

/* Translated by sole

Chapter3——3.4.1.3 (Palm webOS, 1st Edition Page 49——68)

高频词汇翻译: Widget:控件 scene:场景

/*

第三章 控件

Mojo 构架的核心是图形界面的工具集，它包括一系列的动态控件。Mojo 控件由 sceneController 函数和自定义的 CSS 风格配置而成，并由 Mojo events 进行管理（Mojo event 已在第一章中做简要介绍）。

通过控件，您可以构建静态或者动态列表，或者使用各种按钮控件、选择器、以及文本框；您可以选择不同种类的菜单和对话框，利用各种尖端的采集器和查看器专门来处理不同类型的数据。用于声明、初始化和管控控件的公共模块可以让您更加轻松地了解您的程序，并为之编写代码。

控件是由 HTML 中一个空的 div 代码声明的，它的其中一项属性是 x-mojo-element，声明了待显示的控件的类型。通常，您需要在场景的视图中定义这个控件，进行配置，然后使用相应的场景辅助程序的安装函数进行安装。您需要注意与控件相关的事件的发生，它可以由用户使用控件引起，也可以由相关控件的数据更新引起。Mojo 构架将默认风格应用到控件上，您可以用自己的 CSS 覆盖这些默认风格，但是很多情况下，默认风格是最适合的。

在这个章节中，我们将首先介绍下 Mojo 控件的总体设计，然后才会涉及一些基础控件，如按钮和选择器、列表和文本框。在新的程序中，我们会大量地使用这些控件，为了介绍如何在程序中应用这些基础控件，每个种类至少会使用一个基础控件进行转化。

为了完成这篇的教程，我们在多处引用了 webOS Developer Guide 上的资料。在教程上下文引入这些额外的信息会很有帮助。不过，为了获取接口和调用函数的详细信息，您依然应该参阅软件开发工具包中的应用程序接口文档。

3.1 关于控件

控件是能够被集成到任何程序中的用户界面的控件，它们能够为程序所定制，同时可以提供可循环使用的、风格一致的用户界面的功能。在网络开发中，控件一直为人们广泛使用着，然而，Mojo 控件和其他控件却不同。Mojo 控件被定义为具有特定的操作，同时，还附带许多选项；这些控件生成复合的 HTML 代码，并且可以通过 CSS 轻松地改变风格。

它有助于人们了解控件生成的 HTML 代码，尤其是针对列表和对话框，因为程序会在列表和对话框中注明 HTML 模版，而这很大程度上定义了控件的结构。

3.1.1 控件的声明

控件是由 HTML 中一个空的 div 代码声明的，如下所示：

```
<div id="my-toggle" x-mojo-element="ToggleButton"></div>
```

div 中的 x-mojo-element 属性表明了控件的种类，当 HTML 被加入到页面中的时候，x-mojo-element 属性被填充。这时候，以下任意一种情况都可能发生：

1. 场景被触摸，场景的视图 HTML 包括控件。
2. 控件由 HTML 模板指定，而该模版由另一个控件调用。

关于第二种情况，举个例子，如果你的场景包括列表控件，而这个列表控件的项目中又包含了其他控件，这种情况下，只要有新的项目加入，一系列的列表项目的控件都会初始化一次。

3.1.2 控件的创建

在把控件嵌入场景前，必须先创建一个控件。这必须使用场景辅助程序的安装函数，也就是 SceneController.setupWidget() 函数。调用这个函数，必须提供三项参数，如图3-1

表3-1: setupWidget 函数参数

参数列表	描述
控件 ID	在控件的声明中 div 代码元素中的 id 或者名字
属性	包括控件的静态属性，通常是选项或者控件的属性
模型	包括控件的动态属性，通常是控件相关的数据，不过很少有动态属性

例如，我们是这样创建一个开关按钮的：

```
this.toggle = { trueValue:'on', trueLabel:'On', falseValue:'off', falseLabel:'Off'
this.toggleModel = { value:'on', disabled:false };
this.controller.setupWidget('my-toggle', this.toggle, this.toggleModel);
```

my-toggle 参数表明控件被创建。这个参数可以是控件的 div 代码元素的 id，也可以是名字。通常，使用 id 是没问题的，不过，如果控件不止要初始化一次，这也不是唯一的方法。比如为列表项目声明的控件是在模板中的，或者场景被触摸过好几次却不弹出，这种情况下，请使用名字。

第二个参数表明了控件的属性。这些属性将影响控件的操作和显示，不过，和实际数据的显示和编辑没有什么关系。控件一旦初始化，其属性便无法再更改。开关按钮是一个简单的二进制选择器，值得一提的是，它的属性包括 trueValue, trueLabel, falseValue 和

`falseLabel`。这些值允许人们选择开/关、左/右、上/下、内/外等，而且这些标签能跟踪这些值，也能为用户提供不同的 `term`。

最后一个参数表明了控件的数据模型对象。这是控件显示的实际用户数据。模型对象的内容会经常变化，每次都要求控件进行更新。在我们例子中，模型包括开关的值和一个设置为 `false` 的 `disabled` 属性。

属性和模型对象的主要差异在于是否能够允许控件在列表内轻松使用。属性表现在，列表项目和模型共享的安装文件提供了每个项目的数据，这使得你在接下来的章节中，能够更加容易获取列表中的项目。

3.1.3 控件的数据模型的更新

当控件的模型控件外部改变的时候，控件不会自动更新并影响这些变化。应用程序（通常是场景辅助程序）会被要求调用控件场景控制器的 `modelChanged()` 函数，传递改变过的模型对象。接着，场景控制器会提醒所有使用该模型的控件，使得他们能正确地显示现在的模型数据。例如，在我们的例子中，如果你禁用开关按钮，

```
this.toggleModel.disable = true;
this.controller.modelChanged(this.toggleModel, this);
```

第一个参数对于 `modelChanged()` 函数是一个改变过的模型对象，模型会用模型对象改变提醒，以决定哪些控件正在使用这个模型对象，然后通知它们更新。注意，用一个全新的模型对象调用 `modelChange()` 函数不会更新这个模型，相反的，不会有任何变化，因为指定的模型不会被任何已经存在的控件使用，也不会生成或接受到任何提醒。

第二个参数识别是什么对象改变了模型。这保证了对象自身的模型改变时将被提醒。场景辅助程序（还有可适用的控件辅助程序）通常会简单地传递关键字 `this`，只要这个参数不是被控件控制器调用的，那么它就是可选的。

`ModelChange()` 函数是一个间接更新控件模型副本的方法，如果需要直接改变模型的话，你需要使用 `setWidgetModel()`。当 `setupWidget()` 函数应用给定的名字属性的时候，`setWidgetModel()` 函数只应用到一个控件实例。所以，你必须传递控件的 `id` 或者实际的控件 DOM 元素。

以下例子是紧跟 `Tlgggle Button` 案例的：

```
// Use a new model object in place of the old one:
this.newToggleModel = {value:'off'};
// Set the widget to use the new model:
this.controller.setWidgetModel("my-toggle", this.newToggleModel);
```

3.1.4 控件的事件处理

控件由时间支持，可能使用的常用事件有 `Mojo.Event.tap` 和 `Mojo.Event.propertyChange`，不过列表控件指定的事件一般不大可能是 `Mojo.Event.listDelete` 或者 `Mojo.Event.listReorder`，滚动控件指定的事件一般不大可能是 `Mojo.Event.scrollingStarted`

安装控件的时候，必须使用场景辅助程序的安装函数设置事件监听器，可以在声明控件的时候把监听器加入到 `div` 元素中。例如，当控件被开关的时候，开关按钮会发送 `Mojo.Event.propertyChange`，这就意味着开关按钮的模型改变了值。使用我们已经创建的开关按钮，你可以这样设置监听器的代码：

```
This.controller.listen("my-toggle", Mojo.Event.propertyChanged,
this.handleSelectorChange.bindAsEventListener(this));
```

请查看本章最后一节“事件”以获取更多信息，它涵盖了所有的事件模型，你也可以在 Palm 软件开发工具集中查阅 `Mojo.Event` 应用程序接口。

3.2 使用控件

现在，让我们开始进一步使用和讨论独立的控件吧！表3-2概括了 Mojo 1.0中的所有控件，但是请注意，以后新的控件会不断加入到这个平台中。你可以查看 Palm 开发者网站以获取最新的消息。

表3-2 Mojo 控件

集合	控件
按钮&选择器	按钮，复选框，单选按钮，开关，列表选择器，滚动值
列表	列表，过滤列表，网格表
对话框&容器	对话框，显示对话框，错误对话框，拖拽器，滚动器
文本框	文本区域，过滤区域，密码区域，富文本编辑
菜单	程序菜单，命令菜单，查看菜单，子菜单
采集器	日期，文件，集合，时间
查看器	图片查看，网页查看，影音对象
指示器	进度条，进度点，进度滚动，旋转器

我们会把各种控件加入到新的程序中，通过这些案例，你将学会如何在自己的程序中使用 Mojo 控件。但是，我们不会使用 Mojo 提供的所有控件，每类控件挑一个，其他都是相似的。

控件通过 JavaScript 辅助程序在 HTML 场景中声明、安装、转化，并且可以通过 CSS 改变风格。所有的控件都有属性对象，这些属性对象包括控件初始化时应用的静态属性，还

包括拥有控件相关动态值的模型对象。

通过本章和接下来的两章，你会知道如何使用具体的案例。他们都有自己唯一的性能，并且还有一些使用它们的小提示。这些都会有用的。

3.3 按钮和选择器

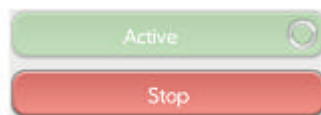
按钮和选择器是 Mojo 控件中最简单的，在前面的“关于控件”章节中，你已经知道如何使用开关按钮，所有其他的按钮和选择器都是按相同的方式工作的。这本节，我们会直接使用按钮控件，为新程序添加一个按钮，然后接触其他控件，比如：开关按钮，复选框，单选按钮，列表选择器和滚动值。

3.3.1 按钮

你可以使用简单的 HTML 按钮作为控件，大多数情况下都可以很好的工作。不过，按钮控件具有动态操作，特别是包括一个可以显示动作的旋转器，你可以通过活动的各个阶段管理按钮的标签。图3-1就是一个按钮控件的例子。

按钮是最基础的 UI 元素，可以在一个区域内操作。当一个按钮被按住，它可以改变状态，然后漂亮地恢复原先的状态，就想门铃一样。按钮可以作为对象，赋予它风格，也可以没有风格，它的标签通常是文本，也可以是图片。Mojo 按钮可以禁用，也可以被配置为用来显示活动的指示器。

图3-1 按钮控件例子



你可以使用 HTML 按钮初始化一个动作，如果你要结合指示器初始化一个动作，请使用按钮控件。正如我们在第二章讲到的转换场景，你可以用常规的 HTML 标记在视图中声明 HTML 按钮：

```
<button id="previousStory" class='palm-button'>Previous</button>
<button id="nextStory" class='palm-button'>Next</button>
```

把按钮的类赋给 .palm-button，按钮会像 Mojo 按钮控件一样显示。Mojo 构架会为类.palm-button 中的 HTML 按钮应用同样的风格。

使用自己的 CSS 覆盖任何类型的按钮（sole 注：Mojo 按钮控件或者 HTML 按钮），最典型的风格修改就是调整按钮的宽度，因为默认风格的按钮是设计为窗口居中的。当按钮作

为首选或次选的时候，便有了额外的风格选择，消除指示器，不管是可取的操作还是不可取的操作。

3.3.1.1

3.3.1.1.1 为新闻添加按钮

在第二章结尾，我们用 HTML 按钮和后退手势为新闻添加了场景。然后这在 UI 中是不可见的，我们现在要用按钮控件替换 HTML 按钮，这样你就知道如何添加一个简单的控件了。如果你已经很清楚如何操作，可以跳过这部分，到下一节中。

第一步很简单：打开 `storyView-scene.html`，用控件声明替换按钮 tag。

```
<div id='storyViewScene'>
  <div class="palm-page-header multi-line">
    <div id="test" class="palm-page-header-wrapper">
      <div id="storyViewTitle" class="title left">#{title}</div>
    </div>
  </div>
  <div id="storyViewSummary" class="itemFull">#{text}</div>
</div>
<div x-mojo-element="Button" id="previousStory"></div>
<div x-mojo-element="Button" id="nextStory"></div>
```

第二步：打开 `storyView-assistant.js`，在监听器的前面为轻击按钮添加控件安装。我们不会改变按钮的 id，所以也没必要改变监听器安装函数。

```
StoryViewAssistant.prototype.setup = function() {
  // Hide Previous Button if first story, and Next Button if last
  if (this.storyIndex > 0) {
    this.controller.setupWidget("previousStory",
      this.attributes = {
        disabledProperty: 'disabled'
      },
      this.model = {
        buttonLabel : "Previous",
        buttonClass: "",
        disabled: false
      });
    this.controller.listen('previousStory', Mojo.Event.tap,
      this.previousStory.bindAsEventListener(this));
  } else {
    $('previousStory').hide();
  }
}
```

```
if (this.storyIndex < this.storyFeed.stories.length-1)    {
    this.controller.setupWidget("nextStory",
        this.attributes = {
            disabledProperty: 'disabled'
        },
        this.model = {
            buttonLabel : "Next",
            buttonClass: "",
            disabled: false
        });
    this.controller.listen('nextStory', Mojo.Event.tap,
        this.nextStory.bindAsEventListener(this));
} else {
    $('nextStory').hide();
}
};
```

其他的代码不变。运行这个版本的程序，它的操作和上个版本是一样的，只不过我们现在用的是按钮控件，而不是 HTML 按钮。

3.3.2 选择器

这个简单的选择器将会被用在新闻程序的其他地方，待会儿的例子中会讲到。现在，接下来的小节将简单介绍各个选择器，以及如何在程序中使用它们。

3.3.2.1 复选框

复选框（图3-2）控件用来控制和指示元素中的一个二进制数值。

图3-2 复选框



轻击复选框会改变它的状态，根据之前的状态显示或取消一个对钩。Mojo 构架会处理这个改变的显示状态，也会处理控件日期模块。安装的时候你可以选择状态。

3.3.2.2 开关按钮

开关按钮是另一个显示和控制二进制数值的控件。和复选框相比，当你轻击开关按钮（图3-3）的时候，它可以在两个状态间交替改变。

图3-3 开关按钮

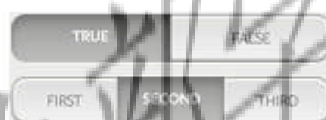


3.3.2.3 单选按钮

如果你需要从很多选项中选择一项，并显示选项的状态，你就需要用到单选按钮（图3-4）。Mojo 提供了一套经典单选按钮，按钮上是标签，横向排列，且一次只能选择一个选项。

选项可以有很多个，这取决于显示的宽度以及可以被轻易选择到的最小按钮的尺寸。在 webOS 设备上，你通常可以操作2-5个状态，但是，Mojo 构架不会在这方面做任何限制。

3-4 单选按钮



3.3.2.4 列表选择器

你也许会希望找到一个和列表控件一样工作的列表选择器，它允许你从众多选项中选出一个，显示在一个弹出式的列表中，而这个列表又不会限制数量。这列斯与子菜单控件的工作方式。图3-5就是一个列表选择器控件的例子。

这些选择项是被定义在一个数组中的，同时定义他们显示的标签和复合标准的值。如果选项是静态的，这意味着他们在场景中永远不会改变，那么你就需要在控件属性中把数组定义为一个属性。如果选项是变化的对象，那么就附加一个模型属性。

表格中的列表选择器

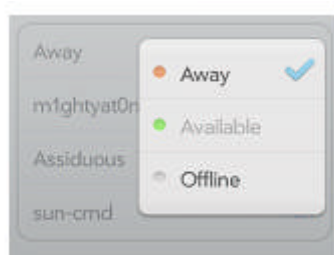
在表格中，需要组合列表选择器的话，请在 `div` 类 `.palm-list` 后面使用 `div` 类 `.palm-group unlabeled`，然后单独的选择器使用变量 `.palm-row classes` 容纳列表选择器控件。例如：

```
<div class="palm-group unlabeled">
  <div class="palm-list">
    <div class="palm-row first">
      <div id="trainSelector" x-mojo-element="ListSelector"></div>
    </div>
    <div class="palm-row">
      <div id="departureSelector" x-mojo-element="ListSelector"></div>
    </div>
    <div class="palm-row">
      <div id="destinationSelector" x-mojo-element="ListSelector"></div>
    </div>
    <div class="palm-row last">
      <div id="timeSelector" x-mojo-element="ListSelector"></div>
    </div>
  </div>
</div>
```

表格相关的小提示：你可以把多个属性不同的模型合并成一个对象，只需要为 `modelProperty` 指定一个合适的名字。用一个对象轻松处理表单。

列表选择器从风格上来看，像列表控件，为了获得 webOS 的风格，你需要为控件声明添加风格说明，你可能还需要用你自己的 CSS 来改变列表类的风格。你可以在下一节“列表”中找到更多的信息。

表3-5 列表选择器



3.3.2.5 滚动值

最后要介绍的控件是滚动之，它在水平方向显示了一个选择范围，控制把手可以被拖动到任何你想要的位置。你必须指定最小值和最大值，也可以指定中间值，不过这会出发额

外的操作。图3-6是一个滚动值的例子。

图3-6 滚动值



3.4 列表

为 Mojo 的设计是从列表开始的。为了验证 webOS 的结构和 Mojo 的概念,信奉 webOS 的设计师们敢于挑战设计列表控件,当用户拥有一台低端 CPU 设备,他们可以在极短的时间内轻击列表或少数数据,这个列表控件会从联系人数据库中拉出动态数据来。毋庸置疑,这是一个不大重要的挑战,却是设计过程中碰到的问题,而围绕设计师的设计,构架的其他部分已经基本上成型。

WebOS 的用户体验将在许多程序中广泛使用列表。考虑到构架的元素和导航模式,大多数的应用程序会想法设法合并列表控件。想最大程度地使用好 Mojo,你就要充分了解列表控件。

其他的列表控件,网格表和过滤列表,都源自列表控件。它们有许多共同的特性,却被设计为处理更专门的事情。我们的案例程序会用到列表和过滤列表,不过,你要在第五章“高级控件”中才能学到过滤列表。

3.4.1 列表控件

列表由使用 HTML 模版插入 DOM 的对象提供,包括列表容器和独立的列表行。列表高度没有限制,并包含单行文本、多行文本、图片和其他控件。列表可以是静态的,其列表项目有控件作为数组直接提供,也可以是动态,用来显示需要的列表项目。列表可以放在一处,由 Mojo 构架为程序删除、排列、添加项目。

列表的核心应用程序包括电子邮件收件箱,信息聊天视图,联系人列表,音乐库等等。可以看出来,列表灵活、快速并且高效。

3.4.1.1 回到新闻程序: 添加故事列表

我们打算在新闻程序中的一些地方使用列表控件。受限,我们要把例子中的新闻源转化为一个列表,然后把它连到 ajax 调用从而为程序获取实时的新闻源。这样,程序便能为我们提供一个基础的新闻阅读器。但是,我们需要添加另一个列表控件作为源,从而让程序处理多个源。这以节中,程序会开始形成初步的样子。

我们会创建一个列表来获取正在使用的例子列表,使用 palm-generate 工具为列表视

图创建一个新的场景，我们叫她 storyList:

```
palm-generate -t new_scene -p "name=storyList" .
```

在视图中, storyList-scene.html 在 palm-header 下声明了一个列表, 包括一个边距 tag, 用来分配列表的标题, div 中的 id='storyListWgt' 是列表控件的声明。

```
<div class='palm-header'>
  <div class='palm-header-center'><span id='feedTitle'></span>
</div>
</div>
<div class="palm-header-spacer"></div>
<div class="palm-list">
  <div id="storyListWgt" x-mojo-element="List"></div>
</div>
```

接下来, 创建一个列表模板。把两个 HTML 文件放入 views/storyList 目录, 调用容器模板 storyListTemplate.html 和行模版 storyRowTemplate.html。

所有的控件都用 HTML 模板布局并格式化列表容器和独立行。通常可以把这些模版作为单独的 HTML 文件放入场景视图文件夹, 不过你也可以指定每个模版的路径名, 这样可以在各场景之间共享模板, 也可以组织他们。路径名用相关记号如 ../scene-dir/template-file 标明, scene-dir 是目前场景视图文件的目录。在模板内, 你可以从列表中找到属性。

注意: Mojo 使用相对路径标明文件所在的地方, 而不是 index.html 所在的文件夹。

listTemplate 是可选的, 它为列表容器定义了 HTML 模板的路径, 如果列表容器不存在, 列表将把列表项目放入场景, 而不需要任何容器。如果 listTemplate 存在, 只能有一个顶级元素。

itemTemplate 是必选的, 它为列表项目设置了 HTML 模板的路径, 使用记号 #{property} 为插入模版的项目标明详细的 items 属性。

storyListTemplate 包括单行, 它使用类 .palm-list 为列表和模板条目 #{listElements} 规定格式:

```
<div class="palm-list">#{listElements}</div>
```

storyRowTemplate 有一点复杂, 它使用类 palm-row 在外 div 格式化行, 这样, 每个列表行都包含一个标题条目和一个文本条目:

```
<div class="palm-row" x-mojo-tap-highlight="momentary">
  <div id="storyTitle" class="listTitle truncating-text #{unreadStyle}">#{title}</div>
  <div id="storyText" class="listText truncating-text">#{text}</div>
```

</div>

每个条目都使用类 `truncating-text`，它使得条目在边界能够被自动裁掉，被裁掉的部分会使用省略号表示。`#{title}`和`#{text}`这两个模板指向模板中被代替了属性的项目的名字。

模板`#{unreadStyle}`指向另一个项目属性，它着重于尚未被阅读的 `story` 标题的特定风格。这个例子证明了模板能和 `div` 属性一起使用，就好像模板能和内容一起使用。未来的 CSS 会将一些风格应用于 `unreadStyle`。

源列表用特定风格将列表控件围住，就像图3-7和图3-8中所展示的那样。你应该再温习下软件开发工具集的用户界面和风格指导文件，以全面了解 Mojo 风格。但是，简要地概括的话，新闻源列表中使用了三种级别的风格。

`.palm-group` 用外框架封装所有的列表元素，它有一个可选类 `.palm-group-title`，用来定义组的标题。

`listTemplate` 用 `.palm-list` 驱动空隙，浅色分隔符规则用来分割列表条目。

`.palm-row` 围住包含列表条目模板的 `div tag`。

现在让我们回到例子，添加 `storyList-assistant.js` 来实现源列表：

```
//
// StoryListAssistant - Displays the feed's stories in a list,
// user taps display the
// selected story in the storyView scene.
//
// Arguments:
// selectedFeed          Feed to be displayed
//
function StoryListAssistant(selectedFeedIndex) {
    this.feed = feedList[selectedFeedIndex];
    this.feedIndex = selectedFeedIndex;
}
StoryListAssistant.prototype.setup = function() {
    // Setup story list with standard news list templates.
    //
    this.controller.setupWidget("storyListWgt",
        this.storyAttr = {
            itemTemplate: "storyList/storyRowTemplate",
            listTemplate: "storyList/storyListTemplate",
            swipeToDelete: false,
            renderLimit: 40,
```

```
        reorderable: false
    },
    this.storyModel = {
        items: this.feed.stories
    }
};
this.controller.listen("storyListWgt", Mojo.Event.listTap,
    this.readStory.bindAsEventListener(this));
};
StoryListAssistant.prototype.activate = function() {
    // Set title into header
    $("feedTitle").innerHTML=this.feed.title;
    // Update story list model in case unreadCount has changed
    this.controller.modelChanged(this.storyModel);
};
// readStory - handler when user taps on displayed story, push that story to
// the storyView scene
StoryListAssistant.prototype.readStory = function(event) {
    Mojo.Controller.stageController.pushScene("storyView", this.feed, event.index);
};
StoryListAssistant.prototype.deactivate = function(event) {
};
StoryListAssistant.prototype.cleanup = function(event) {
};
```

当场景被 **StoryListAssistant** 函数调用进行初始化的时候，传递的源索引用来将选定的源分配到 **this.feed**。场景被按后，安装函数被调用并设置列表控件：分配模板、**renderLimit** 被设置为默认20的两倍。为您的列表考虑，如果有需要的话，你应该使用默认的、经过调试过的。

Render limit（转化限制？？）

列表元素的数目每次都会被列表控件转化进 **DOM**，这是由 **renderLimit** 定义的。通常，它不需要被指定，但是如果你的列表项目非常短，那么默认的20个可能不够，因为翻页显示所有的项目可能导致 **Mojo** 构架负荷运作。

为了保证效率，**Mojo** 构架需要限制被列表控件转化的列表项目。**Mojo** 构架不能转化所有的项目，否则不管是内存还是系统运行都会有重读，但是，仍然需要足够的数量以防止翻页显示列表负荷运作。

理想情况下，**Mojo** 构架会自动计算这个数值，但是现在还不行，所以如果你在翻页的时候发现列表中有空余，你需要调整这个数值。

列表的模板项目被设计为输入源的 **stories** 数组以便显示在列表中，**setupWidget** 会被

调用以完成列表的初始化。列表上的任何轻击都会引起监听。而处理程序 `readStory` 会 `push` 选定的 `story` 项目到 `storyView` 场景。

激活过程中，列表标题会被赋值显示在头部，我们暂时更新了列表模型，改变 `story` 的 `unreadStyle`，便于阅读已选定的 `story`。我们希望状态上的改变是快速的。

最近，我们改变了 `stage-assistant.js`，`push storyList` 场景，而不是 `storyView` 场景：

```
StageAssistant.prototype.setup = function() {  
    this.controller.pushScene("storyList", curFeedIndex);  
};
```

你应该会注意到，`curFeedIndex` 参数仍然被使用着，但 `curFeedSource` 已经被删掉了。现在 `feedList` 是源资源了，它使用所有已选定的 `stories` 了。

最后，在 `index.html` 中加载新的辅助程序，然后启动程序。新的场景包含了所有来自例子源的 `stories`，都显示在图3-7了。

图3-7 故事列表场景

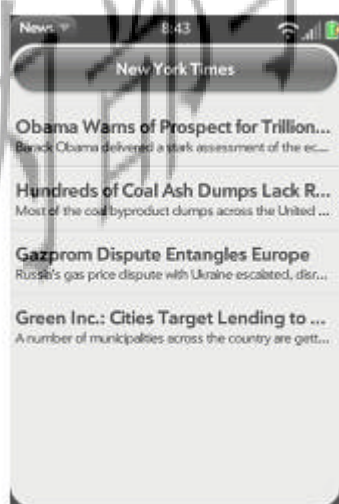


看起来还不是很好，所以，再加点 `CSS` 调整下尺寸和空间，使它看起来更像图3-8。

```
.listTitle {  
    padding-top: 10px;  
    padding-right: 5px;  
    padding-left: 5px;  
    font-size: 14pt;  
    text-align: left;  
    font-weight: normal;  
}
```

```
.listText {  
    vertical-align: bottom;  
    padding-bottom: 10px;  
    padding-right: 10px;  
    padding-left: 5px;  
    font-size: 10pt;  
    height: 20px;  
    font-weight: normal;  
    text-align: left;  
    overflow-x: hidden;  
    overflow-y: hidden;  
}  
.unreadStyle {  
    font-weight: bold;  
}
```

图3-8 风格调整过的 StoryList



3.4.1.2 使用 sources.json

我们刚才加入了第二个场景，这个程序变得越来越丰富了。我们想更了解程序的再如次数。现在正是加入 **sourcecs.json** 的好机会，每一次场景被轻击而不是程序启动的时候，他们都能被载入。你可以在程序的根目录创建 **sources.json**，然后加入以下代码：

```
[  
  {  
    "source": "app\\controllers\\stage-assistant.js"  
  },  
  {  
    "source": "app\\controllers\\storyList-assistant.js",  
  }  
]
```



```

    "scenes": "storyList"
  },
  {
    "source": "app\\controllers\\storyView-assistant.js",
    "scenes": "storyView"
  }
]

```

从 `index.html` 中移除这些文件的脚本载入 `tag`，留下 `mojo.js` 的脚本载入 `tag`。从现在开始，无论何时为程序添加一个新的场景，都添加一个相应的条目到 `sources.json` 中。

3.4.1.3 回到新闻程序：ajax 请求

我们将在第六章详细介绍 `ajax`，不过此处介绍它是为了开启动态源列表。我们已经有了列表，现在要给它增加一个功能，通过 `ajax` 请求访问源资源，从而载入并更新列表。

动态数据是一项非常有用、非常重要的功能，大多数程序都会开发这个功能。用它来更新程序的数据设置，用户可以获得最为及时、准确的信息。倘若没有这个功能，程序就失去了某些价值，它使得人们花费数个小时去关注一些事情。

你可以编写属于自己的 `ajax` 界面，不过考虑到 `ajax` 的简单却强大的功能，`webOS` 已经包含了 `ajax` 的原型库。接下来我们要在一个新的场景 `feedList` 中添加一个 `ajax` 请求，它会向纽约时代周刊的源获取数据，然后把更新数据加载到 `storyList`。`Ajax` 的请求是相当简单的，然而我们必须处理我们接收到的 `RSS&ATOM` 数据，这就稍微复杂点了。

依然是使用 `palm-generate` 添加 `feedList` 场景

```
palm-generate -t new_scene -p "name=feedList".
```

别忘记在 `sources.json` 中添加新的场景哦。我们只需要 `ajax` 请求的网址，然后设置一些回调函数。在 `feedList` 中添加一个新的函数原型：

```

//    feedRequest - function called to setup and make a feed request
//
//    Uses prototype's Ajax.Request and sets up callback routines:
//        onSuccess          feedRequestSuccess
//        onFailure          feedRequestFailure
//
FeedListAssistant.prototype.feedRequest = function(curFeed) {
    var request = new Ajax.Request(curFeed.url, {
        method: 'get',
        evalJSON: 'false',
        onSuccess: this.feedRequestSuccess.bind(this),
        onFailure: this.feedRequestFailure.bind(this)
    });
}

```

```
});
};
```

紧接着在 `feedList` 安装函数中调用它：

```
FeedListAssistant.prototype.setup = function() {
    // Set feedsInitialized to false so that activate knows that it's
    // being launched or pushed from the background
    //
    this.feedsInitialized = false;
    // Start the Ajax Request
    //
    this.feedRequest(curFeedSource);
};
```

Ajax 请求是异步传输的，包括成功的和失败的请求，你不得不为每个请求都创建回调函数。失败的请求只需要简单地记录下错误，稍后，我们会提交一个警报，ajax 请求返回一个 HTTP 状态信息，然后我们使用原型的模版函数把这些信息转化成可读的格式，并记录下结果。

```
// feedRequestFailure
//
// Callback routine from a failed AJAX feed request (feedRequest);
// post a simple failure error message with the http status code.
//
FeedListAssistant.prototype.feedRequestFailure = function(transport) {
    // Use the Prototype template object to generate a string
    // from the return status.
    //
    var t = new Template("Status #{status} returned from newsfeed request.");
    var m = t.evaluate(transport);
    Mojo.Log.info(".....", "Invalid feed - http failure (" , m);
};
```

成功的请求在使用前还需处理下源，我们记录下返回的状态信息，以确保载入成功，然后再次使用模版函数。接下来，我们要处理返回的源数据，这些源数据是文本，我们需要将它们转化成 XML 以方便处理。

我们通过调用全局函数 `ProcessFeed` 来决定源的格式，然后为 `feedList` 提取需要的部分。我们会在接下来的章节讲到它，但是，请注意，如果调用并返回的详细错误状态与 `errorNone` 一致，则说明源已经成功处理。轻击 `storyList`，处理源的过程是这样的。

```
// feedRequestSuccess
//
```

```
// Callback routine from a successful AJAX feed request (feedRequest); use globals:
//      curFeedIndex  Index for the feed being updated
//      feedList      Holds the processed feeds, will have the current feed
//                    updated at exit; has old version of feed at entry
//
FeedListAssistant.prototype.feedRequestSuccess = function(transport) {
    var    feedError = errorNone; //    Return variable for processing feed
    var t = new Template($L("Status #{status} returned from newsfeed request."));
    Mojo.Log.info(".....", "Feed Request Success: ", t.evaluate(transport));
    //    DEBUG - Work around
    if (transport.responseXML === null && transport.responseText !== null) {
        Mojo.Log.info(".....", "Request not in XML format
- manually converting");
        transport.responseXML = new DOMParser().parseFromString
(transport.responseText,
        'text/xml');
    }
    //    Process the feed, identifying the current feed index and passing in the
    //    transport object holding the updated feed data
    //
    feedError = ProcessFeed(transport, curFeedIndex);
    //    If successful processFeed returns errorNone
    if (feedError == errorNone) {
        Mojo.Controller.stageController.pushScene("storyList", curFeedIndex);
    } else {
        //    There was a feed process error
        if (feedError == errorUnsupportedFeedType) {
            Mojo.Log.info(".....", "Feed ", this.nameModel.value,
                " is not a supported type (#", errorUnsupportedFeedType, ").");
        }
    }
};
```

如果你对处理过程感兴趣，可以查看 **Appendix D** 中的 **ProcessFeed**，不过这里没有显示它是因为它不直接影响这里的 **Mojo** 函数。简单地说，**ProcessFeed** 把 **XML** 对象和所有文件传递到了 **feedList**。如果没有索引参数的话，**ProcessFeed** 会在列表底部添加一个新的源。

标题、文本、网址这些得到支持的格式都会被每个 **stories** 提取，**feedList** 的新的源数据、**stories** 和 **unreadCount** 都将得到更新。如果源不是格式化过的 **ATOM**, **RSS (RDF)**, **RSS**，会返回一个 **errorUnsupportedFeedType**。

程序启动的时候，在 **ajax** 请求接收源数据之前，将会有一个新的顶级场景用来简单显示。当 **ajax** 请求发出，构架会处理这个请求，并在 **storyList** 场景中显示一长串的 **stories**。

如图3-9。

图3-9 更新过 stories 的 StoryList

