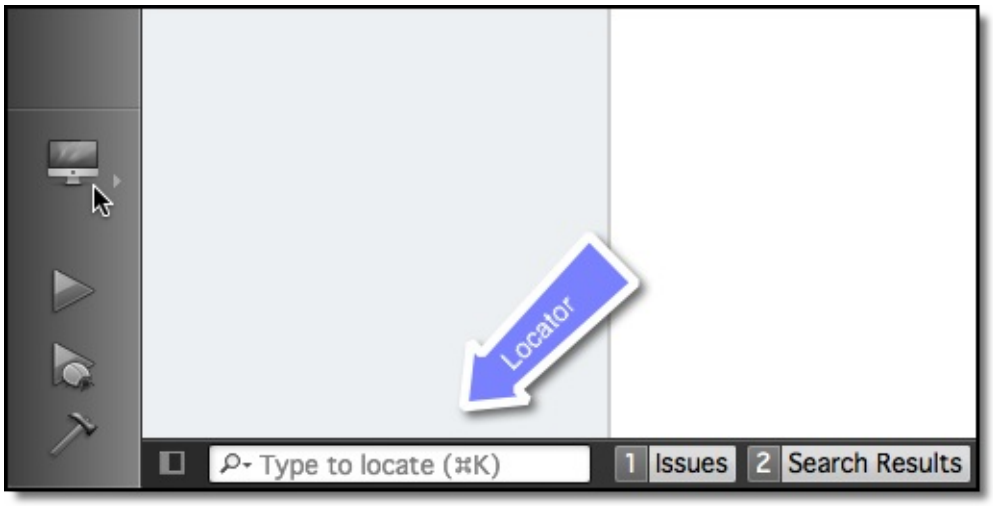
使用編輯器（Using the Editor）

當你打開一個項目或者創建一個新的項目後，Qt Creator將會轉換到編輯模式下。你應該可以在左邊看到你的項目文件，在中央區域看到代碼編輯器。左邊選中的文件將會被編輯器打開。編輯器提供了語法高亮，代碼補全和智能糾錯的功能。也提供幾種代碼重構的命令。當你使用這個編輯器工作時你會覺得它的響應非常的迅速。這感謝與Qt Creaotor的開發者將這個工具做的如此傑出。

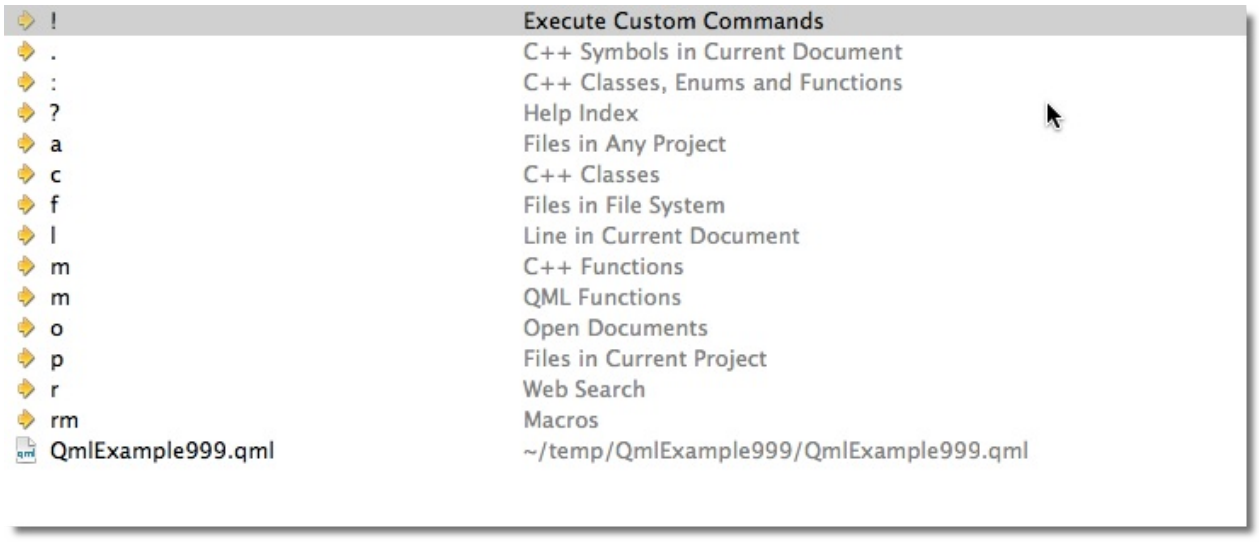


定位器（Locator）

定位器是Qt Creaotor中心的一個組件。它可以讓開發者迅速的找到指定代碼的位置，或者獲得幫助。使用Ctrl+K來打開定位器。



左邊底部可以顯示彈出一系列的選項。如果你只是想搜索你項目中的一個文件，你只需要給出文件第一個字母提示就可以了。定位器也接收通配符，比如*main.qml也可以查找。你也可以通過前綴搜索來搜索指定內容的類型。



試試它，例如尋找一個QML矩形框的幫助，輸入?rectangle。定位器會不停的更新它的建議直到你找到你想要的參考文檔。

調試（Debugging）

Qt Creator支持C++與QML代碼調試。

注意

嗯，我才意識到我還沒有使用過調試。這是一個好的現象。我需要有人對此提出問題，查看**Qt Creator documentation**來獲得更多的幫助吧。

快捷鍵（Shortcuts）

在好使用的系統中和專業系統中，快捷鍵是不同的。作為專業的開發人員，你也許會在你的應用程序上花很多時間，每一個快捷鍵都能使你的工作效率得到提高。Qt Creator的開發者也這樣想，並且在應用程序中加入了許許多多的快捷鍵。

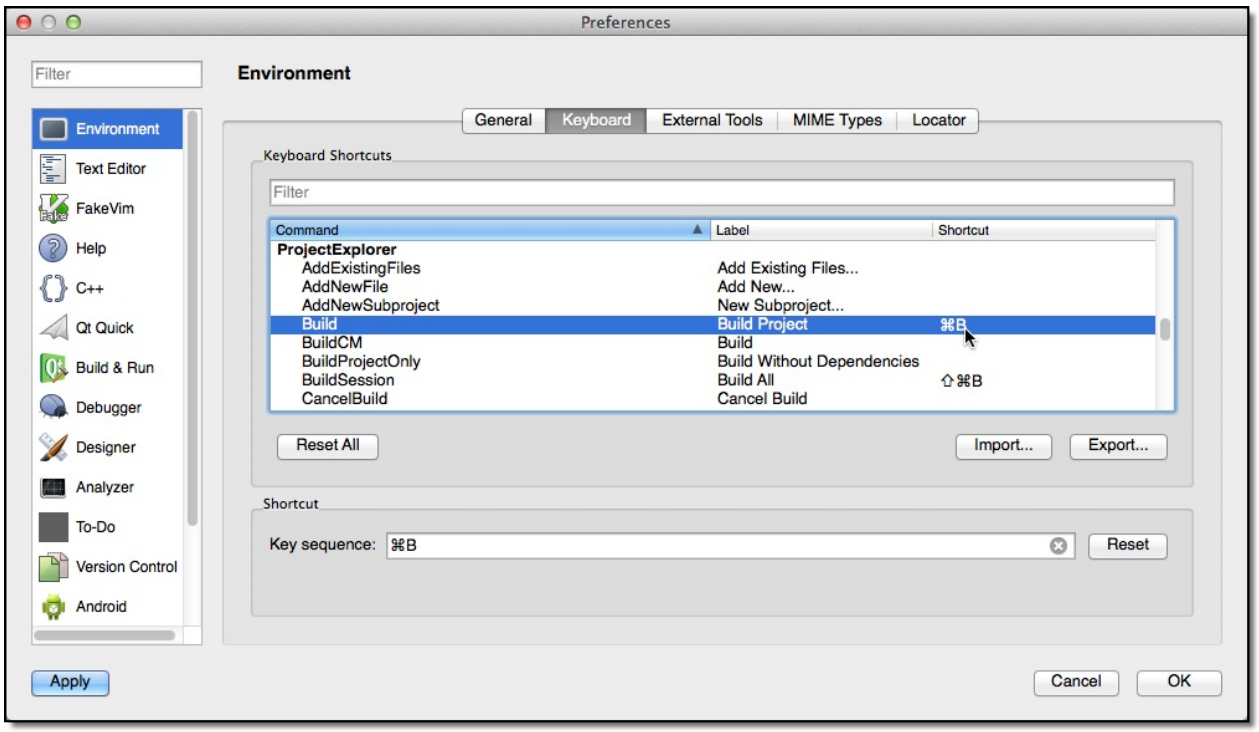
我們列出了一些基本的快捷鍵操作：

- Ctrl+B - 構建項目
- Ctrl+R - 運行項目
- Ctrl+Tab - 切換已打開的文檔
- Ctrl+k - 打開定位器
- Esc - 返回
- F2 - 查找對應的符號解釋。
- F4 - 在頭文件與源文件之間切換（只對c++代碼有效）

這些快捷鍵的定義來自Qt Creator shortcuts這個文檔。

注意

你可以使用設置窗口來編輯你的快捷鍵。



Quick Starter

Quick Starter

注意

最後一次構建：**2014**年**1**月**20**日下午**18:00**。

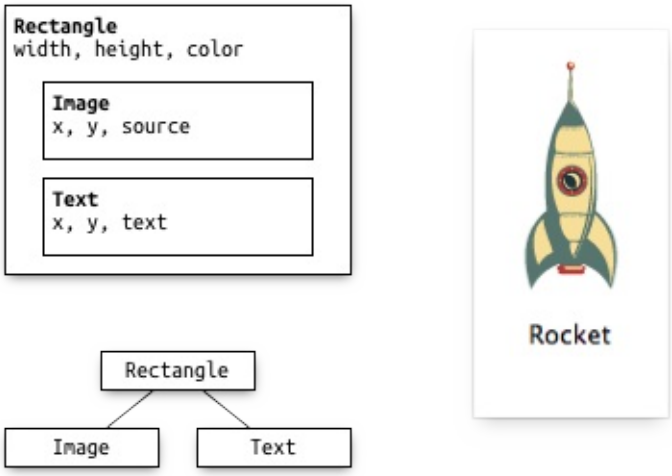
這章的源代碼能夠在**assetts folder**找到。

這章概述了QML語言，Qt5中大量使用了這種聲明用戶界面的語言。我們將會討論QML語言，一個樹形結構的元素，跟著是一些最基本的元素概述。然後我們會簡短的介紹怎樣創建我們自己的元素，這些元素被叫做組件，並如何使用屬性操作來轉換元素。最後我們會介紹如何對元素進行布局，如何向用戶提供輸入。

QML語法（QML Syntax）

QML是一種描述用戶界面的聲明式語言。它將用戶界面分解成一些更小的元素，這些元素能夠結合成一個組件。QML語言描述了用戶界面元素的形狀和行為。用戶界面能夠使用JavaScript來提供修飾，或者增加更加復雜的邏輯。從這個角度來看它遵循HTML-JavaScript模式，但QML是被設計用來描述用戶界面的，而不是文本文檔。

從QML元素的層次結構來理解是最簡單的學習方式。子元素從父元素上繼承了坐標系統，它的x,y坐標總是相對應于它的父元素坐標系統。



讓我們開始用一個簡單的QML文件例子來解釋這個語法。

```
// rectangle.qml

import QtQuick 2.0

// The root element is the Rectangle
Rectangle {
    // name this element root
    id: root

    // properties: <name>: <value>
    width: 120; height: 240

    // color property
    color: "#D8D8D8"

    // Declare a nested element (child of root)
    Image {
        id: rocket

        // reference the parent
        x: (parent.width - width)/2; y: 40

        source: 'assets/rocket.png'
    }

    // Another child of root
    Text {
        // un-named element
```

```
        // reference element by id
        y: rocket.y + rocket.height + 20

        // reference root element
        width: root.width

        horizontalAlignment: Text.AlignHCenter
        text: 'Rocket'
    }
}
```

- import聲明導入了一個指定的模塊版本。一般來說會導入QtQuick2.0來作為初始元素的引用。
- 使用//可以單行注釋，使用/**/可以多行注釋，就像C/C++和JavaScript一樣。
- 每一個QML文件都需要一個根元素，就像HTML一樣。
- 一個元素使用它的類型聲明，然後使用{}進行包含。
- 元素擁有屬性，他們按照name:value的格式來賦值。
- 任何在QML文檔中的元素都可以使用它們的id進行訪問（id是一個任意的標識符）。
- 元素可以嵌套，這意味著一個父元素可以擁有多個子元素。子元素可以通過訪問parent關鍵字來訪問它們的父元素。

建議

你會經常使用**id**或者關鍵字**parent**來訪問你的父對象。有一個比較好的方法是命名你的根元素對象**id**為**root**（**id:root**），這樣就不用去思考你的**QML**文檔中的根元素應該用什麼方式命名了。

提示

你可以在你的操作系統命令行模式下使用**QtQuick**運行環境來運行這個例子，比如像下面這樣：

```
$ $QTDIR/bin/qmlscene rectangle.qml
```

將$QTDIR替換為你的Qt的安裝路徑。qmlscene會執行Qt Quick運行環境初始化，並且解釋這個QML文件。

在Qt Creator中你可以打開對應的項目文件然後運行rectangle.qml文檔。

4.1.1 屬性（Properties）

元素使用他們的元素類型名進行聲明，使用它們的屬性或者創建自定義屬性來定義。一個屬性對應一個值，例如 width:100，text: 'Greeting', color: '#FF0000'。一個屬性有一個類型定義並且需要一個初始值。

```
    Text {
        // (1) identifier
        id: thisLabel

        // (2) set x- and y-position
        x: 24; y: 16
```

```
        // (3) bind height to 2 * width
        height: 2 * width

        // (4) custom property
        property int times: 24

        // (5) property alias
        property alias anotherTimes: thisLabel.times

        // (6) set text appended by value
        text: "Greetings " + times

        // (7) font is a grouped property
        font.family: "Ubuntu"
        font.pixelSize: 24

        // (8) KeyNavigation is an attached property
        KeyNavigation.tab: otherLabel

        // (9) signal handler for property changes
        onHeightChanged: console.log('height:', height)

        // focus is neeed to receive key events
        focus: true

        // change color based on focus value
        color: focus?"red":"black"
    }
```

讓我們來看看不同屬性的特點：

1. id是一個非常特殊的屬性值，它在一個QML文件中被用來引用元素。id不是一個字符串，而是一個標識符和QML語法的一部分。一個id在一個QML文檔中是唯一的，並且不能被設置為其它值，也無法被查詢（它的行為更像C++世界裡的指針）。

2. 一個屬性能夠設置一個值，這個值依賴于它的類型。如果沒有對一個屬性賦值，那麼它將會被初始化為一個默認值。你可以查看特定的元素的文檔來獲得這些初始值的信息。

3. 一個屬性能夠依賴一個或多個其它的屬性，這種操作稱作屬性綁定。當它依賴的屬性改變時，它的值也會更新。這就像訂了一個協議，在這個例子中height始終是width的兩倍。

4. 添加自己定義的屬性需要使用property修飾符，然後跟上類型，名字和可選擇的初始化值（property : ）。如果沒有初始值將會給定一個系統初始值作為初始值。注意如果屬性名與已定義的默認屬性名不重復，使用**default**關鍵字你可以將一個屬性定義為默認屬性。這在你添加子元素時用得著，如果他們是可視化的元素，子元素會自動的添加默認屬性的子類型鏈表（**children property list**）。

5. 另一個重要的聲明屬性的方法是使用alias關鍵字（property alias : ）。alias關鍵字允許我們轉發一個屬性或者轉發一個屬性對象自身到另一個作用域。我們將在後面定義組件導出內部屬性或者引用根級元素id會使用到這個技術。一個屬性別名不需要類型，它使用引用的屬性類型或者對象類型。

6. text屬性依賴于自定義的timers（int整型數據類型）屬性。int整型數據會自動的轉換為string字符串類型數據。這樣的表達方式本身也是另一種屬性綁定的例子，文本結果會在times屬性每次改變時刷新。

7. 一些屬性是按組分配的屬性。當一個屬性需要結構化並且相關的屬性需要聯系在一起時，我們可以這樣使用它。另一個組屬性的編碼方式是 font{family: "UBuntu"; pixelSize: 24 }。

8. 一些屬性是元素自身的附加屬性。這樣做是為了全局的相關元素在應用程序中只出現一次（例如鍵盤輸入）。編碼方式.: 。

9. 對于每個元素你都可以提供一個信號操作。這個操作在屬性值改變時被調用。例如這裡我們完成了當height（高度）改變時會使用控制台輸出一個信息。

警告

一個元素**id**應該只在當前文檔中被引用。**QML**提供了動態作用域的機制，後加載的文檔會覆蓋之前加載文檔的元素**id**號，這樣就可以引用已加載並且沒有被覆蓋的元素**id**，這有點類似創建全局變量。但不幸的是這樣的代碼閱讀性很差。目前這個還沒有辦法解決這個問題，所以你使用這個機制的時候最好仔細一些甚至不要使用這種機制。如果你想向文檔外提供元素的調用，你可以在根元素上使用屬性導出的方式來提供。

4.1.2 腳本（Scripting）

QML與JavaScript是最好的配合。在JavaScrpit的章節中我們將會更加詳細的介紹這種關系，現在我們只需要了解這種關系就可以了。

```
    Text {
        id: label

        x: 24; y: 24

        // custom counter property for space presses
        property int spacePresses: 0

        text: "Space pressed: " + spacePresses + " times"

        // (1) handler for text changes
        onTextChanged: console.log("text changed to:", text)

        // need focus to receive key events
        focus: true

        // (2) handler with some JS
        Keys.onSpacePressed: {
            increment()
        }

        // clear the text on escape
        Keys.onEscapePressed: {
            label.text = ''
        }

        // (3) a JS function
        function increment() {
            spacePresses = spacePresses + 1
        }
    }
```

1. 文本改變操作onTextChanged會將每次空格鍵按下導致的文本改變輸出到控制台。

2. 當文本元素接收到空格鍵操作（用戶在鍵盤上點擊空格鍵），會調用JavaScript函數increment()。

3. 定義一個JavaScript函數使用這種格式function (){....}，在這個例子中是增加spacePressed的計數。每

次spacePressed的增加都會導致它綁定的屬性更新。

注意

QML的：（屬性綁定）與**JavaScript**的**=**（賦值）是不同的。綁定是一個協議，並且存在于整個生命週期。然而**JavaScript**賦值（**=**）只會產生一次效果。當一個新的綁定生效或者使用**JavaScript**賦值給屬性時，綁定的生命週期就會結束。例如一個按鍵的操作設置文本屬性為一個空的字符串將會銷毀我們的增值顯示：

```
Keys.onEscapePressed: {
    label.text = ''
}
```

在點擊取消（**ESC**）後，再次點擊空格鍵（**space-bar**）將不會更新我們的顯示，之前的**text**屬性綁定（**text: "Space pressed:" + spacePresses + "times")**被銷毀。

當你對改變屬性的策略有衝突時（文本的改變基于一個增值的綁定並且可以被**JavaScript**賦值清零），類似于這個例子，你最好不要使用綁定屬性。你需要使用賦值的方式來改變屬性，屬性綁定會在賦值操作後被銷毀（銷毀協議！）。

基本元素（**Basic Elements**）

元素可以被分為可視化元素與非可視化元素。一個可視化元素（例如矩形框Rectangle）有著幾何形狀並且可以在屏幕上顯示。一個非可視化元素（例如計時器Timer）提供了常用的功能，通常用于操作可視化元素。

現在我們將專注于幾個基礎的可視化元素，例如Item（基礎元素對象），Rectangle（矩形框），Text（文本），Image（圖像）和MouseArea（鼠標區域）。

4.2.1 基礎元素對象（**Item Element**）

Item（基礎元素對象）是所有可視化元素的基礎對象，所有其它的可視化元素都繼承自Item。它自身不會有任何繪制操作，但是定義了所有可視化元素共有的屬性：

Group（分組）	**Properties**（屬性）
Geometry（幾何屬性）	x,y（坐標）定義了元素左上角的位置，width，height（長和寬）定義元素的顯示範圍，z（堆疊次序）定義元素之間的重疊順序。
Layout handling（布局操作）	anchors（錨定），包括左（left），右（right），上（top），下（bottom），水平與垂直居中（vertical center，horizontal center），與margins（間距）一起定義了元素與其它元素之間的位置關系。
Key handlikng（按鍵操作）	附加屬性key（按鍵）和keyNavigation（按鍵定位）屬性來控制按鍵操作，處理輸入焦點（focus）可用操作。
Transformation（轉換）	縮放（scale）和rotate（旋轉）轉換，通用的x,y,z屬性列表轉換（transform），旋轉基點設置（transformOrigin）。
Visual（可視化）	不透明度（opacity）控制透明度，visible（是否可見）控制元素是否顯示，clip（裁剪）用來限制元素邊界的繪制，smooth（平滑）用來提高渲染質量。
State definition（狀態定義）	states（狀態列表屬性）提供了元素當前所支持的狀態列表，當前屬性的改變也可以使用transitions（轉變）屬性列表來定義狀態轉變動畫。

為了更好的理解不同的屬性，我們將會在這章中盡量的介紹這些元素的顯示效果。請記住這些基本的屬性在所有可視化元素中都是可以使用的，並且在這些元素中的工作方式都是相同的。

注意

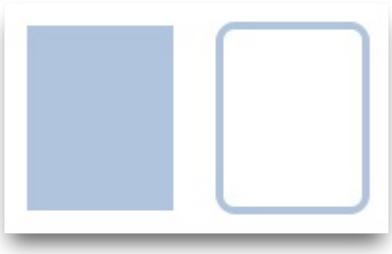
Item（基本元素對象）通常被用來作為其它元素的容器使用，類似**HTML**語言中的**div**元素（**div element**）。

4.2.2 矩形框元素（**Rectangle Element**）

Rectangle（矩形框）是基本元素對象的一個擴展，增加了一個顏色來填充它。它還支持邊界的定義，使用border.color（邊界顏色），border.width（邊界寬度）來自定義邊界。你可以使用radius（半徑）屬性來創建一個圓角矩形。

```
    Rectangle {
        id: rect1
        x: 12; y: 12
        width: 76; height: 96
        color: "lightsteelblue"
    }
    Rectangle {
```

```
        id: rect2
        x: 112; y: 12
        width: 76; height: 96
        border.color: "lightsteelblue"
        border.width: 4
        radius: 8
    }
```

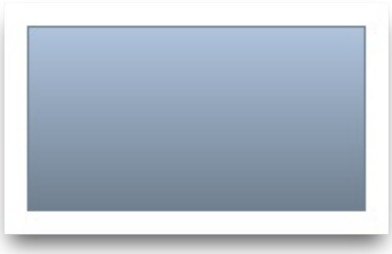


注意

顏色的命名是來自**SVG**顏色的名稱（查看**http://www.w3.org/TR/css3-color/#svg-color**可以獲取更多的顏色名稱）。你也可以使用其它的方法來指定顏色，比如**RGB**字符串（**'#FF4444'**），或者一個顏色名字（例如**'white'**）。

此外，填充的顏色與矩形的邊框也支持自定義的漸變色。

```
    Rectangle {
        id: rect1
        x: 12; y: 12
        width: 176; height: 96
        gradient: Gradient {
            GradientStop { position: 0.0; color: "lightsteelblue" }
            GradientStop { position: 1.0; color: "slategray" }
        }
        border.color: "slategray"
    }
```



一個漸變色是由一系列的梯度值定義的。每一個值定義了一個位置與顏色。位置標記了y軸上的位置（0 = 頂，1 = 底）。GradientStop（傾斜點）的顏色標記了顏色的位置。

注意

一個矩形框如果沒有**width/height**（寬度與高度）將不可見。如果你有幾個相互關聯**width/height**（寬度與高度）的矩形框，在你組合邏輯中出了錯後可能就會發生矩形框不可見，請注意這一點。

注意

這個函數無法創建一個梯形，最好使用一個已有的圖像來創建梯形。有一種可能是在旋轉梯形時，旋轉的

矩形幾何結構不會發生改變，但是這會導致幾何元素相同的可見區域的混淆。從作者的觀點來看類似的情況下最好使用設計好的梯形圖形來完成繪制。

4.2.3 文本元素（Text Element）

顯示文本你需要使用Text元素（Text Element）。它最值得注意的屬性時字符串類型的text屬性。這個元素會使用給出的text（文本）與font（字體）來計算初始化的寬度與高度。可以使用字體屬性組來（font property group）來改變當前的字體，例如font.family，font.pixelSize，等等。改變文本的顏色值只需要改變顏色屬性就可以了。

```
    Text {
        text: "The quick brown fox"
        color: "#303030"
        font.family: "Ubuntu"
        font.pixelSize: 28
    }
```

文本可以使用horizontalAlignment與verticalAlignment屬性來設置它的對齊效果。為了提高文本的渲染效果，你可以使用style和styleColor屬性來配置文字的外框效果，浮雕效果或者凹陷效果。對于過長的文本，你可能需要使用省略號來表示，例如A very ... long text，你可以使用elide屬性來完成這個操作。elide屬性允許你設置文本左邊，右邊或者中間的省略位置。如果你不想'....'省略號出現，並且希望使用文字換行的方式顯示所有的文本，你可以使用wrapMode屬性（這個屬性只在明確設置了寬度後才生效）：

```
Text {
    width: 40; height: 120
    text: 'A very long text'
    // '...' shall appear in the middle
    elide: Text.ElideMiddle
    // red sunken text styling
    style: Text.Sunken
    styleColor: '#FF4444'
    // align text to the top
    verticalAlignment: Text.AlignTop
    // only sensible when no elide mode
    // wrapMode: Text.WordWrap
}
```

一個text元素（text element）只顯示的文本，它不會渲染任何背景修飾。除了顯示的文本，text元素背景是透明的。為一個文本元素提供背景是你自己需要考慮的問題。

注意

知道一個文本元素（**Text Element**）的初始寬度與高度是依賴于文本字符串和設置的字體這一點很重要。一個沒有設置寬度或者文本的文本元素（**Text Element**）將不可見，默認的初始寬度是**0**。

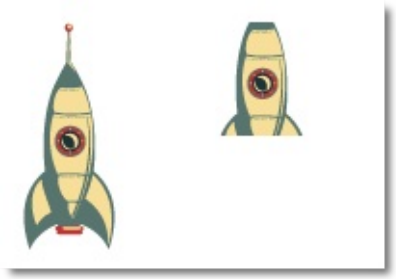
注意

通常你想要對文本元素布局時，你需要區分文本在文本元素內部的邊界對齊和由元素邊界自動對齊。前一種情況你需要使用**horizontalAlignment**和**verticalAlignment**屬性來完成，後一種情況你需要操作元素的幾何形狀或者使用**anchors**（錨定）來完成。

4.2.4 圖像元素（Image Element）

一個圖像元素（Image Element）能夠顯示不同格式的圖像（例如PNG,JPG,GIF,BMP）。想要知道更加詳細的圖像格式支持信息，可以查看Qt的相關文檔。source屬性（source property）提供了圖像文件的鏈接信息，fillMode（文件模式）屬性能夠控制元素對象的大小調整行為。

```
    Image {
        x: 12; y: 12
        // width: 48
        // height: 118
        source: "assets/rocket.png"
    }
    Image {
        x: 112; y: 12
        width: 48
        height: 118/2
        source: "assets/rocket.png"
        fillMode: Image.PreserveAspectCrop
        clip: true
    }
```



注意

一個**URL**可以是使用**'/'**語法的本地路徑（**"./images/home.png"**）或者一個網絡鏈接（**"http://example.org/home.png"**）。

注意

圖像元素（**Image element**）使用**PreserveAspectCrop**可以避免裁剪圖像數據被渲染到圖像邊界外。默認情況下裁剪是被禁用的（**clip:false**）。你需要打開裁剪（**clip:true**）來約束邊界矩形的繪制。這對任何可視化元素都是有效的。

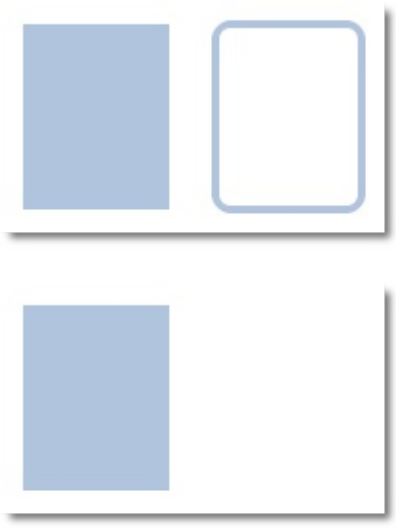
建議

使用**QQmlImageProvider**你可以通過**C++**代碼來創建自己的圖像提供器，這允許你動態創建圖像並且使用線程加載。

4.2.5 鼠標區域元素（MouseArea Element）

為了與不同的元素交互，你通常需要使用MouseArea（鼠標區域）元素。這是一個矩形的非可視化元素對象，你可以通過它來捕捉鼠標事件。當用戶與可視化端口交互時，mouseArea通常被用來與可視化元素對象一起執行命令。

```
    Rectangle {
        id: rect1
        x: 12; y: 12
        width: 76; height: 96
        color: "lightsteelblue"
        MouseArea {
            id: area
            width: parent.width
            height: parent.height
            onClicked: rect2.visible = !rect2.visible
        }
    }

    Rectangle {
        id: rect2
        x: 112; y: 12
        width: 76; height: 96
        border.color: "lightsteelblue"
        border.width: 4
        radius: 8
    }
```



注意

這是**QtQuick**中非常重要的概念，輸入處理與可視化顯示分開。這樣你的交互區域可以比你顯示的區域大很多。

組件（Compontents）

一個組件是一個可以重復使用的元素，QML提供幾種不同的方法來創建組件。但是目前我們只對其中一種方法進行講解：一個文件就是一個基礎組件。一個以文件為基礎的組件在文件中創建了一個QML元素，並且將文件以元素類型來命名（例如Button.qml）。你可以像任何其它的QtQuick模塊中使用元素一樣來使用這個組件。在我們下面的例子中，你將會使用你的代碼作為一個Button（按鈕）來使用。

讓我們來看看這個例子，我們創建了一個包含文本和鼠標區域的矩形框。它類似于一個簡單的按鈕，我們的目標就是讓它足夠簡單。

```
    Rectangle { // our inlined button ui
        id: button
        x: 12; y: 12
        width: 116; height: 26
        color: "lightsteelblue"
        border.color: "slategrey"
        Text {
            anchors.centerIn: parent
            text: "Start"
        }
        MouseArea {
            anchors.fill: parent
            onClicked: {
                status.text = "Button clicked!"
            }
        }
    }

    Text { // text changes when button was clicked
        id: status
        x: 12; y: 76
        width: 116; height: 26
        text: "waiting ..."
        horizontalAlignment: Text.AlignHCenter
    }
```

用戶界面將會看起來像下面這樣。左邊是初始化的狀態，右邊是按鈕點擊後的效果。

我們的目標是提取這個按鈕作為一個可重復使用的組件。我們可以簡單的考慮一下我們的按鈕會有的哪些

API（應用程序接口），你可以自己考慮一下你的按鈕應該有些什麼。下面是我考慮的結果：

```
// my ideal minimal API for a button
Button {
    text: "Click Me"
    onClicked: { // do something }
}
```

我想要使用text屬性來設置文本，然後實現我們自己的點擊操作。我也期望這個按鈕有一個比較合適的初始化大小（例如width:240）。 為了完成我們的目標，我創建了一個Button.qml文件，並且將我們的代碼拷貝了進去。我們在根級添加一個屬性導出方便使用者修改它。

我們在根級導出了文本和點擊信號。通常我們命名根元素為root讓引用更加方便。我們使用了QML的alias（別名）的功能，它可以將內部嵌套的QML元素的屬性導出到外面使用。有一點很重要，只有根級目錄的屬性才能夠被其它文件的組件訪問。

```
// Button.qml

import QtQuick 2.0

Rectangle {
    id: root
    // export button properties
    property alias text: label.text
    signal clicked

    width: 116; height: 26
    color: "lightsteelblue"
    border.color: "slategrey"

    Text {
        id: label
        anchors.centerIn: parent
        text: "Start"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            root.clicked()
        }
    }
}
```

使用我們新的Button元素只需要在我們的文件中簡單的聲明一下就可以了，之前的例子將會被簡化。

```
    Button { // our Button component
        id: button
        x: 12; y: 12
        text: "Start"
        onClicked: {
            status.text = "Button clicked!"
        }
    }

    Text { // text changes when button was clicked
        id: status
```

```
        x: 12; y: 76
        width: 116; height: 26
        text: "waiting ..."
        horizontalAlignment: Text.AlignHCenter
    }
```

現在你可以在你的用戶界面代碼中隨意的使用Button{ ...}來作為按鈕了。一個真正的按鈕將更加復雜，比如提供按鍵反饋或者添加一些裝飾。

注意

就個人而言，可以更進一步的使用基礎元素對象（**Item**）作為根元素。這樣可以防止用戶改變我們設計的按鈕的顏色，並且可以提供出更多相關控制的**API**（應用程序接口）。我們的目標是導出一個最小的**API**（應用程序接口）。實際上我們可以將根矩形框（**Rectangle**）替換為一個基礎元素（**Item**），然後將一個矩形框（**Rectangle**）嵌套在這個根元素（**root item**）就可以完成了。

```
Item {
    id: root
    Rectangle {
        anchors.fill parent
        color: "lightsteelblue"
        border.color: "slategrey"
    }
    ...
}
```

使用這項技術可以很簡單的創建一系列可重用的組件。

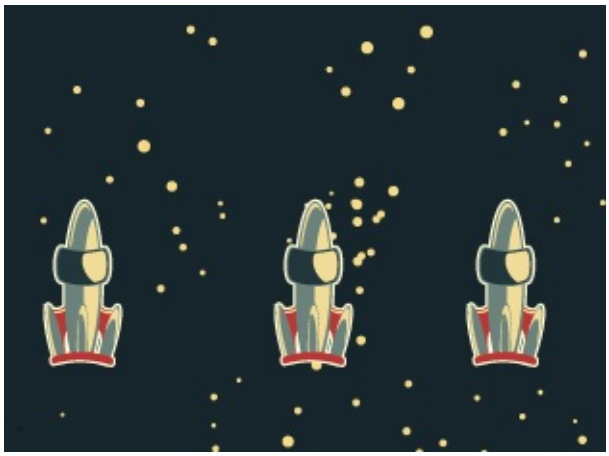
簡單的轉換（Simple Transformations）

轉換操作改變了一個對象的幾何狀態。QML元素對象通常能夠被平移，旋轉，縮放。下面我們將講解這些簡單的操作和一些更高級的用法。 我們先從一個簡單的轉換開始。用下面的場景作為我們學習的開始。

簡單的位移是通過改變x,y坐標來完成的。旋轉是改變rotation（旋轉）屬性來完成的，這個值使用角度作為單位（0~360）。縮放是通過改變scale（比例）的屬性來完成的，小于1意味著縮小，大于1意味著放大。旋轉與縮放不會改變對象的幾何形狀，對象的x,y（坐標）與width/height（寬/高）也類似。只有繪制指令是被轉換的對象。

在我們展示例子之前我想要介紹一些東西：ClickableImage元素（ClickableImage element），ClickableImage僅僅是一個包含鼠標區域的圖像元素。我們遵循一個簡單的原則，三次使用相同的代碼描述一個用戶界面最好可以抽象為一個組件。

```
// ClickableImage.qml

// Simple image which can be clicked

import QtQuick 2.0

Image {
    id: root
    signal clicked

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



我們使用我們可點擊圖片元素來顯示了三個火箭。當點擊時，每個火箭執行一種簡單的轉換。點擊背景將會重置場景。

```
// transformation.qml


import QtQuick 2.0

Item {
    // set width based on given background
```

```
    width: bg.width
    height: bg.height

    Image { // nice background image
        id: bg
        source: "assets/background.png"
    }

    MouseArea {
        id: backgroundClicker
        // needs to be before the images as order matters
        // otherwise this mousearea would be before the other elements
        // and consume the mouse events
        anchors.fill: parent
        onClicked: {
            // reset our little scene
            rocket1.x = 20
            rocket2.rotation = 0
            rocket3.rotation = 0
            rocket3.scale = 1.0
        }
    }

    ClickableImage {
        id: rocket1
        x: 20; y: 100
        source: "assets/rocket.png"
        onClicked: {
            // increase the x-position on click
            x += 5
        }
    }

    ClickableImage {
        id: rocket2
        x: 140; y: 100
        source: "assets/rocket.png"
        smooth: true // need antialising
        onClicked: {
            // increase the rotation on click
            rotation += 5
        }
    }

    ClickableImage {
        id: rocket3
        x: 240; y: 100
        source: "assets/rocket.png"
        smooth: true // need antialising
        onClicked: {
            // several transformations
            rotation += 5
            scale -= 0.05
        }
    }
}
```

火箭1在每次點擊後X軸坐標增加5像素，火箭2每次點擊後會旋轉。火箭3每次點擊後會縮小。對于縮放和旋轉操作我們都設置了smooth:true來增加反鋸齒，由于性能的原因通常是被關閉的（與剪裁屬性clip類似）。當你看到你的圖形中出現鋸齒時，你可能就需要打開平滑（smooth）。

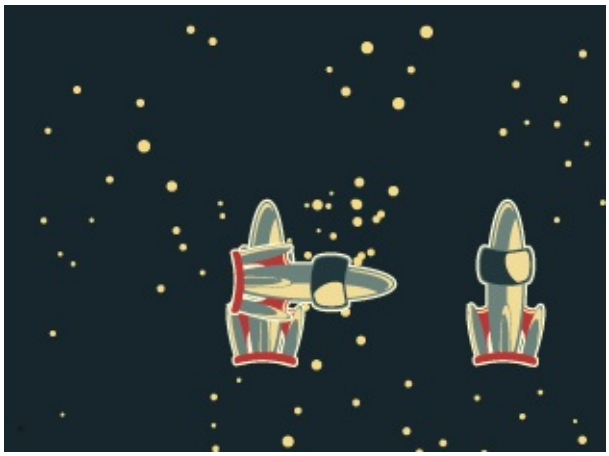
注意

為了獲得更好的顯示效果，當縮放圖片時推薦使用已縮放的圖片來替代，過量的放大可能會導致圖片模糊不清。當你在縮放圖片時你最好考慮使用**smooth:true**來提高圖片顯示質量。

使用MouseArea來覆蓋整個背景，點擊背景可以初始化火箭的值。

注意

在代碼中先出現的元素有更低的堆疊順序（叫做**z**順序值**z-order**），如果你點擊火箭**1**足夠多次，你會看見火箭**1**移動到了火箭**2**下面。**z**軸順序也可以使用元素對象的**z-property**來控制。



由于火箭**2**後出現在代碼中，火箭**2**將會放在火箭**1**上面。這同樣適用于**MouseArea**（鼠標區域），一個後出現在代碼中的鼠標區域將會與之前的鼠標區域重疊，後出現的鼠標區域才能捕捉到鼠標事件。

請記住：文檔中元素的順序很重要。

定位元素（**Positioning Element**）

有一些QML元素被用于放置元素對象，它們被稱作定位器，QtQuick模塊提供了Row，Column，Grid，Flow用來作為定位器。你可以在下面的插圖中看到它們使用相同內容的顯示效果。

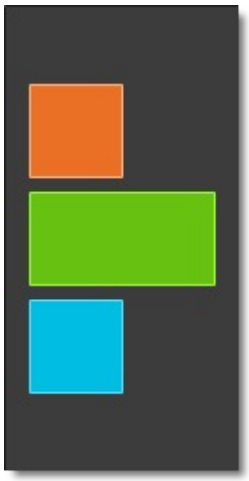
注意

在我們詳細介紹前，我們先介紹一些相關的元素，紅色（**red**），藍色（**blue**），綠色（**green**），高亮（**lighter**）與黑暗（**darker**）方塊，每一個組件都包含了一個**48**乘**48**的著色區域。下面是關于**RedSquare**（紅色方塊）的代碼：

```
// RedSquare.qml

import QtQuick 2.0

Rectangle {
    width: 48
    height: 48
    color: "#ea7025"
    border.color: Qt.lighter(color)
}
```

請注意使用了**Qt.lighter**（**color**）來指定了基于填充色的邊界高亮色。我們將會在後面的例子中使用到這些元素，希望後面的代碼能夠容易讀懂。請記住每一個矩形框的初始化大小都是**48**乘**48**像素大小。

Column（列）元素將它的子對象通過頂部對齊的列方式進行排列。spacing屬性用來設置每個元素之間的間隔大小。



```
// column.qml

import QtQuick 2.0

DarkSquare {
    id: root
    width: 120
    height: 240

    Column {
        id: column
```

```
        anchors.centerIn: parent
        spacing: 8
        RedSquare { }
        GreenSquare { width: 96 }
        BlueSquare { }
    }
}

// M1<<
```

Row（行）元素將它的子對象從左到右，或者從右到左依次排列，排列方式取決于layoutDirection屬性。spacing屬性用來設置每個元素之間的間隔大小。

```
// row.qml

import QtQuick 2.0

BrightSquare {
    id: root
    width: 400; height: 120

    Row {
        id: row
        anchors.centerIn: parent
        spacing: 20
        BlueSquare { }
        GreenSquare { }
        RedSquare { }
    }
}
```

Grid（柵格）元素通過設置rows（行數）和columns（列數）將子對象排列在一個柵格中。可以只限制行數或者列數。如果沒有設置它們中的任意一個，柵格元素會自動計算子項目總數來獲得配置，例如，設置rows（行數）為3，添加了6個子項目到元素中，那麼會自動計算columns（列數）為2。屬性flow（流）與layoutDirection（布局方向）用來控制子元素的加入順序。spacing屬性用來控制所有元素之間的間隔。

```
// grid.qml
```

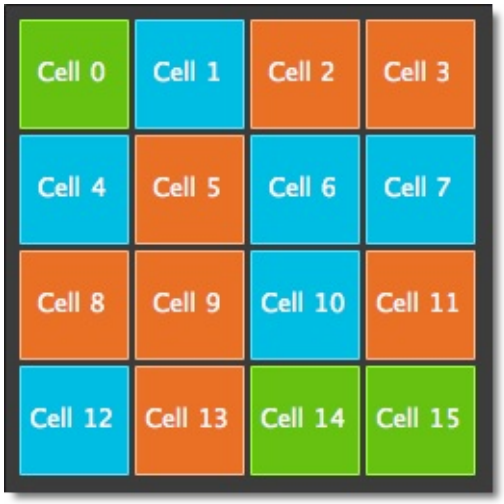
```
import QtQuick 2.0

BrightSquare {
    id: root
    width: 160
    height: 160

    Grid {
        id: grid
        rows: 2
        columns: 2
        anchors.centerIn: parent
        spacing: 8
        RedSquare { }
        RedSquare { }
        RedSquare { }
        RedSquare { }
    }

}
```

最後一個定位器是Flow（流）。通過flow（流）屬性和layoutDirection（布局方向）屬性來控制流的方向。它能夠從頭到底的橫向布局，也可以從左到右或者從右到左進行布局。作為加入流中的子對象，它們在需要時可以被包裝成新的行或者列。為了讓一個流可以工作，必須指定一個寬度或者高度，可以通過屬性直接設定，或者通過anchor（錨定）布局設置。

```
// flow.qml

import QtQuick 2.0

BrightSquare {
    id: root
    width: 160
    height: 160

    Flow {
        anchors.fill: parent
        anchors.margins: 20
        spacing: 20
        RedSquare { }
        BlueSquare { }
        GreenSquare { }
    }
}
```

通常Repeater（重復元素）與定位器一起使用。它的工作方式就像for循環與迭代器的模式一樣。在這個最

簡單的例子中，僅僅提供了一個循環的例子。




```
// repeater.qml

import QtQuick 2.0

DarkSquare {
    id: root
    width: 252
    height: 252
    property variant colorArray: ["#00bde3", "#67c111", "#ea7025"]


    Grid{
        anchors.fill: parent
        anchors.margins: 8
        spacing: 4
        Repeater {
            model: 16
            Rectangle {
                width: 56; height: 56
                property int colorIndex: Math.floor(Math.random()*3)
                color: root.colorArray[colorIndex]
                border.color: Qt.lighter(color)
                Text {
                    anchors.centerIn: parent
                    color: "#f0f0f0"
                    text: "Cell " + index
                }
            }
        }
    }
}
```

在這個重復元素的例子中，我們使用了一些新的方法。我們使用一個顏色數組定義了一組顏色屬性。重復元素能夠創建一連串的矩形框（16個，就像模型中定義的那樣）。每一次的循環都會創建一個矩形框作為repeater的子對象。在矩形框中，我們使用了JS數學函數Math.floor(Math.random()*3)來選擇顏色。這個函數會給我們生成一個0~2的隨機數，我們使用這個數在我們的顏色數組中選擇顏色。注意之前我們說過JavaScript是QtQuick中的一部分，所以這些典型的庫函數我們都可以使用。

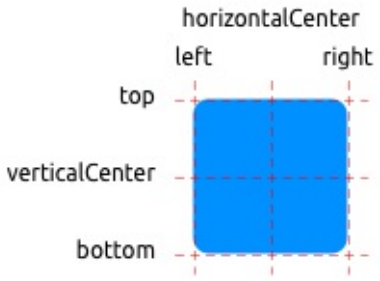
一個重復元素循環時有一個index（索引）屬性值。當前的循環索引（0,1,2,....15）。我們可以使用這個索引值來做一些操作，例如在我們這個例子中使用Text（文本）顯示當前索引值。

注意

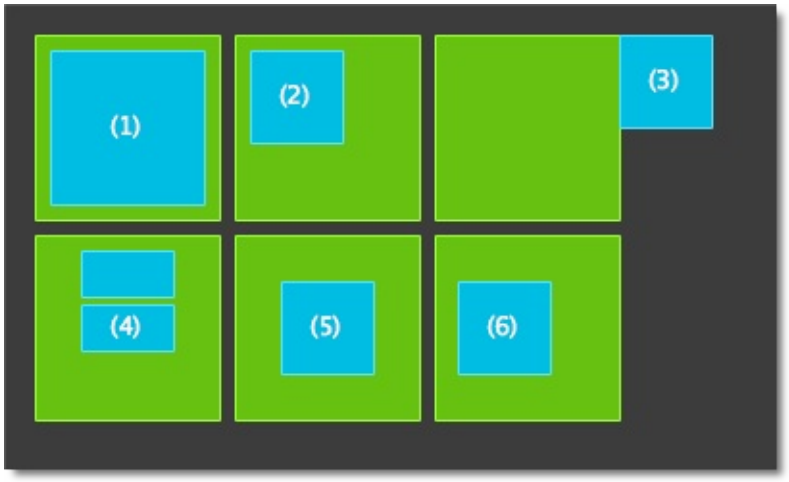
高級的大數據模型處理和使用動態代理的動態視圖會在模型與視圖（**model-view**）章節中講解。當有一小部分的靜態數據需要顯示時，使用重復元素是最好的方式。

布局元素（**Layout Items**）

QML使用anchors（錨）對元素進行布局。anchoring（錨定）是基礎元素對象的基本屬性，可以被所有的可視化QML元素使用。一個anchors（錨）就像一個協議，並且比幾何變化更加強大。Anchors（錨）是相對關系的表達式，你通常需要與其它元素搭配使用。



一個元素有6條錨定線（top頂，bottom底，left左，right右，horizontalCenter水平中，verticalCenter垂直中）。在文本元素（Text Element）中有一條文本的錨定基線（baseline）。每一條錨定線都有一個偏移（offset）值，在top（頂），bottom（底），left（左），right（右）的錨定線中它們也被稱作邊距。對于horizontalCenter（水平中）與verticalCenter（垂直中）與baseline（文本基線）中被稱作偏移值。



1. 元素填充它的父元素。

```
GreenSquare {
    BlueSquare {
        width: 12
        anchors.fill: parent
        anchors.margins: 8
        text: '(1)'
    }
}
```

2. 元素左對齊它的父元素。

```
GreenSquare {
    BlueSquare {
        width: 48
        y: 8
        anchors.left: parent.left
```

```
            anchors.leftMargin: 8
            text: '(2)'
        }
    }
```

3. 元素的左邊與它父元素的右邊對齊。

```
    GreenSquare {
        BlueSquare {
            width: 48
            anchors.left: parent.right
            text: '(3)'
        }
    }
```

4. 元素中間對齊。Blue1與它的父元素水平中間對齊。Blue2與Blue1中間對齊，並且它的頂部對齊Blue1的底部。

```
    GreenSquare {
        BlueSquare {
            id: blue1
            width: 48; height: 24
            y: 8
            anchors.horizontalCenter: parent.horizontalCenter
        }
        BlueSquare {
            id: blue2
            width: 72; height: 24
            anchors.top: blue1.bottom
            anchors.topMargin: 4
            anchors.horizontalCenter: blue1.horizontalCenter
            text: '(4)'
        }
    }
```

5. 元素在它的父元素中居中。

```
    GreenSquare {
        BlueSquare {
            width: 48
            anchors.centerIn: parent
            text: '(5)'
        }
    }
```

6. 元素水平方向居中對齊父元素並向後偏移12像素，垂直方向居中對齊。

```
    GreenSquare {
        BlueSquare {
            width: 48
            anchors.horizontalCenter: parent.horizontalCenter
            anchors.horizontalCenterOffset: -12
            anchors.verticalCenter: parent.verticalCenter
            text: '(6)'
```

```
        }
    }
```

注意

我們的方格都打開了拖拽。試著拖放幾個方格。你可以發現第一個方格無法被拖拽因為它每個邊都被固定了，當然第一個方格的父元素能夠被拖拽是因為它的父元素沒有被固定。第二個方格能夠在垂直方向上拖拽是因為它只有左邊被固定了。類似的第三個和第四個方格也只能在垂直方向上拖拽是因為它們都使用水平居中對齊。第五個方格使用居中布局，它也無法被移動，第六個方格與第五個方格類似。拖拽一個元素意味著會改變它的**x,y**坐標。**anchoring**（錨定）比幾何變化（例如**x,y**坐標變化）更強大是因為錨定線（**anchored lines**）的限制，我們將在後面討論動畫時看到這些功能的強大。

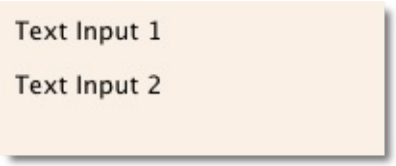
輸入元素（Input Element）

我們已經使用過MouseArea（鼠標區域）作為鼠標輸入元素。這裡我們將更多的介紹關于鍵盤輸入的一些東西。我們開始介紹文本編輯的元素：TextInput（文本輸入）和TextEdit（文本編輯）。

4.7.1 文本輸入（TextInput）

文本輸入允許用戶輸入一行文本。這個元素支持使用正則表達式驗證器來限制輸入和輸入掩碼的模式設置。

```
// textinput.qml

import QtQuick 2.0

Rectangle {
    width: 200
    height: 80
    color: "linen"

    TextInput {
        id: input1
        x: 8; y: 8
        width: 96; height: 20
        focus: true
        text: "Text Input 1"
    }

    TextInput {
        id: input2
        x: 8; y: 36
        width: 96; height: 20
        text: "Text Input 2"
    }
}
```



用戶可以通過點擊TextInput來改變焦點。為了支持鍵盤改變焦點，我們可以使用KeyNavigation（按鍵向導）這個附加屬性。

```
// textinput2.qml

import QtQuick 2.0

Rectangle {
    width: 200
    height: 80
    color: "linen"

    TextInput {
```

```
        id: input1
        x: 8; y: 8
        width: 96; height: 20
        focus: true
        text: "Text Input 1"
        KeyNavigation.tab: input2
    }

    TextInput {
        id: input2
        x: 8; y: 36
        width: 96; height: 20
        text: "Text Input 2"
        KeyNavigation.tab: input1
    }
}
```

KeyNavigation（按鍵向導）附加屬性可以預先設置一個元素id綁定切換焦點的按鍵。

一個文本輸入元素（text input element）只顯示一個閃爍符和已經輸入的文本。用戶需要一些可見的修飾來鑑別這是一個輸入元素，例如一個簡單的矩形框。當你放置一個TextInput（文本輸入）在一個元素中時，你需要確保其它的元素能夠訪問它導出的大多數屬性。

我們提取這一段代碼作為我們自己的組件，稱作TLineEditV1用來重復使用。

```
// TLineEditV1.qml

import QtQuick 2.0

Rectangle {
    width: 96; height: input.height + 8
    color: "lightsteelblue"
    border.color: "gray"

    property alias text: input.text
    property alias input: input

    TextInput {
        id: input
        anchors.fill: parent
        anchors.margins: 4
        focus: true
    }
}
```

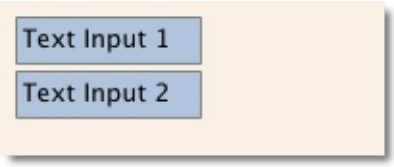
注意

如果你想要完整的導出**TextInput**元素，你可以使用**property alias input: input**來導出這個元素。第一個**input**是屬性名字，第二個**input**是元素**id**。

我們使用TLineEditV1組件重寫了我們的KeyNavigation（按鍵向導）的例子。

```
    Rectangle {
        ...
        TLineEditV1 {
            id: input1
            ...
```

```
        }
        TLineEditV1 {
            id: input2
            ...
        }
    }
```



嘗試使用Tab按鍵來導航，你會發現焦點無法切換到input2上。這個例子中使用focus:true的方法不正確，這個問題是因為焦點被轉移到input2元素時，包含TlineEditV1的頂部元素接收了這個焦點並且沒有將焦點轉發給TextInput（文本輸入）。為了防止這個問題，QML提供了FocusScope（焦點區域）。

4.7.2 焦點區域（FocusScope）

一個焦點區域（focus scope）定義了如果焦點區域接收到焦點，它的最後一個使用focus:true的子元素接收焦點，它將會把焦點傳遞給最後申請焦點的子元素。我們創建了第二個版本的TLineEdit組件，稱作TLineEditV2，使用焦點區域（focus scope）作為根元素。

```
// TLineEditV2.qml

import QtQuick 2.0

FocusScope {
    width: 96; height: input.height + 8
    Rectangle {
        anchors.fill: parent
        color: "lightsteelblue"
        border.color: "gray"

    }

    property alias text: input.text
    property alias input: input

    TextInput {
        id: input
        anchors.fill: parent
        anchors.margins: 4
        focus: true
    }
}
```

現在我們的例子將像下面這樣：

```
    Rectangle {
        ...
        TLineEditV2 {
            id: input1
            ...
        }
```

```
        TLineEditV2 {
            id: input2
            ...
        }
    }
```

按下Tab按鍵可以成功的在兩個組件之間切換焦點，並且能夠正確的將焦點鎖定在組件內部的子元素中。

4.7.3 文本編輯（TextEdit）

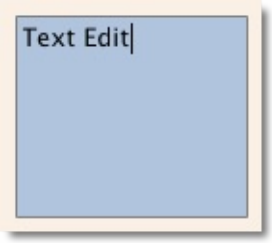
文本編輯（TextEdit）元素與文本輸入（TextInput）非常類似，它支持多行文本編輯。它不再支持文本輸入的限制，但是提供了已繪制文本的大小查詢（paintedHeight，paintedWidth）。我們也創建了一個我們自己的組件TTextEdit，可以編輯它的背景，使用focus scope（焦點區域）來更好的切換焦點。

```
// TTextEdit.qml

import QtQuick 2.0

FocusScope {
    width: 96; height: 96
    Rectangle {
        anchors.fill: parent
        color: "lightsteelblue"
        border.color: "gray"

    }

    property alias text: input.text
    property alias input: input

    TextEdit {
        id: input
        anchors.fill: parent
        anchors.margins: 4
        focus: true
    }
}
```

你可以像下面這樣使用這個組件：

```
// textedit.qml

import QtQuick 2.0

Rectangle {
    width: 136
    height: 120
    color: "linen"

    TTextEdit {
        id: input
        x: 8; y: 8
        width: 120; height: 104
        focus: true
        text: "Text Edit"
    }
```

```
}
```



4.7.4 按鍵元素（Key Element）

附加屬性key允許你基于某個按鍵的點擊來執行代碼。例如使用up，down按鍵來移動一個方塊，left，right按鍵來旋轉一個元素，plus，minus按鍵來縮放一個元素。

```
// keys.qml

import QtQuick 2.0

DarkSquare {
    width: 400; height: 200

    GreenSquare {
        id: square
        x: 8; y: 8
    }
    focus: true
    Keys.onLeftPressed: square.x -= 8
    Keys.onRightPressed: square.x += 8
    Keys.onUpPressed: square.y -= 8
    Keys.onDownPressed: square.y += 8
    Keys.onPressed: {
        switch(event.key) {
            case Qt.Key_Plus:
                square.scale += 0.2
                break;
            case Qt.Key_Minus:
                square.scale -= 0.2
                break;
        }

    }
}
```

高級用法（**Advanced Techniques**）

後續添加。

Fluid Elements

注意

最後一次構建：**2014**年**1**月**20**日下午**18:00**。

這章的源代碼能夠在**assetts folder**找到。

到目前為止，我們已經介紹了簡單的圖形元素和怎樣布局，怎樣操作它們。這一章介紹如何控制屬性值的變化，通過動畫的方式在一段時間內來改變屬性值。這項技術是建立一個現代化的平滑界面的基礎，通過使用狀態和過渡來擴展你的用戶界面。每一種狀態定義了屬性的改變，與動畫聯系起來的狀態改變稱作過渡。

動畫（Animations）

動畫被用于屬性的改變。一個動畫定義了屬性值改變的曲線，將一個屬性值變化從一個值過渡到另一個值。動畫是由一連串的目標屬性活動定義的，平緩的曲線算法能夠引發一個定義時間內屬性的持續變化。所有在QtQuick中的動畫都由同一個計時器來控制，因此它們始終都保持同步，這也提高了動畫的性能和顯示效果。

注意

動畫控制了屬性的改變，也就是值的插入。這是一個基本的概念，**QML**是基于元素，屬性與腳本的。每一個元素都提供了許多的屬性，每一個屬性都在等待使用動畫。在這本書中你將會看到這是一個壯闊的場景，你會發現你自己在看一些動畫時欣賞它們的美麗並且肯定自己的創造性想法。然後請記住：動畫控制了屬性的改變，每個元素都有大量的屬性供你任意使用。

```
// animation.qml

import QtQuick 2.0

Image {
    source: "assets/background.png"

    Image {
        x: 40; y: 80
        source: "assets/rocket.png"

        NumberAnimation on x {
            to: 240
            duration: 4000
            loops: Animation.Infinite
        }
        RotationAnimation on rotation {
            to: 360
            duration: 4000
            loops: Animation.Infinite
        }
    }
}
```

上面這個例子在x坐標和旋轉屬性上應用了一個簡單的動畫。每一次動畫持續4000毫秒並且永久循環。x軸坐標動畫展示了火箭的x坐標逐漸移至240，旋轉動畫展示了當前角度到360度的旋轉。兩個動畫同時運行，並且在加載用戶界面完成後開始。

現在你可以通過to屬性和duration屬性來實現動畫效果。或者你可以在opacity或者scale上添加動畫作為例子，集成這兩個參數，你可以實現火箭逐漸消失在太空中，試試吧!

5.1.1 動畫元素（Animation Elements）

有幾種類型的動畫，每一種都在特定情況下都有最佳的效果，下面列出了一些常用的動畫：

- PropertyAnimation（屬性動畫）- 使用屬性值改變播放的動畫
- NumberAnimation（數字動畫）- 使用數字改變播放的動畫
- ColorAnimation（顏色動畫）- 使用顏色改變播放的動畫
- RotationAnimation（旋轉動畫）- 使用旋轉改變播放的動畫

除了上面這些基本和通常使用的動畫元素，QtQuick還提供了一切特殊場景下使用的動畫：

- PauseAnimation（停止動畫）- 運行暫停一個動畫
- SequentialAnimation（順序動畫）- 允許動畫有序播放
- ParallelAnimation（並行動畫）- 允許動畫同時播放
- AnchorAnimation（錨定動畫）- 使用錨定改變播放的動畫
- ParentAnimation（父元素動畫）- 使用父對象改變播放的動畫
- SmotthedAnimation（平滑動畫）- 跟蹤一個平滑值播放的動畫
- SpringAnimation（彈簧動畫）- 跟蹤一個彈簧變換的值播放的動畫
- PathAnimation（路徑動畫）- 跟蹤一個元素對象的路徑的動畫
- Vector3dAnimation（3D容器動畫）- 使用QVector3d值改變播放的動畫

我們將在後面學習怎樣創建一連串的動畫。當使用更加復雜的動畫時，我們可能需要在播放一個動畫時中改變一個屬性或者運行一個腳本。對于這個問題，QtQuick提供了一個動作元素：

- PropertyAction（屬性動作）- 在播放動畫時改變屬性
- ScriptAction（腳本動作）- 在播放動畫時運行腳本

在這一章中我們將會使用一些小的例子來討論大多數類型的動畫。

5.1.2 應用動畫（Applying Animations）

動畫可以通過以下幾種方式來應用：

- 屬性動畫 - 在元素完整加載後自動運行
- 屬性動作 - 當屬性值改變時自動運行
- 獨立運行動畫 - 使用start()函數明確指定運行或者running屬性被設置為true（比如通過屬性綁定）

後面我們會談論如何在狀態變換時播放動畫。

擴展可點擊圖像元素版本**2**（**ClickableImage Version2**）

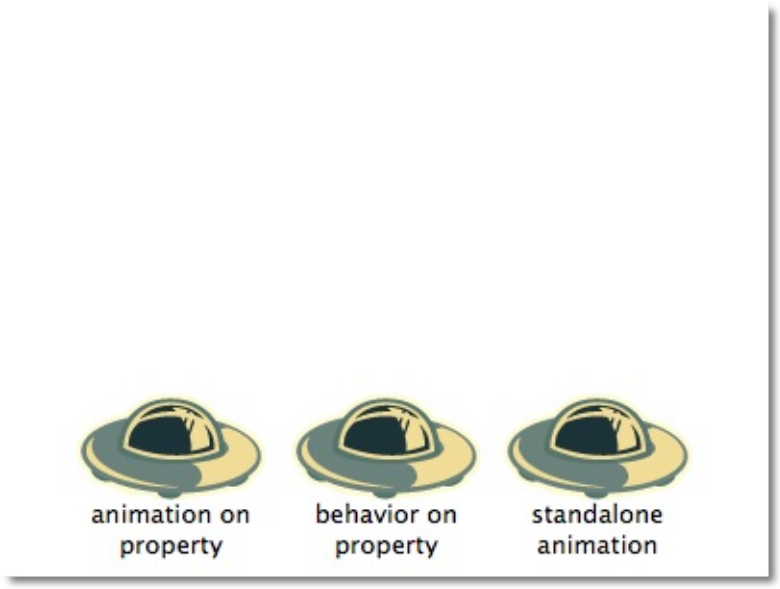
為了演示動畫的使用方法，我們重新實現了ClickableImage組件並且使用了一個文本元素（Text Element）來擴展它。

```
// ClickableImageV2.qml
// Simple image which can be clicked

import QtQuick 2.0

Item {
    id: root
    width: container.childrenRect.width
    height: container.childrenRect.height
    property alias text: label.text
    property alias source: image.source
    signal clicked

    Column {
        id: container
        Image {
            id: image
        }
        Text {
            id: label
            width: image.width
            horizontalAlignment: Text.AlignHCenter
            wrapMode: Text.WordWrap
            color: "#111111"
        }
    }

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

為了給圖片下面的元素定位，我們使用了Column（列）定位器，並且使用基于列的子矩形（childRect）屬性來計算它的寬度和高度（width and height）。我們導出了文本（text）和圖形源（source）屬性，一個點擊信號（clicked signal）。我們使用文本元素的wrapMode屬性來設置文本與圖像一樣寬並且可以自動換行。

注意

由于幾何依賴關系的反向（父幾何對象依賴于子幾何對象）我們不能對**ClickableImageV2**設置寬度**/**高度（**width/height**），因為這樣將會破壞我們已經做好的屬性綁定。這是我們**內**部設計的限制，作為一個設計組件的人你需要明白這一點。通常我們更喜歡**內**部幾何圖像依賴于父幾何對象。



三個火箭位于相同的y軸坐標（y = 200）。它們都需要移動到y = 40。每一個火箭都使用了一種的方法來完成這個功能。

```
ClickableImageV3 {
    id: rocket1
    x: 40; y: 200
    source: "assets/rocket2.png"
    text: "animation on property"
    NumberAnimation on y {
        to: 40; duration: 4000
    }
}
```

第一個火箭

第一個火箭使用了Animation on 屬性變化的策略來完成。動畫會在加載完成後立即播放。點擊火箭可以重置它回到開始的位置。在動畫播放時重置第一個火箭不會有任何影響。在動畫開始前的幾分之一秒設置一個新的y軸坐標讓人感覺挺不安全的，應當避免這樣的屬性值競爭的變化。

```
ClickableImageV3 {
    id: rocket2
    x: 152; y: 200
    source: "assets/rocket2.png"
    text: "behavior on property"
    Behavior on y {
        NumberAnimation { duration: 4000 }
    }

    onClicked: y = 40
    // random y on each click
//        onClicked: y = 40+Math.random()*(205-40)
}
```

第二個火箭

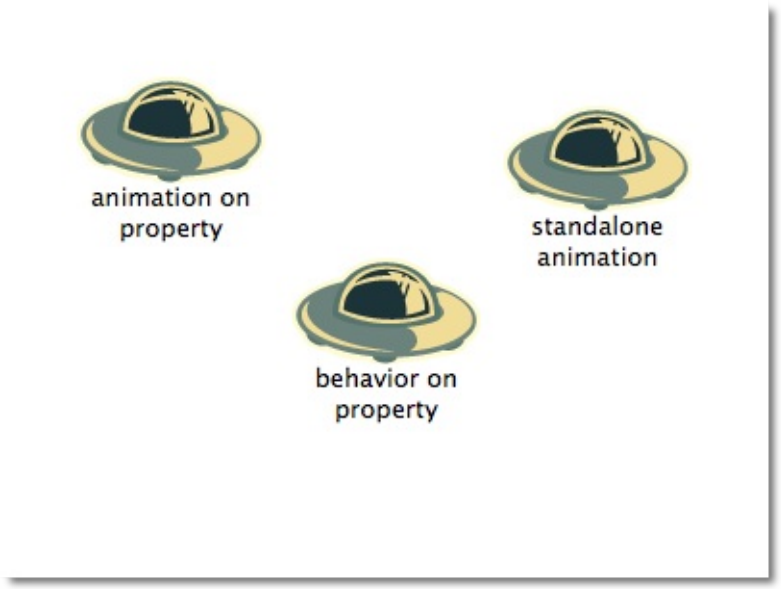
第二個火箭使用了behavior on 屬性行為策略的動畫。這個行為告訴屬性值每時每刻都在變化，通過動畫的方式來改變這個值。可以使用行為元素的enabled : false來設置行為失效。當你點擊這個火箭時它將會開始

運行（y軸坐標逐漸移至40）。然後其它的點擊對于位置的改變沒有任何的影響。你可以試著使用一個隨機值（例如 40+(Math.random()*(205-40)）來設置y軸坐標。你可以發現動畫始終會將移動到新位置的時間匹配在4秒內完成。

```
    ClickableImageV3 {
        id: rocket3
        x: 264; y: 200
        source: "assets/rocket2.png"
        onClicked: anim.start()
    //        onClicked: anim.restart()

        text: "standalone animation"

        NumberAnimation {
            id: anim
            target: rocket3
            properties: "y"
            from: 205
            to: 40
            duration: 4000
        }
    }
```

第三個火箭

第三個火箭使用standalone animation獨立動畫策略。這個動畫由一個私有的元素定義並且可以寫在文檔的任何地方。點擊火箭調用動畫函數start()來啟動動畫。每一個動畫都有start()，stop()，resume()，restart()函數。這個動畫自身可以比其他類型的動畫更早的獲取到更多的相關信息。我們只需要定義目標和目標元素的屬性需要怎樣改變的一個動畫。我們定義一個to屬性的值，在這個例子中我們也定義了一個from屬性的值允許動畫可以重復運行。



點擊背景能夠重新設置所有的火箭回到它們的初始位置。第一個火箭無法被重置，只有重啟程序重新加載元素才能重置它。

注意

另一個啟動/停止一個動畫的方法是綁定一個動畫的**running**屬性。當需要用戶輸入控制屬性時這種方法非常有用：

```
    NumberAnimation {
        ...
        // animation runs when mouse is pressed
        running: area.pressed
    }
    MouseArea {
        id: area
    }
```

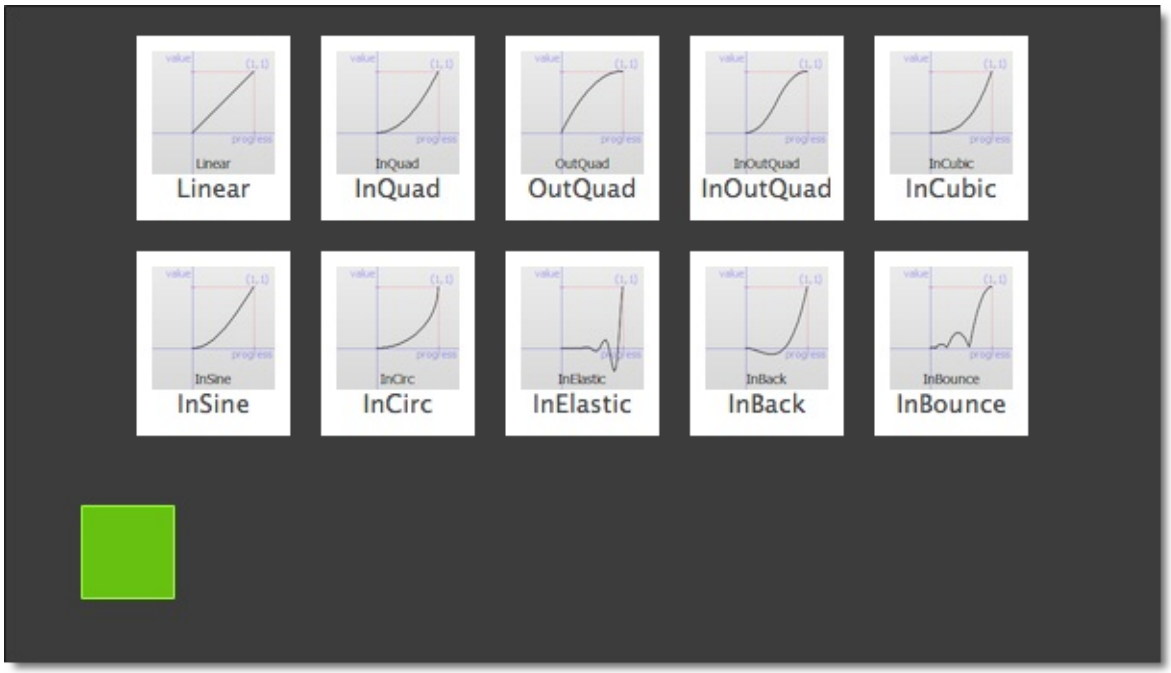
5.1.3 緩衝曲線（Easing Curves）

屬性值的改變能夠通過一個動畫來控制，緩衝曲線屬性影響了一個屬性值改變的插值算法。我們現在已經定義的動畫都使用了一種線性的插值算法，因為一個動畫的默認緩衝類型是Easing.Linear。在一個小場景下的x軸與y軸坐標改變可以得到最好的視覺效果。一個線性插值算法將會在動畫開始時使用from的值到動畫結束時使用的to值繪制一條直線，所以緩衝類型定義了曲線的變化情況。精心為一個移動的對象挑選一個合適的緩衝類型將會使界面更加自然，例如一個頁面的滑出，最初使用緩慢的速度滑出，然後在最後滑出時使用高速滑出，類似翻書一樣的效果。

注意

不要過度的使用動畫。用戶界面動畫的設計應該盡量小心，動畫是讓界面更加生動而不是充滿整個界面。眼睛對于移動的東西非常敏感，很容易幹擾用戶的使用。

在下面的例子中我們將會使用不同的緩衝曲線，每一種緩衝曲線都都使用了一個可點擊圖片來展示，點擊將會在動畫中設置一個新的緩衝類型並且使用這種曲線重新啟動動畫。



擴展可點擊圖像**V3**（**ClickableImage V3**）

我們給圖片和文本添加了一個小的外框來增強我們的ClickableImage。添加一個屬性property bool framed: false來作為我們的API，基于framed的值我們能夠設置這個框是否可見，並且不破壞之前用戶的使用。下面是我們做的修改。

```
// ClickableImageV2.qml
```

```
// Simple image which can be clicked

import QtQuick 2.0

Item {
    id: root
    width: container.childrenRect.width + 16
    height: container.childrenRect.height + 16
    property alias text: label.text
    property alias source: image.source
    signal clicked

    // M1>>
    // ... add a framed rectangle as container
    property bool framed : false

    Rectangle {
        anchors.fill: parent
        color: "white"
        visible: root.framed
    }
```

這個例子的代碼非常簡潔。我們使用了一連串的緩衝曲線的名稱（property variant easings）並且在一個Repeater（重復元素）中將它們分配給一個ClickableImage。圖片的源路徑通過一個命名方案來定義，一個叫做“InQuad”的緩衝曲線在“curves/InQuad.png”中有一個對應的圖片。如果你點擊一個曲線圖，這個點擊將會分配一個緩衝類型給動畫然後重新啟動動畫。動畫自身是用來設置方塊的x坐標屬性在2秒內變化的獨立動畫。

```
// easingtypes.qml

import QtQuick 2.0

DarkSquare {
    id: root
    width: 600
    height: 340

    // A list of easing types
    property variant easings : [
        "Linear", "InQuad", "OutQuad", "InOutQuad",
        "InCubic", "InSine", "InCirc", "InElastic",
        "InBack", "InBounce" ]


    Grid {
        id: container
        anchors.top: parent.top
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.margins: 16
        height: 200
        columns: 5
        spacing: 16
        // iterates over the 'easings' list
        Repeater {
            model: easings
            ClickableImageV3 {
                framed: true
                // the current data entry from 'easings' list
                text: modelData
```

```
                source: "curves/" + modelData + ".png"
                onClicked: {
                    // set the easing type on the animation
                    anim.easing.type = modelData
                    // restart the animation
                    anim.restart()
                }
            }
        }
    }

    // The square to be animated
    GreenSquare {
        id: square
        x: 40; y: 260
    }

    // The animation to test the easing types
    NumberAnimation {
        id: anim
        target: square
        from: 40; to: root.width - 40 - square.width
        properties: "x"
        duration: 2000
    }
}
```

當你運行這個例子時，請注意觀察動畫的改變速度。一些動畫對于這個對象看起來很自然，一些看起來非常惱火。

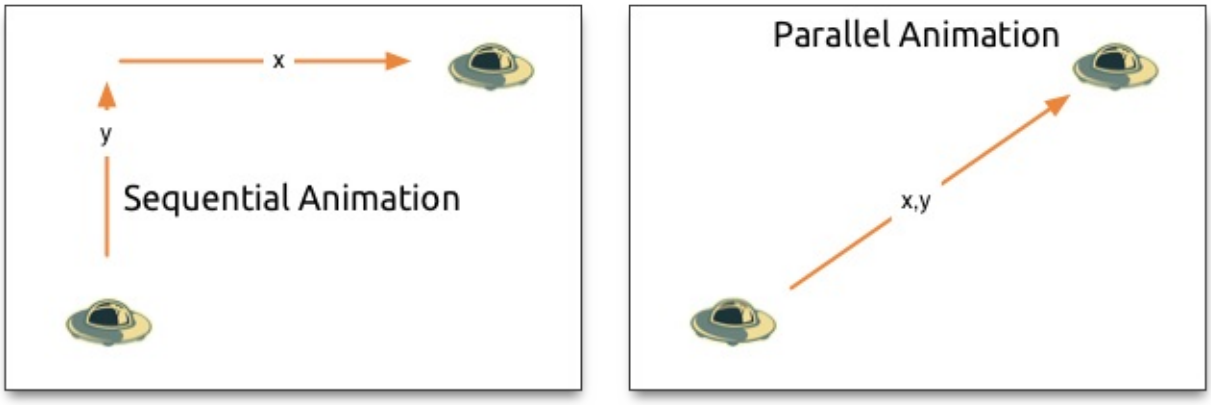
除了duration屬性與easing.type屬性，你也可以對動畫進行微調。例如PropertyAnimation屬性，大多數動畫都支持附加的easing.amplitude（緩衝振幅），easing.overshoot（緩衝溢出），easing.period（緩衝週期），這些屬性允許你對個別的緩衝曲線進行微調。不是所有的緩衝曲線都支持這些參數。可以查看Qt PropertyAnimation文檔中的緩衝列表（easing table）來查看一個緩衝曲線的相關參數。

注意

對于用戶界面正確的動畫非常重要。請記住動畫是幫助用戶界面更加生動而不是刺激用戶的眼睛。

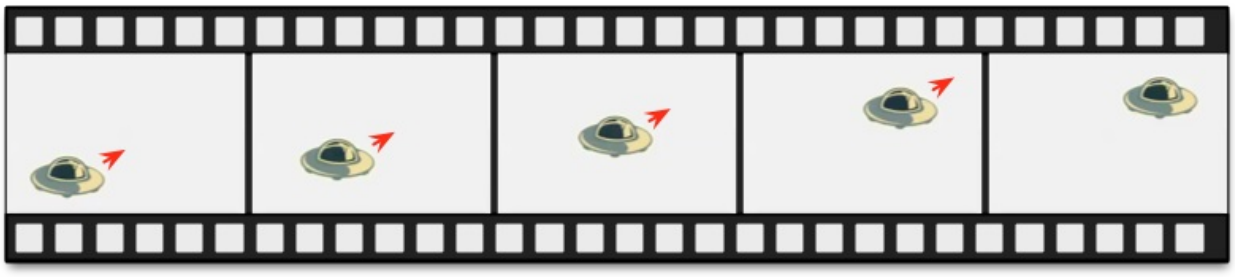
5.1.4 動畫分組（Grouped Animations）

通常使用的動畫比一個屬性的動畫更加復雜。例如你想同時運行幾個動畫並把他們連接起來，或者在一個一個的運行，或者在兩個動畫之間執行一個腳本。動畫分組提供了很好的幫助，作為命名建議可以叫做一組動畫。有兩種方法來分組：平行與連續。你可以使用SequentialAnimation（連續動畫）和ParallelAnimation（平行動畫）來實現它們，它們作為動畫的容器來包含其它的動畫元素。



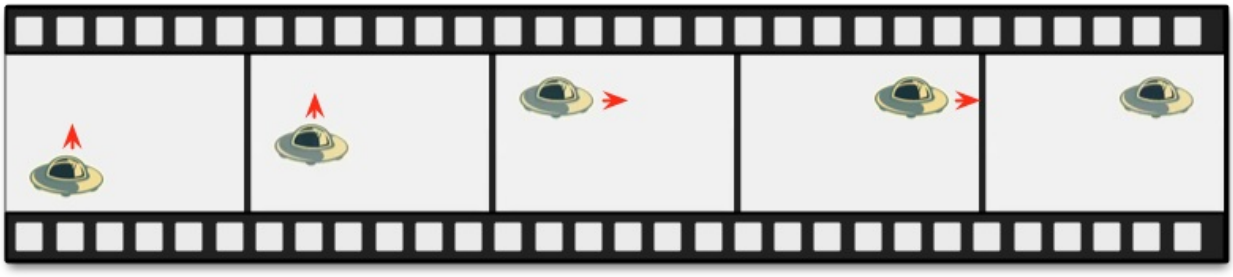
當開始時，平行元素的所有子動畫都會平行運行，它允許你在同一時間使用不同的屬性來播放動畫。

```
// parallelanimation.qml
import QtQuick 2.0

BrightSquare {
    id: root
    width: 300
    height: 200
    property int duration: 3000

    ClickableImageV3 {
        id: rocket
        x: 20; y: 120
        source: "assets/rocket2.png"
        onClicked: anim.restart()
    }

    ParallelAnimation {
        id: anim
        NumberAnimation {
            target: rocket
            properties: "y"
            to: 20
            duration: root.duration
        }
        NumberAnimation {
            target: rocket
            properties: "x"
            to: 160
            duration: root.duration
        }
    }
}
```

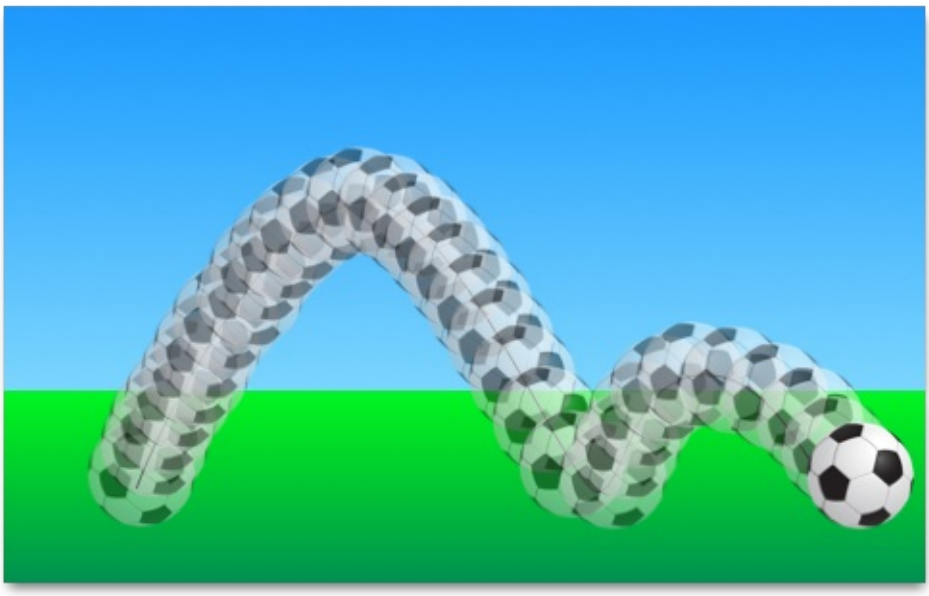


一個連續的動畫將會一個一個的運行子動畫。

```
// sequentialanimation.qml
import QtQuick 2.0

BrightSquare {
    id: root
    width: 300
    height: 200
    property int duration: 3000

    ClickableImageV3 {
        id: rocket
        x: 20; y: 120
        source: "assets/rocket2.png"
        onClicked: anim.restart()
    }

    SequentialAnimation {
        id: anim
        NumberAnimation {
            target: rocket
            properties: "y"
            to: 20
            // 60% of time to travel up
            duration: root.duration*0.6
        }
        NumberAnimation {
            target: rocket
            properties: "x"
            to: 160
            // 40% of time to travel sideways
            duration: root.duration*0.4
        }
    }
}
```



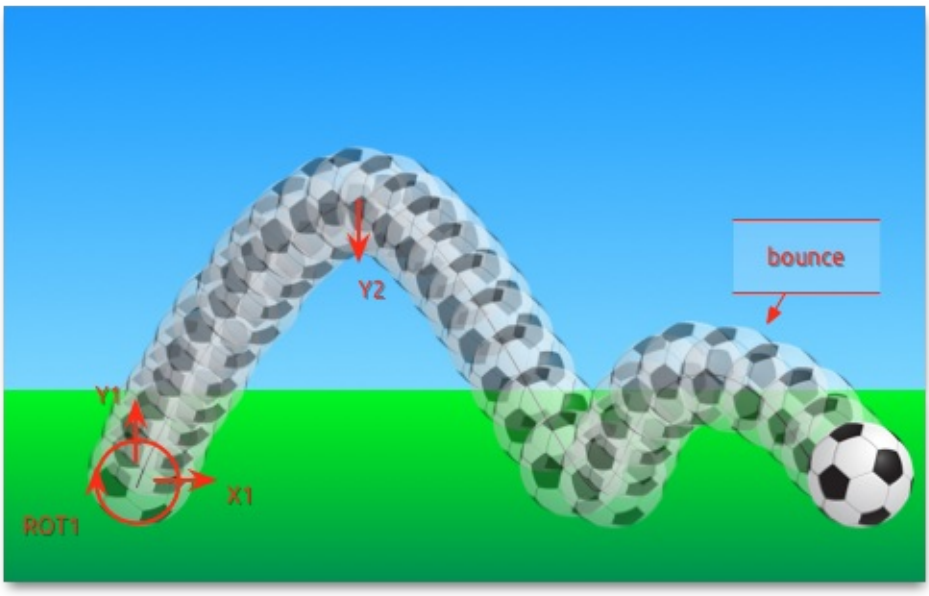
分組動畫也可以被嵌套，例如一個連續動畫可以擁有兩個平行動畫作為子動畫。我們來看看這個足球的例子。這個動畫描述了一個從左向右扔一個球的行為：



要弄明白這個動畫我們需要剖析這個目標的運動過程。我們需要記住這個動畫是通過屬性變化來實現的動畫，下面是不同部分的轉換：

- 從左向右的x坐標轉換（X1）。
- 從下往上的y坐標轉換（Y1）然後跟著一個從上往下的Y坐標轉換（Y2）。
- 整個動畫過程中360度旋轉。

這個動畫將會花掉3秒鐘的時間。



我們使用一個空的基本元素對象（Item）作為根元素，它的寬度為480，高度為300。

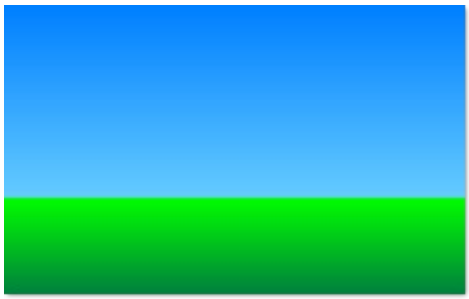
```
import QtQuick 1.1

Item {
    id: root
    width: 480
    height: 300
    property int duration: 3000
```

```
    ...
}
```

我們定義動畫的總持續時間作為參考，以便更好的同步各部分的動畫。

下一步我們需需要添加一個背景，在我們這個例子中有兩個矩形框分別使用了綠色漸變和藍色漸變填充。

```
    Rectangle {
        id: sky
        width: parent.width
        height: 200
        gradient: Gradient {
            GradientStop { position: 0.0; color: "#0080FF" }
            GradientStop { position: 1.0; color: "#66CCFF" }
        }
    }
    Rectangle {
        id: ground
        anchors.top: sky.bottom
        anchors.bottom: root.bottom
        width: parent.width
        gradient: Gradient {
            GradientStop { position: 0.0; color: "#00FF00" }
            GradientStop { position: 1.0; color: "#00803F" }
        }
    }
```



上面部分的藍色區域高度為200像素，下面部分的區域使用上面的藍色區域的底作為錨定的頂，使用根元素的底作為底。

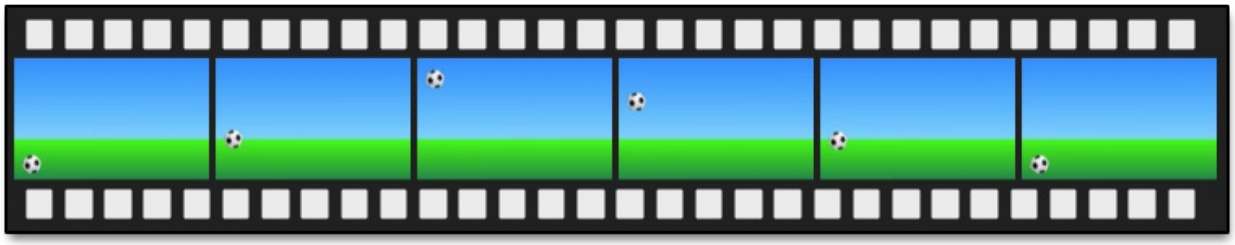
讓我們將足球加入到屏幕上，足球是一個圖片，位于路徑“assets/soccer_ball.png”。首先我們需要將它放置在左下角接近邊界處。

```
    Image {
        id: ball
        x: 20; y: 240
        source: "assets/soccer_ball.png"

        MouseArea {
            anchors.fill: parent
            onClicked: {
                ball.x = 20; ball.y = 240
                anim.restart()
            }
        }
    }
```

圖片與鼠標區域連接，點擊球將會重置球的狀態，並且動畫重新開始。

首先使用一個連續的動畫來播放兩次的y軸變換。

```
    SequentialAnimation {
        id: anim
        NumberAnimation {
            target: ball
            properties: "y"
            to: 20
            duration: root.duration * 0.4
        }
        NumberAnimation {
            target: ball
            properties: "y"
            to: 240
            duration: root.duration * 0.6
        }
    }
```

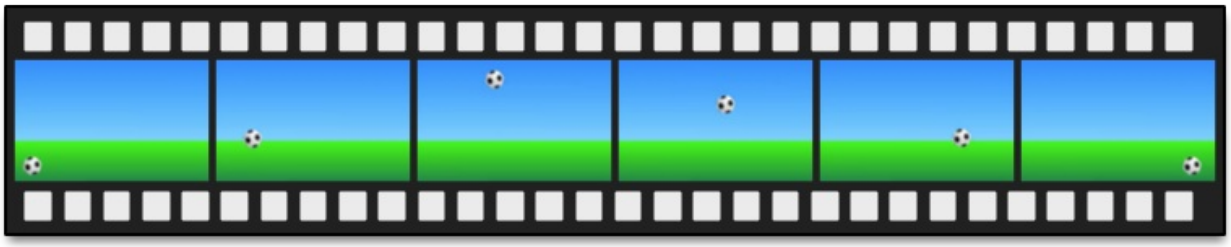


在動畫總時間的40%的時間裡完成上升部分，在動畫總時間的60%的時間裡完成下降部分，一個動畫完成後播放下一個動畫。目前還沒有使用任何緩衝曲線。緩衝曲線將在後面使用easing curves來添加，現在我們只關心如何使用動畫來完成過渡。

現在我們需要添加x軸坐標轉換。x軸坐標轉換需要與y軸坐標轉換同時進行，所以我們需要將y軸坐標轉換的連續動畫和x軸坐標轉換一起壓縮進一個平行動畫中。

```
    ParallelAnimation {
        id: anim
        SequentialAnimation {
            // ... our Y1, Y2 animation
        }
        NumberAnimation { // X1 animation
            target: ball
            properties: "x"
            to: 400
            duration: root.duration
        }
```

```
    }
```



最後我們想要旋轉這個球，我們需要向平行動畫中添加一個新的動畫，我們選擇RotationAnimation來實現旋轉。

```
ParallelAnimation {
    id: anim
    SequentialAnimation {
        // ... our Y1, Y2 animation
    }
    NumberAnimation { // X1 animation
        // X1 animation
    }
    RotationAnimation {
        target: ball
        properties: "rotation"
        to: 720
        duration: root.duration
    }
}
```

我們已經完成了整個動畫鏈表，然後我們需要給動畫提供一個正確的緩衝曲線來描述一個移動的球。對于Y1動畫我們使用Easing.OutCirc緩衝曲線，它看起來更像是一個圓週運動。Y2使用了Easing.OutBounce緩衝曲線，因為在最後球會發生反彈。（試試使用Easing.InBounce，你會發現反彈將會立刻開始。）。X1和ROT1動畫都使用線性曲線。

下面是這個動畫最後的代碼，提供給你作為參考：

```
ParallelAnimation {
    id: anim
    SequentialAnimation {
        NumberAnimation {
            target: ball
            properties: "y"
            to: 20
            duration: root.duration * 0.4
            easing.type: Easing.OutCirc
        }
        NumberAnimation {
            target: ball
            properties: "y"
            to: 240
            duration: root.duration * 0.6
            easing.type: Easing.OutBounce
        }
    }
    NumberAnimation {
        target: ball
```

```
            properties: "x"
            to: 400
            duration: root.duration
        }
        RotationAnimation {
            target: ball
            properties: "rotation"
            to: 720
            duration: root.duration * 1.1
        }
    }
```

狀態與過渡（States and Transitions）

通常我們將用戶界面描述為一種狀態。一個狀態定義了一組屬性的改變，並且會在一定的條件下被觸發。另外在這些狀態轉化的過程中可以有一個過渡，定義了這些屬性的動畫或者一些附加的動作。當進入一個新的狀態時，動作也可以被執行。

5.2.1 狀態（States）

在QML中，使用State元素來定義狀態，需要與基礎元素對象（Item）的states序列屬性連接。狀態通過它的狀態名來鑑別，由組成它的一系列簡單的屬性來改變元素。默認的狀態在初始化元素屬性時定義，並命名為“”（一個空的字符串）。

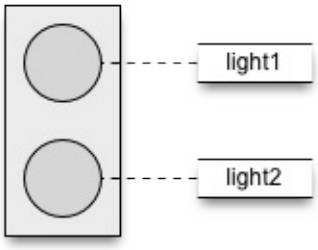
```
Item {
    id: root
    states: [
        State {
            name: "go"
            PropertyChanges { ... }
        },
        State {
            name: "stop"
            PropertyChanges { ... }
        }
    ]
}
```

狀態的改變由分配一個元素新的狀態屬性名來完成。

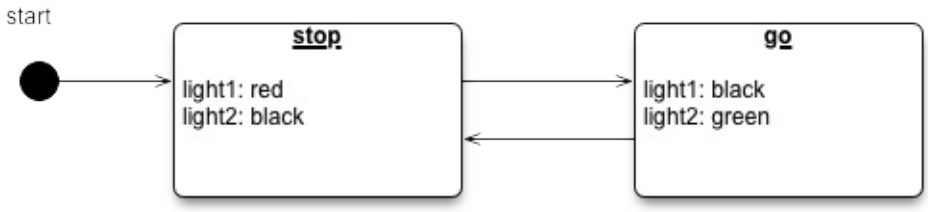
注意

另一種切換屬性的方法是使用狀態元素的**when**屬性。**when**屬性能夠被設置為一個表達式的結果，當結果為**true**時，狀態被使用。

```
Item {
    id: root
    states: [
        ...
    ]

    Button {
        id: goButton
        ...
        onClicked: root.state = "go"
    }
}
```



例如一個交通信號燈有兩個信號燈。上面的一個信號燈使用紅色，下面的信號燈使用綠色。在這個例子中，兩個信號燈不會同時發光。讓我們看看狀態圖。



當系統啟動時，它會自動切換到停止模式作為默認狀態。停止狀態改變了light1為紅色並且light2為黑色（關閉）。一個外部的事件能夠觸發現在的狀態變換為“go”狀態。在go狀態下，我們改變顏色屬性，light1變為黑色（關閉），light2變為綠色。

為了實現這個方案，我們給這兩個燈繪制一個用戶界面的草圖，為了簡單起見，我們使用兩個包含圓邊的矩形框，設置圓半徑為寬度的一半（寬度與高度相同）。

```
Rectangle {
    id: light1
    x: 25; y: 15
    width: 100; height: width
    radius: width/2
    color: "black"
}

Rectangle {
    id: light2
    x: 25; y: 135
    width: 100; height: width
    radius: width/2
    color: "black"
}
```

就像在狀態圖中定義的一樣，我們有一個“go”狀態和一個“stop”狀態，它們將會分別將交通燈改變為紅色和綠色。我們設置state屬性到stop來確保初始化狀態為stop狀態。

注意

我們可以只使用**“go”**狀態來達到同樣的效果，設置顏色**light1**為紅色，顏色**light2**為黑色。初始化狀態**“”**（空字符串）定義初始化屬性，並且扮演類似**“stop”**狀態的角色。

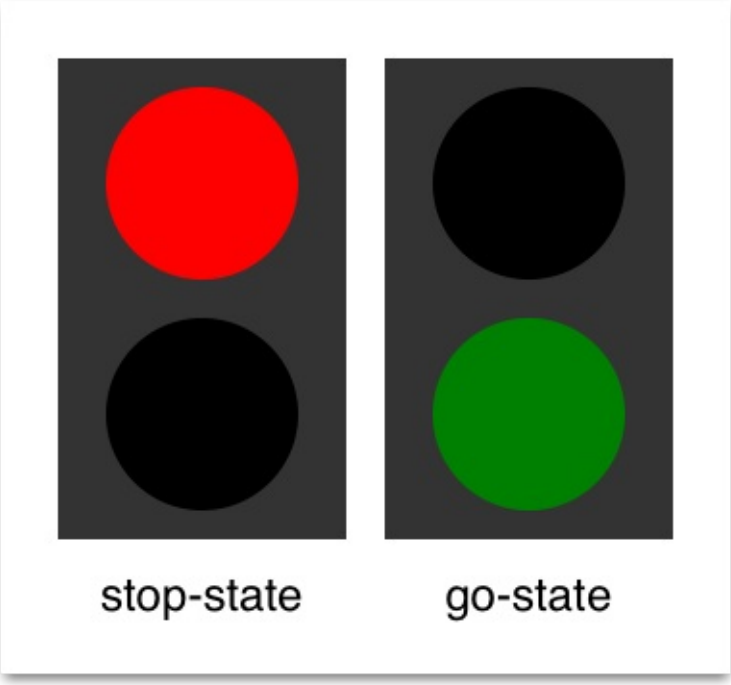
```
state: "stop"

states: [
    State {
        name: "stop"
        PropertyChanges { target: light1; color: "red" }
```

```
            PropertyChanges { target: light2; color: "black" }
        },
        State {
            name: "go"
            PropertyChanges { target: light1; color: "black" }
            PropertyChanges { target: light2; color: "green" }
        }
    ]
```

PropertyChanges{ target: light2; color: "black" }在這個例子中不是必要的，因為light2初始化顏色已經是黑色了。在一個狀態中，只需要描述屬性如何從它們的默認狀態改變（而不是前一個狀態的改變）。

使用鼠標區域覆蓋整個交通燈，並且綁定在點擊時切換go和stop狀態。

```
    MouseArea {
        anchors.fill: parent
        onClicked: parent.state = (parent.state == "stop"? "go" : "stop")
    }
```



我們現在已經成功實現了交通燈的狀態切換。為了讓用戶界面看起來更加自然，我們需要使用動畫效果來增加一些過渡。一個過渡能夠被狀態的改變觸發。

注意

可以使用一個簡單邏輯的腳本來替換**QML**狀態。開發人員很容易落入這種陷阱，寫的代碼更像一個**JavaScript**程序而不是一個**QML**程序。

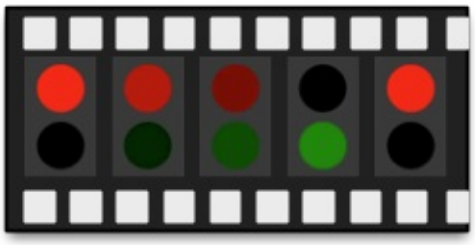
5.2.2 過渡（Transitions）

一系列的過渡能夠被加入任何元素， 一個過渡由狀態的改變觸發執行。你可以使用屬性的from:和to:來定義狀態改變的指定過渡。這兩個屬性就像一個過濾器，當過濾器為true時，過渡生效。你也可以使用“”來表示任何狀態。例如*from:*""; to:"*"表示從任一狀態到另一個任一狀態的默認值，這意味著過渡用于每個狀態的切換。

在這個例子中，我們期望從狀態“go”到“stop”轉換時實現一個顏色改變的動畫。對于從“stop”到“go”狀態的改變，我們期望保持顏色的直接改變，不使用過渡。我們使用from和to來限制過渡只在從“go”到“stop”時生效。在過渡中我們給每個燈添加兩個顏色的動畫，這個動畫將按照狀態的描述來改變屬性。

```
    transitions: [
        Transition {
            from: "stop"; to: "go"
            ColorAnimation { target: light1; properties: "color"; duration: 2000 }
            ColorAnimation { target: light2; properties: "color"; duration: 2000 }
        }
    ]
```

你可以點擊用戶界面來改變狀態。試試點擊用戶界面，當狀態從“stop”到“go”時，你將會發現改變立刻發生了。



接下來，你可以修改下這個例子，例如縮小未點亮的等來突出點亮的等。為此，你需要在狀態中添加一個屬性用來縮放，並且操作一個動畫來播放縮放屬性的過渡。另一個選擇是可以添加一個“attention”狀態，燈會出現黃色閃爍，為此你需要添加為這個過渡添加一個一秒連續的動畫來顯示黃色（使用“to”屬性來實現，一秒後變為黑色）。也許你也可以改變緩衝曲線來使這個例子更加生動。

高級用法（**Advanced Techniques**）

後續添加。

Model-View-Delegate

注意

最後一次構建：**2014**年**1**月**20**日下午**18:00**。

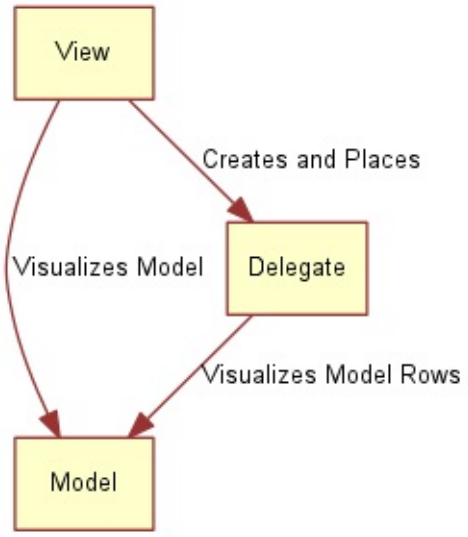
這章的源代碼能夠在**assetts folder**找到。

在QtQuick中，數據通過model-view（模型-視圖）分離。對于每個view（視圖），每個數據元素的可視化都分給一個代理（delegate）。QtQuick附帶了一組預定義的模型與視圖。想要使用這個系統，必須理解這些類，並且知道如何創建合適的代理來獲得正確的顯示和交互。

概念（**Concept**）

對于開發用戶界面，最重要的一方面是保持數據與可視化的分離。例如，一個電話薄可以使用一個垂直文本鏈表排列或者使用一個網格聯系人圖片排列。在這兩個案例中，數據都是相同的，但是可視化效果卻是不同的。這種方法通常被稱作model-view（模型-視圖）模式。在這種模式中，數據通常被稱作model（模型），可視化處理稱作view（視圖）。

在QML中，model（模型）與view（視圖）都通過delegate（代理）連接起來。功能劃分如下，model（模型）提供數據。對于每個數據項，可能有多個值。在上面的電話薄例子中，每個電話薄條目對應一個名字，一個圖片和一個號碼。顯示在view（視圖）中的每項數據,都是通過delegate（代理）來實現可視化。view（視圖）的任務是排列這些delegate（代理），每個delegate（代理）將model item（模型項）的值顯示給用戶。

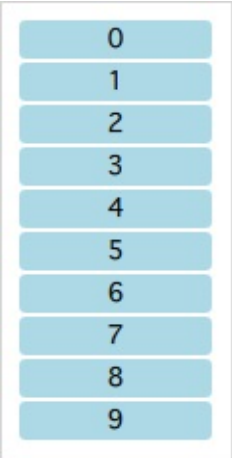


基礎模型（Basic Model）

最基本的分離數據與顯示的方法是使用Repeater元素。它被用于實例化一組元素項，並且很容易與一個用于填充用戶界面的定位器相結合。

最基本的實現舉例，repeater元素用于實現子元素的標號。每個子元素都擁有一個可以訪問的屬性index，用于區分不同的子元素。在下面的例子中，一個repeater元素創建了10個子項，子項的數量由model屬性控制。對于每個子項Rectangle包含了一個Text元素，你可以將text屬性設置為index的值，因此可以看到子項的編號是0~9。

```
import QtQuick 2.0

Column {
    spacing: 2

    Repeater {
        model: 10

        Rectangle {
            width: 100
            height: 20

            radius: 3

            color: "lightBlue"

            Text {
                anchors.centerIn: parent
                text: index
            }
        }
    }
}
```



這是一個不錯的編號列表，有時我們想顯示一些更復雜的數據。使用一個JavaScript序列來替換整形變量model的值可以達到我們的目的。序列可以使用任何類型的內容，可以是字符串，整數，或者對象。在下面的例子中，使用了一個字符串鏈表。我們仍然使用index的值作為變量，並且我們也訪問modelData中包含的每個元素的數據。

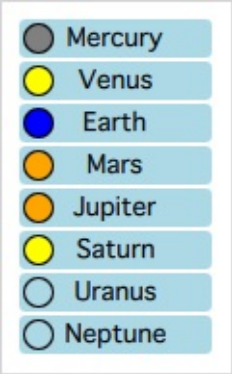
```
import QtQuick 2.0

Column {
    spacing: 2

    Repeater {
        model: ["Enterprise", "Colombia", "Challenger", "Discovery", "Endeavour", "Atlant

        Rectangle {
            width: 100
            height: 20

            radius: 3

            color: "lightBlue"

            Text {
                anchors.centerIn: parent
                text: index +": "+modelData
            }
        }
    }
}
```

將數據暴露成一組序列，你可以通過標號迅速的找到你需要的信息。想象一下這個模型的草圖，這是一個最簡單的模型，也是通常都會使用的模型，ListModel（鏈表模型）。一個鏈表模型由許多ListElement（鏈表元素）組成。在每個鏈表元素中，可以綁定值到屬性上。例如在下面這個例子中，每個元素都提供了一個名字和一個顏色。

每個元素中的屬性綁定連接到repeater實例化的子項上。這意味著變量name和surfaceColor可以被repeater創建的每個Rectangle和Text項引用。這不僅可以方便的訪問數據，也可以使源代碼更加容易閱讀。surfaceColor是名字左邊圓的顏色，而不是模糊的數據序列列i或者行j。

```
import QtQuick 2.0

Column {
    spacing: 2

    Repeater {
        model: ListModel {
            ListElement { name: "Mercury"; surfaceColor: "gray" }
            ListElement { name: "Venus"; surfaceColor: "yellow" }
            ListElement { name: "Earth"; surfaceColor: "blue" }
            ListElement { name: "Mars"; surfaceColor: "orange" }
            ListElement { name: "Jupiter"; surfaceColor: "orange" }
            ListElement { name: "Saturn"; surfaceColor: "yellow" }
            ListElement { name: "Uranus"; surfaceColor: "lightBlue" }
```

```
            ListElement { name: "Neptune"; surfaceColor: "lightBlue" }
        }

        Rectangle {
            width: 100
            height: 20

            radius: 3

            color: "lightBlue"

            Text {
                anchors.centerIn: parent
                text: name
            }

            Rectangle {
                anchors.left: parent.left
                anchors.verticalCenter: parent.verticalCenter
                anchors.leftMargin: 2

                width: 16
                height: 16

                radius: 8

                border.color: "black"
                border.width: 1

                color: surfaceColor
            }
        }
    }
}
```



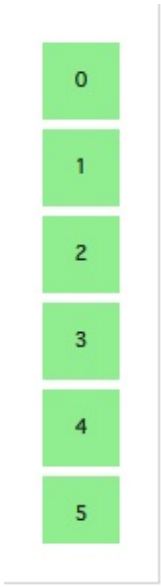
repeater的內容的每個子項實例化時綁定了默認的屬性delegate（代理）。這意味著例1（第一個代碼段）的代碼與下面顯示的代碼是相同的。注意，唯一的不同是delegate屬性名，將會在後面詳細講解。

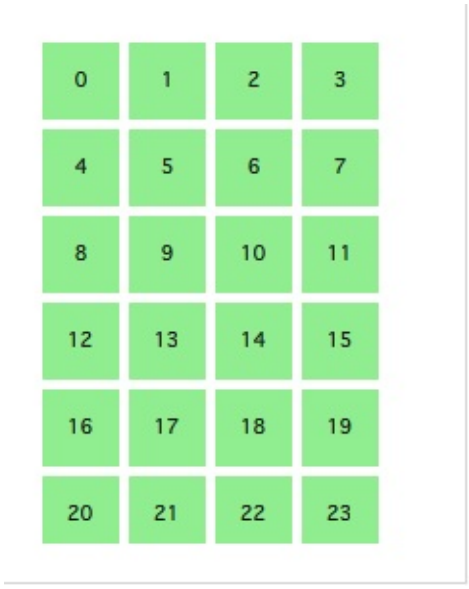
```
import QtQuick 2.0

Column {
    spacing: 2

    Repeater {
        model: 10

        delegate: Rectangle {
            width: 100
```

```
            height: 20

            radius: 3

            color: "lightBlue"

            Text {
                anchors.centerIn: parent
                text: index
            }
        }
    }
}
```

動態視圖（**Dynamic Views**）

Repeater元素適合有限的靜態數據，但是在真正使用時，模型通常更加復雜和龐大，我們需要一個更加智能的解決方案。QtQuick提供了ListView和GridView元素，這兩個都是基于Flickable（可滑動）區域的元素，因此用戶可以放入更大的數據。同時，它們限制了同時實例化的代理數量。對于一個大型的模型，這意味著在同一個場景下只會加載有限的元素。



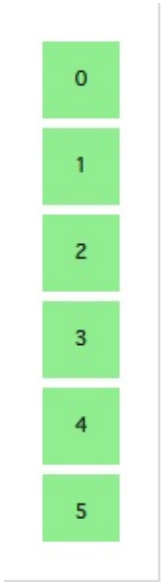


這兩個元素的用法非常類似，我們由ListView開始，然後會描述GridView的模型起點來進行比較。

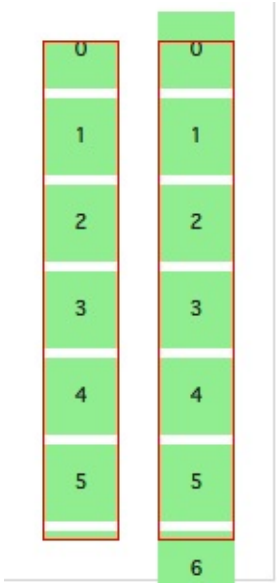
ListView與Repeater元素像素，它使用了一個model，使用delegate來實例化，並且在兩個delegate之間能夠設置間隔sapcing。下面的列表顯示了怎樣設置一個簡單的鏈表。

```
import QtQuick 2.0

Rectangle {
    width: 80
    height: 300

    color: "white"
```

```
    ListView {
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: 100

        delegate: numberDelegate
        spacing: 5
    }

    Component {
        id: numberDelegate

        Rectangle {
            width: 40
            height: 40

            color: "lightGreen"

            Text {
                anchors.centerIn: parent

                font.pixelSize: 10

                text: index
            }
        }
    }
}
```



如果模型包含的數據比屏幕上顯示的更多，ListView元素只會顯示部分的鏈表內容。然後由于QtQuick的默認行為導致的問題，列表視圖不會限制被顯示的代理項（delegates）只在限制區域內顯示。這意味著代理項可以在列表視圖外顯示，用戶可以看見在列表視圖外動態的創建和銷毀這些代理項（delegates）。為了防止這個問題，ListView通過設置clip屬性為true，來激活裁剪功能。下面的圖片展示了這個結果，左邊是clip屬性設置為false的對比。



對于用戶，ListView（列表視圖）是一個滾動區域。它支持慣性滾動，這意味著它可以快速的翻閱內容。默認模式下，它可以在內容最後繼續伸展，然後反彈回去，這個信號告訴用戶已經到達內容的末尾。

視圖末尾的行為是由到boundsBehavior屬性的控制的。這是一個枚舉值，並且可以配置為默認的Flickable.DragAndOvershootBounds，視圖可以通過它的邊界線來拖拽和翻閱，配置為Flickable.StopAtBounds，視圖將不再可以移動到它的邊界線之外。配置為Flickable.DragOverBounds，用戶可以將視圖拖拽到它的邊界線外，但是在邊界線上翻閱將無效。

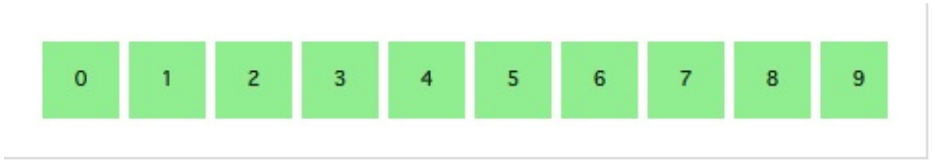
使用snapMode屬性可以限制一個視圖內元素的停止位置。默認行為下是ListView.NoSnap，允許視圖內元素在任何位置停止。將snapMode屬性設置為ListView.SnapToItem，視圖頂部將會與元素對象的頂部對齊排列。使用ListView.SnapOneItem，當鼠標或者觸摸釋放時，視圖將會停止在第一個可見的元素，這種模式對于瀏覽頁面非常便利。

6.3.1 方向（Orientation）

默認的鏈表視圖只提供了一個垂直方向的滾動條，但是水平滾動條也是需要的。鏈表視圖的方向由屬性orientation控制。它能夠被設置為默認值ListView.Vertical或者ListView.Horizontal。下面是一個水平鏈表視圖。

```
import QtQuick 2.0

Rectangle {
    width: 480
    height: 80

    color: "white"

    ListView {
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: 100

        orientation: ListView.Horizontal

        delegate: numberDelegate
```

```
        spacing: 5
    }

    Component {
        id: numberDelegate

        Rectangle {
            width: 40
            height: 40

            color: "lightGreen"

            Text {
                anchors.centerIn: parent

                font.pixelSize: 10

                text: index
            }
        }
    }
}
```



按照上面的設置，水平鏈表視圖默認的元素順序方向是由左到右。可以通過設置layoutDirection屬性來控制元素順序方向，它可以設置為Qt.LeftToRight或者Qt.RightToLeft。

6.3.2 鍵盤導航和高亮

當使用基于觸摸方式的鏈表視圖時，默認提供的視圖已經足夠使用。在使用鍵盤甚至僅僅通過方向鍵選擇一個元素的場景下，需要有標識當前選中元素的機制。在QML中，這被叫做高亮。

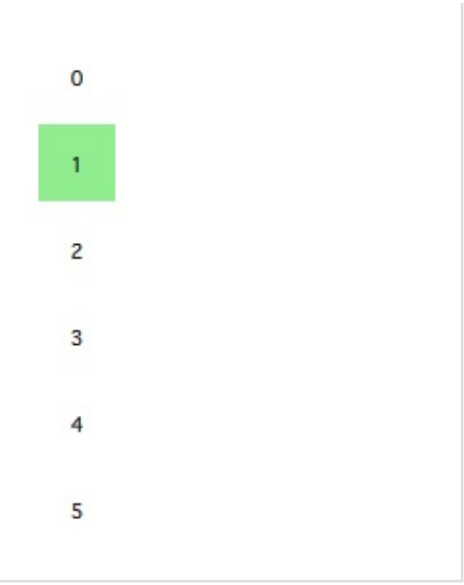
視圖支持設置一個當前視圖中顯示代理元素中的高亮代理。它是一個附加的代理元素，這個元素僅僅只實例化一次，並移動到與當前元素相同的位置。

在下面例子的演示中，有兩個屬性來完成這個工作。首先是focus屬性設置為true，它設置鏈表視圖能夠獲得鍵盤焦點。然後是highlight屬性，指出使用的高亮代理元素。高亮代理元素的x,y與height屬性由當前元素指定。如果寬度沒有特別指定，當前元素的寬度也可以用于高亮代理元素。

在例子中，ListView.view.width屬性被綁定用于高亮元素的寬度。關于代理元素的使綁定屬性將在後面的章節討論，但是最好知道相同的綁定屬性也可以用于高亮代理元素。

```
import QtQuick 2.0

Rectangle {
    width: 240
    height: 300

    color: "white"

    ListView {
        anchors.fill: parent
```

```
        anchors.margins: 20

        clip: true

        model: 100

        delegate: numberDelegate
        spacing: 5

        highlight: highlightComponent
        focus: true
    }

    Component {
        id: highlightComponent

        Rectangle {
            width: ListView.view.width
            color: "lightGreen"
        }
    }

    Component {
        id: numberDelegate

        Item {
            width: 40
            height: 40

            Text {
                anchors.centerIn: parent

                font.pixelSize: 10

                text: index
            }
        }
    }
}
// M1>>
```



當使用高亮與鏈表視圖（ListView）結合時，一些屬性可以用來控制它的行為。highlightRangeMode控制

了高亮如何影響視圖中當前的顯示。默認設置ListView.NoHighLighRange意味著高亮與視圖中的元素距離不相關。

ListView.StrictlyEnforceRnage確保了高亮始終可見，如果某個動作嘗試將高亮移出當前視圖可見範圍，當前元素將會自動切換，確保了高亮始終可見。

ListView.ApplyRange，它嘗試保持高亮代理始終可見，但是不會強制切換當前元素始終可見。如果在需要的情況下高亮代理允許被移出當前視圖。

在默認配置下，視圖負責高亮移動到指定位置，移動的速度與大小的改變能夠被控制，使用一個速度值或者一個動作持續時間來完成它。這些屬性包括highlightMoveSpeed，highlightMoveDuration，highlightResizeSpeed和highlightResizeDuration。默認下速度被設置為每秒400像素，動作持續時間為-1，表明速度和距離控制了動作的持續時間。如果速度與動作持續時間都被設置，動畫將會採用速度較快的結果來完成。

為了更加詳細的控制高亮的移動，highlightFollowCurrentItem屬性設置為false。這意味著視圖將不再負責高亮代理的移動。取而代之可以通過一個行為（Bahavior）或者一個動畫來控制它。

在下面的例子中，高亮代理的y坐標屬性與ListView.view.currentItem.y屬性綁定。這確保了高亮始終跟隨當前元素。然而，由于我們沒有讓視圖來移動這個高亮代理，我們需要控制這個元素如何移動，通過Behavior on y來完成這個操作，在下面的例子中，移動分為三步完成：淡出，移動，淡入。注意怎樣使用SequentialAnimation和PropertyAnimation元素與NumberAnimation結合創建更加復雜的移動效果。

```
    Component {
        id: highlightComponent

        Item {
            width: ListView.view.width
            height: ListView.view.currentItem.height

            y: ListView.view.currentItem.y

            Behavior on y {
                SequentialAnimation {
                    PropertyAnimation { target: highlightRectangle; property: "opacity";
                    NumberAnimation { duration: 1 }
                    PropertyAnimation { target: highlightRectangle; property: "opacity";
                }
            }

            Rectangle {
                id: highlightRectangle
                anchors.fill: parent
                color: "lightGreen"
            }
        }
    }
```

6.3.3 頁眉與頁腳（Header and Footer）

這一節是鏈表視圖最後的內容，我們能夠向鏈表視圖中插入一個頁眉（header）元素和一個頁腳（footer）元素。這部分是鏈表的開始或者結尾處被作為代理元素特殊的區域。對于一個水平鏈表視圖，不會存在頁眉或者頁腳，但是也有開始和結尾處，這取決于layoutDirection的設置。

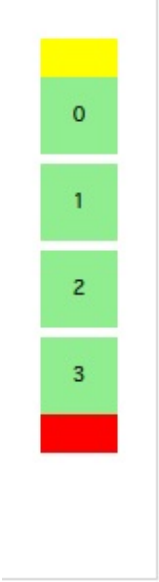
下面這個例子展示了如何使用一個頁眉和頁腳來突出鏈表的開始與結尾。這些特殊的鏈表元素也有其它的作用，例如，它們能夠保持鏈表中的按鍵加載更多的內容。

```
import QtQuick 2.0

Rectangle {
    width: 80
    height: 300

    color: "white"

    ListView {
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: 4

        delegate: numberDelegate
        spacing: 5

        header: headerComponent
        footer: footerComponent
    }

    Component {
        id: headerComponent

        Rectangle {
            width: 40
            height: 20

            color: "yellow"
        }
    }

    Component {
        id: footerComponent

        Rectangle {
            width: 40
            height: 20

            color: "red"
        }
    }

    Component {
        id: numberDelegate

        Rectangle {
            width: 40
            height: 40

            color: "lightGreen"

            Text {
                anchors.centerIn: parent

                font.pixelSize: 10
```

```
                text: index
            }
        }
    }
}
```

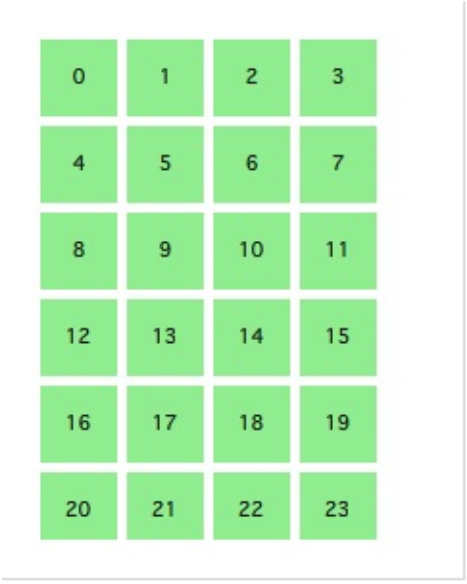
注意

頁眉與頁腳代理元素不遵循鏈表視圖（**ListView**）的間隔（**spacing**）屬性，它們被直接放在相鄰的鏈表元素之上或之下。這意味著頁眉與頁腳的間隔必須通過頁眉與頁腳元素自己設置。



6.3.4 網格視圖（The GridView）

使用網格視圖（GridView）與使用鏈表視圖（ListView）的方式非常類似。真正不同的地方是網格視圖（GridView）使用了一個二維數組來存放元素，而鏈表視圖（ListView）是使用的線性鏈表來存放元素。



與鏈表視圖（ListView）比較，網格視圖（GridView）不依賴于元素間隔和大小來配置元素。它使用單元寬度（cellWidth）與單元高度（cellHeight）屬性來控制數組內的二維元素的內容。每個元素從左上角開始依次放入單元格。

```
import QtQuick 2.0

Rectangle {
    width: 240
    height: 300

    color: "white"

    GridView {
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: 100

        cellWidth: 45
        cellHeight: 45

        delegate: numberDelegate
    }

    Component {
        id: numberDelegate

        Rectangle {
            width: 40
            height: 40

            color: "lightGreen"

            Text {
                anchors.centerIn: parent

                font.pixelSize: 10

                text: index
            }
        }
    }
}
```

一個網格視圖（GridView）也包含了頁腳與頁眉，也可以使用高亮代理並且支持捕捉模式（snap mode）的多種反彈行為。它也可以使用不同的方向（orientations）與定向（directions）來定位。

定向使用flow屬性來控制。它可以被設置為GridView.LeftToRight或者GridView.TopToBottom。模型的值從左往右向網格中填充，行添加是從上往下。視圖使用一個垂直方向的滾動條。後面添加的元素也是由上到下，由左到右。

此外還有flow屬性和layoutDirection屬性，能夠適配網格從左到右或者從右到左，這依賴于你使用的設置值。

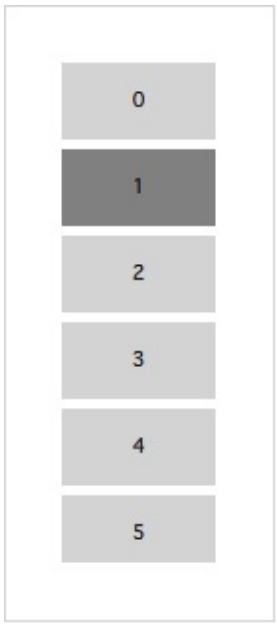
代理（**Delegate**）

當使用模型與視圖來自定義用戶界面時，代理在創建顯示時扮演了大量的角色。在模型中的每個元素通過代理來實現可視化，用戶真實可見的是這些代理元素。

每個代理訪問到索引號或者綁定的屬性，一些是來自數據模型，一些來自視圖。來自模型的數據將會通過屬性傳遞到代理。來自視圖的數據將會通過屬性傳遞視圖中與代理相關的狀態信息。

通常使用的視圖綁定屬性是ListView.isCurrentItem和ListView.view。第一個是一個布爾值，標識這個元素是否是視圖當前元素，這個值是只讀的，引用自當前視圖。通過訪問視圖，可以創建可復用的代理，這些代理在被包含時會自動匹配視圖的大小。在下面這個例子中，每個代理的width（寬度）屬性與視圖的width（寬度）屬性綁定，每個代理的背景顏色color依賴于綁定的屬性ListView.isCurrentItem屬性。

```
import QtQuick 2.0

Rectangle {
    width: 120
    height: 300

    color: "white"

    ListView {
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: 100

        delegate: numberDelegate
        spacing: 5

        focus: true
    }

    Component {
        id: numberDelegate

        Rectangle {
            width: ListView.view.width
            height: 40

            color: ListView.isCurrentItem?"gray":"lightGray"

            Text {
                anchors.centerIn: parent

                font.pixelSize: 10

                text: index
            }
        }
    }
}
```



如果在模型中的每個元素與一個動作相關，例如點擊作用于一個元素時，這個功能是代理完成的。這是由事件管理分配給視圖的，這個操作控制了視圖中元素的導航，代理控制了特定元素上的動作。

最基礎的方法是在每個代理中創建一個MouseArea（鼠標區域）並且響應onClicked信號。在後面章節中將會演示這個例子。

6.4.1 動畫添加與移除元素（Animating Added and Removed Items）

在某些情況下，視圖中的顯示內容會隨著時間而改變。由于模型數據的改變，元素會添加或者移除。在這些情況下，一個比較好的做法是使用可視化隊列給用戶一個方向的感覺來幫助用戶知道哪些數據被加入或者移除。

為了方便使用，QML視圖為每個代理綁定了兩個信號，onAdd和onRemove。使用動畫連接它們，可以方便創建識別哪些內容被添加或刪除的動畫。

下面這個例子演示了如何動態填充一個鏈表模型（ListModel）。在屏幕下方，有一個添加新元素的按鈕。當點擊它時，會調用模型的append方法來添加一個新的元素。這個操作會觸發視圖創建一個新的代理，並發送GridView.onAdd信號。SequentialAnimation隊列動畫與這個信號連接綁定，使用代理的scale屬性來放大視圖元素。

當視圖中的一個代理點擊時，將會調用模型的remove方法將一個元素從模型中移除。這個操作將會導致GridView.onRemove信號的發送，觸發另一個SequentialAnimation。這時，代理的銷毀將會延遲直到動畫完成。為了完成這個操作，PropertyAction元素需要在動畫前設置GridView.delayRemove屬性為true，並在動畫後設置為false。這樣確保了動畫在代理項移除前完成。

```
import QtQuick 2.0

Rectangle {
    width: 480
    height: 300

    color: "white"

    ListModel {
```

```
        id: theModel

        ListElement { number: 0 }
        ListElement { number: 1 }
        ListElement { number: 2 }
        ListElement { number: 3 }
        ListElement { number: 4 }
        ListElement { number: 5 }
        ListElement { number: 6 }
        ListElement { number: 7 }
        ListElement { number: 8 }
        ListElement { number: 9 }
    }

    Rectangle {
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        anchors.margins: 20

        height: 40

        color: "darkGreen"

        Text {
            anchors.centerIn: parent

            text: "Add item!"
        }

        MouseArea {
            anchors.fill: parent

            onClicked: {
                theModel.append({"number": ++parent.count});
            }
        }

        property int count: 9
    }

    GridView {
        anchors.fill: parent
        anchors.margins: 20
        anchors.bottomMargin: 80

        clip: true

        model: theModel

        cellWidth: 45
        cellHeight: 45

        delegate: numberDelegate
    }

    Component {
        id: numberDelegate

        Rectangle {
            id: wrapper

            width: 40
```

```
            height: 40

            color: "lightGreen"

            Text {
                anchors.centerIn: parent

                font.pixelSize: 10

                text: number
            }

            MouseArea {
                anchors.fill: parent

                onClicked: {
                    if (!wrapper.GridView.delayRemove)
                        theModel.remove(index);
                }
            }

            GridView.onRemove: SequentialAnimation {
                PropertyAction { target: wrapper; property: "GridView.delayRemove"; value
                NumberAnimation { target: wrapper; property: "scale"; to: 0; duration: 25
                PropertyAction { target: wrapper; property: "GridView.delayRemove"; value
            }

            GridView.onAdd: SequentialAnimation {
                NumberAnimation { target: wrapper; property: "scale"; from: 0; to: 1; dur
            }
        }
    }
}
```

6.4.2 形變的代理（Shape-Shifting Delegates）

在使用鏈表時通常會使用當前項激活時展開的機制。這個操作可以被用于動態的將當前項目填充到整個屏幕來添加一個新的用戶界面，或者為鏈表中的當前項提供更多的信息。

在下面的例子中，當點擊鏈表項時，鏈表項都會展開填充整個鏈表視圖（ListView）。額外的間隔區域被用于添加更多的信息，這種機制使用一個狀態來控制，當一個鏈表項展開時，代理項都能輸入expanded（展開）狀態，在這種狀態下一些屬性被改變。

首先，包裝器（wrapper）的高度（height）被設置為鏈表視圖（ListView）的高度。標簽圖片被放大並且下移，使圖片從小圖片的位置移向大圖片的位置。除了這些之外，兩個隱藏項，實際視圖（factsView）與關閉按鍵（closeButton）切換它的opactiy（透明度）顯示出來。最後設置鏈表視圖（ListView）。

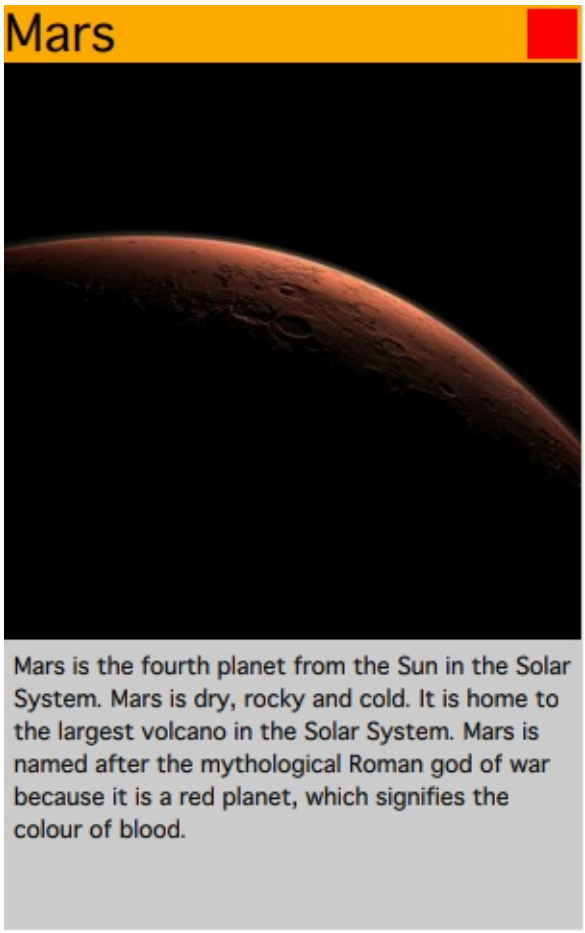
設置鏈表視圖（ListView）包含了設置內容Y坐標（contentsY），這是視圖頂部可見的部分代理的Y軸坐標。另一個變化是設置視圖的交互（interactive）為false。這個操作阻止了視圖的移動，用戶不再能夠通過滾動條切換當前項。

由于設置第一個鏈表項為可點擊，向它輸入一個expanded（展開）狀態，導致了它的代理項被填充到整個鏈表並且內容重置。當點擊關閉按鈕時，清空狀態，導致它的代理項返回上一個狀態，並且重新設置鏈表視圖（ListView）有效。

```
import QtQuick 2.0

Item {
    width: 300
    height: 480

    ListView {
        id: listView

        anchors.fill: parent

        delegate: detailsDelegate
        model: planets
    }

    ListModel {
        id: planets

        ListElement { name: "Mercury"; imageSource: "images/mercury.jpeg"; facts: "Mercur
        ListElement { name: "Venus"; imageSource: "images/venus.jpeg"; facts: "Venus is t
        ListElement { name: "Earth"; imageSource: "images/earth.jpeg"; facts: "The Earth
        ListElement { name: "Mars"; imageSource: "images/mars.jpeg"; facts: "Mars is the
    }

    Component {
        id: detailsDelegate

        Item {
            id: wrapper

            width: listView.width
            height: 30

            Rectangle {
                anchors.left: parent.left
                anchors.right: parent.right
                anchors.top: parent.top

                height: 30

                color: "#ffaa00"

                Text {
                    anchors.left: parent.left
                    anchors.verticalCenter: parent.verticalCenter

                    font.pixelSize: parent.height-4

                    text: name
                }
            }

            Rectangle {
                id: image

                color: "black"

                anchors.right: parent.right
                anchors.top: parent.top
                anchors.rightMargin: 2
                anchors.topMargin: 2
```

```
                width: 26
                height: 26

                Image {
                    anchors.fill: parent

                    fillMode: Image.PreserveAspectFit

                    source: imageSource
                }
            }

            MouseArea {
                anchors.fill: parent
                onClicked: parent.state = "expanded"
            }

            Item {
                id: factsView

                anchors.top: image.bottom
                anchors.left: parent.left
                anchors.right: parent.right
                anchors.bottom: parent.bottom

                opacity: 0

                Rectangle {
                    anchors.fill: parent

                    color: "#cccccc"

                    Text {
                        anchors.fill: parent
                        anchors.margins: 5

                        clip: true
                        wrapMode: Text.WordWrap

                        font.pixelSize: 12

                        text: facts
                    }
                }
            }

            Rectangle {
                id: closeButton

                anchors.right: parent.right
                anchors.top: parent.top
                anchors.rightMargin: 2
                anchors.topMargin: 2

                width: 26
                height: 26

                color: "red"

                opacity: 0

                MouseArea {
                    anchors.fill: parent
```

```
                    onClicked: wrapper.state = ""
                }
            }

            states: [
                State {
                    name: "expanded"

                    PropertyChanges { target: wrapper; height: listView.height }
                    PropertyChanges { target: image; width: listView.width; height: listV
                    PropertyChanges { target: factsView; opacity: 1 }
                    PropertyChanges { target: closeButton; opacity: 1 }
                    PropertyChanges { target: wrapper.ListView.view; contentY: wrapper.y;
                }
            ]

            transitions: [
                Transition {
                    NumberAnimation {
                        duration: 200;
                        properties: "height,width,anchors.rightMargin,anchors.topMargin,o
                    }
                }
            ]
        }
    }
}
```



這個技術展示了展開代理來填充視圖能夠簡單的通過代理的形變來完成。例如當瀏覽一個歌曲的鏈表時，可以通過放大當前項來對該項添加更多的說明。

高級用法（**Advanced Techniques**）

6.5.1 路徑視圖（**The PathView**）

路徑視圖（PathView）非常強大，但也非常復雜，這個視圖由QtQuick提供。它創建了一個可以讓子項沿著任意路徑移動的視圖。沿著相同的路徑，使用縮放（scale），透明（opacity）等元素可以更加詳細的控制過程。

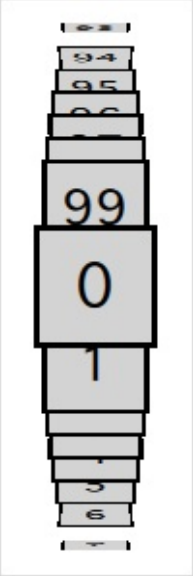
當使用路徑視圖（PathView）時，你必須定義一個代理和一個路徑。在這些之上，路徑視圖（PathView）本身也可以自定義一些屬性的區間。通常會使用pathItemCount屬性，它控制了一次可見的子項總數。preferredHighLightBegin屬性控制了高亮區間，preferredHighlightEnd與highlightRangeMode，控制了當前項怎樣沿著路徑顯示。

在關注高亮區間之前，我們必須先看看路徑（path）這個屬性。路徑（path）屬性使用一個路徑（path）元素來定義路徑視圖（PathView）內代理的滾動路徑。路徑使用startx與starty屬性來鏈接路徑（path）元素，例如PathLine,PathQuad和PathCubic。這些元素都使用二維數組來構造路徑。

當路徑定義好之後，可以使用PathPercent和PathAttribute元素來進一步設置。它們被放置在路徑元素之間，並且為經過它們的路徑和代理提供更加細致的控制。PathPercent提供了如何控制每個元素之間覆蓋區域部分的路徑，然後反過來控制分布在這條路徑上的代理元素，它們被按比例的分布播放。

preferredHightlightBegin與preferredHighlightEnd屬性由PathView（路徑視圖）輸入到圖片元素中。它們的值在0~1之間。結束值大于等于開始值。例如設置這些屬性值為0.5，當前項只會顯示當前百分之50的圖像在這個路徑上。

在Path中，PathAttribute元素也是被放置在元素之間的，就像PathPercent元素。它們可以讓你指定屬性的值然後插入的路徑中去。這些屬性與代理綁定可以用來控制任意的屬性。



下面這個例子展示了路徑視圖（PathView）如何創建一個卡片視圖，並且用戶可以滑動它。我們使用了一些技巧來完成這個例子。路徑由PathLine元素組成。使用PathPercent元素，它確保了中間的元素居中，並且給其它的元素提供了足夠的空間。使用PathAttribute元素來控制旋轉，大小和深度值（z-value）。

在這個路徑之上（path），需要設置路徑視圖（PathView）的pathItemCount屬性。它控制了路徑的濃密度。路徑視圖的路徑（PathView.onPath）使用preferredHighlightBegin與preferredHighlightEnd來控制可

見的代理項。

```
    PathView {
        anchors.fill: parent

        delegate: flipCardDelegate
        model: 100

        path: Path {
            startX: root.width/2
            startY: 0

            PathAttribute { name: "itemZ"; value: 0 }
            PathAttribute { name: "itemAngle"; value: -90.0; }
            PathAttribute { name: "itemScale"; value: 0.5; }
            PathLine { x: root.width/2; y: root.height*0.4; }
            PathPercent { value: 0.48; }
            PathLine { x: root.width/2; y: root.height*0.5; }
            PathAttribute { name: "itemAngle"; value: 0.0; }
            PathAttribute { name: "itemScale"; value: 1.0; }
            PathAttribute { name: "itemZ"; value: 100 }
            PathLine { x: root.width/2; y: root.height*0.6; }
            PathPercent { value: 0.52; }
            PathLine { x: root.width/2; y: root.height; }
            PathAttribute { name: "itemAngle"; value: 90.0; }
            PathAttribute { name: "itemScale"; value: 0.5; }
            PathAttribute { name: "itemZ"; value: 0 }
        }

        pathItemCount: 16

        preferredHighlightBegin: 0.5
        preferredHighlightEnd: 0.5
    }
```

代理如下面所示，使用了一些從PathAttribute中鏈接的屬性，itemZ,itemAngle和itemScale。需要注意代理鏈接的屬性只在wrapper中可用。因此，rotxs屬性在Rotation元素中定義為可訪問值。

另一個需要注意的是路徑視圖（PathView）鏈接的PathView.onPath屬性的用法。通常對于這個屬性都綁定為可見，這樣允許路徑視圖（PathView）緩衝不可見的元素。這不是通過剪裁處理來實現的，因為路徑視圖（PathView）的代理比其它的視圖，例如鏈表視圖（ListView）或者柵格視圖（GridView）放置更加隨意。

```
    Component {
        id: flipCardDelegate

        Item {
            id: wrapper

            width: 64
            height: 64

            visible: PathView.onPath

            scale: PathView.itemScale
            z: PathView.itemZ

            property variant rotX: PathView.itemAngle
```

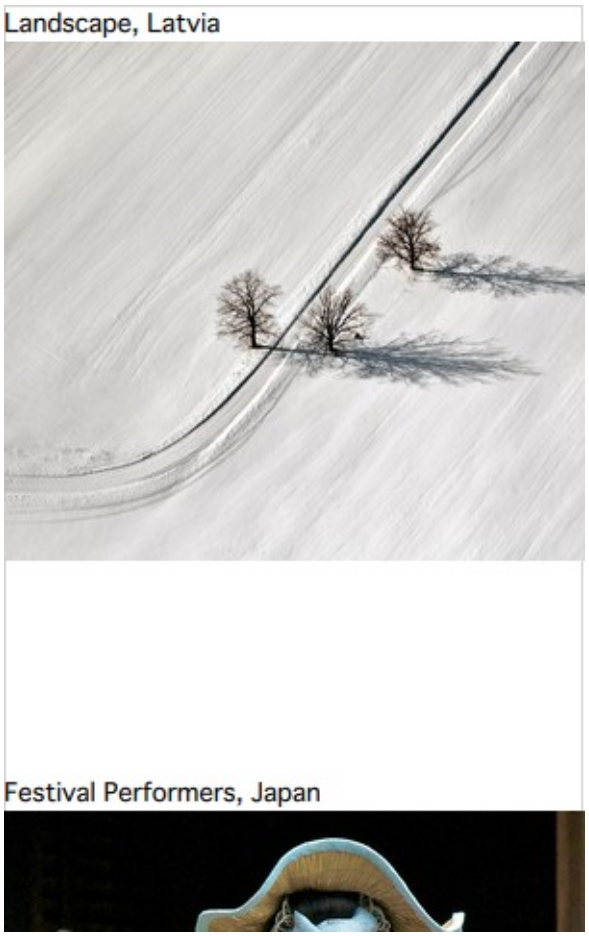
```
            transform: Rotation { axis { x: 1; y: 0; z: 0 } angle: wrapper.rotX; origin {

            Rectangle {
                anchors.fill: parent
                color: "lightGray"
                border.color: "black"
                border.width: 3
            }

            Text {
                anchors.centerIn: parent
                text: index
                font.pixelSize: 30
            }
        }
    }
```

當在路徑視圖（PathView）上使用圖像轉換或者其它更加復雜的元素時，有一個性能優化的技巧是綁定圖像元素（Image）的smooth屬性與PathView.view.moving屬性。這意味著圖像在移動時可能不夠完美，但是能夠比較平滑的轉換。當視圖在移動時，對于平滑縮放的處理是沒有意義的，因為用戶根本看不見這個過程。

6.5.2 XML模型（A Model from XML）

由于XML是一種常見的數據格式，QML提供了XmlListModel元素來包裝XML數據。這個元素能夠獲取本地或者網絡上的XML數據，然後通過XPath解析這些數據。

下面這個例子展示了從RSS流中獲取圖片，源屬性（source）引用了一個網絡地址，這個數據會自動下載。



當數據下載完成後，它會被加工作為模型的子項。查詢屬性（query）是一個XPath代理的基礎查詢，用來創建模型項。在這個例子中，這個路徑是/rss/channel/item，所以，在一個模型子項創建後，每一個子項的標簽，都包含了一個頻道標簽，包含一個RSS標簽。

每一個模型項，一些規則需要被提取，由XmlRole元素來代理。每一個規則都需要一個名稱，這樣代理才能夠通過屬性綁定來訪問。每個這樣的屬性的值都通過XPath查詢來確定。例如標題屬性（title）符合title/string()查詢，返回內容中在之間的值。

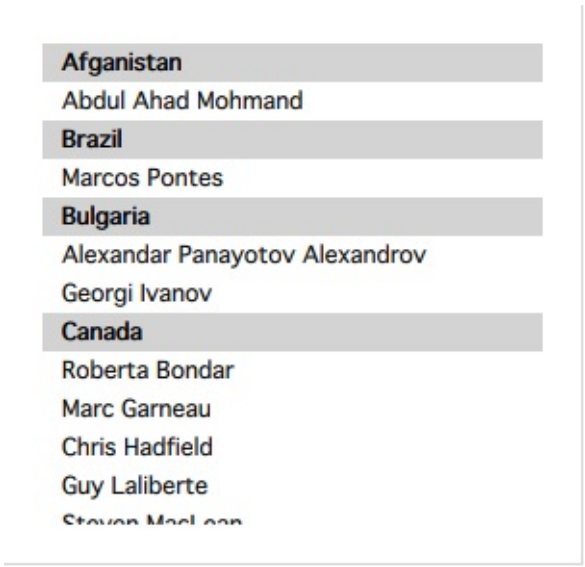
圖像源屬性（imageSource）更加有趣，因為它不僅僅是從XML中提取字符串，也需要加載它。在流數據的支持下，每個子項包含了一個圖片。使用XPath的函數substring-after與substring-before，可以提取本地的圖片資源。這樣imageSource屬性就可以直接被作為一個Image元素的source屬性使用。

```
import QtQuick 2.0
import QtQuick.XmlListModel 2.0

Item {
    width: 300
    height: 480

    Component {
        id: imageDelegate

        Item {
            width: listView.width
            height: 400

        Column {
                Text {
                    text: title
                }
```

```
                Image {
                    source: imageSource
                }
            }
        }
    }

    XmlListModel {
        id: imageModel

        source: "http://feeds.nationalgeographic.com/ng/photography/photo-of-the-day/"
        query: "/rss/channel/item"

        XmlRole { name: "title"; query: "title/string()" }
        XmlRole { name: "imageSource"; query: "substring-before(substring-after(descripti
    }

    ListView {
        id: listView

        anchors.fill: parent

        model: imageModel
        delegate: imageDelegate
    }
}
```

6.5.3 鏈表分段（Lists with Sections）

有時，鏈表的數據需要劃分段。例如使用首字母來劃分聯系人，或者音樂。使用鏈表視圖可以把平面列表按類別劃分。



為了使用分段，section.property與section.criteria必須安裝。section.property定義了哪些屬性用于內容的劃分。在這裡，最重要的是知道每一段由哪些連續的元素構成，否則相同的屬性名可能出現在幾個不同的地方。

section.criteria能夠被設置為ViewSection.FullString或者ViewSection.FirstCharacter。默認下使用第一個值，能夠被用于模型中有清晰的分段，例如音樂專輯。第二個是使用一個屬性的首字母來分段，這說明任

何屬性都可以被使用。通常的例子是用于聯系人名單的姓。

當段被定義好後，每個子項能夠使用綁定屬性ListView.section，ListView.previousSection與ListView.nextSection來訪問。使用這些屬性，可以檢測段的第一個與最後一個子項。

使用鏈表視圖（ListView）的section.delegate屬性可以給段指定代理組件。它能夠創建段標題，並且可以在任意子項之前插入這個段代理。使用綁定屬性section可以訪問當前段的名稱。

下面這個例子使用國際分類展示了分段的一些概念。國籍（nation）作為section.property，段代理組件（section.delegate）使用每個國家作為標題。在每個段中，spacemen模型中的名字使用spaceManDelegate組件來代理顯示。

```
import QtQuick 2.0

Rectangle {
    width: 300
    height: 290

    color: "white"

    ListView {
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: spaceMen

        delegate: spaceManDelegate

        section.property: "nation"
        section.delegate: sectionDelegate
    }

    Component {
        id: spaceManDelegate

        Item {
            width: 260
            height: 20

            Text {
                anchors.left: parent.left
                anchors.verticalCenter: parent.verticalCenter
                anchors.leftMargin: 10

                font.pixelSize: 12

                text: name
            }
        }
    }

    Component {
        id: sectionDelegate

        Rectangle {
            width: 260
            height: 20
```

```
            color: "lightGray"

            Text {
                anchors.left: parent.left
                anchors.verticalCenter: parent.verticalCenter
                anchors.leftMargin: 10

                font.pixelSize: 12
                font.bold: true

                text: section
            }
        }
    }


    ListModel {
        id: spaceMen

        ListElement { name: "Abdul Ahad Mohmand"; nation: "Afganistan"; }
        ListElement { name: "Marcos Pontes"; nation: "Brazil"; }
        ListElement { name: "Alexandar Panayotov Alexandrov"; nation: "Bulgaria"; }
        ListElement { name: "Georgi Ivanov"; nation: "Bulgaria"; }
        ListElement { name: "Roberta Bondar"; nation: "Canada"; }
        ListElement { name: "Marc Garneau"; nation: "Canada"; }
        ListElement { name: "Chris Hadfield"; nation: "Canada"; }
        ListElement { name: "Guy Laliberte"; nation: "Canada"; }
        ListElement { name: "Steven MacLean"; nation: "Canada"; }
        ListElement { name: "Julie Payette"; nation: "Canada"; }
        ListElement { name: "Robert Thirsk"; nation: "Canada"; }
        ListElement { name: "Bjarni Tryggvason"; nation: "Canada"; }
        ListElement { name: "Dafydd Williams"; nation: "Canada"; }
    }
}
```

6.5.4 性能協調（Tunning Performance）

一個模型視圖的性能很大程度上依賴于代理的創建。例如滾動下拉一個鏈表視圖時，代理從外部加入到視圖底部，並且從視圖頂部移出。如果設置剪裁（clip）屬性為false，並且代理項花了很多時間來初始化，用戶會感覺到視圖滾動體驗很差。

為了優化這個問題，你可以在滾動時使用像素來調整。使用cacheBuffer屬性，在上訴情況下的垂直滾動，它將會調整在鏈表視圖的上下需要預先準備好多少像素的代理項，結合異步加載圖像元素（Image），例如在它們進入視圖之前加載。

創建更多的代理項將會犧牲一些流暢的體驗，並且花更多的時間來初始化每個代理。這並不代表可以解決一些更加復雜的代理項的問題。在每次實例化代理時，它的內容都會被評估和編輯。這需要花費時間，如果它花費了太多的時間，它將會導致一個很差的滾動體驗。在一個代理中包含太多的元素也會降低滾動的性能。

為了補救這個問題，我們推薦使用動態加載元素。當它們需要時，可以初始化這些附加的元素。例如，一個展開代理可能推遲它的詳細內容的實例化，直到需要使用它時。每個代理中最好減少JavaScript的數量。將每個代理中復雜的JavaScript調用放在外面來實現。這將會減少每個代理在創建時編譯JavaScript。

總結（**Summary**）

在這個章節中，我們學習了模型，視圖與代理。每個數據的入口是模型，視圖通過可視化代理來實現數據的可視化。將數據從顯示中分離出來。

一個模型可以是一個整數，提供給代理使用的索引值（index ）。如果JavaScript數組被作為一個模型，模型數據變量（modelData）代表了數組的數據的當前索引。對于更加復雜的情況，每個數據項需要提供多個值，使用鏈表模型（ListModel）與鏈表元素（ListElement）是一個更好的解決辦法。

對于靜態模型，一個Repeater可以被用作視圖。它可以非常方便的使用行（Row），列（Column），柵格（Grid），或者流（Flow）來創建用戶界面。對于動態或者大的數據模型，使用ListView或者GridView更加適合。它們會在需要時動態的創建代理，減少在場景下一次顯示的元素的數量。

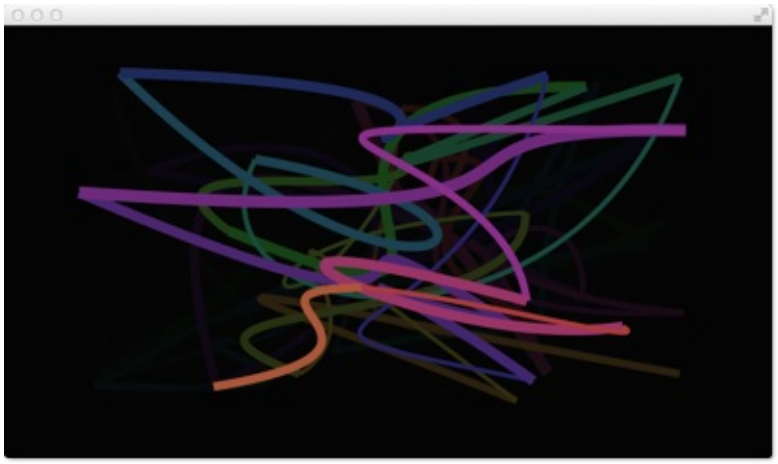
在視圖中的代理可以與數據模型中的屬性靜態綁定，或者動態綁定。使用視圖的onAdd與onRemove信號，可以動態播放的它們的顯示與消失。

Canvas Element

注意

最後一次構建：**2014**年**1**月**20**日下午**18:00**。

這章的源代碼能夠在**assetts folder**找到。



在早些時候的Qt4中加入QML時，一些開發者討論如何在QtQuick中繪制一個圓形。類似圓形的問題，一些開發者也對于其它的形狀的支持進行了討論。在QtQuick中沒有圓形，只有矩形。在Qt4中，如果你需要一個除了矩形外的形狀，你需要使用圖片或者使用你自己寫的C++圓形元素。

Qt5中引進了畫布元素（canvas element），允許腳本繪制。畫布元素（canvas element）提供了一個依賴于分辨率的位圖畫布，你可以使用JavaScript腳本來繪制圖形，制作遊戲或者其它的動態圖像。畫布元素（canvas element）是基于HTML5的畫布元素來完成的。

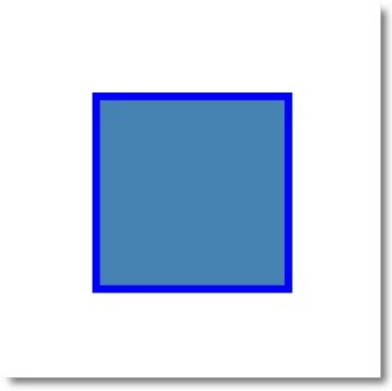
畫布元素（canvas element）的基本思想是使用一個2D對象來渲染路徑。這個2D對象包括了必要的繪圖函數，畫布元素（canvas element）充當繪制畫布。2D對象支持畫筆，填充，漸變，文本和繪制路徑創建命令。

讓我們看看一個簡單的路徑繪制的例子：

```
import QtQuick 2.0

Canvas {
    id: root
    // canvas size
    width: 200; height: 200
    // handler to override for drawing
    onPaint: {
        // get context to draw with
        var ctx = getContext("2d")
        // setup the stroke
        ctx.lineWidth = 4
        ctx.strokeStyle = "blue"
        // setup the fill
        ctx.fillStyle = "steelblue"
        // begin a new path to draw
        ctx.beginPath()
        // top-left start point
        ctx.moveTo(50,50)
```

```
        // upper line
        ctx.lineTo(150,50)
        // right line
        ctx.lineTo(150,150)
        // bottom line
        ctx.lineTo(50,150)
        // left line through path closing
        ctx.closePath()
        // fill using fill style
        ctx.fill()
        // stroke using line width and stroke style
        ctx.stroke()
    }
}
```

這個例子產生了一個在坐標（50,50），高寬為100的填充矩形框，並且使用了畫筆來修飾邊界。



畫筆的寬度被設置為4個像素，並且定義strokeStyle（畫筆樣式）為藍色。最後的形狀由設置填充樣式（fillStyle）為steelblue顏色，然後填充完成的。只有調用stroke或者fill函數，創建的路徑才會繪制，它們與其它的函數使用是相互獨立的。調用stroke或者fill將會繪制當前的路徑，創建的路徑是不可重用的，只有繪制狀態能夠被存儲和恢復。

在QML中，畫布元素（canvas element）充當了繪制的容器。2D繪制對象提供了實際繪制的方法。繪制需要在onPaint事件中完成。

```
Canvas {
    width: 200; height: 200
    onPaint: {
        var ctx = getContext("2d")
        // setup your path
        // fill or/and stroke
    }
}
```

畫布自身提供了典型的二維笛卡爾坐標系統，左上角是（0,0）坐標。Y軸坐標軸向下，X軸坐標軸向右。

典型繪制命令調用如下：

1. 裝載畫筆或者填充模式
2. 創建繪制路徑
3. 使用畫筆或者填充繪制路徑

```
    onPaint: {
        var ctx = getContext("2d")

        // setup the stroke
        ctx.strokeStyle = "red"

        // create a path
        ctx.beginPath()
        ctx.moveTo(50,50)
        ctx.lineTo(150,50)

        // stroke path
        ctx.stroke()
    }
```

這將產生一個從P1（50，50）到P2（150,50）水平線。

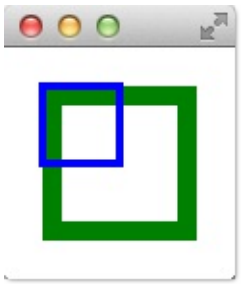
注意

通常在你重置了路徑後你將會設置一個開始點，所以，在**beginPath()**這個操作後，你需要使用**moveTo**來設置開始點。

便捷的接口（**Convenient API**）

在繪制矩形時，我們提供了一個便捷的接口，而不需要調用stroke或者fill來完成。

```
// convenient.qml

import QtQuick 2.0

Canvas {
    id: root
    width: 120; height: 120
    onPaint: {
        var ctx = getContext("2d")
        ctx.fillStyle = 'green'
        ctx.strokeStyle = "blue"
        ctx.lineWidth = 4

        // draw a filles rectangle
        ctx.fillRect(20, 20, 80, 80)
        // cut our an inner rectangle
        ctx.clearRect(30,30, 60, 60)
        // stroke a border from top-left to
        // inner center of the larger rectangle
        ctx.strokeRect(20,20, 40, 40)
    }
}
```



注意

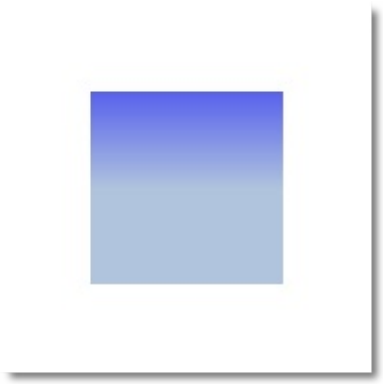
畫筆的繪制區域由中間向兩邊延展。一個寬度為**4**像素的畫筆將會在繪制路徑的裡面繪制**2**個像素，外面繪制**2**個像素。

漸變（Gradients）

畫布中可以使用顏色填充也可以使用漸變或者圖像來填充。

```
    onPaint: {
        var ctx = getContext("2d")

        var gradient = ctx.createLinearGradient(100,0,100,200)
        gradient.addColorStop(0, "blue")
        gradient.addColorStop(0.5, "lightsteelblue")
        ctx.fillStyle = gradient
        ctx.fillRect(50,50,100,100)
    }
```

在這個例子中，漸變色定義在開始點（100,0）到結束點（100,200）。在我們畫布中是一個中間垂直的線。漸變色在停止點定義一個顏色，範圍從0.0到1.0。這裡我們使用一個藍色作為0.0（100,0），一個高亮剛藍色作為0.5（100,200）。漸變色的定義比我們想要繪制的矩形更大，所以矩形在它定義的範圍內對漸變進行了裁剪。



注意

漸變色是在畫布坐標下定義的，而不是在繪制路徑相對坐標下定義的。畫布中沒有相對坐標的概念。

陰影（**Shadows**）

注意

在**Qt5**的**alpha**版本中，我們使用陰影遇到了一些問題。

2D對象的路徑可以使用陰影增強顯示效果。陰影是一個區域的輪廓線使用偏移量，顏色和模糊來實現的。所以你需要指定一個陰影顏色（shadowColor），陰影X軸偏移值（shadowOffsetX），陰影Y軸偏移值（shadowOffsetY）和陰影模糊（shadowBlur）。這些參數的定義都使用2D context來定義。2D context是唯一的繪制操作接口。

陰影也可以用來創建發光的效果。在下面的例子中我們使用白色的光創建了一個“Earth”的文本。在一個黑色的背景上可以有更加好的顯示效果。

首先我們繪制黑色背景：

```
        // setup a dark background
        ctx.strokeStyle = "#333"
        ctx.fillRect(0,0,canvas.width,canvas.height);
```

然後定義我們的陰影配置：

```
        ctx.shadowColor = "blue";
        ctx.shadowOffsetX = 2;
        ctx.shadowOffsetY = 2;
        // next line crashes
        // ctx.shadowBlur = 10;
```

最後我們使用加粗的，80像素寬度的Ubuntu字體來繪制“Earth”文本：

```
        ctx.font = 'Bold 80px Ubuntu';
        ctx.fillStyle = "#33a9ff";
        ctx.fillText("Earth",30,180);
```

圖片（**Images**）

QML畫布支持多種資源的圖片繪制。在畫布中使用一個圖片需要先加載圖片資源。在我們的例子中我們使用Component.onCompleted操作來加載圖片。

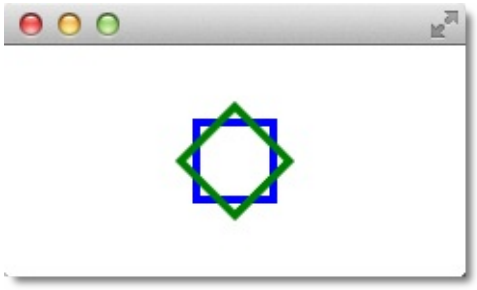
```
    onPaint: {
        var ctx = getContext("2d")


        // draw an image
        ctx.drawImage('assets/ball.png', 10, 10)

        // store current context setup
        ctx.save()
        ctx.strokeStyle = 'red'
        // create a triangle as clip region
        ctx.beginPath()
        ctx.moveTo(10,10)
        ctx.lineTo(55,10)
        ctx.lineTo(35,55)
        ctx.closePath()
        // translate coordinate system
        ctx.translate(100,0)
        ctx.clip()  // create clip from triangle path
        // draw image with clip applied
        ctx.drawImage('assets/ball.png', 10, 10)
        // draw stroke around path
        ctx.stroke()
        // restore previous setup
        ctx.restore()

    }

    Component.onCompleted: {
        loadImage("assets/ball.png")
    }
```

在左邊，足球圖片使用10×10的大小繪制在左上方的位置。在右邊我們對足球圖片進行了裁剪。圖片或者輪廓路徑都可以使用一個路徑來裁剪。裁剪需要定義一個裁剪路徑，然後調用clip()函數來實現裁剪。在clip()之前所有的繪制操作都會用來進行裁剪。如果還原了之前的狀態或者定義裁剪區域為整個畫布時，裁剪是無效的。

轉換（Transformation）

畫布有多種方式來轉換坐標系。這些操作非常類似于QML元素的轉換。你可以通過縮放（scale），旋轉（rotate），translate（移動）來轉換坐標系。與QML元素的轉換不同的是，轉換原點通常就是畫布原點。例如，從中心點放大一個封閉的路徑，你需要先將畫布原點移動到整個封閉的路徑的中心點上。使用這些轉換的方法你可以創建一些更加復雜的轉換。

```
// transform.qml

import QtQuick 2.0

Canvas {
    id: root
    width: 240; height: 120
    onPaint: {
        var ctx = getContext("2d")
        ctx.strokeStyle = "blue"
        ctx.lineWidth = 4

        ctx.beginPath()
        ctx.rect(-20, -20, 40, 40)
        ctx.translate(120,60)
        ctx.stroke()

        // draw path now rotated
        ctx.strokeStyle = "green"
        ctx.rotate(Math.PI/4)
        ctx.stroke()
    }
}
```



除了移動畫布外，也可以使用scale(x,y)來縮放x,y坐標軸。旋轉使用rotate(angle)，angle是角度（360度=2*Math.PI）。使用setTransform(m11,m12,m21,m22,dx,dy)來完成矩陣轉換。

警告

QML畫布中的轉換與**HTML5**畫布中的機制有些不同。不確定這是不是一個**Bug**。

注意

重置矩陣你可以調用**resetTransform()**函數來完成，這個函數會將轉換矩陣還原為單位矩陣。

組合模式（**Composition Mode**）

組合允許你繪制一個形狀然後與已有的像素點集合混合。畫布提供了多種組合模式，使用globalCompositeOperation(mode)來設置。

- "source-over"
- "source-in"
- "source-out"
- "source-atop"

```
    onPaint: {
        var ctx = getContext("2d")
        ctx.globalCompositeOperation = "xor"
        ctx.fillStyle = "#33a9ff"

        for(var i=0; i<40; i++) {
            ctx.beginPath()
            ctx.arc(Math.random()*400, Math.random()*200, 20, 0, 2*Math.PI)
            ctx.closePath()
            ctx.fill()
        }
    }
```

下面這個例子遍歷了列表中的組合模式，使用對應的組合模式生成了一個矩形與圓形的組合。

```
    property var operation : [
        'source-over', 'source-in', 'source-over',
        'source-atop', 'destination-over', 'destination-in',
        'destination-out', 'destination-atop', 'lighter',
        'copy', 'xor', 'qt-clear', 'qt-destination',
        'qt-multiply', 'qt-screen', 'qt-overlay', 'qt-darken',
        'qt-lighten', 'qt-color-dodge', 'qt-color-burn',
        'qt-hard-light', 'qt-soft-light', 'qt-difference',
        'qt-exclusion'
        ]

    onPaint: {
        var ctx = getContext('2d')

        for(var i=0; i<operation.length; i++) {
            var dx = Math.floor(i%6)*100
            var dy = Math.floor(i/6)*100
            ctx.save()
            ctx.fillStyle = '#33a9ff'
            ctx.fillRect(10+dx,10+dy,60,60)
            // TODO: does not work yet
            ctx.globalCompositeOperation = root.operation[i]
            ctx.fillStyle = '#ff33a9'
            ctx.globalAlpha = 0.75
            ctx.beginPath()
            ctx.arc(60+dx, 60+dy, 30, 0, 2*Math.PI)
            ctx.closePath()
            ctx.fill()
```

```
            ctx.restore()
        }
    }
```

像素緩衝（**Pixels Buffer**）

當你使用畫布時，你可以檢索讀取畫布上的像素數據，或者操作畫布上的像素。讀取圖像數據使用createImageData(sw,sh)或者getImageData(sx,sy,sw,sh)。這兩個函數都會返回一個包含寬度（width），高度（height）和數據（data）的圖像數據（ImageData）對象。圖像數據包含了一維數組像素數據，使用RGBA格式進行檢索。每個數據的數據範圍在0到255之間。設置畫布的像素數據你可以使用putImageData(imagedata,dx,dy)函數來完成。

另一種檢索畫布內容的方法是將畫布的數據存儲進一張圖片中。可以使用畫布的函數save(path)或者toDataURL(mimeType)來完成，toDataURL(mimeType)會返回一個圖片的地址，這個鏈接可以直接用Image元素來讀取。

```
import QtQuick 2.0

Rectangle {
    width: 240; height: 120
    Canvas {
        id: canvas
        x: 10; y: 10
        width: 100; height: 100
        property real hue: 0.0
        onPaint: {
            var ctx = getContext("2d")
            var x = 10 + Math.random(80)*80
            var y = 10 + Math.random(80)*80
            hue += Math.random()*0.1
            if(hue > 1.0) { hue -= 1 }
            ctx.globalAlpha = 0.7
            ctx.fillStyle = Qt.hsla(hue, 0.5, 0.5, 1.0)
            ctx.beginPath()
            ctx.moveTo(x+5,y)
            ctx.arc(x,y, x/10, 0, 360)
            ctx.closePath()
            ctx.fill()
        }
        MouseArea {
            anchors.fill: parent
            onClicked: {
                var url = canvas.toDataURL('image/png')
                print('image url=', url)
                image.source = url
            }
        }
    }

    Image {
        id: image
        x: 130; y: 10
        width: 100; height: 100
    }

    Timer {
        interval: 1000
        running: true
        triggeredOnStart: true
        repeat: true
        onTriggered: canvas.requestPaint()
```

```
    }
}
```

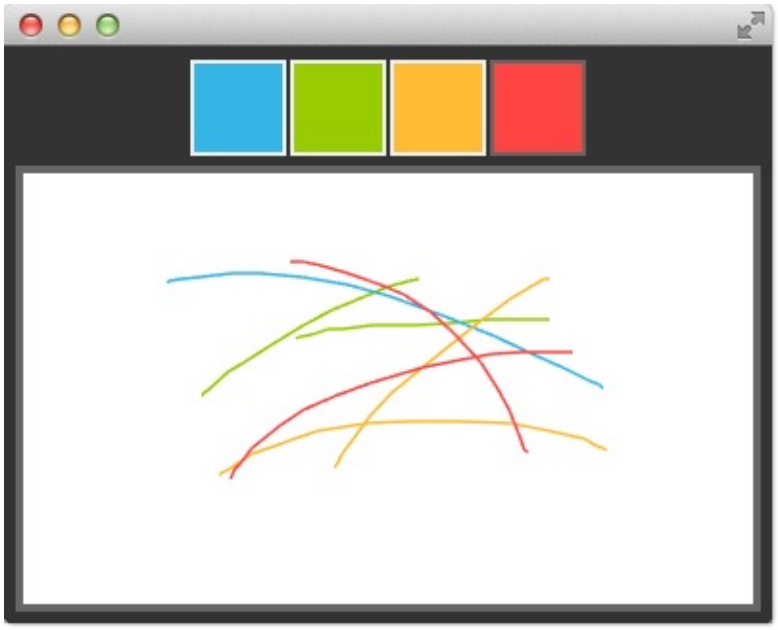
在我們這個例子中，我們每秒在左邊的畫布中繪制一個的圓形。當使用鼠標點擊畫布內容時，會將內容存儲為一個圖片鏈接。在右邊將會展示這個存儲的圖片。

注意

在**Qt5**的**AIpha**版本中，檢索圖像數據似乎不能工作。

畫布繪制（Canvas Paint）

在這個例子中我們將使用畫布（Canvas）創建一個簡單的繪制程序。



在我們場景的頂部我們使用行定位器排列四個方形的顏色塊。一個顏色塊是一個簡單的矩形，使用鼠標區域來檢測點擊。

```
    Row {
        id: colorTools
        anchors {
            horizontalCenter: parent.horizontalCenter
            top: parent.top
            topMargin: 8
        }
        property variant activeSquare: red
        property color paintColor: "#33B5E5"
        spacing: 4
        Repeater {
            model: ["#33B5E5", "#99CC00", "#FFBB33", "#FF4444"]
            ColorSquare {
                id: red
                color: modelData
                active: parent.paintColor == color
                onClicked: {
                    parent.paintColor = color
                }
            }
        }
    }
```

顏色存儲在一個數組中，作為繪制顏色使用。當用戶點擊一個矩形時，矩形內的顏色被設置為colorTools的paintColor屬性。

為了在畫布上跟蹤鼠標事件，我們使用鼠標區域（MouseArea）覆蓋畫布元素，並連接點擊和移動操作。

```
    Canvas {
```

```
        id: canvas
        anchors {
            left: parent.left
            right: parent.right
            top: colorTools.bottom
            bottom: parent.bottom
            margins: 8
        }
        property real lastX
        property real lastY
        property color color: colorTools.paintColor

        onPaint: {
            var ctx = getContext('2d')
            ctx.lineWidth = 1.5
            ctx.strokeStyle = canvas.color
            ctx.beginPath()
            ctx.moveTo(lastX, lastY)
            lastX = area.mouseX
            lastY = area.mouseY
            ctx.lineTo(lastX, lastY)
            ctx.stroke()
        }
        MouseArea {
            id: area
            anchors.fill: parent
            onPressed: {
                canvas.lastX = mouseX
                canvas.lastY = mouseY
            }
            onPositionChanged: {
                canvas.requestPaint()
            }
        }
    }
```

鼠標點擊存儲在laxstX與lastY屬性中。每次鼠標位置的改變會觸發畫布的重繪，這將會調用onPaint操作。

最後繪制用戶的筆劃，在onPaint操作中，我們繪制從最近改變的點上開始繪制一條新的路徑，然後我們從鼠標區域採集新的點，使用選擇的顏色繪制線段到新的點上。鼠標位置被存儲為新改變的位置。

HTML5畫布移植（Porting from HTML5 Canvas）

- https://developer.mozilla.org/en/Canvas_tutorial/Transformations
- http://en.wikipedia.org/wiki/Spirograph

移植一個HTML5畫布圖像到QML畫布非常簡單。在成百上千的例子中，我們選擇了一個來移植。

螺旋圖形（**Spiro Graph**）

我們使用一個來自Mozila項目的螺旋圖形例子來作為我們的基礎示例。原始的HTML5代碼被作為畫布教程發布。

下面是我們需要修改的代碼：

- Qt Quick要求定義變量使用，所以我們需要添加var的定義：

```
for (var i=0;i<3;i++) {
  ...
}
```

- 修改繪制方法接收Context2D對象：

```
function draw(ctx) {
  ...
}
```

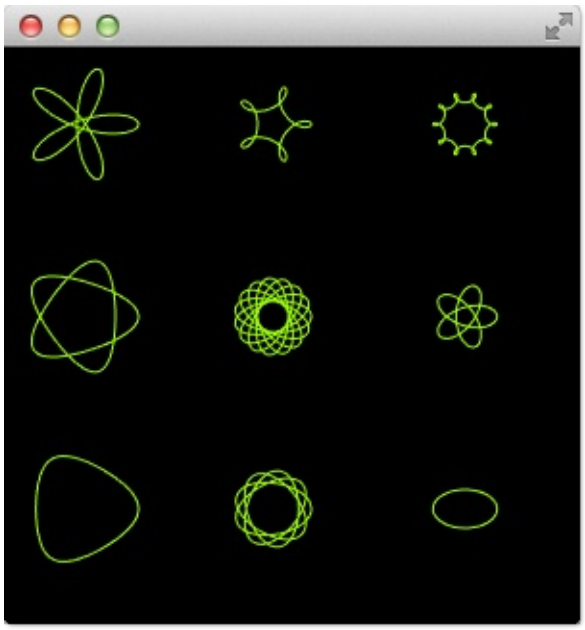
- 由于不同的大小，我們需要對每個螺旋適配轉換：

```
ctx.translate(20+j*50,20+i*50);
```

最後我們實現onPaint操作。在onPaint中我們請求一個context，並且調用我們的繪制方法。

```
    onPaint: {
        var ctx = getContext("2d");
        draw(ctx);
    }
```

下面這個結果就是我們使用QML畫布移植的螺旋圖形。



發光線（**Glowing Lines**）

下面有一個更加復雜的移植來自W3C組織。原始的發光線有些很不錯的地方，這使得移植更加具有挑戰性。


```
<!DOCTYPE HTML>
<html lang="en">
<head>
    <title>Pretty Glowing Lines</title>
</head>
<body>

<canvas width="800" height="450"></canvas>
<script>
var context = document.getElementsByTagName('canvas')[0].getContext('2d');

// initial start position
var lastX = context.canvas.width * Math.random();
var lastY = context.canvas.height * Math.random();
var hue = 0;

// closure function to draw
// a random bezier curve with random color with a glow effect
```

```
function line() {

    context.save();

    // scale with factor 0.9 around the center of canvas
    context.translate(context.canvas.width/2, context.canvas.height/2);
    context.scale(0.9, 0.9);
    context.translate(-context.canvas.width/2, -context.canvas.height/2);

    context.beginPath();
    context.lineWidth = 5 + Math.random() * 10;

    // our start position
    context.moveTo(lastX, lastY);

    // our new end position
    lastX = context.canvas.width * Math.random();
    lastY = context.canvas.height * Math.random();

    // random bezier curve, which ends on lastX, lastY
    context.bezierCurveTo(context.canvas.width * Math.random(),
    context.canvas.height * Math.random(),
    context.canvas.width * Math.random(),
    context.canvas.height * Math.random(),
    lastX, lastY);

    // glow effect
    hue = hue + 10 * Math.random();
    context.strokeStyle = 'hsl(' + hue + ', 50%, 50%)';
    context.shadowColor = 'white';
    context.shadowBlur = 10;
    // stroke the curve
    context.stroke();
    context.restore();
}

// call line function every 50msecs
setInterval(line, 50);

function blank() {
    // makes the background 10% darker on each call
    context.fillStyle = 'rgba(0,0,0,0.1)';
    context.fillRect(0, 0, context.canvas.width, context.canvas.height);
}

// call blank function every 50msecs
setInterval(blank, 40);

</script>
</body>
</html>
```

在HTML5中，context2D對象可以隨意在畫布上繪制。在QML中，只能在onPaint操作中繪制。在HTML5中，通常調用setInterval使用計時器觸發線段的繪制或者清屏。由于QML中不同的操作方法，僅僅只是調用這些函數不能實現我們想要的結果，因為我們需要通過onPaint操作來實現。我們也需要修改顏色的格式。讓我們看看需要改變哪些東西。

修改從畫布元素開始。為了簡單，我們使用畫布元素（Canvas）作為我們QML文件的根元素。

```
import QtQuick 2.0

Canvas {
   id: canvas
   width: 800; height: 450

   ...
}
```

代替直接調用的setInterval函數，我們使用兩個計時器來請求重新繪制。一個計時器觸發間隔較短，允許我們可以執行一些代碼。我們無法告訴繪制函數哪個操作是我想觸發的，我們為每個操作定義一個布爾標識，當重新繪制請求時，我們請求一個操作並且觸發它。

下面是線段繪制的代碼，清屏操作類似。

```
...
property bool requestLine: false

Timer {
    id: lineTimer
    interval: 40
    repeat: true
    triggeredOnStart: true
    onTriggered: {
        canvas.requestLine = true
        canvas.requestPaint()
    }
}

Component.onCompleted: {
    lineTimer.start()
}
...
```

現在我們已經有了告訴onPaint操作中我們需要執行哪個操作的指示。當我們進入onPaint處理每個繪制請求時，我們需要提取畫布元素中的初始化變量。

```
Canvas {
    ...
    property real hue: 0
    property real lastX: width * Math.random();
    property real lastY: height * Math.random();
    ...
}
```

現在我們的繪制函數應該像這樣：

```
onPaint: {
    var context = getContext('2d')
    if(requestLine) {
        line(context)
        requestLine = false
    }
    if(requestBlank) {
```

```
        blank(context)
        requestBlank = false
    }
}
```

線段繪制函數提取畫布作為一個參數。

```
function line(context) {
    context.save();
    context.translate(canvas.width/2, canvas.height/2);
    context.scale(0.9, 0.9);
    context.translate(-canvas.width/2, -canvas.height/2);
    context.beginPath();
    context.lineWidth = 5 + Math.random() * 10;
    context.moveTo(lastX, lastY);
    lastX = canvas.width * Math.random();
    lastY = canvas.height * Math.random();
    context.bezierCurveTo(canvas.width * Math.random(),
        canvas.height * Math.random(),
        canvas.width * Math.random(),
        canvas.height * Math.random(),
        lastX, lastY);

    hue += Math.random()*0.1
    if(hue > 1.0) {
        hue -= 1
    }
    context.strokeStyle = Qt.hsla(hue, 0.5, 0.5, 1.0);
    // context.shadowColor = 'white';
    // context.shadowBlur = 10;
    context.stroke();
    context.restore();
}
```

最大的變化是使用QML的Qt.rgba()和Qt.hsla()。在QML中需要把變量值適配在0.0到1.0之間。

同樣應用在清屏函數中。

```
function blank(context) {
    context.fillStyle = Qt.rgba(0,0,0,0.1)
    context.fillRect(0, 0, canvas.width, canvas.height);
}
```

下面是最終結果（目前沒有陰影）類似下面這樣。

查看下面的鏈接獲得更多的信息：

- W3C HTML Canvas 2D Context Specification
- Mozilla Canvas Documentation
- HTML5 Canvas Tutorial

Particle Simulations

注意

最後一次構建：**2014**年**1**月**20**日下午**18:00**。

這章的源代碼能夠在**assetts folder**找到。

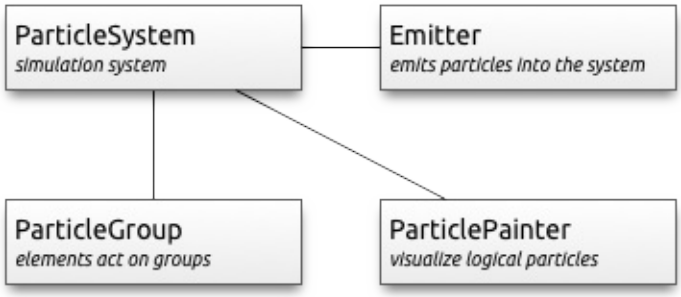
粒子模擬是計算機圖形技術的可視化圖形效果。典型的效果有：落葉，火燄，爆炸，流星，雲等等。

它不同于其它圖形渲染，粒子是基于模糊來渲染。它的結果在基于像素下是不可預測的。粒子系統的參數描述了隨機模擬的邊界。傳統的渲染技術實現粒子渲染效果很困難。有一個好消息是你可以使用QML元素與粒子系統交互。同時參數也可以看做是屬性，這些參數可以使用傳統的動畫技術來實現動態效果。

概念（**Concept**）

粒子模擬的核心是粒子系統（ParticleSystem），它控制了共享時間線。一個場景下可以有多個粒子系統，每個都有自己獨立的時間線。一個粒子使用發射器元素（Emitter）發射，使用粒子畫筆（ParticlePainter）實現可視化，它可以是一張圖片，一個QML項或者一個著色項（shader item）。一個發射器元素（Emitter）也提供向量來控制粒子方向。一個粒子被發送後就再也無法控制。粒子模型提供粒子控制器（Affector），它可以控制已發射粒子的參數。

在一個系統中，粒子可以使用粒子群元素（ParticleGroup）來共享移動時間。默認下，每個例子都屬于空（""）組。



- 粒子系統（ParticleSystem）- 管理發射器之間的共享時間線。
- 發射器（Emitter）- 向系統中發射邏輯粒子。
- 粒子畫筆（ParticlePainter）- 實現粒子可視化。
- 方向（Direction）- 已發射粒子的向量空間。
- 粒子組（ParticleGroup）- 每個粒子是一個粒子組的成員。
- 粒子控制器（Affector）- 控制已發射粒子。

簡單的模擬（Simple Simulation）

讓我們從一個簡單的模擬開始學習。Qt Quick使用簡單的粒子渲染非常簡單。下面是我們需要的：

- 綁定所有元素到一個模擬的粒子系統（ParticleSystem）。
- 一個向系統發射粒子的發射器（Emitter）。
- 一個ParticlePainter派生元素，用來實現粒子的可視化。

```
import QtQuick 2.0
import QtQuick.Particles 2.0

Rectangle {
    id: root
    width: 480; height: 160
    color: "#1f1f1f"

    ParticleSystem {
        id: particleSystem
    }

    Emitter {
        id: emitter
        anchors.centerIn: parent
        width: 160; height: 80
        system: particleSystem
        emitRate: 10
        lifeSpan: 1000
        lifeSpanVariation: 500
        size: 16
        endSize: 32
        Tracer { color: 'green' }
    }

    ImageParticle {
        source: "assets/particle.png"
        system: particleSystem
    }
}
```

例子的運行結果如下所示：

我們使用一個80x80的黑色矩形框作為我們的根元素和背景。然後我們定義一個粒子系統（ParticleSystem）。這通常是粒子系統綁定所有元素的第一步。下一個元素是發射器（Emitter），它定義了基于矩形框的發射區域和發射粒子的基礎屬性。發射器使用system屬性與粒子系統進行綁定。

在這個例子中，發射器每秒發射10個粒子（emitRate:10）到發射器的區域，每個粒子的生命週期是1000毫秒（lifeSpan:1000），一個已發射粒子的生命週期變化是500毫秒（lifeSpanVariation:500）。一個粒子開

始的大小是16個像素（size:16），生命週期結束時的大小是32個像素（endSize:32）。

綠色邊框的矩形是一個跟蹤元素，用來顯示發射器的幾何形狀。這個可視化展示了粒子在發射器矩形框內發射，但是渲染效果不被限制在發射器的矩形框內。渲染位置依賴于粒子的壽命和方向。這將幫助我們更加清楚的知道如何改變粒子的方向。

發射器發射邏輯粒子。一個邏輯粒子的可視化使用粒子畫筆（ParticlePainter）來實現，在這個例子中我們使用了圖像粒子（ImageParticle），使用一個圖片鏈接作為源屬性。圖像粒子也有其它的屬性用來控制粒子的外觀。

- 發射頻率（emitRate）- 每秒粒子發射數（默認為10個）。
- 生命週期（lifeSpan）- 粒子持續時間（單位毫秒，默認為1000毫秒）。
- 初始大小（size），結束大小（endSize）- 粒子在它的生命週期的開始和結束時的大小（默認為16像素）。

改變這些屬性將會徹底改變顯示結果：

```
    Emitter {
        id: emitter
        anchors.centerIn: parent
        width: 20; height: 20
        system: particleSystem
        emitRate: 40
        lifeSpan: 2000
        lifeSpanVariation: 500
        size: 64
        sizeVariation: 32
        Tracer { color: 'green' }
    }
```

增加發射頻率為40，生命週期增加到2秒，開始大小為64像素，結束大小減少到32像素。

增加結束大小（endSize）可能會導致白色的背景出現。請注意粒子只有發射被限制在發射器定義的區域內，而粒子渲染是不會考慮這個參數的。

粒子參數（Particle Parameters）

我們已經知道通過改變發射器的行為就可以改變我們的粒子模擬。粒子畫筆被用來繪制每一個粒子。 回到我們之前的粒子中，我們更新一下我們的圖片粒子畫筆（ImageParticle）。首先我們改變粒子圖片為一個小的星形圖片：

```
ImageParticle {
    ...
    source: 'assets/star.png'
}
```

粒子使用金色來進行初始化，不同的粒子顏色變化範圍為+/- 20%。

```
color: '#FFD700'
colorVariation: 0.2
```

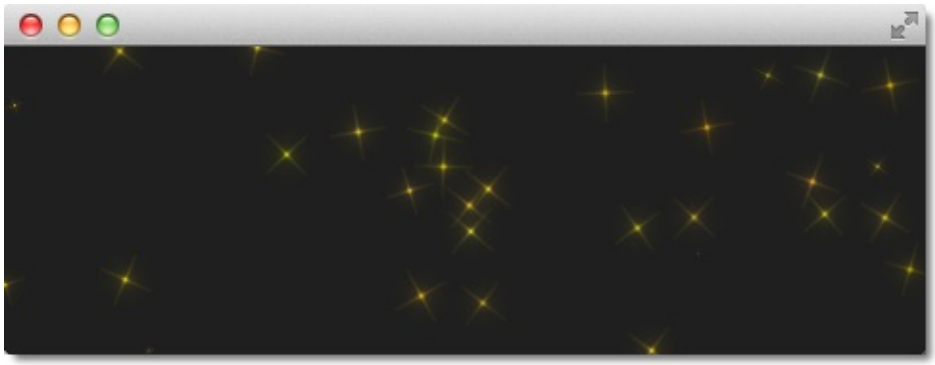
為了讓場景更加生動，我們需要旋轉粒子。每個粒子首先按順時針旋轉15度，不同的粒子在+/-5度之間變化。每個例子會不斷的以每秒45度旋轉。每個粒子的旋轉速度在+/-15度之間變化：

```
rotation: 15
rotationVariation: 5
rotationVelocity: 45
rotationVelocityVariation: 15
```

最後，我們改變粒子的入場效果。 這個效果是粒子產生時的效果，在這個例子中，我們希望使用一個縮放效果：

```
entryEffect: ImageParticle.Scale
```

現在我們可以看到旋轉的星星出現在我們的屏幕上。



下面是我們如何改變圖片粒子畫筆的代碼段。

```
    ImageParticle {
        source: "assets/star.png"
        system: particleSystem
        color: '#FFD700'
```

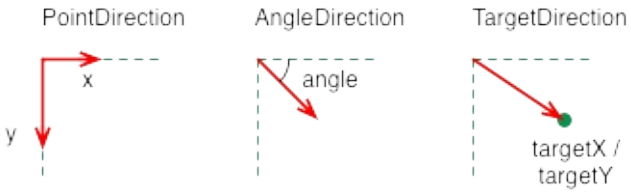
```
        colorVariation: 0.2
        rotation: 0
        rotationVariation: 45
        rotationVelocity: 15
        rotationVelocityVariation: 15
        entryEffect: ImageParticle.Scale
    }
```

粒子方向（**Directed Particle**）

我們已經看到了粒子的旋轉，但是我們的粒子需要一個軌蹟。軌蹟由速度或者粒子隨機方向的加速度指定，也可以叫做矢量空間。

有多種可用矢量空間用來定義粒子的速度或加速度：

- 角度方向（AngleDirection）- 使用角度的方向變化。
- 點方向（PointDirection）- 使用x,y組件組成的方向變化。
- 目標方向（TargetDirection）- 朝著目標點的方向變化。



讓我們在場景下試著用速度方向將粒子從左邊移動到右邊。

首先使用角度方向（AngleDirection）。我們使用AngleDirection元素作為我們的發射器（Emitter）的速度屬性：

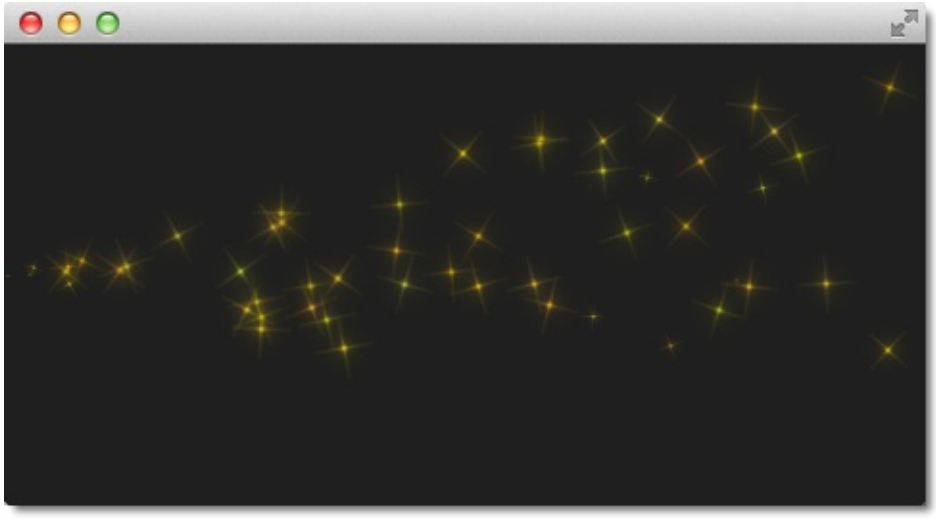
```
velocity: AngleDirection { }
```

粒子的發射將會使用指定的角度屬性。角度值在0到360度之間，0度代表指向右邊。在我們的例子中，例子將會移動到右邊，所以0度已經指向右邊方向。粒子的角度變化在+/-15度之間：

```
velocity: AngleDirection {
    angle: 0
    angleVariation: 15
}
```

現在我們已經設置了方向，下面是指定粒子的速度。它由一個梯度值定義，這個梯度值定義了每秒像素的變化。正如我們設置大約640像素，梯度值為100，看起來是一個不錯的值。這意味著平均一個6.4秒生命週期的粒子可以穿越我們看到的區域。為了讓粒子的穿越看起來更加有趣，我們使用magnitudeVariation來設置梯度值的變化，這個值是我們的梯度值的一半：

```
velocity: AngleDirection {
    ...
    magnitude: 100
    magnitudeVariation: 50
}
```



下面是完整的源碼，平均的生命週期被設置為6..4秒。我們設置發射器的寬度和高度為1個像素，這意味著所有的粒子都從相同的位置發射出去，然後基于我們給定的軌蹟運動。

```
    Emitter {
        id: emitter
        anchors.left: parent.left
        anchors.verticalCenter: parent.verticalCenter
        width: 1; height: 1
        system: particleSystem
        lifeSpan: 6400
        lifeSpanVariation: 400
        size: 32
        velocity: AngleDirection {
            angle: 0
            angleVariation: 15
            magnitude: 100
            magnitudeVariation: 50
        }
    }
```

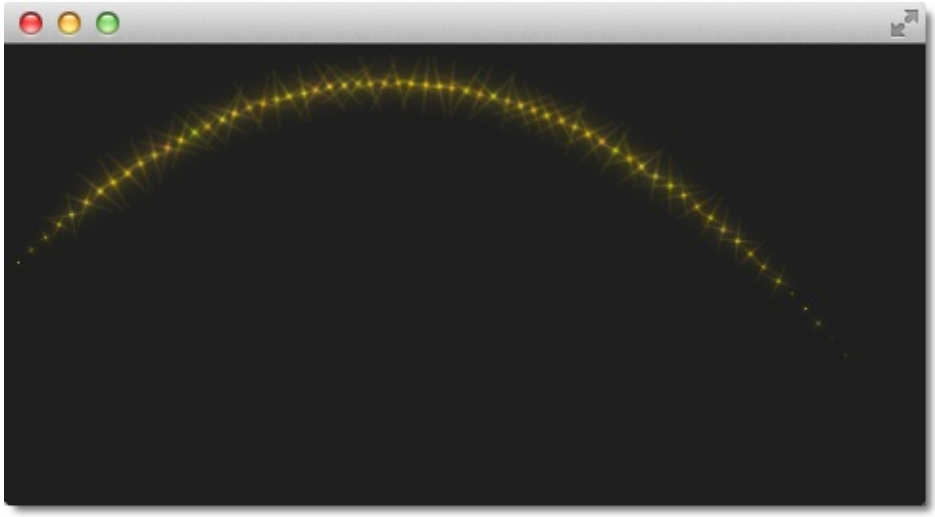
那麼加速度做些什麼？加速度是每個粒子加速度矢量，它會在運動的時間中改變速度矢量。例如我們做一個星星按照弧形運動的軌蹟。我們將會改變我們的速度方向為-45度，然後移除變量，可以得到一個更連貫的弧形軌蹟：

```
velocity: AngleDirection {
    angle: -45
    magnitude: 100
}
```

加速度的方向為90度（向下），加速度為速度的四分之一：

```
acceleration: AngleDirection {
    angle: 90
    magnitude: 25
}
```

結果是中間左方到右下的一個弧。



這個值是在不斷的嘗試與錯誤中發現的。

下面是發射器完整的代碼。

```
    Emitter {
        id: emitter
        anchors.left: parent.left
        anchors.verticalCenter: parent.verticalCenter
        width: 1; height: 1
        system: particleSystem
        emitRate: 10
        lifeSpan: 6400
        lifeSpanVariation: 400
        size: 32
        velocity: AngleDirection {
            angle: -45
            angleVariation: 0
            magnitude: 100
        }
        acceleration: AngleDirection {
            angle: 90
            magnitude: 25
        }
    }
```

在下一個例子中，我們將使用點方向（PointDirection）矢量空間來再一次演示粒子從左到右的運動。

一個點方向（PointDirection）是由x和y組件組成的矢量空間。例如，如果你想粒子以45度的矢量運動，你需要為x，y指定相同的值。

在我們的例子中，我們希望粒子在從左到右的例子中建立一個15度的圓錐。我們指定一個坐標方向（PointDirection）作為我們速度矢量空間：

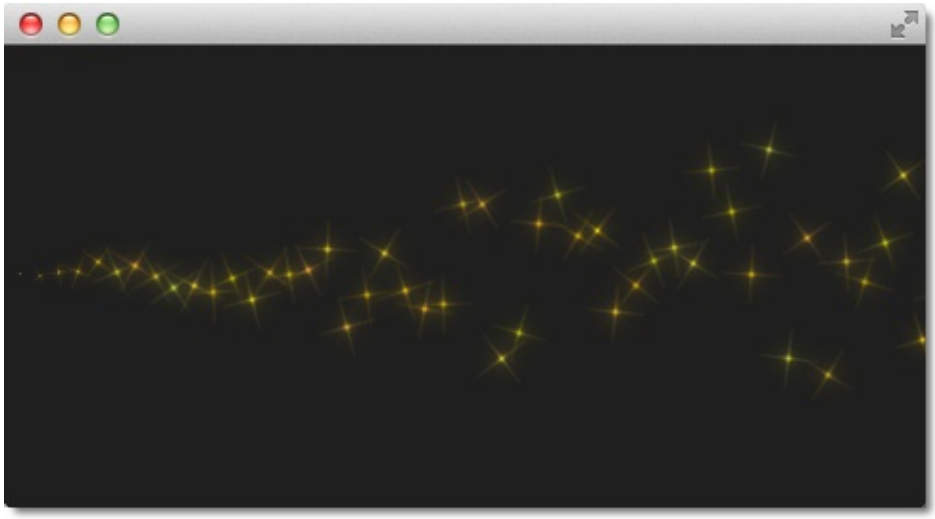
```
velocity: PointDirection { }
```

為了達到運動速度每秒100個像素，我們設置x為100,。15度角（90度的1/6）,我們指定y變量為100/6：

```
velocity: PointDirection {
    x: 100
    y: 0
```

```
    xVariation: 0
    yVariation: 100/6
}
```

結果是粒子的運動從左到右構成了一個15度的圓錐。



現在是最後一個方案，目標方向（TargetDirection）。目標方向允許我們指定發射器或者一個QML項的x,y坐標值。當一個QML項的中心點成為一個目標點時，你可以指定目標變化值是x目標值的1/6來完成一個15度的圓錐：

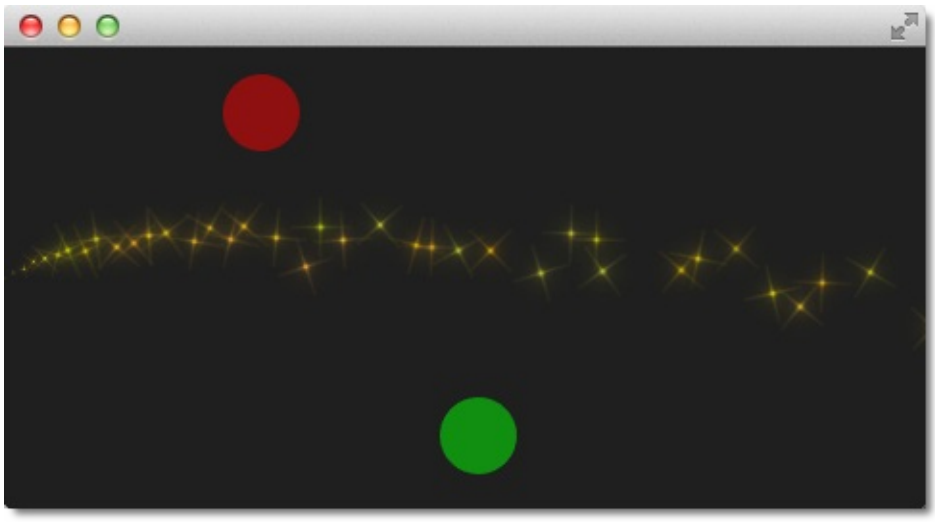
```
velocity: TargetDirection {
    targetX: 100
    targetY: 0
    targetVariation: 100/6
    magnitude: 100
}
```

注意

當你期望發射粒子朝著指定的**x,y**坐標值流動時，目標方向是非常好的方案。

我沒有再貼出結果圖，因為它與前一個結果相同，取而代之的有一個問題留給你。

在下圖的紅色和綠色圓指定每個目標項的目標方向速度的加速屬性。每個目標方向有相同的參數。那麼哪一個負責速度，哪一個負責加速度？



粒子畫筆（Particle Painter）

到目前為止我們只使用了基于粒子畫筆的圖像來實現粒子可視化。Qt也提供了一些其它的粒子畫筆：

- 粒子項（ItemParticle）：基于粒子畫筆的代理
- 自定義粒子（CustomParticle）：基于粒子畫筆的著色器

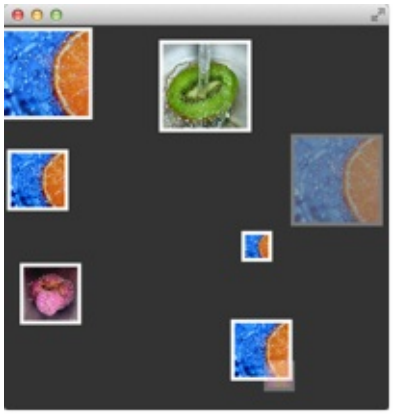
粒子項可以將QML元素項作為粒子發射。你需要制定自己的粒子代理。

```
    ItemParticle {
        id: particle
        system: particleSystem
        delegate: itemDelegate
    }
```

在這個例子中，我們的代理是一個隨機圖片（使用Math.random()完成），有著白色邊框和隨機大小。

```
    Component {
        id: itemDelegate
        Rectangle {
            id: container
            width: 32*Math.ceil(Math.random()*3); height: width
            color: 'white'
            Image {
                anchors.fill: parent
                anchors.margins: 4
                source: 'assets/fruits'+Math.ceil(Math.random()*10)+'.jpg'
            }
        }
    }
```

每秒發出四個粒子，每個粒子擁有4秒的生命週期。粒子自動淡入淡出。



對于更多的動態情況，也可以由你自己創建一個子項，讓粒子系統來控制它，使用take(item, priority)來完成。粒子系統控制你的粒子就像控制普通的粒子一樣。你可以使用give(item)來拿回子項的控制權。你也可以操作子項粒子，甚至可以使用freeze(item)來停止它，使用unfreeze(item)來恢復它。

粒子控制（**Affecting Particles**）

粒子由粒子發射器發出。在粒子發射出後，發射器無法再改變粒子。粒子控制器允許你控制發射後的粒子參數。

控制器的每個類型使用不同的方法來影響粒子：

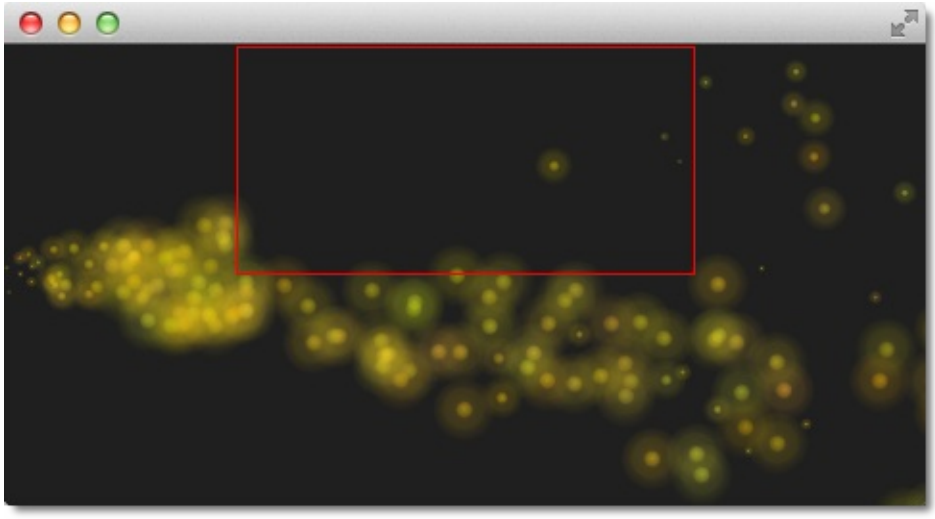
- 生命週期（Age）- 修改粒子的生命週期
- 吸引（Attractor）- 吸引粒子朝向指定點
- 摩擦（Friction）- 按當前粒子速度成正比減慢運動
- 重力（Gravity）- 設置一個角度的加速度
- 紊流（Turbulence）- 強制基于噪聲圖像方式的流動
- 漂移（Wander）- 隨機變化的軌蹟
- 組目標（GroupGoal）- 改變一組粒子群的狀態
- 子粒子（SpriteGoal）- 改變一個子粒子的狀態

生命週期（**Age**）

允許粒子老得更快，lifeLeft屬性指定了粒子的有多少的生命週期。

```
    Age {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 240; height: 120
        system: particleSystem
        advancePosition: true
        lifeLeft: 1200
        once: true
        Tracer {}
    }
```

在這個例子中，當粒子的生命週期達到1200毫秒後，我們將會縮短上方的粒子的生命週期一次。由于我們設置了advancePosition為true，當粒子的生命週期到達1200毫秒後，我們將會再一次在這個位置看到粒子出現。

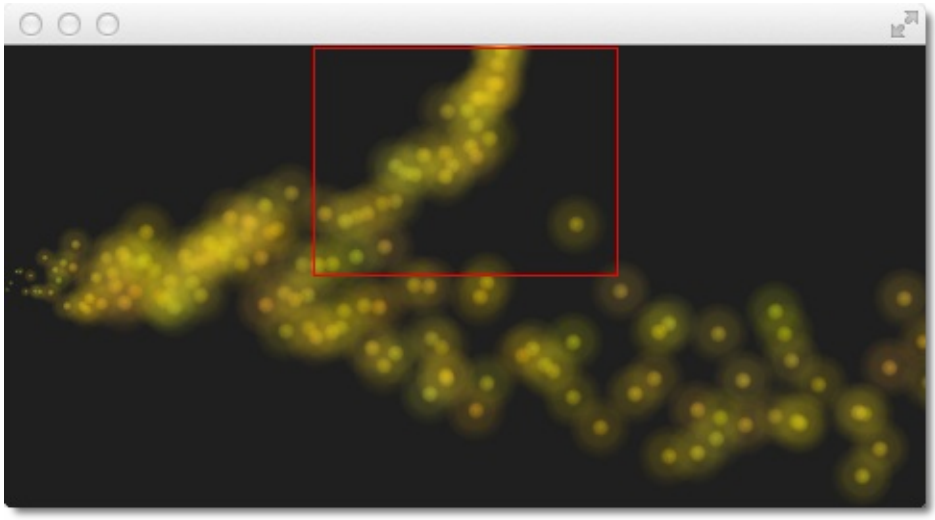


吸引（**Attractor**）

吸引會將粒子朝指定的點上吸引。這個點使用pointX與pointY來指定，它是與吸引區域的幾何形狀相對的。strength指定了吸引的力度。在我們的例子中，我們讓粒子從左向右運動，吸引放在頂部，有一半運動的粒子會穿過吸引區域。控制器只會影響在它們幾何形狀內的粒子。這種分離讓我們可以同步看到正常的流動與受影響的流動。

```
    Attractor {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 160; height: 120
        system: particleSystem
        pointX: 0
        pointY: 0
        strength: 1.0
        Tracer {}
    }
```

很容易看出上半部分粒子受到吸引。吸引點被設置為吸引區域的左上角（0/0點），吸引力為1.0。



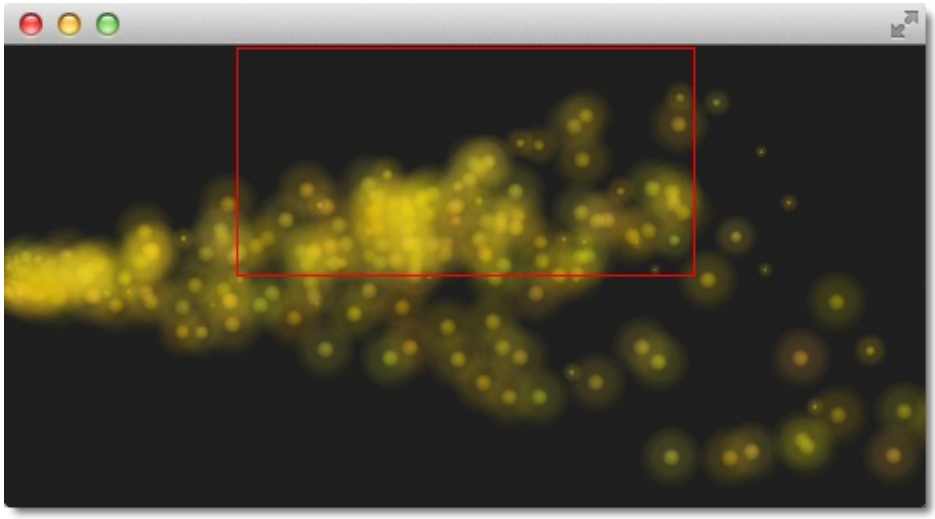
摩擦（**Friction**）

摩擦控制器使用一個參數（factor）減慢粒子運動，直到達到一個閾值。

```
    Friction {
```

```
        anchors.horizontalCenter: parent.horizontalCenter
        width: 240; height: 120
        system: particleSystem
        factor : 0.8
        threshold: 25
        Tracer {}
    }
```

在上部的摩擦區域，粒子被按照0.8的參數（factor）減慢，直到粒子的速度達到25像素每秒。這個閾值像一個過濾器。粒子運動速度高于閾值將會按照給定的參數來減慢它。

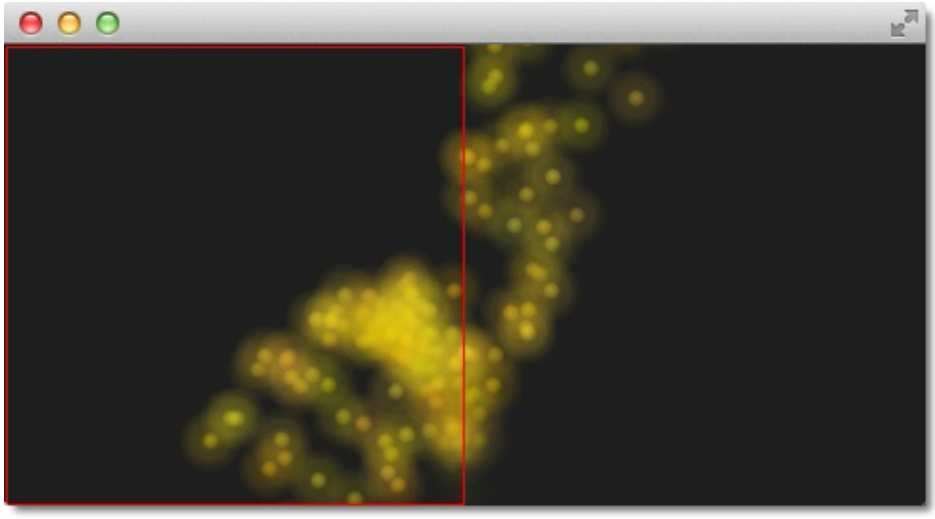


重力（**Gravity**）

重力控制器應用在加速度上，在我們的例子中，我們使用一個角度方向將粒子從底部發射到頂部。右邊是為控制區域，左邊使用重力控制器控制，重力方向為90度方向（垂直向下），梯度值為50。

```
    Gravity {
        width: 240; height: 240
        system: particleSystem
        magnitude: 50
        angle: 90
        Tracer {}
    }
```

左邊的粒子試圖爬上去，但是穩定向下的加速度將它們按照重力的方向拖拽下來。

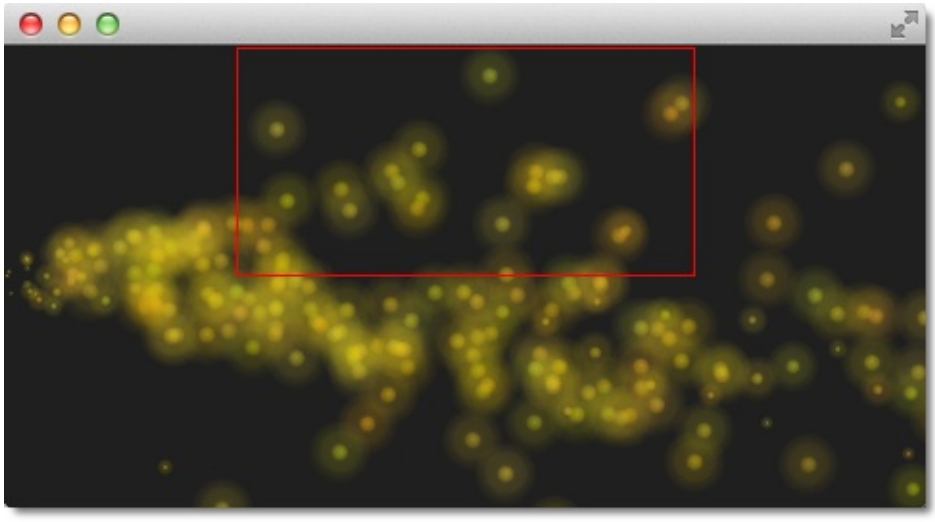


紊流（**Turbulence**）

紊流控制器，對粒子應用了一個混亂映射方向力的矢量。這個混亂映射是由一個噪聲圖像定義的。可以使用noiseSource屬性來定義噪聲圖像。strength定義了矢量對于粒子運動的影響有多大。

```
    Turbulence {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 240; height: 120
        system: particleSystem
        strength: 100
        Tracer {}
    }
```

在這個例子中，上部區域被紊流影響。它們的運動看起來是不穩定的。不穩定的粒子偏差值來自原路徑定義的strength。



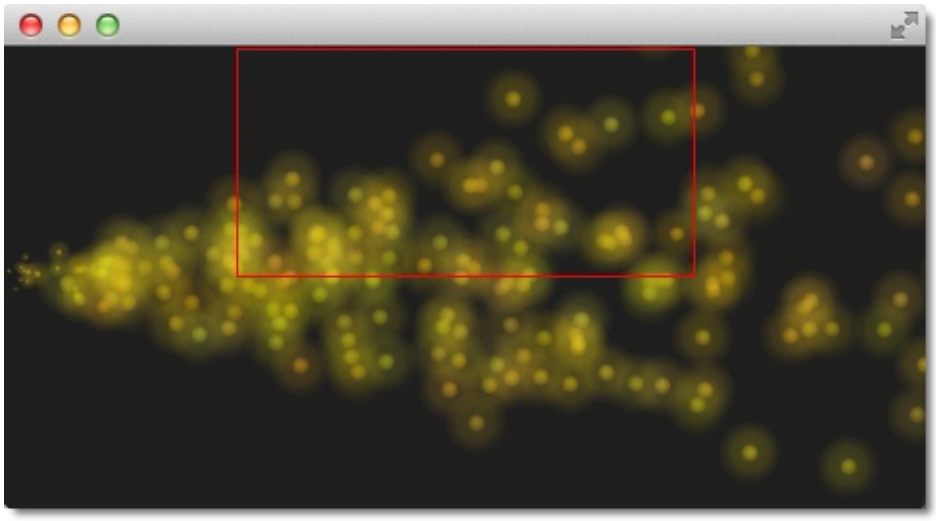
漂移（**Wander**）

漂移控制器控制了軌蹟。affectedParameter屬性可以指定哪個參數控制了漂移（速度，位置或者加速度）。pace屬性制定了每秒最多改變的屬性。yVariance指定了y組件對粒子軌蹟的影響。

```
    Wander {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 240; height: 120
```

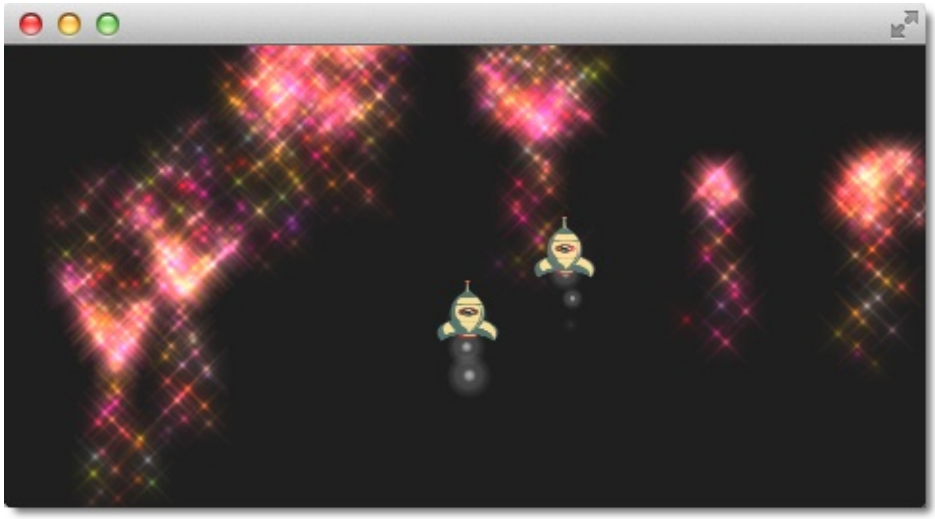
```
        system: particleSystem
        affectedParameter: Wander.Position
        pace: 200
        yVariance: 240
        Tracer {}
    }
```

在頂部漂移控制器的粒子被隨機的軌蹟改變。在這種情境下，每秒改變粒子y方向的位置200次。



粒子組（**Particle Group**）

在本章開始時，我們已經介紹過粒子組了，默認下，粒子都屬于空組（""）。使用GroupGoal控制器可以改變粒子組。為了實現可視化，我們創建了一個煙花示例，火箭進入，在空中爆炸形成壯觀的煙火。



這個例子分為兩部分。第一部分叫做“發射時間（Launch Time）”連接場景，加入粒子組，第二部分叫做“爆炸煙花（Let there be firework）”，專注于粒子組的變化。

讓我們看看這兩部分。

發射時間（**Launch Time**）

首先我們創建一個典型的黑色場景：

```
import QtQuick 2.0
import QtQuick.Particles 2.0

Rectangle {
    id: root
    width: 480; height: 240
    color: "#1F1F1F"
    property bool tracer: false
}
```

tracer使用被用作場景追蹤的開關，然後定義我們的粒子系統：

```
ParticleSystem {
    id: particleSystem
}
```

我們添加兩種粒子圖片畫筆（一個用于火箭，一個用于火箭噴射煙霧）：

```
ImageParticle {
    id: smokePainter
    system: particleSystem
    groups: ['smoke']
```

```
    source: "assets/particle.png"
    alpha: 0.3
    entryEffect: ImageParticle.None
}

ImageParticle {
    id: rocketPainter
    system: particleSystem
    groups: ['rocket']
    source: "assets/rocket.png"
    entryEffect: ImageParticle.None
}
```

你可以看到在這些畫筆定義中，它們使用groups屬性來定義粒子的歸屬。只需要定義一個名字，Qt Quick將會隱式的創建這個分組。

現在我們需要將這些火箭發射到空中。我們在場景底部創建一個粒子發射器，將速度設置為朝上的方向。為了模擬重力，我們設置一個向下的加速度：

```
Emitter {
    id: rocketEmitter
    anchors.bottom: parent.bottom
    width: parent.width; height: 40
    system: particleSystem
    group: 'rocket'
    emitRate: 2
    maximumEmitted: 4
    lifeSpan: 4800
    lifeSpanVariation: 400
    size: 32
    velocity: AngleDirection { angle: 270; magnitude: 150; magnitudeVariation: 10 }
    acceleration: AngleDirection { angle: 90; magnitude: 50 }
    Tracer { color: 'red'; visible: root.tracer }
}
```

發射器屬于'rocket'粒子組，與我們的火箭粒子畫筆相同。通過粒子組將它們聯系在一起。發射器將粒子發射到'rocket'粒子組中，火箭畫筆將會繪制它們。

對于煙霧，我們使用一個追蹤發射器，它將會跟在火箭的後面。我們定義'smoke'組，並且它會跟在'rocket'粒子組後面：

```
TrailEmitter {
    id: smokeEmitter
    system: particleSystem
    emitHeight: 1
    emitWidth: 4
    group: 'smoke'
    follow: 'rocket'
    emitRatePerParticle: 96
    velocity: AngleDirection { angle: 90; magnitude: 100; angleVariation: 5 }
    lifeSpan: 200
    size: 16
    sizeVariation: 4
    endSize: 0
}
```

向下模擬從火箭裡面噴射出的煙。emitHeight與emitWidth指定了圍繞跟隨在煙霧粒子發射後的粒子。如果不指定這個值，跟隨的粒子將會被拿掉，但是對于這個例子，我們想要提升顯示效果，粒子流從一個接近于火箭尾部的中間點發射出。

如果你運行這個例子，你會發現一些火箭正常飛起，一些火箭卻飛出場景。這不是我們想要的，我們需要在它們離開場景前讓他們慢下來，這裡可以使用摩擦控制器來設置一個最小閾值：

```
Friction {
    groups: ['rocket']
    anchors.top: parent.top
    width: parent.width; height: 80
    system: particleSystem
    threshold: 5
    factor: 0.9
}
```

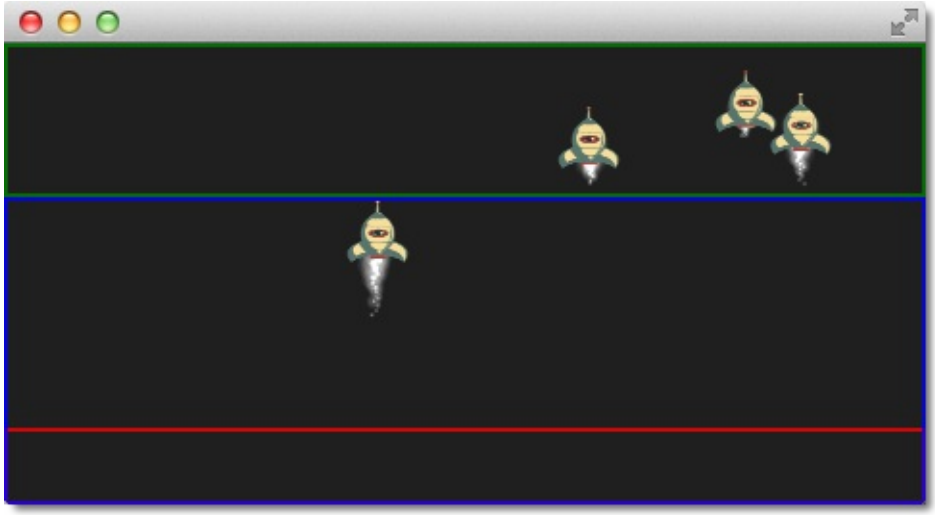
在摩擦控制器中，你也需要定義哪個粒子組受控制器影響。當火箭經過從頂部向下80像素的區域時，所有的火箭將會以0.9的factor減慢（你可以試試100，你會發現它們立即停止了），直到它們的速度達到每秒5個像素。隨著火箭粒子向下的加速度繼續生效，火箭開始向地面下沉，直到它們的生命週期結束。

由于在空氣中向上運動是非常困難的，並且非常不穩定，我們在火箭上升時模擬一些紊流：

```
Turbulence {
    groups: ['rocket']
    anchors.bottom: parent.bottom
    width: parent.width; height: 160
    system: particleSystem
    strength: 25
    Tracer { color: 'green'; visible: root.tracer }
}
```

當然，紊流控制器也需要定義它會影響哪些粒子組。紊流控制器的區域從底部向上160像素（直到摩擦控制器邊界上），它們也可以相互覆蓋。

當你運行程序時，你可以看到火箭開始上升，然後在摩擦控制器區域開始減速，向下的加速度仍然生效，火箭開始後退。下一步我們開始制作爆炸煙花。



注意

使用**tracers**跟蹤區域可以顯示場景中的不同區域。火箭粒子發射的紅色區域，藍色區域是紊流控制器區域，最後在綠色的摩擦控制器區域減速，並且再次下降是由于向下的加速度仍然生效。

爆炸煙花（**Let there be fireworks**）

讓火箭變成美麗的煙花，我們需要添加一個粒子組來封裝這個變化：

```
ParticleGroup {
    name: 'explosion'
    system: particleSystem
}
```

我們使用GroupGoal控制器來改變粒子組。這個組控制器被放置在屏幕中間垂直線附近，它將會影響'rocket'粒子組。使用groupGoal屬性，我們設置目標組改變為我們之前定義的'explosion'組：

```
GroupGoal {
    id: rocketChanger
    anchors.top: parent.top
    width: parent.width; height: 80
    system: particleSystem
    groups: ['rocket']
    goalState: 'explosion'
    jump: true
    Tracer { color: 'blue'; visible: root.tracer }
}
```

jump屬性定義了粒子組的變化是立即變化而不是在某個時間段後變化。

注意

在**Qt5**的**alpha**發布版中，粒子組的持續改變無法工作，有好的建議嗎？

由于火箭粒子變為我們的爆炸粒子，當火箭粒子進入GroupGoal控制器區域時，我們需要在粒子組中添加一個煙花：

```
// inside particle group
TrailEmitter {
    id: explosionEmitter
    anchors.fill: parent
    group: 'sparkle'
    follow: 'rocket'
    lifeSpan: 750
    emitRatePerParticle: 200
    size: 32
    velocity: AngleDirection { angle: -90; angleVariation: 180; magnitude: 50 }
}
```

爆炸釋放粒子到'sparkle'粒子組。我們稍後會定義這個組的粒子畫筆。軌蹟發射器跟隨火箭粒子每秒發射200個火箭爆炸粒子。粒子的方向向上，並改變180度。

由于向'sparkle'粒子組發射粒子，我們需要定義一個粒子畫筆用于繪制這個組的粒子：

```
ImageParticle {
```

```
    id: sparklePainter
    system: particleSystem
    groups: ['sparkle']
    color: 'red'
    colorVariation: 0.6
    source: "assets/star.png"
    alpha: 0.3
}
```

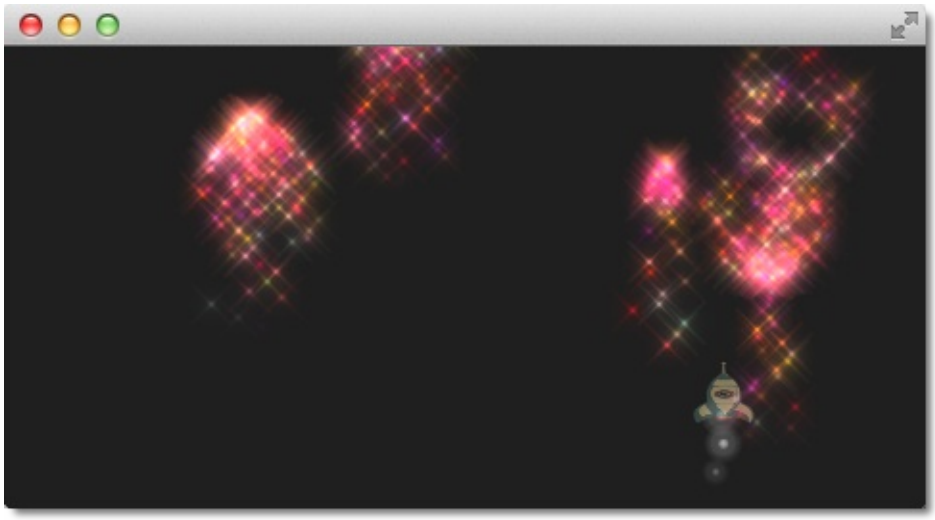
閃爍的煙花是紅色的星星，使用接近透明的顏色來渲染出發光的效果。

為了使煙花更加壯觀，我們也需要添加給我們的粒子組添加第二個軌蹟發射器，它向下發射錐形粒子：

```
// inside particle group
TrailEmitter {
    id: explosion2Emitter
    anchors.fill: parent
    group: 'sparkle'
    follow: 'rocket'
    lifeSpan: 250
    emitRatePerParticle: 100
    size: 32
    velocity: AngleDirection { angle: 90; angleVariation: 15; magnitude: 400 }
}
```

其它的爆炸軌蹟發射器與這個設置類似，就這樣。

下面是最終結果。



下面是火箭煙花的所有代碼。

```
import QtQuick 2.0
import QtQuick.Particles 2.0

Rectangle {
    id: root
    width: 480; height: 240
    color: "#1F1F1F"
    property bool tracer: false

    ParticleSystem {
```

```
        id: particleSystem
    }

    ImageParticle {
        id: smokePainter
        system: particleSystem
        groups: ['smoke']
        source: "assets/particle.png"
        alpha: 0.3
    }

    ImageParticle {
        id: rocketPainter
        system: particleSystem
        groups: ['rocket']
        source: "assets/rocket.png"
        entryEffect: ImageParticle.Fade
    }

    Emitter {
        id: rocketEmitter
        anchors.bottom: parent.bottom
        width: parent.width; height: 40
        system: particleSystem
        group: 'rocket'
        emitRate: 2
        maximumEmitted: 8
        lifeSpan: 4800
        lifeSpanVariation: 400
        size: 32
        velocity: AngleDirection { angle: 270; magnitude: 150; magnitudeVariation: 10 }
        acceleration: AngleDirection { angle: 90; magnitude: 50 }
        Tracer { color: 'red'; visible: root.tracer }
    }

    TrailEmitter {
        id: smokeEmitter
        system: particleSystem
        group: 'smoke'
        follow: 'rocket'
        size: 16
        sizeVariation: 8
        emitRatePerParticle: 16
        velocity: AngleDirection { angle: 90; magnitude: 100; angleVariation: 15 }
        lifeSpan: 200
        Tracer { color: 'blue'; visible: root.tracer }
    }

    Friction {
        groups: ['rocket']
        anchors.top: parent.top
        width: parent.width; height: 80
        system: particleSystem
        threshold: 5
        factor: 0.9

    }

    Turbulence {
        groups: ['rocket']
        anchors.bottom: parent.bottom
        width: parent.width; height: 160
        system: particleSystem
```

```
        strength:25
        Tracer { color: 'green'; visible: root.tracer }
    }


    ImageParticle {
        id: sparklePainter
        system: particleSystem
        groups: ['sparkle']
        color: 'red'
        colorVariation: 0.6
        source: "assets/star.png"
        alpha: 0.3
    }

    GroupGoal {
        id: rocketChanger
        anchors.top: parent.top
        width: parent.width; height: 80
        system: particleSystem
        groups: ['rocket']
        goalState: 'explosion'
        jump: true
        Tracer { color: 'blue'; visible: root.tracer }
    }

    ParticleGroup {
        name: 'explosion'
        system: particleSystem

        TrailEmitter {
            id: explosionEmitter
            anchors.fill: parent
            group: 'sparkle'
            follow: 'rocket'
            lifeSpan: 750
            emitRatePerParticle: 200
            size: 32
            velocity: AngleDirection { angle: -90; angleVariation: 180; magnitude: 50 }
        }

        TrailEmitter {
            id: explosion2Emitter
            anchors.fill: parent
            group: 'sparkle'
            follow: 'rocket'
            lifeSpan: 250
            emitRatePerParticle: 100
            size: 32
            velocity: AngleDirection { angle: 90; angleVariation: 15; magnitude: 400 }
        }
    }
}
```

總結（**Summary**）

粒子是一個非常強大且有趣的方法，用來表達圖像現象的一種方式，比如煙， 火花，隨機可視元素。Qt5的擴展API非常強大，我們僅僅只使用了一些淺顯的。有一些元素我們還沒有使用過，比如精靈（spirites），尺寸表（size tables），顏色表（color tables）。粒子看起來非常有趣，它在界面上創建引人注目的東西是非常有潛力的。在一個用戶界面中使用非常多的粒子效果將會導致用戶對它產生這是一個遊戲的印象。粒子的真正力量也是用來創建遊戲。

Shader Effect

注意

最後一次構建：**2014**年**1**月**20**日下午**18:00**。

這章的源代碼能夠在**assetts folder**找到。

- http://labs.qt.nokia.com/2012/02/02/qt-graphical-effects-in-qt-labs/
- http://labs.qt.nokia.com/2011/05/03/qml-shadereffectitem-on-qgraphicsview/
- http://qt-project.org/doc/qt-4.8/declarative-shadereffects.html
- http://www.opengl.org/registry/doc/GLSLangSpec.4.20.6.clean.pdf
- http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf
- http://www.lighthouse3d.com/opengl/glsl/
- http://wiki.delphigl.com/index.php/Tutorial_glsl
- Qt5Doc qtquick-shaders

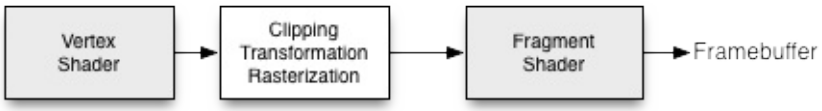
著色器允許我們利用SceneGraph的接口直接調用在強大的GPU上運行的OpenGL來創建渲染效果。著色器使用ShaderEffect與ShaderEffectSource元素來實現。著色器本身的算法使用OpenGL Shading Language（OpenGL著色語言）來實現。

實際上這意味著你需要混合使用QML代碼與著色器代碼。執行時，會將著色器代碼發送到GPU，並在GPU上編譯執行。QML著色器元素（Shader QML Elements）允許你與OpenGL著色器程序的屬性交互。

讓我們首先來看看OpenGL著色器。

OpenGL著色器（OpenGL Shader）

OpenGL的渲染管線分為幾個步驟。一個簡單的OpenGL渲染管線將包含一個頂點著色器和一個片段著色器。



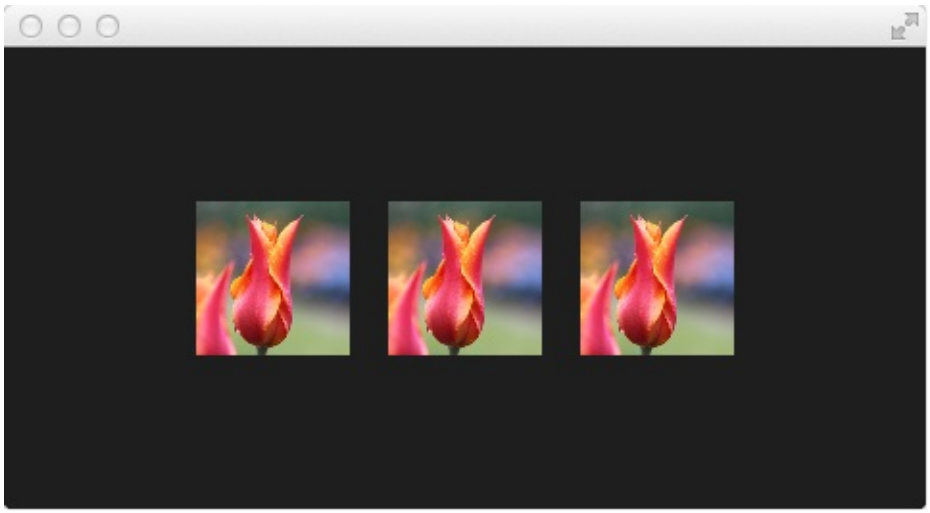
頂點著色器接收頂點數據，並且在程序最後賦值給gl_Position。然後，頂點將會被裁剪，轉換和柵格化後作為像素輸出。 片段（像素）進入片段著色器，進一步對片段操作並將結果的顏色賦值給gl_FragColor。頂點著色器調用多邊形每個角的點（頂點=3D中的點），負責這些點的3D處理。片段（片度=像素）著色器調用每個像素並決定這個像素的顏色。

著色器元素（**Shader Elements**）

為了對著色器編程，Qt Quick提供了兩個元素。ShaderEffectSource與ShaderEffect。ShaderEffect將會使用自定義的著色器，ShaderEffectSource可以將一個QML元素渲染為一個紋理然後再渲染這個紋理。由于ShaderEffect能夠應用自定義的著色器到它的矩形幾何形狀，並且能夠使用在著色器中操作資源。一個資源可以是一個圖片，它被作為一個紋理或者著色器資源。

默認下著色器使用這個資源並且不作任何改變進行渲染。

```
import QtQuick 2.0

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    Row {
        anchors.centerIn: parent
        spacing: 20
        Image {
            id: sourceImage
            width: 80; height: width
            source: 'assets/tulips.jpg'
        }
        ShaderEffect {
            id: effect
            width: 80; height: width
            property variant source: sourceImage
        }
        ShaderEffect {
            id: effect2
            width: 80; height: width
            // the source where the effect shall be applied to
            property variant source: sourceImage
            // default vertex shader code
            vertexShader: "
                uniform highp mat4 qt_Matrix;
                attribute highp vec4 qt_Vertex;
                attribute highp vec2 qt_MultiTexCoord0;
                varying highp vec2 qt_TexCoord0;
                void main() {
                    qt_TexCoord0 = qt_MultiTexCoord0;
                    gl_Position = qt_Matrix * qt_Vertex;
                }"
            // default fragment shader code
            fragmentShader: "
                varying highp vec2 qt_TexCoord0;
                uniform sampler2D source;
                uniform lowp float qt_Opacity;
                void main() {
                    gl_FragColor = texture2D(source, qt_TexCoord0) * qt_Opacity;
                }"
        }
    }
}
```



在上邊這個例子中，我們在一行中顯示了3張圖片，第一張是原始圖片，第二張使用默認的著色器渲染出來的圖片，第三張使用了Qt5源碼中默認的頂點與片段著色器的代碼進行渲染的圖片。

注意

如果你不想看到原始圖片，而只想看到被著色器渲染後的圖片，你可以設置**Image**為不可見（**visible:false**）。著色器仍然會使用圖片數據，但是圖像元素（**Image Element**）將不會被渲染。

讓我們仔細看看著色器代碼。

```
vertexShader: "
    uniform highp mat4 qt_Matrix;
    attribute highp vec4 qt_Vertex;
    attribute highp vec2 qt_MultiTexCoord0;
    varying highp vec2 qt_TexCoord0;
    void main() {
        qt_TexCoord0 = qt_MultiTexCoord0;
        gl_Position = qt_Matrix * qt_Vertex;
    }"
```

著色器代碼來自Qt這邊的一個字符串，綁定了頂點著色器（vertexShader）與片段著色器（fragmentShader）屬性。每個著色器代碼必須有一個main(){....}函數，它將被GPU執行。Qt已經默認提供了以qt_開頭的變量。

下面是這些變量簡短的介紹：

- uniform-在處理過程中不能夠改變的值。
- attribute-連接外部數據
- varying-著色器之間的共享數據
- highp-高精度值
- lowp-低精度值
- mat4-4x4浮點數（float）矩陣
- vec2-包含兩個浮點數的向量

- sampler2D-2D紋理
- float-浮點數

可以查看OpenGL ES 2.0 API Quick Reference Card獲得更多信息。

現在我們可以更好的理解下面這些變量：

- qt_Matrix：model-view-projection（模型-視圖-投影）矩陣
- qt_Vertex：當前頂點坐標
- qt_MultiTexCoord0：紋理坐標
- qt_TexCoord0：共享紋理坐標

我們已經有可以使用的投影矩陣（projection matrix），當前頂點與紋理坐標。紋理坐標與作為資源（source）的紋理相關。在main()函數中，我們保存紋理坐標，留在後面的片段著色器中使用。每個頂點著色器都需要賦值給gl_Postion，在這裡使用項目矩陣乘以頂點，得到我們3D坐標系中的點。

片段著色器從頂點著色器中接收我們的紋理坐標，這個紋理仍然來自我們的QML資源屬性（source property）。在著色器代碼與QML之間傳遞變量是如此的簡單。此外我們的透明值，在著色器中也可以使用，變量是qt_Opacity。每 個片段著色器需要給gl_FragColor變量賦值，在這裡默認著色器代碼使用資源紋理（source texture）的像素顏色與透明值相乘。

```
fragmentShader: "
    varying highp vec2 qt_TexCoord0;
    uniform sampler2D source;
    uniform lowp float qt_Opacity;
    void main() {
        gl_FragColor = texture2D(source, qt_TexCoord0) * qt_Opacity;
    }"
```

在後面的例子中，我們將會展示一些簡單的著色器例子。首先我們會集中在片段著色器上，然後在回到頂點著色器上。

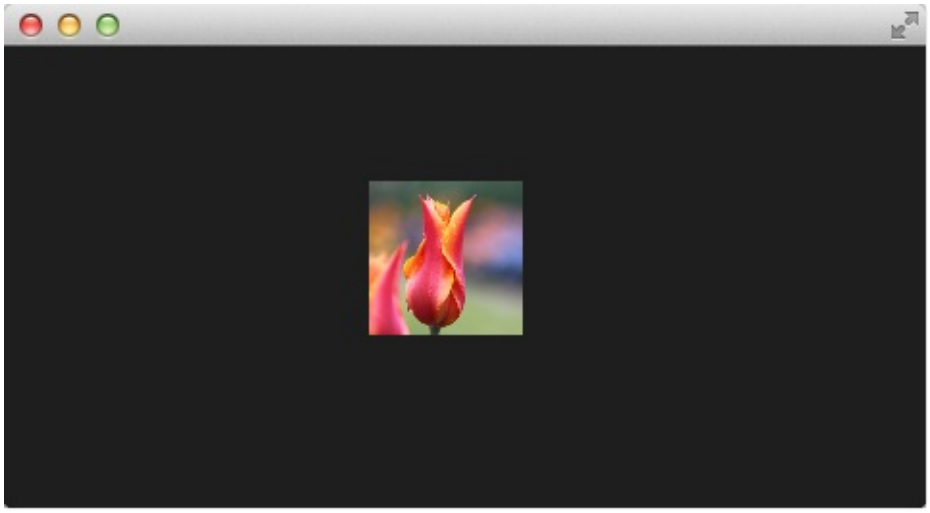
片段著色器（**Fragement Shader**）

片段著色器調用每個需要渲染的像素。我們將開發一個紅色透鏡，它將會增加圖片的紅色通道的值。

配置場景（**Setting up the scene**）

首先我們配置我們的場景，在區域中央使用一個網格顯示我們的源圖片（source image）。

```
import QtQuick 2.0

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    Grid {
        anchors.centerIn: parent
        spacing: 20
        rows: 2; columns: 4
        Image {
            id: sourceImage
            width: 80; height: width
            source: 'assets/tulips.jpg'
        }
    }
}
```



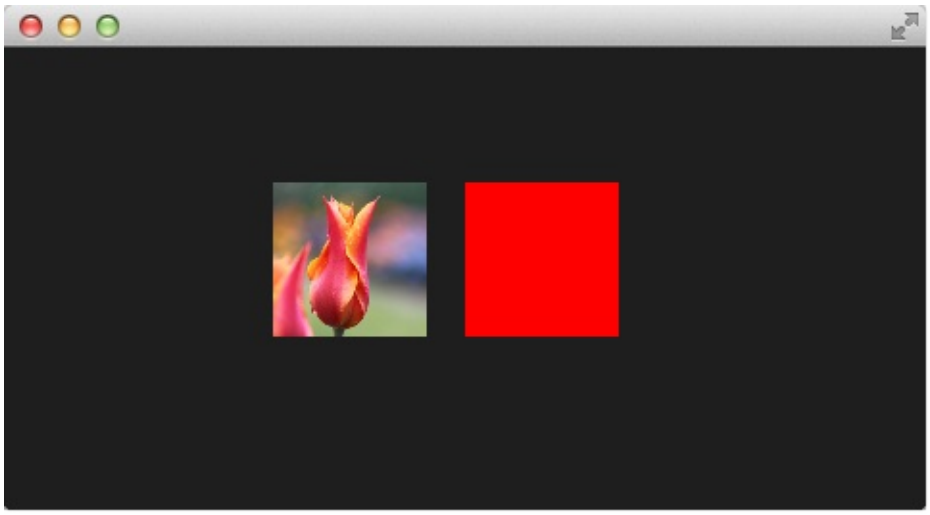
紅色著色器（**A red Shader**）

下一步我們添加一個著色器，顯示一個紅色矩形框。由于我們不需要紋理，我們從頂點著色器中移除紋理。

```
            vertexShader: "
                uniform highp mat4 qt_Matrix;
                attribute highp vec4 qt_Vertex;
                void main() {
                    gl_Position = qt_Matrix * qt_Vertex;
                }"
            fragmentShader: "
                uniform lowp float qt_Opacity;
```

```
            void main() {
                gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0) * qt_Opacity;
            }"
```

在片段著色器中，我們簡單的給gl_FragColor賦值為vec4(1.0, 0.0, 0.0, 1.0)，它代表紅色， 並且不透明（alpha=1.0）。



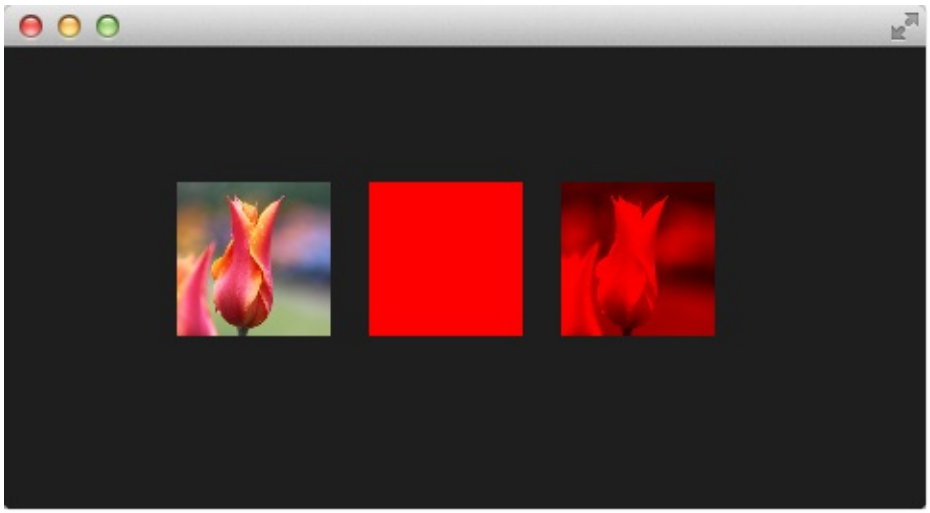
使用紋理的紅色著色器（**A red shader with texture**）

現在我們想要將這個紅色應用在紋理的每個像素上。我們需要將紋理加回頂點著色器。由于我們不再在頂點著色器中做任何其它的事情，所以默認的頂點著色器已經滿足我們的要求。

```
    ShaderEffect {
        id: effect2
        width: 80; height: width
        property variant source: sourceImage
        visible: root.step>1
        fragmentShader: "
            varying highp vec2 qt_TexCoord0;
            uniform sampler2D source;
            uniform lowp float qt_Opacity;
            void main() {
                gl_FragColor = texture2D(source, qt_TexCoord0) * vec4(1.0, 0.0, 0.0,
            }"
    }
```

完整的著色器重新包含我們的源圖片作為屬性，由于我們沒有特殊指定，使用默認的頂點著色器，我沒有重寫頂點著色器。

在片段著色器中，我們提取紋理片段texture2D(source,qt_TexCoord0)，並且與紅色一起應用。

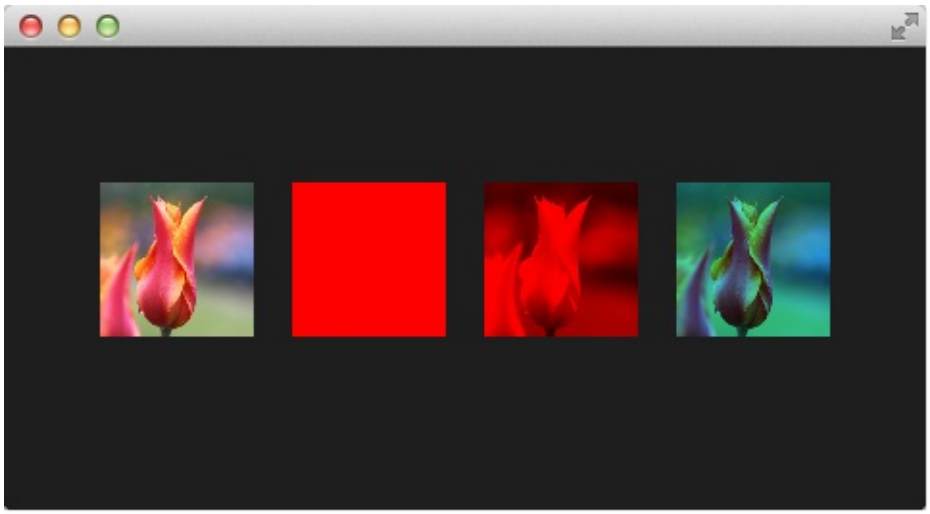


紅色通道屬性（**The red channel property**）

這樣的代碼用來修改紅色通道的值看起來不是很好，所以我們想要將這個值包含在QML這邊。我們在ShaderEffect中增加一個redChannel屬性，並在我們的片段著色器中申明一個uniform lowpfloat redChannel。這就是從一個著色器代碼中標記一個值到QML這邊的方法，非常簡單。

```
        ShaderEffect {
            id: effect3
            width: 80; height: width
            property variant source: sourceImage
            property real redChannel: 0.3
            visible: root.step>2
            fragmentShader: "
                varying highp vec2 qt_TexCoord0;
                uniform sampler2D source;
                uniform lowp float qt_Opacity;
                uniform lowp float redChannel;
                void main() {
                    gl_FragColor = texture2D(source, qt_TexCoord0) * vec4(redChannel, 1.0
                }"
        }
```

為了讓這個透鏡更真實，我們改變vec4顏色為vec4(redChannel, 1.0, 1.0, 1.0)，這樣其它顏色與1.0相乘，只有紅色部分使用我們的redChannel變量。

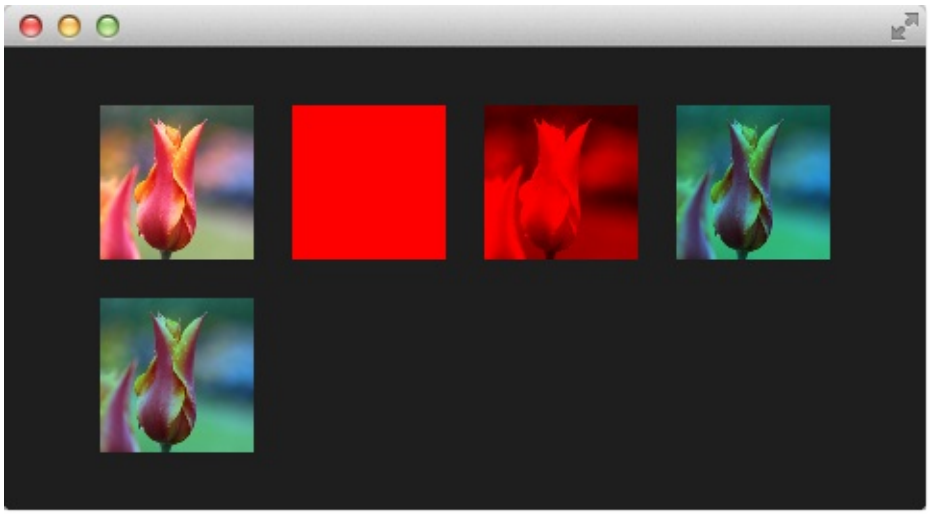


紅色通道的動畫（**The red channel animated**）

由于redChannel屬性僅僅是一個正常的屬性，我們也可以像其它QML中的屬性一樣使用動畫。我們使用QML屬性在GPU上改變這個值，來影響我們的著色器，這真酷！

```
        ShaderEffect {
            id: effect4
            width: 80; height: width
            property variant source: sourceImage
            property real redChannel: 0.3
            visible: root.step>3
            NumberAnimation on redChannel {
                from: 0.0; to: 1.0; loops: Animation.Infinite; duration: 4000
            }

            fragmentShader: "
                varying highp vec2 qt_TexCoord0;
                uniform sampler2D source;
                uniform lowp float qt_Opacity;
                uniform lowp float redChannel;
                void main() {
                    gl_FragColor = texture2D(source, qt_TexCoord0) * vec4(redChannel, 1.0
                }"
        }
```

下面是最後的結果。



在這4秒內，第二排的著色器紅色通道的值從0.0到1.0。圖片從沒有紅色信息（0.0 red）到一個正常的圖片（1.0 red）。

波浪效果（**Wave Effect**）

在這個更加復雜的例子中，我們使用片段著色器創建一個波浪效果。波浪的形成是基于sin曲線，並且它影響了使用的紋理坐標的顏色。

```
import QtQuick 2.0

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    Row {
        anchors.centerIn: parent
        spacing: 20
        Image {
            id: sourceImage
            width: 160; height: width
            source: "assets/coastline.jpg"
        }
        ShaderEffect {
            width: 160; height: width
            property variant source: sourceImage
            property real frequency: 8
            property real amplitude: 0.1
            property real time: 0.0
            NumberAnimation on time {
                from: 0; to: Math.PI*2; duration: 1000; loops: Animation.Infinite
            }

            fragmentShader: "
                varying highp vec2 qt_TexCoord0;
                uniform sampler2D source;
                uniform lowp float qt_Opacity;
                uniform highp float frequency;
                uniform highp float amplitude;
                uniform highp float time;
                void main() {
                    highp vec2 pulse = sin(time - frequency * qt_TexCoord0);
                    highp vec2 coord = qt_TexCoord0 + amplitude * vec2(pulse.x, -pulse.x)
                    gl_FragColor = texture2D(source, coord) * qt_Opacity;
                }"
        }
    }
}
```

波浪的計算是基于一個脈衝與紋理坐標的操作。我們使用一個基于當前時間與使用的紋理坐標的sin波浪方程式來實現脈衝。

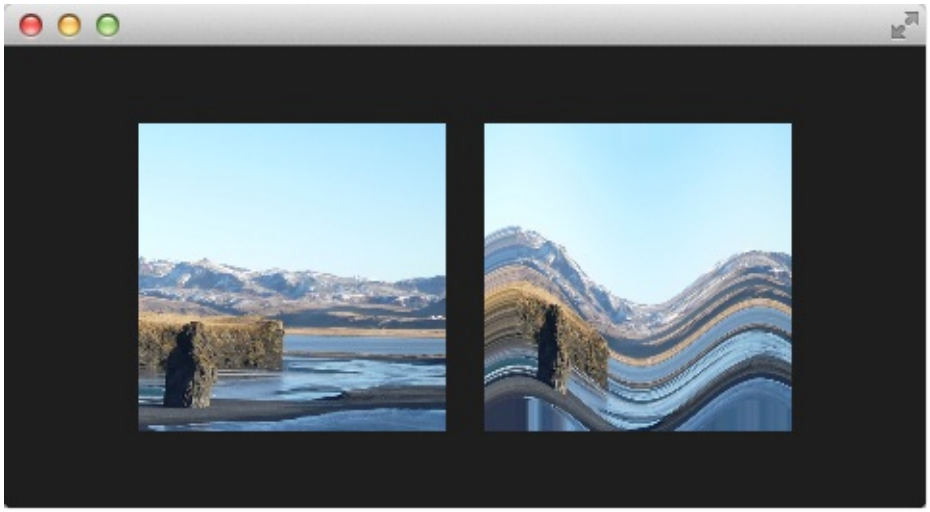
```
highp vec2 pulse = sin(time - frequency * qt_TexCoord0);
```

離開了時間的因素，我們僅僅只有扭曲，而不是像波浪一樣運動的扭曲。

我們使用不同的紋理坐標作為顏色。

```
highp vec2 coord = qt_TexCoord0 + amplitude * vec2(pulse.x, -pulse.x);
```

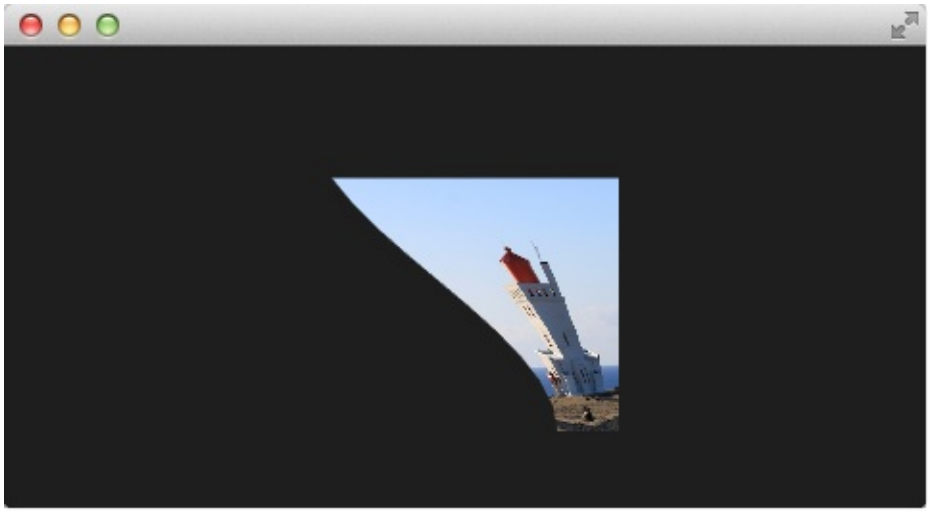
紋理坐標受我們的x脈衝值影響，結果就像一個移動的波浪。



如果我們沒有在片段著色器中使用像素的移動，這個效果可以首先考慮使用頂點著色器來完成。

頂點著色器（**Vertex Shader**）

頂點著色器用來操作ShaderEffect提供的頂點。正常情況下，ShaderEffect有4個頂點（左上top-left，右上top-right，左下bottom-left，右下bottom-right）。每個頂點使用vec4類型記錄。為了實現頂點著色器的可視化，我們將編寫一個吸收的效果。這個效果通常被用來讓一個矩形窗口消失為一個點。

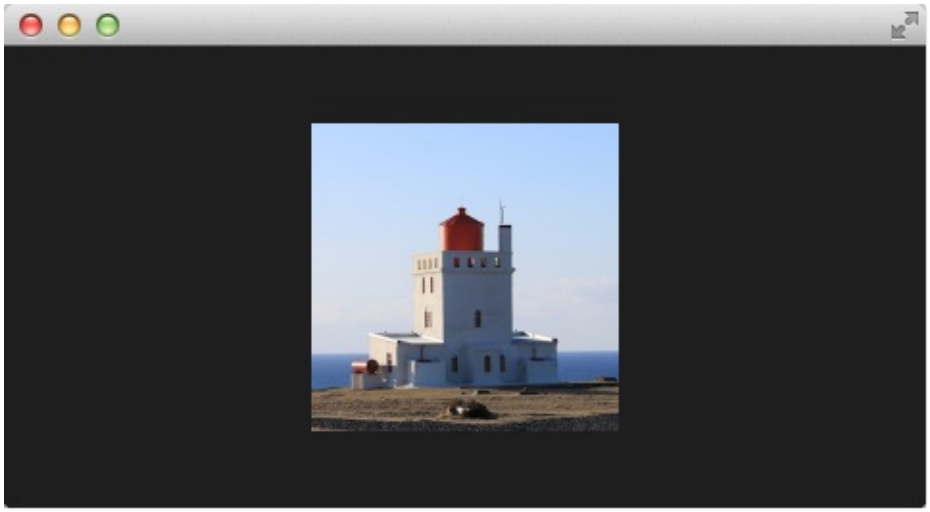


配置場景（**Setting up the scene**）

首先我們再一次配置場景。

```
import QtQuick 2.0

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    Image {
        id: sourceImage
        width: 160; height: width
        source: "assets/lighthouse.jpg"
        visible: false
    }
    Rectangle {
        width: 160; height: width
        anchors.centerIn: parent
        color: '#333333'
    }
    ShaderEffect {
        id: genieEffect
        width: 160; height: width
        anchors.centerIn: parent
        property variant source: sourceImage
        property bool minimized: false
        MouseArea {
            anchors.fill: parent
            onClicked: genieEffect.minimized = !genieEffect.minimized
        }
    }
}
```

這個場景使用了一個黑色背景，並且提供了一個使用圖片作為資源紋理的ShaderEffect。使用image元素的原圖片是不可見的，只是給我們的吸收效果提供資源。此外我們在ShaderEffect的位置添加了一個同樣大小的黑色矩形框，這樣我們可以更加明確的知道我們需要點擊哪裡來重置效果。



點擊圖片將會觸發效果，MouseArea覆蓋了ShaderEffect。在onClicked操作中，我們綁定了自定義的布爾變量屬性minimized。我們稍後使用這個屬性來觸發效果。

最小化與正常化（**Minimize and normalize**）

在我們配置好場景後，我們定義一個real類型的屬性，叫做minimize，這個屬性包含了我們當前最小化的值。這個值在0.0到1.0之間，由一個連續的動畫來控制它。

```
        property real minimize: 0.0

        SequentialAnimation on minimize {
            id: animMinimize
            running: genieEffect.minimized
            PauseAnimation { duration: 300 }
            NumberAnimation { to: 1; duration: 700; easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1000 }
        }

        SequentialAnimation on minimize {
            id: animNormalize
            running: !genieEffect.minimized
            NumberAnimation { to: 0; duration: 700; easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1300 }
        }
```

這個動畫綁定了由minimized屬性觸發。現在我們已經配置好我們的環境，最後讓我們看看頂點著色器的代碼。

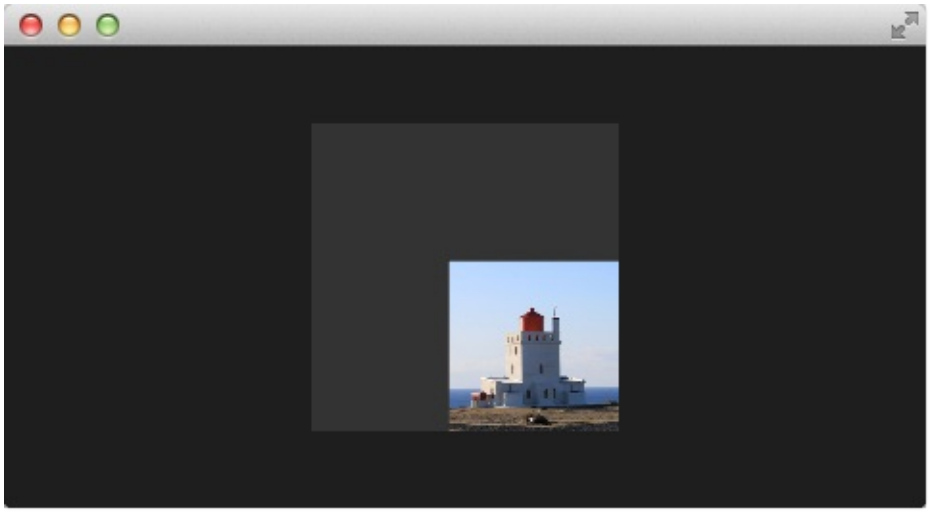
```
        vertexShader: "
            uniform highp mat4 qt_Matrix;
            attribute highp vec4 qt_Vertex;
            attribute highp vec2 qt_MultiTexCoord0;
            varying highp vec2 qt_TexCoord0;
            uniform highp float minimize;
            uniform highp float width;
            uniform highp float height;
            void main() {
```

```
                qt_TexCoord0 = qt_MultiTexCoord0;
                highp vec4 pos = qt_Vertex;
                pos.y = mix(qt_Vertex.y, height, minimize);
                pos.x = mix(qt_Vertex.x, width, minimize);
                gl_Position = qt_Matrix * pos;
            }"
```

頂點著色器被每個頂點調用，在我們這個例子中，一共調用了四次。默認下提供qt已定義的參數，如qt_Matrix，qt_Vertex，qt_MultiTexCoord0，qt_TexCoord0。我們在之前已經討論過這些變量。此外我們從ShaderEffect中鏈接minimize，width與height的值到我們的頂點著色器代碼中。在main函數中，我們將當前紋理值保存在qt_TexCoord()中，讓它在片段著色器中可用。現在我們拷貝當前位置，並修改頂點的x,y的位置。

```
highp vec4 pos = qt_Vertex;
pos.y = mix(qt_Vertex.y, height, minimize);
pos.x = mix(qt_Vertex.x, width, minimize);
```

mix(...)函數提供了一種在兩個參數之間（0.0到1.0）的線性插值的算法。在我們的例子中，在當前y值與高度值之間基于minimize的值插值獲得y值，x的值獲取類似。記住minimize的值是由我們的連續動畫控制，並且在0.0到1.0之間（反之亦然）。



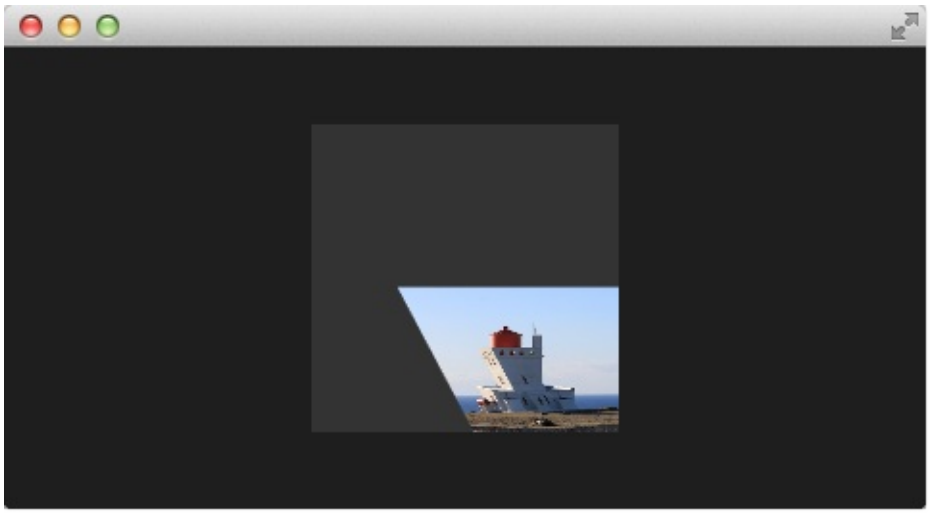
這個結果的效果不是真正吸收效果，但是已經能朝著這個目標完成了一大步。

基礎彎曲（**Primitive Bending**）

我們已經完成了最小化我們的坐標。現在我們想要修改一下對x值的操作，讓它依賴當前的y值。這個改變很簡單。y值計算在前。x值的插值基于當前頂點的y坐標。

```
highp float t = pos.y / height;
pos.x = mix(qt_Vertex.x, width, t * minimize);
```

這個結果造成當y值比較大時，x的位置更靠近width的值。也就是說上面2個頂點根本不受影響，它們的y值始終為0，下面兩個頂點的x坐標值更靠近width的值，它們最後轉向同一個x值。



```
import QtQuick 2.0

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    Image {
        id: sourceImage
        width: 160; height: width
        source: "assets/lighthouse.jpg"
        visible: false
    }
    Rectangle {
        width: 160; height: width
        anchors.centerIn: parent
        color: '#333333'
    }
    ShaderEffect {
        id: genieEffect
        width: 160; height: width
        anchors.centerIn: parent
        property variant source: sourceImage
        property real minimize: 0.0
        property bool minimized: false


        SequentialAnimation on minimize {
            id: animMinimize
            running: genieEffect.minimized
            PauseAnimation { duration: 300 }
            NumberAnimation { to: 1; duration: 700; easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1000 }
        }

        SequentialAnimation on minimize {
            id: animNormalize
            running: !genieEffect.minimized
            NumberAnimation { to: 0; duration: 700; easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1300 }
        }


        vertexShader: "
            uniform highp mat4 qt_Matrix;
            uniform highp float minimize;
```

```
            uniform highp float height;
            uniform highp float width;
            attribute highp vec4 qt_Vertex;
            attribute highp vec2 qt_MultiTexCoord0;
            varying highp vec2 qt_TexCoord0;
            void main() {
                qt_TexCoord0 = qt_MultiTexCoord0;
                // M1>>
                highp vec4 pos = qt_Vertex;
                pos.y = mix(qt_Vertex.y, height, minimize);
                highp float t = pos.y / height;
                pos.x = mix(qt_Vertex.x, width, t * minimize);
                gl_Position = qt_Matrix * pos;
```

更好的彎曲（**Better Bending**）

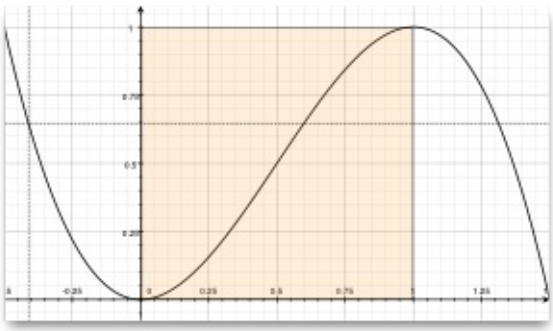
現在簡單的彎曲並不能真正的滿足我們的要求，我們將添加幾個部件來提升它的效果。首先我們增加動畫，支持一個自定義的彎曲屬性。這是非常必要的，由于彎曲立即發生，y值的最小化需要被推遲。兩個動畫在同一持續時間計算總和（300+700+100與700+1300）。

```
        property real bend: 0.0
        property bool minimized: false


        // change to parallel animation
        ParallelAnimation {
            id: animMinimize
            running: genieEffect.minimized
            SequentialAnimation {
                PauseAnimation { duration: 300 }
                NumberAnimation {
                    target: genieEffect; property: 'minimize';
                    to: 1; duration: 700;
                    easing.type: Easing.InOutSine
                }
                PauseAnimation { duration: 1000 }
            }
            // adding bend animation
            SequentialAnimation {
                NumberAnimation {
                    target: genieEffect; property: 'bend'
                    to: 1; duration: 700;
                    easing.type: Easing.InOutSine }
                PauseAnimation { duration: 1300 }
            }
        }
```

此外，為了使彎曲更加平滑，不再使用y值影響x值的彎曲函數，pos.x現在依賴新的彎曲屬性動畫：

```
highp float t = pos.y / height;
t = (3.0 - 2.0 * t) * t * t;
pos.x = mix(qt_Vertex.x, width, t * bend);
```

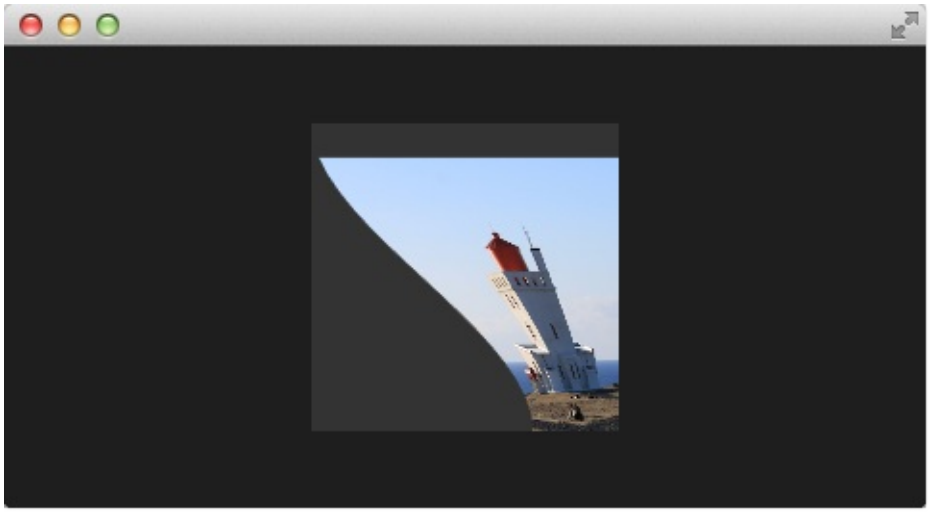
彎曲從0.0平滑開始，逐漸加快，在1.0時逐漸平滑。下面是這個函數在指定範圍內的曲線圖。對于我們，只需要關注0到1的區間。



想要獲得最大化的視覺改變，需要增加我們的頂點數量。可以使用網眼（mesh）來增加頂點：

```
mesh: GridMesh { resolution: Qt.size(16, 16) }
```

現在ShaderEffect被分布為16x16頂點的網格，替換了之前2x2的頂點。這樣頂點之間的插值將會看起來更加平滑。

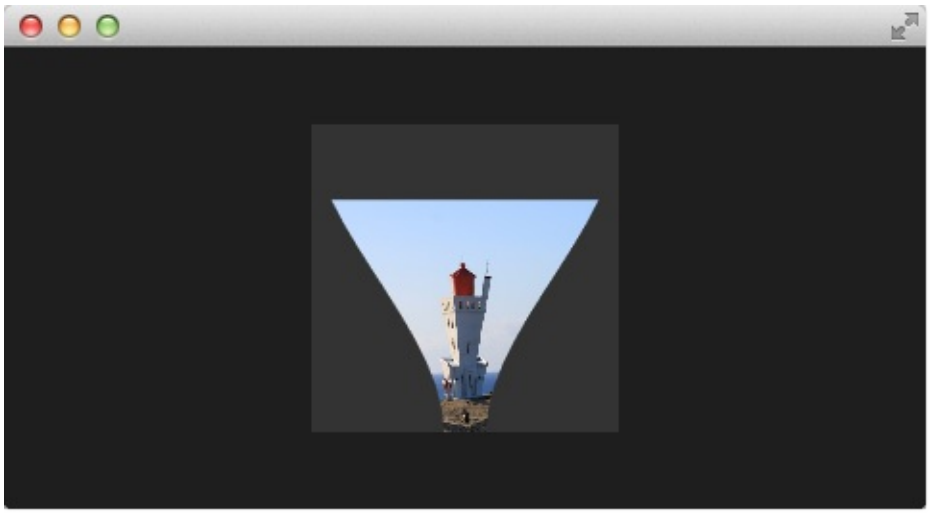


你可以看見曲線的變化，在最後讓彎曲變得非常平滑。這讓彎曲有了更加強大的效果。

側面收縮（**Choosing Sides**）

最後一個增強，我們希望能夠收縮邊界。邊界朝著吸收的點消失。直到現在它總是在朝著width值的點消失。添加一個邊界屬性，我們能夠修改這個點在0到width之間。

```
ShaderEffect {
    ...
    property real side: 0.5

    vertexShader: "
        ...
        uniform highp float side;
        ...
        pos.x = mix(qt_Vertex.x, side * width, t * bend);
    "
}
```



包裝（**Packing**）

最後將我們的效果包裝起來。將我們吸收效果的代碼提取到一個叫做GenieEffect的自定義組件中。它使用ShaderEffect作為根元素。移除掉MouseArea，這不應該放在組件中。綁定minimized屬性來觸發效果。

```
import QtQuick 2.0

ShaderEffect {
    id: genieEffect
    width: 160; height: width
    anchors.centerIn: parent
    property variant source
    mesh: GridMesh { resolution: Qt.size(10, 10) }
    property real minimize: 0.0
    property real bend: 0.0
    property bool minimized: false
    property real side: 1.0


    ParallelAnimation {
        id: animMinimize
        running: genieEffect.minimized
        SequentialAnimation {
            PauseAnimation { duration: 300 }
            NumberAnimation {
                target: genieEffect; property: 'minimize';
                to: 1; duration: 700;
                easing.type: Easing.InOutSine
            }
            PauseAnimation { duration: 1000 }
        }
        SequentialAnimation {
            NumberAnimation {
                target: genieEffect; property: 'bend'
                to: 1; duration: 700;
                easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1300 }
        }
    }

    ParallelAnimation {
        id: animNormalize
        running: !genieEffect.minimized
        SequentialAnimation {
```

```
            NumberAnimation {
                target: genieEffect; property: 'minimize';
                to: 0; duration: 700;
                easing.type: Easing.InOutSine
            }
            PauseAnimation { duration: 1300 }
        }
        SequentialAnimation {
            PauseAnimation { duration: 300 }
            NumberAnimation {
                target: genieEffect; property: 'bend'
                to: 0; duration: 700;
                easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1000 }
        }
    }

    vertexShader: "
        uniform highp mat4 qt_Matrix;
        attribute highp vec4 qt_Vertex;
        attribute highp vec2 qt_MultiTexCoord0;
        uniform highp float height;
        uniform highp float width;
        uniform highp float minimize;
        uniform highp float bend;
        uniform highp float side;
        varying highp vec2 qt_TexCoord0;
        void main() {
            qt_TexCoord0 = qt_MultiTexCoord0;
            highp vec4 pos = qt_Vertex;
            pos.y = mix(qt_Vertex.y, height, minimize);
            highp float t = pos.y / height;
            t = (3.0 - 2.0 * t) * t * t;
            pos.x = mix(qt_Vertex.x, side * width, t * bend);
            gl_Position = qt_Matrix * pos;
        }"
}
```

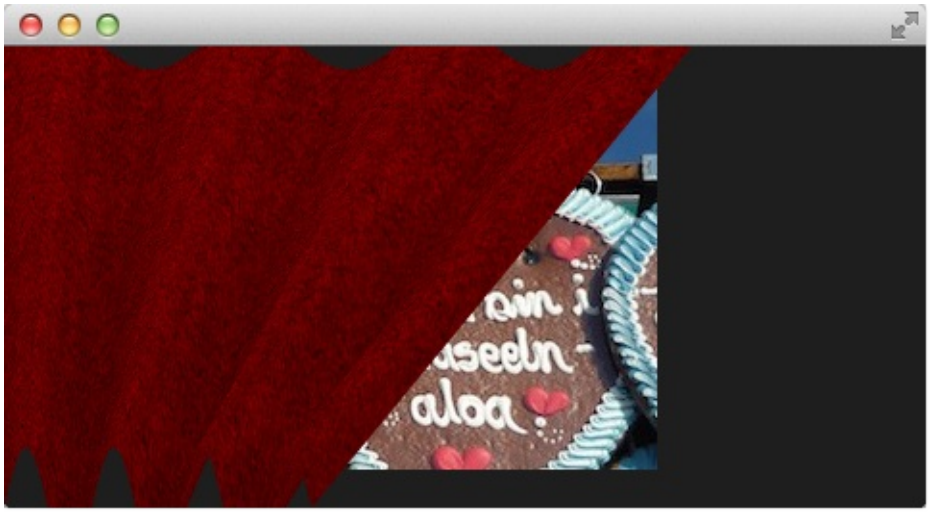
你現在可以像這樣簡單的使用這個效果：

```
import QtQuick 2.0

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    GenieEffect {
        source: Image { source: 'assets/lighthouse.jpg' }
        MouseArea {
            anchors.fill: parent
            onClicked: parent.minimized = !parent.minimized
        }
    }
}
```

我們簡化了代碼，移除了背景矩形框，直接使用圖片完成效果，替換了在一個單獨的圖像元素中加載它。

劇幕效果（Curtain Effect）

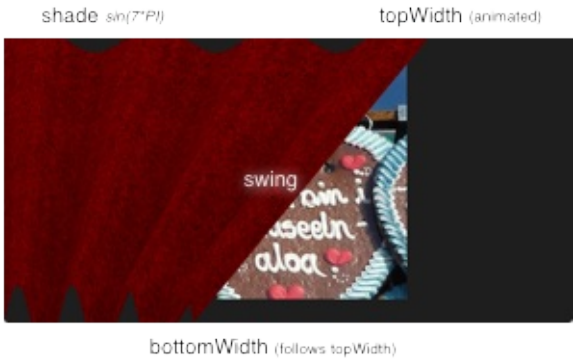
在最後的自定義效果例子中，我們將帶來一個劇幕效果。這個效果是2011年5月Qt實驗室發布的著色器效果中的一部分。目前網址已經轉到blog.qt.digia.com，不知道還能不能找到。



當時我非常喜歡這些效果，劇幕效果是我最喜愛的一個。我喜歡劇幕打開然後遮擋後面的背景對象。

我將代碼移植適配到Qt5上，這非常簡單。同時我做了一些簡化讓它能夠更好的展示。如果你對整個例子有興趣，可以訪問Qt實驗室的博客。

只有一個小組件作為背景，劇幕實際上是一張圖片，叫做fabric.jpg，它是ShaderEffect的資源。整個效果使用頂點著色器來擺動劇幕，使用片段著色器提供陰影的效果。下面是一個簡單的圖片，讓你更加容易理解代碼。



劇幕的波形陰影通過一個在劇幕寬度上的sin曲線使用7的振幅來計算（7*PI=221.99..）另一個重要的部分是擺動，當劇幕打開或者關閉時，使用動畫來播放劇幕的topWidth。bottomWidth使用SpringAnimation來跟隨topWidth變化。這樣我們就能創建出底部擺動的劇幕效果。計算得到的swing提供了搖擺的強度，用來對頂點的y值進行插值。

劇幕效果放在CurtainEffect.qml組件中，fabric圖像作為紋理資源。在陰影的使用上沒有新的東西加入，唯一不同的是在頂點著色器中操作gl_Postion和片段著色器中操作gl_FragColor。

```
import QtQuick 2.0

ShaderEffect {
    anchors.fill: parent
```

```
mesh: GridMesh {
    resolution: Qt.size(50, 50)
}

property real topWidth: open?width:20
property real bottomWidth: topWidth
property real amplitude: 0.1
property bool open: false
property variant source: effectSource

Behavior on bottomWidth {
    SpringAnimation {
        easing.type: Easing.OutElastic;
        velocity: 250; mass: 1.5;
        spring: 0.5; damping: 0.05
    }
}

Behavior on topWidth {
    NumberAnimation { duration: 1000 }
}


ShaderEffectSource {
    id: effectSource
    sourceItem: effectImage;
    hideSource: true
}

Image {
    id: effectImage
    anchors.fill: parent
    source: "assets/fabric.jpg"
    fillMode: Image.Tile
}

vertexShader: "
    attribute highp vec4 qt_Vertex;
    attribute highp vec2 qt_MultiTexCoord0;
    uniform highp mat4 qt_Matrix;
    varying highp vec2 qt_TexCoord0;
    varying lowp float shade;

    uniform highp float topWidth;
    uniform highp float bottomWidth;
    uniform highp float width;
    uniform highp float height;
    uniform highp float amplitude;

    void main() {
        qt_TexCoord0 = qt_MultiTexCoord0;

        highp vec4 shift = vec4(0.0, 0.0, 0.0, 0.0);
        highp float swing = (topWidth - bottomWidth) * (qt_Vertex.y / height);
        shift.x = qt_Vertex.x * (width - topWidth + swing) / width;

        shade = sin(21.9911486 * qt_Vertex.x / width);
        shift.y = amplitude * (width - topWidth + swing) * shade;

        gl_Position = qt_Matrix * (qt_Vertex - shift);

        shade = 0.2 * (2.0 - shade ) * ((width - topWidth + swing) / width);
```

```
        }"

    fragmentShader: "
        uniform sampler2D source;
        varying highp vec2 qt_TexCoord0;
        varying lowp float shade;
        void main() {
            highp vec4 color = texture2D(source, qt_TexCoord0);
            color.rgb *= 1.0 - shade;
            gl_FragColor = color;
        }"
}
```

這個效果在curtaindemo.qml文件中使用。

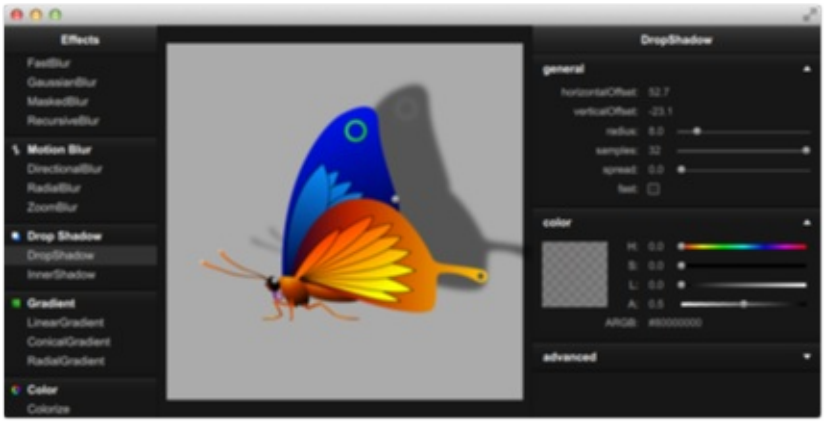
```
import QtQuick 2.0

Rectangle {
    id: root
    width: 480; height: 240
    color: '#1e1e1e'

    Image {
        anchors.centerIn: parent
        source: 'assets/wiesn.jpg'
    }

    CurtainEffect {
        id: curtain
        anchors.fill: parent
    }

    MouseArea {
        anchors.fill: parent
        onClicked: curtain.open = !curtain.open
    }
}
```

劇幕效果通過自定義的open屬性打開。我們使用了一個MouseArea來觸發打開和關閉劇幕。

Qt圖像效果庫（Qt GraphicsEffect Library）

圖像效果庫是一個著色器效果的集合，是由Qt開發者提供制作的。它是一個很好的工具，你可以將它應用在你的程序中，它也是一個學習如何創建著色器的例子。

圖像效果庫附帶了一個手動測試平台，這個工具可以幫助你測試發現不同的效果 測試工具在$QTDIR/qtgraphicaleffects/tests/manual/testbed下。



效果庫包含了大約20種效果，下面是效果列表和一些簡短的描述。

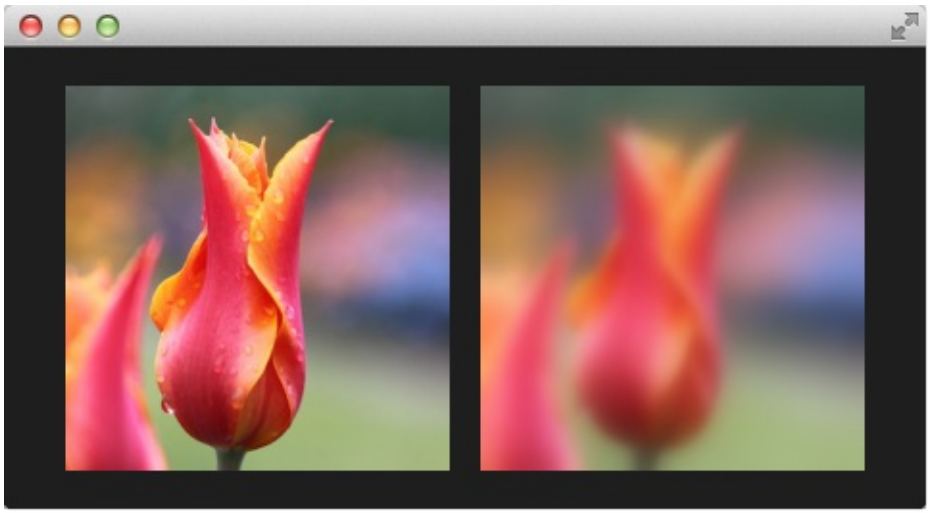
種類	效果	描述
混合（Blend）	混合（Blend）	使用混合模式合並兩個資源項
顏色（Color）	亮度與對比度（BrightnessContrast）	調整亮度與對比度
	著色（Colorize）	設置HSL顏色空間顏色
	顏色疊加（ColorOverlay）	應用一個顏色層
	降低飽和度（Desaturate）	減少顏色飽和度
	伽馬調整（GammaAdjust）	調整發光度
	色調飽和度（HueSaturation）	調整HSL顏色空間顏色
	色階調整（LevelAdjust）	調整RGB顏色空間顏色
漸變（Gradient）	圓錐漸變（ConicalGradient）	繪制一個圓錐漸變
	線性漸變（LinearGradient）	繪制一個線性漸變
	射線漸變（RadialGradient）	繪制一個射線漸變
失真（Distortion）	置換（Displace）	按照指定的置換源移動源項的像素
陰影（Drop Shadow）	陰影 （DropShadow）	繪制一個陰影
	內陰影（InnerShadow）	繪制一個內陰影
模糊 （Blur）	快速模糊（FastBlur）	應用一個快速模糊效果
	高斯模糊（GaussianBlur）	應用一個高質量模糊效果
	蒙版模糊（MaskedBlur）	應用一個多種強度的模糊效果
	遞歸模糊（RecursiveBlur）	重復模糊，提供一個更強的模糊效果
運動模糊（Motion Blur）	方向模糊（DirectionalBlur）	應用一個方向的運動模糊效果
	放射模糊（RadialBlur）	應用一個放射運動模糊效果
	變焦模糊（ZoomBlur）	應用一個變焦運動模糊效果

發光（Glow）	發光（Glow）	繪制一個外發光效果
	矩形發光（RectangularGlow）	繪制一個矩形外發光效果
蒙版（Mask）	透明蒙版（OpacityMask）	使用一個源項遮擋另一個源項
	閾值蒙版（ThresholdMask）	使用一個閾值，一個源項遮擋另一個源項

下面是一個使用快速模糊效果的例子：

```
import QtQuick 2.0
import QtGraphicalEffects 1.0

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    Row {
        anchors.centerIn: parent
        spacing: 16

        Image {
            id: sourceImage
            source: "assets/tulips.jpg"
            width: 200; height: width
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
        }

        FastBlur {
            width: 200; height: width
            source: sourceImage
            radius: blurred?32:0
            property bool blurred: false

            Behavior on radius {
                NumberAnimation { duration: 1000 }
            }

            MouseArea {
                id: area
                anchors.fill: parent
                onClicked: parent.blurred = !parent.blurred
            }
        }
    }
}
```

左邊是原圖片。點擊右邊的圖片將會觸發blurred屬性，模糊在1秒內從0到32。左邊顯示模糊後的圖片。



Multimedia

在QtMultimedia模塊中的multimedia元素可以播放和記錄媒體資源，例如聲音，視頻，或者圖片。解碼和編碼的操作由特定的後台完成。例如在Linux上的gstreamer框架，Windows上的DirectShow，和OS X上的QuickTime。 multimedia元素不是QtQuick核心的接口。它的接口通過導入QtMultimedia 5.0來加入，如下所示：

```
import QtMultimedia 5.0
```

媒體播放（**Playing Media**）

在QML應用程序中，最基本的媒體應用是播放媒體。使用MediaPlayer元素可以完成它，如果源是一個圖片或者視頻，可以選擇結合VideoOutput元素。MediaPlayer元素有一個source屬性指向需要播放的媒體。當媒體源被綁定後，簡單的調用play函數就可以開始播放。

如果你想播放一個可視化的媒體，例如圖片或者視頻等，你需要配置一個VideoOutput元素。MediaPlayer播放通過source屬性與視頻輸出綁定。

在下面的例子中，給MediaPlayer元素一個視頻文件作為source。一個VideoOutput被創建和綁定到媒體播放器上。一旦主要部件完全初始化，例如在Component.onCompleted中，播放器的play函數被調用。

```
import QtQuick 2.0
import QtMultimedia 5.0
import QtSystemInfo 5.0

Item {
    width: 1024
    height: 600

    MediaPlayer {
        id: player
        source: "trailer_400p.ogg"
    }

    VideoOutput {
        anchors.fill: parent
        source: player
    }

    Component.onCompleted: {
        player.play();
    }

    ScreenSaver {
        screenSaverEnabled: false;
    }
}
// M1>>
```

除了上面介紹的視頻播放，這個例子也包括了一小段代碼用于禁止屏幕保護。這將阻止視頻被中斷。通過設置ScreenSaver元素的screenSaverEnabled屬性為false來完成。通過導入QtSystemInfo 5.0可以使用ScreenSaver元素。

基礎操作例如當播放媒體時可以通過MediaPlayer元素的volume屬性來控制音量。還有一些其它有用的屬性。例如，duration與position屬性可以用來創建一個進度條。如果seekable屬性為true，當撥動進度條時可以更新position屬性。下面這個例子展示了在上面的例子基礎上如何添加基礎播放。

```
    Rectangle {
        id: progressBar

        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
```

```
        anchors.margins: 100

        height: 30

        color: "lightGray"

        Rectangle {
            anchors.left: parent.left
            anchors.top: parent.top
            anchors.bottom: parent.bottom

            width: player.duration>0?parent.width*player.position/player.duration:0

            color: "darkGray"
        }

        MouseArea {
            anchors.fill: parent

            onClicked: {
                if (player.seekable)
                    player.position = player.duration * mouse.x/width;
            }
        }
    }
```

默認情況下position屬性每秒更新一次。這意味著進度條將只會在大跨度下的時間週期下才會更新，需要媒體持續時間足夠長，進度條像素足夠寬。然而，這個可以通過mediaObject屬性的notifyInterval屬性改變。它可以設置每個position之間更新的毫秒數，增加用戶界面的平滑度。

```
    Connections {
        target: player
        onMediaObjectChanged: {
            if (player.mediaObject)
                player.mediaObject.notifyInterval = 50;
        }
    }
```

當使用MediaPlayer創建一個媒體播放器時，最好使用status屬性來監聽播放器。這個屬性是一個枚舉，它枚舉了播放器可能出現的狀態，從MediaPlayer.Buffered到MediaPlayer.InvalidMedia。下面是這些狀態值的總結：

- MediaPlayer.UnknownStatus - 未知狀態
- MediaPlayer.NoMedia - 播放器沒有指定媒體資源，播放停止
- MediaPlayer.Loading - 播放器正在加載媒體
- MediaPlayer.Loaded - 媒體已經加載完畢，播放停止
- MediaPlayer.Stalled - 加載媒體已經停止
- MediaPlayer.Buffering - 媒體正在緩衝
- MediaPlayer.Buffered - 媒體緩衝完成

- MediaPlayer.EndOfMedia - 媒體播放完畢，播放停止
- MediaPlayer.InvalidMedia - 無法播放媒體，播放停止

正如上面提到的這些枚舉項，播放狀態會隨著時間變化。調用play，pause或者stop將會切換狀態，但由于媒體的原因也會影響這些狀態。例如，媒體播放完畢，它將會無效，導致播放停止。當前的播放狀態可以使用playbackState屬性跟蹤。這個值可能是MediaPlayer.PlayingState，MediaPlayer.PasuedState或者MediaPlayer.StoppedState。

使用autoPlay屬性，MediaPlayer在source屬性改變時將會嘗試進入播放狀態。類似的屬性autoLoad將會導致播放器在source屬性改變時嘗試加載媒體。默認下autoLoad是被允許的。

當然也可以讓MediaPlayer循環播放一個媒體項。loops屬性控制source將會被重復播放多少次。設置屬性為MediaPlayer.Infinite將會導致不停的重播。非常適合持續的動畫或者一個重復的背景音樂。

聲音效果（**Sounds Effects**）

當播放聲音效果時，從請求播放到真實響應播放的響應時間非常重要。在這種情況下，SoundEffect元素將會派上用場。設置source屬性，一個簡單調用play函數會直接開始播放。

當敲擊屏幕時，可以使用它來完成音效反饋，如下所示：

```
SoundEffect {
    id: beep
    source: "beep.wav"
}

Rectangle {
    id: button

    anchors.centerIn: parent

    width: 200
    height: 100

    color: "red"

    MouseArea {
        anchors.fill: parent
        onClicked: beep.play()
    }
}
```

這個元素也可以用來完成一個配有音效的轉換。為了從轉換觸發，使用ScriptAction元素。

```
SoundEffect {
    id: swosh
    source: "swosh.wav"
}

transitions: [
    Transition {
        ParallelAnimation {
            ScriptAction { script: swosh.play(); }
            PropertyAnimation { properties: "rotation"; duration: 200; }
        }
    }
]
```

除了調用play函數，在MediaPlayer中類似屬性也可以使用。比如volume和loops。loops可以設置為SoundEffect.Infinite來提供無限重復播放。停止播放調用stop函數。

注意

當後台使用**PulseAudio**時，**stop**將不會立即停止，但會阻止繼續循環。這是由于底層**API**的限制造成的。

視頻流（**Video Streams**）

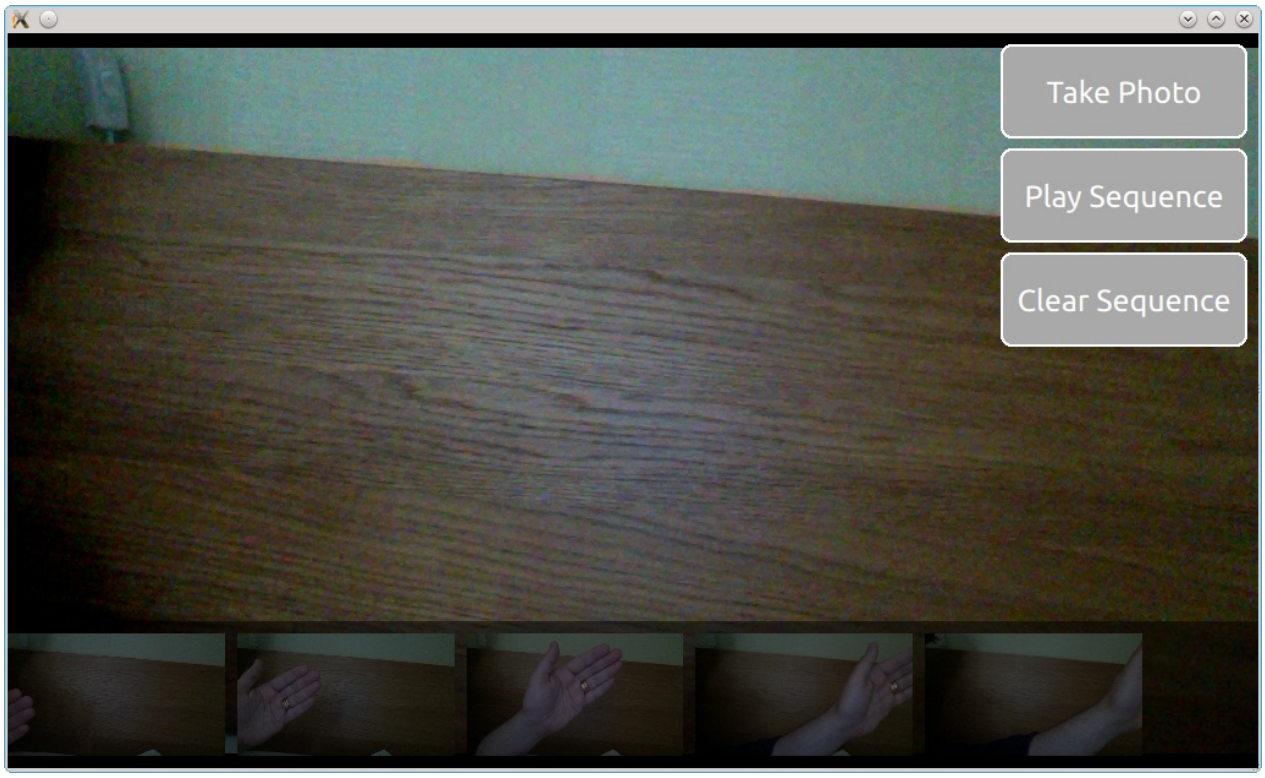
VideoOutput元素不被限制與MediaPlayer元素綁定使用的。它也可以直接用來加載實時視頻資源顯示一個流媒體。應用程序使用Camera元素作為資源。來自Camera的視頻流給用戶提供了一個實時流媒體。

```
import QtQuick 2.0
import QtMultimedia 5.0

Item {
    width: 1024
    height: 600

    VideoOutput {
        anchors.fill: parent
        source: camera
    }

    Camera {
        id: camera
    }
}
```

捕捉圖像（**Capturing Images**）

Camera元素一個關鍵特性就是可以用來拍照。我們將在一個簡單的定格動畫程序中使用到它。在這章中，你將學習如何顯示一個視圖查找器，截圖和追蹤拍攝的圖片。

用戶界面如下所示。它由三部分組成，背景是一個視圖查找器，右邊有一列按鈕，底部有一連串拍攝的圖片。我們想要拍攝一系列的圖片，然後點擊Play Sequence按鈕。這將回放圖片，並創建一個簡單的定格電影。



相機的視圖查找器部分是在VideoOutput中使用一個簡單的Camera元素作為資源。這將給用戶顯示一個來自相機的流媒體視頻。

```
    VideoOutput {
        anchors.fill: parent
        source: camera
    }

    Camera {
        id: camera
    }
```

使用一個水平放置的ListView顯示來自ListModel的圖片，這個部件叫做imagePaths。在背景中使用一個半透明的Rectangle。

```
    ListModel {
        id: imagePaths
    }

    ListView {
        id: listView
```

```
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        anchors.bottomMargin: 10

        height: 100

        orientation: ListView.Horizontal
        spacing: 10

        model: imagePaths

        delegate: Image { source: path; fillMode: Image.PreserveAspectFit; height: 100; }

        Rectangle {
            anchors.fill: parent
            anchors.topMargin: -10

            color: "black"
            opacity: 0.5
        }
    }
```

為了拍攝圖像，你需要知道Camera元素包含了一組子對象用來完成各種工作。使用Camera.imageCapture用來捕捉圖像。當你調用capture方法時，一張圖片就被拍攝下來了。Camera.imageCapture的結果將會發送imageCaptured信號，接著發送imageSaved信號。

```
        Button {
            id: shotButton

            width: 200
            height: 75

            text: "Take Photo"
            onClicked: {
                camera.imageCapture.capture();
            }
        }
```

為了攔截子元素的信號，需要一個Connections元素。在這個例子中，我們不需要顯示預覽圖片，僅僅只是將結果圖片加入底部的ListView中。就如下面的例子展示的一樣，圖片保存的路徑由信號的path參數提供。

```
    Connections {
        target: camera.imageCapture

        onImageSaved: {
            imagePaths.append({"path": path})
            listView.positionViewAtEnd();
        }
    }
```

為了顯示預覽，連接imageCaptured信號，並且使用preview信號參數作為Image元素的source。requestId信號參數與imageCaptured和imageSaved一起發送。這個值由capture方法返回。這樣，就可以完整的跟蹤拍攝的圖片了。預覽的圖片首先被使用，然後替換為保存的圖片。然而在這個例子中我們不需要這樣做。

最後是自動回放的部分。使用Timer元素來驅動它，並且加上一些JavaScript。_imageIndex變量被用來跟蹤當前顯示的圖片。當最後一張圖片被顯示時，回放停止。在例子中，當播放序列時，root.state被用來隱藏用戶界面。

```
property int _imageIndex: -1

function startPlayback()
{
    root.state = "playing";
    setImageIndex(0);
    playTimer.start();
}

function setImageIndex(i)
{
    _imageIndex = i;

    if (_imageIndex >= 0 && _imageIndex < imagePaths.count)
        image.source = imagePaths.get(_imageIndex).path;
    else
        image.source = "";
}

Timer {
    id: playTimer

    interval: 200
    repeat: false

    onTriggered: {
        if (_imageIndex + 1 < imagePaths.count)
        {
            setImageIndex(_imageIndex + 1);
            playTimer.start();
        }
        else
        {
            setImageIndex(-1);
            root.state = "";
        }
    }
}
```

高級用法（Advanced Techniques）

10.5.1 實現一個播放列表（Implementing a Playlist）

Qt 5 multimedia接口沒有提供播放列表。幸好，它非常容易實現。通過設置模型子項與MediaPlayer元素可以實現它，如下所示。當playstate通過player控制時，Playlist元素負責設置MediaPlayer的source。

```
    Playlist {
        id: playlist

        mediaPlayer: player

        items: ListModel {
            ListElement { source: "trailer_400p.ogg" }
            ListElement { source: "trailer_400p.ogg" }
            ListElement { source: "trailer_400p.ogg" }
        }
    }

    MediaPlayer {
        id: player
    }
```

Playlist元素的第一部分如下，注意使用setIndex函數來設置source元素的索引值。我們也實現了next與previous函數來操作鏈表。

```
Item {
    id: root

    property int index: 0
    property MediaPlayer mediaPlayer
    property ListModel items: ListModel {}

    function setIndex(i)
    {
        console.log("setting index to: " + i);

        index = i;

        if (index < 0 || index >= items.count)
        {
            index = -1;
            mediaPlayer.source = "";
        }
        else
            mediaPlayer.source = items.get(index).source;
    }

    function next()
    {
        setIndex(index + 1);
    }

    function previous()
    {
```

```
        setIndex(index + 1);
    }
```

讓播放列表自動播放下一個元素的訣竅是使用MediaPlayer的status屬性。當得到MediaPlayer.EndOfMedia狀態時，索引值增加，恢復播放，或者當列表達到最後時，停止播放。

```
    Connections {
        target: root.mediaPlayer

        onStopped: {
            if (root.mediaPlayer.status == MediaPlayer.EndOfMedia)
            {
                root.next();
                if (root.index == -1)
                    root.mediaPlayer.stop();
                else
                    root.mediaPlayer.play();
            }
        }
    }
```

總結（**Summary**）

Qt的媒體應用程序接口提供了播放和捕捉視頻和音頻的機制。通過VideoOutput元素，視頻源能夠在我們的用戶界面上顯示。通過MediaPlayer元素，可以操作大多數的播放，SoundEffect被用于低延遲的聲音。Camera元素被用來截圖或者顯示一個實時的視頻流。

Networking

Qt5在C++中有豐富的網絡相關的類。例如在http協議層上使用請求回答方式的高級封裝類如QNetworkRequest，QNetworkReply，QNetworkAccessManageer。也有在TCP/IP或者UDP協議層封裝的低級類如QTcpSocket，QTcpServer和QUdpSocket。還有一些額外的類用來管理代理，網絡緩衝和系統網絡配置。

這章將不再闡述關于C++網絡方面的知識，這章是關于QtQuick與網絡的知識。我們應該怎樣連接QML/JS用戶界面與網絡服務，或者如何通過網絡服務來為我們用戶界面提供服務。已經有很好的教材和示例覆蓋了關于Qt/C++的網絡編程。然後你只需要閱讀這章相關的C++集成來滿足你的QtQuick就可以了。

通過HTTP服務UI（Serving UI via HTTP）

通過HTTP加載一個簡單的用戶界面，我們需要一個web服務器，它為UI文件服務。但是首先我們需要有用戶界面，我們在項目裡創建一個創建了紅色矩形框的main.qml。

```
// main.qml
import QtQuick 2.0

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'
}
```

我們加載一段python腳本來提供這個文件：

```
$ cd <PROJECT>
# python -m SimpleHTTPServer 8080
```

現在我們可以通過http://localhost:8000/main.qml來訪問，你可以像下面這樣測試：

```
$ curl http://localhost:8000/main.qml
```

或者你可以用瀏覽器來訪問。瀏覽器無法識別QML，並且無法通過文檔來渲染。我們需要創建一個可以瀏覽QML文檔的瀏覽器。為了渲染文檔，我們需要指出qmlscene的位置。不幸的是qmlscene只能讀取本地文件。我們為了突破這個限制，我們可以使用自己寫的qmlscene或者使用QML動態加載。我們選擇動態加載的方式。我們選擇一個加載元素來加載遠程的文檔。

```
// remote.qml
import QtQuick 2.0

Loader {
    id: root
    source: 'http://localhost:8080/main2.qml'
    onLoaded: {
        root.width = item.width
        root.height = item.height
    }
}
```

我們現在可以使用qmlscene來加載remote.qml文檔。這裡仍然有一個小問題。加載器將會調整加載項的大小。我們的qmlscene需要適配大小。可以使用--resize-to-root選項來運行qmlscene。

```
$ qmlscene --resize-to-root remote.qml
```

按照root元素調整大小，告訴qmlscene按照root元素的大小調它的窗口大小。remote現在從本地服務器加載main.qml，並且可以自動調整加載的用戶界面。方便且簡單。

注意

如果你不想使用一個本地服務器，你可以使用來自**GitHub**的**gist**服務。**Gist**是一個在線剪切板服務，就像**PasteBin**等等。可以在**https://gist.github.com**下使用。我創建了一個簡單的**gist**例子，地址是**https://gist.github.com/jryannel/7983492**。這將會返回一個綠色矩形框。由于**gist**連接提供的是**HTML**代碼，我們需要連接一個**/raw**來讀取原始文件而不是**HTML**代碼。

```
// remote.qml
import QtQuick 2.0

Loader {
    id: root
    source: 'https://gist.github.com/jryannel/7983492/raw'
    onLoaded: {
        root.width = item.width
        root.height = item.height
    }
}
```

從網絡加載另一個文件，你只需要引用組件名。例如一個Button.qml，只要它們在同一個遠程文件夾下就能夠像正常一樣訪問。

11.1.1 網絡組件（Networked Components）

我們做了一個小實驗。我們在遠程端添加一個按鈕作為可以復用的組件。

```
- src/main.qml
- src/Button.qml
```

我們修改main.qml來使用button：

```
import QtQuick 2.0

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'

    Button {
        anchors.centerIn: parent
        text: 'Click Me'
        onClicked: Qt.quit()
    }
}
```

再次加載我們的web服務器：

```
$ cd src
# python -m SimpleHTTPServer 8080
```

再次使用http加載遠mainQML文件：

```
$ qmlscene --resize-to-root remote.qml
```

我們看到一個錯誤：

```
http://localhost:8080/main2.qml:11:5: Button is not a type
```

所以，在遠程加載時，QML無法解決Button組件的問題。如果代碼使用本地加載qmlscene src/main.qml，將不會有問題。Qt能夠直接解析本地文件，並且檢測哪些組件可用，但是使用http的遠程訪問沒有“list-dir”函數。我們可以在main.qml中使用import聲明來強制QML加載元素：

```
import "http://localhost:8080" as Remote

...

Remote.Button { ... }
```

再次運行qmlscene後，它將正常工作：

```
$ qmlscene --resize-to-root remote.qml
```

這是完整的代碼：

```
// main2.qml
import QtQuick 2.0
import "http://localhost:8080" 1.0 as Remote

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'

    Remote.Button {
        anchors.centerIn: parent
        text: 'Click Me'
        onClicked: Qt.quit()
    }
}
```

一個更好的選擇是在服務器端使用qmldir文件來控制輸出：

```
// qmldir
Button 1.0 Button.qml
```

然後更新main.qml：

```
import "http://localhost:8080" 1.0 as Remote

...
```

```
  Remote.Button { ... }
```

當從本地文件系統使用組件時，它們的創建沒有延遲。當組件通過網絡加載時，它們的創建是異步的。創建時間的影響是未知的，當其它組件已經完成時，一個組件可能還沒有完成加載。當通過網絡加載組件時，需要考慮這些。

模板（**Templating**）

當使用HTML項目時，通常需要使用模板驅動開發。服務器使用模板機制生成代碼在服務器端對一個HTML根進行擴展。例如一個照片列表的列表頭將使用HTML編碼，動態圖片鏈表將會使用模板機制動態生成。通常這也可以使用QML解決，但是仍然有一些問題。

首先，HTML開發者這樣做的原因是為了克服HTML後端的限制。在HTML中沒有組件模型，動態機制方面不得不使用這些機制或者在客戶端邊使用javascript編程。很多的JS框架產生（jQuery，dojo，backbone，angular，...）可以用來解決這個問題，把更多的邏輯問題放在使用網絡服務連接的客戶端瀏覽器。客戶端使用一個web服務的接口（例如JSON服務，或者XML數據服務）與服務器通信。這也適用于QML。

第二個問題是來自QML的組件緩衝。當QML訪問一個組件時，緩衝渲染樹（render-tree），並且只加載緩衝版本來渲染。磁盤上的修改版本或者遠程的修改在沒有重新啟動客戶端時不會被檢測到。為了克服這個問題，我們需要跟蹤。我們使用URL後綴來加載鏈接（例如 http://localhost:8080/main.qml#1234），“#1234”就是後綴標識。HTTP服務器總是為相同的文檔服務，但是QML將使用完整的鏈接來保存這個文檔，包括鏈接標識。每次我們訪問的這個鏈接的標識獲得改變，QML緩衝無法獲得這個信息。這個後綴標識可以是當前時間的毫秒或者一個隨機數。

```
Loader {
    source: 'http://localhost:8080/main.qml#' + new Date().getTime()
}
```

總之，模板可以實現，但是不推薦，無法完整發揮QML的長處。一個更好的方法是使用web服務提供JSON或者XML數據服務。

HTTP請求（HTTP Requests）

從c++方面來看，Qt中完成http請求通常是使用QNetworkRequest和QNetworkReply，然後使用Qt/C++將響應推送到集成的QML。所以我們嘗試使用QtQuick的工具給我們的網絡信息尾部封裝了小段信息，然後推送這些信息。為此我們使用一個幫助對象來構造http請求，和循環響應。它使用java腳本的XMLHttpRequest對象的格式。

XMLHttpRequest對象允許用戶注冊一個響應操作函數和一個鏈接。一個請求能夠使用http動作來發送（如get，post，put，delete，等等）。當響應到達時，會調用注冊的操作函數。操作函數會被調用多次。每次調用請求的狀態都已經改變（例如信息頭部已接收，或者響應完成）。

下面是一個簡短的例子：

```
function request() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED) {
            print('HEADERS_RECEIVED');
        } else if(xhr.readyState === XMLHttpRequest.DONE) {
            print('DONE');
        }
    }
    xhr.open("GET", "http://example.com");
    xhr.send();
}
```

從一個響應中你可以獲取XML格式的數據或者是原始文本。可以遍歷XML結果但是通常使用原始文本來匹配JSON格式響應。使用JSON.parse(text）可以JSON文檔將轉換為JS對象使用。

```
...
} else if(xhr.readyState === XMLHttpRequest.DONE) {
    var object = JSON.parse(xhr.responseText.toString());
    print(JSON.stringify(object, null, 2));
}
```

在響應操作中，我們訪問原始響應文本並且將它轉換為一個javascript對象。JSON對象是一個可以使用的JS對象（在javascript中，一個對象可以是對象或者一個數組）。

注意

toString()轉換似乎讓代碼更加穩定。在不使用顯式的轉換下我有幾次都解析錯誤。不確定是什麼問題引起的。

11.3.1 Flickr調用（Flickr Call）

讓我們看看更加真實的例子。一個典型的例子是使用網絡相冊服務來取得公共訂閱中新上傳的圖片。我們可以使用http://api.flicker.com/services/feeds/photos_public.gne鏈接。不幸的是它默認返回XML流格式的數據，在qml中可以很方便的使用XmlListModel來解析。為了達到只關注JSON數據的目的，我們需要在請求中附加一些參數可以得到JSON響應：http://api.flickr.com/services/feeds/photo_public.gne?format=json&nojsoncallback=1。這將會返回一個沒有JSON回調的JSON響應。

注意 一個**JSON**回調將**JSON**響應包裝在一個函數調用中。這是一個**HTML**編程中的快捷方式，使用腳本標記來創建一個**JSON**請求。響應將觸發本地定義的回調函數。在**QML**中沒有**JSON**回調的工作機制。

使用curl來查看響應：

```
curl "http://api.flickr.com/services/feeds/photos_public.gne?format=json&nojsoncallback=1
```

響應如下：

```
{
    "title": "Recent Uploads tagged munich",
    ...
    "items": [
        {
        "title": "Candle lit dinner in Munich",
        "media": {"m":"http://farm8.staticflickr.com/7313/11444882743_2f5f87169f_m.jpg"},
        ...
        },{
        "title": "Munich after sunset: a train full of \"must haves\" =",
        "media": {"m":"http://farm8.staticflickr.com/7394/11443414206_a462c80e83_m.jpg"},
        ...
        }
    ]
    ...
}
```

JSON文檔已經定義了結構體。一個對象包含一個標題和子項的屬性。標題是一個字符串，子項是一組對象。當轉換文本為一個JSON文檔後，你可以單獨訪問這些條目，它們都是可用的JS對象或者結構體數組。

```
// JS code
obj = JSON.parse(response);
print(obj.title) // => "Recent Uploads tagged munich"
for(var i=0; i<obj.items.length; i++) {
    // iterate of the items array entries
    print(obj.items[i].title) // title of picture
    print(obj.items[i].media.m) // url of thumbnail
}
```

我們可以使用obj.items數組將JS數組作為鏈表視圖的模型，試著完成這個操作。首先我們需要取得響應並且將它轉換為可用的JS對象。然後設置response.items屬性作為鏈表視圖的模型。

```
function request() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if(...) {
            ...
        } else if(xhr.readyState === XMLHttpRequest.DONE) {
            var response = JSON.parse(xhr.responseText.toString());
            // set JS object as model for listview
            view.model = response.items;
        }
```

```
    }
    xhr.open("GET", "http://api.flickr.com/services/feeds/photos_public.gne?format=json&n
    xhr.send();
}
```

下面是完整的源代碼，當組件加載完成後，我們創建請求。然後使用請求的響應作為我們鏈表視圖的模型。

```
import QtQuick 2.0

Rectangle {
    width: 320
    height: 480
    ListView {
        id: view
        anchors.fill: parent
        delegate: Thumbnail {
            width: view.width
            text: modelData.title
            iconSource: modelData.media.m
        }
    }

    function request() {
        var xhr = new XMLHttpRequest();
        xhr.onreadystatechange = function() {
            if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED) {
                print('HEADERS_RECEIVED')
            } else if(xhr.readyState === XMLHttpRequest.DONE) {
                print('DONE')
                var json = JSON.parse(xhr.responseText.toString())
                view.model = json.items
            }
        }
        xhr.open("GET", "http://api.flickr.com/services/feeds/photos_public.gne?format=js
        xhr.send();
    }

    Component.onCompleted: {
        request()
    }
}
```

當文檔完整加載後（Component.onCompleted）,我們從Flickr請求最新的訂閱內容。我們解析JSON的響應並且設置item數組作為我們視圖的模型。鏈表視圖有一個代理可以在一行中顯示圖標縮略圖和標題文本。

另一種方法是添加一個ListModel，並且將每個子項添加到鏈表模型中。為了支持更大的模型，需要支持分頁和懶加載。

本地文件（**Local files**）

使用XMLHttpRequest也可以加載本地文件（XML/JSON）。例如加載一個本地名為“colors.json”的文件可以這樣使用：

```
xhr.open("GET", "colors.json");
```

我們使用它讀取一個顏色表並且使用表格來顯示。從QtQuick這邊無法修改文件。為了將源數據存儲回去，我們需要一個基于HTTP服務器的REST服務支持或者一個用來訪問文件的QtQuick擴展。

```
import QtQuick 2.0

Rectangle {
    width: 360
    height: 360
    color: '#000'

    GridView {
        id: view
        anchors.fill: parent
        cellWidth: width/4
        cellHeight: cellWidth
        delegate: Rectangle {
            width: view.cellWidth
            height: view.cellHeight
            color: modelData.value
        }
    }

    function request() {
        var xhr = new XMLHttpRequest();
        xhr.onreadystatechange = function() {
            if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED) {
                print('HEADERS_RECEIVED')
            } else if(xhr.readyState === XMLHttpRequest.DONE) {
                print('DONE');
                var obj = JSON.parse(xhr.responseText.toString());
                view.model = obj.colors
            }
        }
        xhr.open("GET", "colors.json");
        xhr.send();
    }

    Component.onCompleted: {
        request()
    }
}
```

也可以使用XmlListModel來替代XMLHttpRequest訪問本地文件。

```
import QtQuick.XmlListModel 2.0

XmlListModel {
```

```
    source: "http://localhost:8080/colors.xml"
    query: "/colors"
    XmlRole { name: 'color'; query: 'name/string()' }
    XmlRole { name: 'value'; query: 'value/string()' }
}
```

XmlListModel只能用來讀取XML文件，不能讀取JSON文件。

REST接口（REST API）

為了使用web服務，我們首先需要創建它。我們使用Flask（http://flask.pocoo.org），一個基于python創建簡單的顏色web服務的HTTP服務器應用。你也可以使用其它的web服務器，只要它接收和返回JSON數據。通過web服務來管理一組已經命名的顏色。在這個例子中，管理意味著CRUD（創建-讀取-更新-刪除）。

在Flask中一個簡單的web服務可以寫入一個文件。我們使用一個空的服務器.py文件開始，在這個文件中我們創建一些規則並且從額外的JSON文件中加載初始顏色。你可以查看Flask文檔獲取更多的幫助。

```
from flask import Flask, jsonify, request
import json

colors = json.load(file('colors.json', 'r'))

app = Flask(__name__)

# ... service calls go here

if __name__ == '__main__':
    app.run(debug = True)
```

當你運行這個腳本後，它會在http://localhost:5000。

我們開始添加我們的CRUD（創建，讀取，更新，刪除）到我們的web服務。

11.5.1 讀取請求（Read Request）

從web服務讀取數據，我們提供GET方法來讀取所有的顏色。

```
@app.route('/colors', methods = ['GET'])
def get_colors():
    return jsonify( { "colors" :  colors })
```

這將會返回‘/colors’下的顏色。我們使用curl來創建一個http請求測試。

```
curl -i -GET http://localhost:5000/colors
```

這將會返回給我們JSON數據的顏色鏈表。

11.5.2 讀取接口（Read Entry）

為了通過名字讀取顏色，我們提供更加詳細的後綴，定位在‘/colors/’下。名稱是後綴的參數，用來識別一個獨立的顏色。

```
@app.route('/colors/<name>', methods = ['GET'])
def get_color(name):
    for color in colors:
```

```
        if color["name"] == name:
            return jsonify( color )
```

我們再次使用curl測試，例如獲取一個紅色的接口。

```
curl -i -GET http://localhost:5000/colors/red
```

這將返回一個JSON數據的顏色。

11.5.3 創建接口（Create Entry）

目前我們僅僅使用了HTTP GET方法。為了在服務器端創建一個接口，我們使用POST方法，並且將新的顏色信息發使用POST數據發送。後綴與獲取所有顏色相同，但是我們需要使用一個POST請求。

```
@app.route('/colors', methods= ['POST'])
def create_color():
    color = {
        'name': request.json['name'],
        'value': request.json['value']
    }
    colors.append(color)
    return jsonify( color ), 201
```

curl非常靈活，允許我們使用JSON數據作為新的接口包含在POST請求中。

```
curl -i -H "Content-Type: application/json" -X POST -d '{"name":"gray1","value":"#333"}'
```

11.5.4 更新接口（Update Entry）

我們使用PUT HTTP方法來添加新的update接口。後綴與取得一個顏色接口相同。當顏色更新後，我們獲取更新後JSON數據的顏色。

```
@app.route('/colors/<name>', methods= ['PUT'])
def update_color(name):
    for color in colors:
        if color["name"] == name:
            color['value'] = request.json.get('value', color['value'])
            return jsonify( color )
```

在curl請求中，我們用JSON數據來定義更新值，後綴名用來識別哪個顏色需要更新。

```
curl -i -H "Content-Type: application/json" -X PUT -d '{"value":"#666"}' http://localhost
```

11.5.5 刪除接口（Delete Entry）

使用DELETE HTTP來完成刪除接口。使用與顏色相同的後綴，但是使用DELETE HTTP方法。

```
@app.route('/colors/<name>', methods=['DELETE'])
def delete_color(name):
    success = False
    for color in colors:
        if color["name"] == name:
            colors.remove(color)
            success = True
    return jsonify( { 'result' : success } )
```

這個請求看起來與GET請求一個顏色類似。

```
curl -i -X DELETE http://localhost:5000/colors/red
```

現在我們能夠讀取所有顏色，讀取指定顏色，創建新的顏色，更新顏色和刪除顏色。我們知道使用HTTP後綴來訪問我們的接口。

動作	HTTP協議	後綴
讀取所有	GET	http://localhost:5000/colors
創建接口	POST	http://localhost:5000/colors
讀取接口	GET	http://localhost:5000/colors/name
更新接口	PUT	http://localhost:5000/colors/name
刪除接口	DELETE	http://localhost:500/colors/name

REST服務已經完成，我們現在只需要關注QML和客戶端。為了創建一個簡單好用的接口，我們需要映射每個動作為一個獨立的HTTP請求，並且給我們的用戶提供一個簡單的接口。

11.5.6 REST客戶端（REST Client）

為了展示REST客戶端，我們寫了一個小的顏色表格。這個顏色表格顯示了通過HTTP請求從web服務取得的顏色。我們的用戶界面提供以下命令：

- 獲取顏色鏈表
- 創建顏色
- 讀取最後的顏色
- 更新最後的顏色
- 刪除最後的顏色

我們將我們的接口包裝在一個JS文件中，叫做colorservice.js，並將它導入到我們的UI中作為服務（Service）。在服務模塊中，我們創建了幫助函數來為我們構造HTTP請求：

```
// colorservice.js
function request(verb, endpoint, obj, cb) {
    print('request: ' + verb + ' ' + BASE + (endpoint?'/' + endpoint:''))
```

```
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        print('xhr: on ready state change: ' + xhr.readyState)
        if(xhr.readyState === XMLHttpRequest.DONE) {
            if(cb) {
                var res = JSON.parse(xhr.responseText.toString())
                cb(res);
            }
        }
    }
    xhr.open(verb, BASE + (endpoint?'/' + endpoint:''));
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.setRequestHeader('Accept', 'application/json');
    var data = obj?JSON.stringify(obj):''
    xhr.send(data)
}
```

包含四個參數。verb，定義了使用HTTP的動作（GET，POST，PUT，DELETE）。第二個參數是作為基礎地址的後綴（例如'http://localhost:5000/colors'）。第三個參數是可選對象，作為JSON數據發送給服務的數據。最後一個選項是定義當響應返回時的回調。回調接收一個響應數據的響應對象。

在我們發送請求前，我們需要明確我們發送和接收的JSON數據修改的請求頭。

```
// colorservice.js
function get_colors(cb) {
    // GET http://localhost:5000/colors
    request('GET', null, null, cb)
}

function create_color(entry, cb) {
    // POST http://localhost:5000/colors
    request('POST', null, entry, cb)
}

function get_color(name, cb) {
    // GET http://localhost:5000/colors/<name>
    request('GET', name, null, cb)
}

function update_color(name, entry, cb) {
    // PUT http://localhost:5000/colors/<name>
    request('PUT', name, entry, cb)
}

function delete_color(name, cb) {
    // DELETE http://localhost:5000/colors/<name>
    request('DELETE', name, null, cb)
}
```

這些代碼在服務實現中。在UI中我們使用服務來實現我們的命令。我們有一個存儲id的ListModel和存儲數據的gridModel為GridView提供數據。命令使用Button元素來發送。

讀取服務器顏色鏈表。

```
// rest.qml
import "colorservice.js" as Service
...
```

```
// read colors command
Button {
    text: 'Read Colors';
    onClicked: {
        Service.get_colors( function(resp) {
            print('handle get colors resp: ' + JSON.stringify(resp));
            gridModel.clear();
            var entries = resp.data;
            for(var i=0; i<entries.length; i++) {
                gridModel.append(entries[i]);
            }
        });
    }
}
```

在服務器上創建一個新的顏色。

```
// rest.qml
import "colorservice.js" as Service
...
// create new color command
Button {
    text: 'Create New';
    onClicked: {
        var index = gridModel.count-1
        var entry = {
            name: 'color-' + index,
            value: Qt.hsla(Math.random(), 0.5, 0.5, 1.0).toString()
        }
        Service.create_color(entry, function(resp) {
            print('handle create color resp: ' + JSON.stringify(resp))
            gridModel.append(resp)
        });
    }
}
```

基于名稱讀取一個顏色。

```
// rest.qml
import "colorservice.js" as Service
...
// read last color command
Button {
    text: 'Read Last Color';
    onClicked: {
        var index = gridModel.count-1
        var name = gridModel.get(index).name
        Service.get_color(name, function(resp) {
            print('handle get color resp:' + JSON.stringify(resp))
            message.text = resp.value
        });
    }
}
```

基于顏色名稱更新服務器上的一個顏色。

```
// rest.qml
import "colorservice.js" as Service
...
// update color command
Button {
    text: 'Update Last Color'
    onClicked: {
        var index = gridModel.count-1
        var name = gridModel.get(index).name
        var entry = {
            value: Qt.hsla(Math.random(), 0.5, 0.5, 1.0).toString()
        }
        Service.update_color(name, entry, function(resp) {
            print('handle update color resp: ' + JSON.stringify(resp))
            var index = gridModel.count-1
            gridModel.setProperty(index, 'value', resp.value)
        });
    }
}
```

基于顏色名稱刪除一個顏色。

```
// rest.qml
import "colorservice.js" as Service
...
// delete color command
Button {
    text: 'Delete Last Color'
    onClicked: {
        var index = gridModel.count-1
        var name = gridModel.get(index).name
        Service.delete_color(name)
        gridModel.remove(index, 1)
    }
}
```

在CRUD（創建，讀取，更新，刪除）操作使用REST接口。也可以使用其它的方法來創建web服務接口。可以基于模塊，每個模塊都有自己的後綴。可以使用JSON RPC（http://www.jsonrpc.org/）來定義接口。當然基于XML的接口也可以使用，但是JSON在作為JavaScript部分解析進QML/JS中更有優勢。

使用開放授權登陸驗證（**Authentication using OAuth**）

OAuth是一個開放協議，允許簡單的安全驗證，是來自web的典型方法，用于移動和桌面應用程序。使用OAuth對通常的web服務的客戶端進行身份驗證，例如Google，Facebook和Twitter。

注意

對于自定義的**web**服務，你也可以使用典型的**HTTP**身份驗證，例如使用**XMLHttpRequest**的用戶名和密碼的獲取方法（比如**xhr.open(verb,url,true,username,password**））。

Auth目前不是QML/JS的接口，你需要寫一些C++代碼並且將身份驗證導入到QML/JS中。另一個問題是安全的存儲訪問密碼。

下面這些是我找到的有用的連接：

- http://oauth.net
- http://hueniverse.com/oauth/
- https://github.com/pipacs/o2
- http://www.johanpaul.com/blog/2011/05/oauth2-explained-with-qt-quick/

Engine IO

Engine IO是DIGIA運行的一個web服務。它允許Qt/QML應用程序訪問來自Engin.IO的NoSQL存儲。這是一個基于雲存儲對象的Qt/QML接口和一個管理平台。如果你想存儲一個QML應用程序的數據到雲存儲中，它可以提供非常方便的QML/JS的接口。

查看EnginIO的文檔獲得更多的幫助。

Web Sockets

webSockets不是Qt提供的。將WebSockets加入到Qt/QML中需要花費一些工作。從作者的角度來看WebSockets有巨大的潛力來添加HTTP服務缺少的功能-通知。HTTP給了我們get和post的功能，但是post還不是一個通知。目前客戶端輪詢服務器來獲得應用程序的服務，服務器也需要能通知客戶端變化和事件。你可以與QML接口比較：屬性，函數，信號。也可以叫做獲取/設置/調用和通知。

QML WebSocket插件將會在Qt5中加入。你可以試試來自qt playground的web sockets插件。為了測試，我們使用一個現有的web socket服務實現了echo server。

首先確保你使用的Qt5.2.x。

```
$ qmake --version
... Using Qt version 5.2.0 ...
```

然後你需要克隆web socket的代碼庫，並且編譯它。

```
$ git clone git@gitorious.org:qtplayground/websockets.git
$ cd websockets
$ qmake
$ make
$ make install
```

現在你可以在qml模塊中使用web socket。

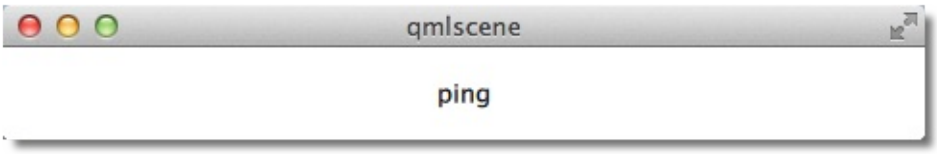
```
import Qt.WebSockets 1.0

WebSocket {
    id: socket
}
```

測試你的web socket，我們使用來自http://websocket.org的echo server 。

```
import QtQuick 2.0
import Qt.WebSockets 1.0

Text {
    width: 480
    height: 48

    horizontalAlignment: Text.AlignHCenter
    verticalAlignment: Text.AlignVCenter

    WebSocket {
        id: socket
        url: "ws://echo.websocket.org"
        active: true
        onTextMessageReceived: {
            text = message
        }
        onStatusChanged: {
```

```
            if (socket.status == WebSocket.Error) {
                console.log("Error: " + socket.errorString)
            } else if (socket.status == WebSocket.Open) {
                socket.sendTextMessage("ping")
            } else if (socket.status == WebSocket.Closed) {
                text += "\nSocket closed"
            }
        }
    }
}
```

你可以看到我們使用socket.sendTextMessage("ping")作為響應在文本區域中。



11.8.1 WS Server

你可以使用Qt WebSocket的C++部分來創建你自己的WS Server或者使用一個不同的WS實現。它非常有趣，是因為它允許連接使用大量擴展的web應用程序服務的高質量渲染的QML。在這個例子中，我們將使用基于web socket的ws模塊的Node JS。你首先需要安裝node.js。然後創建一個ws_server文件夾，使用node package manager（npm）安裝ws包。

```
$ cd ws_server
$ npm install ws
```

npm工具下載並安裝了ws包到你的本地依賴文件夾中。

一個server.js文件是我們服務器的實現。服務器代碼將在端口3000創建一個web socket服務並監聽連接。在一個連接加入後，它將會發送一個歡迎並等待客戶端信息。每個客戶端發送到socket信息都會發送回客戶端。

```
var WebSocketServer = require('ws').Server;

var server = new WebSocketServer({ port : 3000 });

server.on('connection', function(socket) {
    console.log('client connected');
    socket.on('message', function(msg) {
        console.log('Message: %s', msg);
        socket.send(msg);
    });
    socket.send('Welcome to Awesome Chat');
});

console.log('listening on port ' + server.options.port);
```

你需要獲取使用的JavaScript標記和回調函數。

11.8.2 WS Client

在客戶端我們需要一個鏈表視圖來顯示信息，和一個文本輸入來輸入新的聊天信息。

在例子中我們使用一個白色的標簽。

```
// Label.qml
import QtQuick 2.0

Text {
    color: '#fff'
    horizontalAlignment: Text.AlignLeft
    verticalAlignment: Text.AlignVCenter
}
```

我們的聊天視圖是一個鏈表視圖，文本被加入到鏈表模型中。每個條目顯示使用行前綴和信息標簽。我們使用單元將它分為24列。

```
// ChatView.qml
import QtQuick 2.0

ListView {
    id: root
    width: 100
    height: 62

    model: ListModel {}

    function append(prefix, message) {
        model.append({prefix: prefix, message: message})
    }

    delegate: Row {
        width: root.width
        height: 18
        property real cw: width/24
        Label {
            width: cw*1
            height: parent.height
            text: model.prefix
        }
        Label {
            width: cw*23
            height: parent.height
            text: model.message
        }
    }
}
```

聊天輸入框是一個簡單的使用顏色包裹邊界的文本輸入。

```
// ChatInput.qml
import QtQuick 2.0

FocusScope {
    id: root
```

```
    width: 240
    height: 32
    Rectangle {
        anchors.fill: parent
        color: '#000'
        border.color: '#fff'
        border.width: 2
    }

    property alias text: input.text

    signal accepted(string text)

    TextInput {
        id: input
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.verticalCenter: parent.verticalCenter
        anchors.leftMargin: 4
        anchors.rightMargin: 4
        onAccepted: root.accepted(text)
        color: '#fff'
        focus: true
    }
}
```

當web socket返回一個信息後，它將會把信息添加到聊天視圖中。這也同樣適用于狀態改變。也可以當用戶輸入一個聊天信息，將聊天信息拷貝添加到客戶端的聊天視圖中，並將信息發送給服務器。

```
// ws_client.qml
import QtQuick 2.0
import Qt.WebSockets 1.0

Rectangle {
    width: 360
    height: 360
    color: '#000'

    ChatView {
        id: box
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.top: parent.top
        anchors.bottom: input.top
    }
    ChatInput {
        id: input
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        focus: true
        onAccepted: {
            print('send message: ' + text)
            socket.sendTextMessage(text)
            box.append('>', text)
            text = ''
        }
    }
    WebSocket {
        id: socket
```

```
        url: "ws://localhost:3000"
        active: true
        onTextMessageReceived: {
            box.append('<', message)
        }
        onStatusChanged: {
            if (socket.status == WebSocket.Error) {
                box.append('#', 'socket error ' + socket.errorString)
            } else if (socket.status == WebSocket.Open) {
                box.append('#', 'socket open')
            } else if (socket.status == WebSocket.Closed) {
                box.append('#', 'socket closed')
            }
        }
    }
}
```

你首先需要運行服務器，然後是客戶端。在我們簡單例子中沒有客戶端重連的機制。

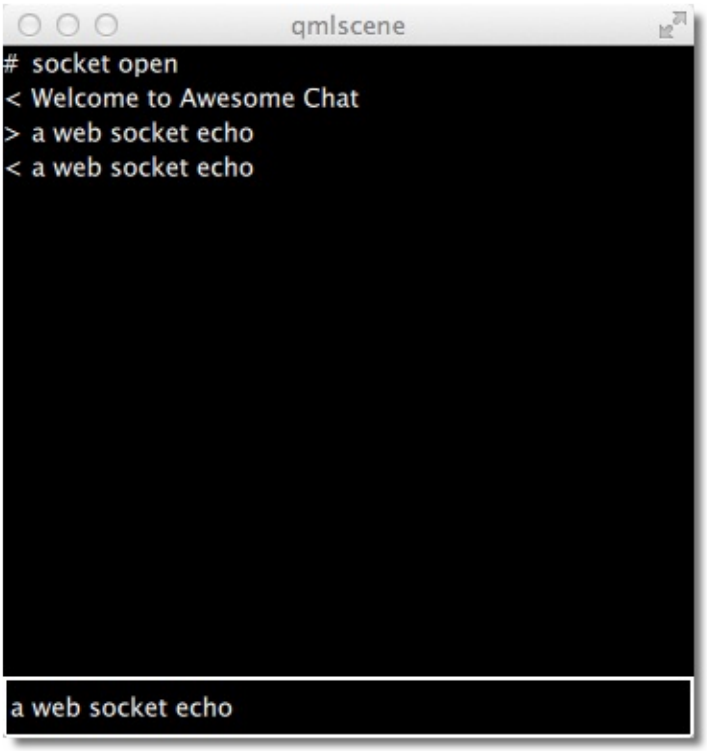
運行服務器

```
$ cd ws_server
$ node server.js
```

運行客戶端

```
$ cd ws_client
$ qmlscene ws_client.qml
```

當輸入文本並點擊發送後，你可以看到類似下面這樣。



總結（**Summary**）

這章我們討論了關于QML的網絡應用。請記住Qt已在本地端提供了豐富的網絡接口可以在QML中使用。但是這一章的我們是想推動QML的網絡運用和如何與雲服務集成。