

/* Translated by [sole](#)

Chapter4 (Palm webOS, 1st Edition Page 87——116)

*/

第四章 对话框&菜单

对话框和菜单是我们非常熟悉的组件，几乎所有的程序上都有它们的身影。Mojo 构架的对话框和菜单控件除了提供常规功能外，还具备一些特有的、额外的功能。对话框可以用作子场景，它允许开发人员在对话框中放置任何网页内容；而菜单也可以为场景所定制，以传统的下拉菜单方式显示，也可以以浮动元素方式显示。

虽然对话框和菜单都是基本的控件，却比第三章提到的那些要复杂得多，而且它们的访问方式和管理方式也和其他控件不同。多数基础控件都会调用 `setupWidget` 和 `showDialog` 函数，并需要辅助这个组建，而对话框是由控制器函数初始化的。

菜单是由 `setupWidget` 函数初始化的，但是它使用 `Commander Chain` 在舞台辅助和场景辅助中传递菜单命令。Mojo 构架提供了一个在程序、舞台、场景之间传递命令的模型，叫做 `Commander Chain`，本章结束部分将详细介绍它。

一如第三章所讲，我们会把他们加入到新闻程序中，并辅之一些文字描述和截图。

4.1 对话框

对话框允许开发人员创建模态视图做任何事情。自定义对话框就是一个传统对话框，它需要自身场景，也就是说，需要声明辅助和场景视图。对话框场景被作为子场景传送到调用它的场景辅助，所以，不管是开发还是运行的时候，都会产生大量的 `overhead`（可以理解为校验码）。开发人员可以使用错误对话框处理错误，使用警告对话框显示一些简单的选项。我们首先介绍构架内建的简单对话框，然后介绍如何使用 `showDialog` 函数创建自定义对话框。

4.1.1 错误对话框

错误对话框是一个用来传递错误信息的模态对话框，包括一个固定标题 `Error`，一个自定义的信息和一个确认按钮。错误对话框只能用来处理错误，因为开发人员无法改变标题。图4-1就是一个例子。它是一个被调用的函数，唯一的参数就是 `Error` 标题和 `OK` 按钮间的那条显示给用户的看的字符串。

图4-1 错误对话框



4.1.1.1 回到新闻程序：添加一个错误对话框

为了响应 ajax 发出的源同步的失败请求，开发人员可以在 feedlist-assistant.html 的 feedRequestFailure 函数后面添加一个 eerrorDialog 调用，这样就发布了一个错误对话框：

```
// 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);
    // Log error & post dialog
    //
    Mojo.Log.info("Invalid feed - http failure (" + m);
    Mojo.Controller.errorDialog("Invalid feed - http failure (" + m);
};
```

这个对话框显示了 ajax 的错误代码，并附加了一个静态错误信息。在新闻程序中，你可以在 ProcessFeed 函数(同样在 feedlist-assistant.html 中)中加入一个类似的错误对话框，以便遇到不支持的源格式的时候可以给出错误信息。

Logging Methods

Mojo 构架自带一个 logging method，它能高效地生成控制台输出，而又不降低程序或系统的运行效率。

共有三个 log 级别：

```
Mojo.log.info();      // Mojo.Log.LOG_LEVEL_INFO = 20
Mojo.log.warn();      // Mojo.Log.LOG_LEVEL_WARNING = 10
Mojo.log.error();     // Mojo.Log.LOG_LEVEL_ERROR = 0
```

只有同级或低级的信息才会被生成，当前 logging 级别是 appinfo.json 中的一个配置项目，如果想允许所有的 log 级别的话，必须把 logLevel 属性设置为99。

```
{
  "logLevel": 99
}
```

不要把传送代码限制在0以上，因为 logging 校验码会降低程序和系统的运行效率。

和 console.log 不同，Mojo.log 的参数是单独传送 log 函数的，如果信息是显示在控制台的，它只会被转成字符串。例如：

```
Mojo.Log.info("I have", 3, "eggs.");
```

会输出：

I have 3 eggs.

同时，系统还支持格式化部分字符，并把 log methods 添加到独立对象中。例如：

```
var favoriteColor = 'blue';
Mojo.Log.info("My favorite color is %s.", favoriteColor);
```

会输出：

My favorite color is blue.

开发人员可以使用%s, %d, %f, %i, %o 和%j，前四个会生成相同的结果，强制把适当的参数转化成用于 log 的字符串。%o 使用原型的 Object.inspect()函数把参数转化为字符串，同理，%j 使用的是 Object.toJSON()函数。

在桌面浏览器上，基于相应级别的信息是频繁出现的。对系统信息来说，添加文本和一些定义符号会使得信息更加突出。

4.1.2 警告对话框

开发人员可以使用警告对话框显示短信息，并用一个或多个 HTML 按钮显示带选项。如果用来显示对用户来说没有错误的信息或者用按钮来选择选项，警告对话框无疑是最好的选择。如图4-2

图4-2 警告对话框



4.1.3 自定义对话框

如果以上两个简单的对话框不能满足你的需求，还有一个 `showDialog` 函数可以用来在模态对话框内显示任何类型的内容。你可以把任何想要放入场景的内容放到对话框中，这几乎包括了几乎所有的网页内容和 Mojo 用户界面内容。

4.1.3.1 回到新闻程序：添加一个源对话框

在上一章，我们在源列表辅助中添加了一个绘图，用来支持添加源这个功能。把这个类型的功能放在对话框中更好。我们会使用 `showDialog` 函数创建一个对话框，然后把绘图中用到的代码放到这个对话框中。

通过调用 `showDialog()`，把 `feedlist-assistant.js` 中的 `addNewFeed` 替换掉：

```
// addNewFeed - triggered by "Add..." item in feed list and invokes the AddDialog
// Assistant defined above.
//
FeedListAssistant.prototype.addNewFeed = function() {
    this.controller.showDialog({
```

```

        template: 'feedList/addFeed-dialog',
        assistant: new AddDialogAssistant(this)
    });
};

```

参数指定了对话框模版和管理对话框的辅助的引用。现在，我们创建一个 AddDialogAssistant 的实例，传递到源列表辅助的引用，包括我们的 addFeed-dialog 模版的引用。对话框模版是一个简单的 HTML 模版，不过，你还是必须使用一些标准的对话框风格，比如 .palm-dialog-box, .palm-dialog-title 和 .palm-dialog-content，用来格式化和风格化对话框，以适合 webOS 用户界面。

为 addFeed-dialog 模版创建 HTML，并使用上一章源列表视图用到的代码，views/feedList/addFeed-dialog.html

```

<div>
  <span id="add-feed-title" class="palm-dialog-title">Add News Feed Source</span>
  <div class="palm-group unlabeled">
    <div class="palm-list">
      <div class='palm-row first'>
        <div class="palm-row-wrapper textfield-group"
x-mojo-focus-highlight="true">
          <div class="title">
            <div class="label">URL</div>
            <div id="newFeedURL" x-mojo-element="TextField" align="left"></div>
          </div>
        </div>
      </div>
      <div class='palm-row last'>
        <div class="palm-row-wrapper textfield-group"
x-mojo-focus-highlight="true">
          <div class="title">
            <div class="label">Title</div>
            <div id="newFeedName" x-mojo-element="TextField" align="left"></div>
          </div>
        </div>
      </div>
    </div>
  </div>
  <div x-mojo-element="Button" id="okButton"></div>
</div>

```

HTML 中唯一改变的是移除了绘图和封装的 .palm-group 风格代码。

对话框辅助必须像场景辅助一样定义，使用新建函数和标准场景函数：安装、激活、发

激活、清除。

使用对话框辅助，开发人员可以安装控件，传递场景，基本上可以用场景辅助做的，在这里都可以做到。有一个主要的区别，那就是对话框辅助并非使用 `sceneController` 函数扩展的，相反的，对话框辅助必须使用场景辅助的函数调用，比如 `setupWidget`。为了改进它，当对话框的创建函数被调用的时候，`showDialog` 参数对象中的 `assistant` 属性把 `thisuanjianzi` 作为一个关键字传递。

要创建 `AddDialogAssistant`，我们会移除上一章中使用的代码，以便在绘图控件中生成小表格。以下是 `AddDialogAssistant` 中改进过的代码。

```
// AddDialogAssistant - simple controller for adding new feeds to the list.
//     Invoked by the FeedListAssistant when the "Add..." list item is selected
//     on the feed list.
//
//     The dialog displays two text fields (URL and Name) and an OK button.
//     Either the user enters a feed URL and a name, followed by OK or a
//     back swipe to close. If OK, the feed header is checked through an Ajax
//     request and if valid, the feed updated and dialog closed. If an error,
//     the posted in place of title and dialog remains open.
//     Swipe back cancels and returns back to the FeedListAssistant.
//
function AddDialogAssistant(sceneAssistant) {
    this.sceneAssistant = sceneAssistant;
}
AddDialogAssistant.prototype.setup = function(widget) {
    this.widget = widget;
    //     Setup text field for the new feed's URL
    //
    this.sceneAssistant.controller.setupWidget("newFeedURL",
        this.urlAttributes = {
            property: "value",
            hintText: "RSS or ATOM feed",
            focus: true,
            limitResize: true,
            textReplacement: false,
            enterSubmits: false
        },
        this.urlModel = {value: ""});
    //     Setup text field for the new feed's name
    //
    this.sceneAssistant.controller.setupWidget("newFeedName",
        this.nameAttributes = {
            property: "value",
```

```
        hintText: "Optional",
        limitResize: true,
        textReplacement: false,
        enterSubmits: false
    },
    this.nameModel = {value : ""};
    // Setup button and event handler
    //
    this.sceneAssistant.controller.setupWidget("okButton",
        this.attributes = {},
        this.model = {
            buttonLabel: "Add Feed",
            buttonClass: "addFeedButton",
            disabled: false
        });
    Mojo.Event.listen($('okButton'), Mojo.Event.tap,
        this.checkIt.bindAsEventListener(this));
};
// -----
// Add Feed Functions
//
// checkIt - called when Add feed OK button is clicked
//
AddDialogAssistant.prototype.checkIt = function() {
    // Check for 'http://' on front or other legal prefix
    // assume any string of 1 to 5 alpha characters followed
    // by ':' is legal, otherwise prepend "http://"
    //
    var url = this.urlModel.value;
    if (/^[a-z]{1,5}:/ .test(url) === false) {
        url = url.replace(/^\w{1,2}/, "");
        url = "http://" + url;
    }
    // Update the submitted URL text model
    this.urlModel.value = url;
    this.sceneAssistant.controller.modelChanged(this.urlModel);
    var request = new Ajax.Request(url, {
        method: 'get',
        evalJSON: 'false',
        onSuccess: this.checkOK.bind(this),
        onFailure: this.checkFailure.bind(this)
    });
};
```

```
//
//   checkOK - called when the Ajax request is successful.
//
AddDialogAssistant.prototype.checkOK = function(transport)    {
    //   DEBUG - log the result
    //
    var t = new Template("Status #{status} returned from newsfeed request.");
    var success =      "Feed Request Success: "+t.evaluate(transport);
    Mojo.Log.info(success);
    //   Work around when XML comes back in text response
    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');
    }
    var feedError = errorNone;
    //   Push the entered feed onto feedlist and call processFeed to evaluate it.
    //   This is mainly to record the user's TITLE entry before processing
    //
    feedList.push({title:this.nameModel.value, url:this.urlModel.value, type:"",
        acFreq:0, numUnRead:0, stories:[]});
    feedError = ProcessFeed(transport);
    //   If successful processFeed returns errorNone
    //
    if (feedError === errorNone)    {
        this.sceneAssistant.feedWgtModel.items = feedList;
        this.sceneAssistant.controller.modelChanged
(this.sceneAssistant.feedWgtModel);
        this.widget.mojo.close();
    }
    else    {
        feedList.pop();
        if (feedError == errorUnsupportedFeedType)    {
            Mojo.Log.info("Feed ", this.urlModel.value,
                " isn't a recognized feed type (#", errorUnsupportedFeedType, ")");
            $("add-feed-title").innerHTML =
                "Invalid Feed - not a supported feed type";
        }
    }
};
AddDialogAssistant.prototype.checkFailure = function(transport) {
```



```
//    Use the Prototype template object to generate a string from the return
//
var t = new Template($L("#{status}"));
var m = t.evaluate(transport);
//    Log error and put message in status
//
Mojo.Log.info("Invalid feed (Status ", m, " returned).");
$("#add-feed-title").innerHTML = "Invalid feed - http failure (" + m + ")";
};
```

为了在对话框中创建这个版本的绘图控件，我们对上个版本做了些改变：

取消：删掉了“取消”按钮和 `cancelIt` 函数，因为返回手势会被用来提供取消功能，而不是按钮。

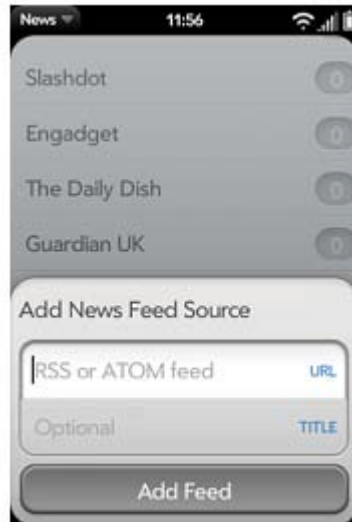
场景辅助函数：`this.controller.*` 引用改为 `this.sceneAssistant.controller.*` 引用，因为 `AddDialogAssistant` 必须使用为这些函数传递到场景辅助到引用。

关闭：在 `checkOk` 中添加源后，添加 `this.widget.mojo.close()` 函数。开发人员必须调用对话框控件上的 `close()` 函数来直接关闭对话框。注意，控件元素被当作参数传递到对话框辅助安装函数。

文本框的清理：退出时删除代码。这些代码不需要是因为对话框场景已经从 DOM 中完全删除。

在默认对话框中重击后退，会关闭这个对话框，所以不再需要取消按钮。如果新手不知道取消按钮为什么不在这里，开发人员也许就会想要在大多数的对话框中保留取消按钮了。你可以把 `showDialog` 调用参数中的 `preventCancel` 选项设置为 `true`，这样就可以禁用返回手势的取消功能。默认情况下，`preventCancel` 设置为 `false`。图4-3显示了改变后的结果，以及添加源的对话框。

图4-3 添加源的对话框。



4.2 菜单

Mojo 构架支持四种菜单控件，每一种的设计都相当不同，不过他们也会共用一些设计元素，并且使用方式相似。你可以参考下用户界面指导，看看如何应用各种菜单类型和为程序设计菜单是最好的。简单地说：

- 程序菜单：传统的桌面菜单风格是屏幕左上角的下拉菜单。
- 视图菜单：横跨屏幕顶部，用来显示标题栏，动作按钮，弹出子菜单或者选择设置。
- 命名菜单：横跨屏幕底部，用来设置菜单或典型按钮，可以弹出子菜单或者选择设置。
- 子菜单：可连接其他菜单类型以提供更多的选项，可以附加页面中的任何元素。

从技术上说，程序菜单、视图菜单、命令菜单是非常相似的，他们使用一个包含菜单项目数组的模型定义，由 `setupWidget()` 函数初始化。选择菜单后生成命令，由 `Commander Chain` 传送给注册过的命令。我们会在下一节介绍这三个控件。

子菜单和菜单控件共用很多模型属性，只不过，子菜单通过一个直接的函数调用初始化，并且，处理方式也不同。子菜单控件控件的地址被放置在菜单控件地址的后面。

系统用户界面包括另一个菜单，叫做连接菜单，它的外观和程序菜单差不多，被固定在屏幕右上方。它只能由系统使用，程序不可以使用它。

4.2.1 菜单控件

和其他控件不同，菜单控件菜单控件不是在视图文件中声明的，却是由辅助程序初始化并处理的。从设计角度来看，菜单控件浮动在其他场景元素上方，附着在场景的窗口上，而非场景上的一个点。正因为如此，在 HTML 中决定位置是不可行的。它们被放置在 DOM 中，

这样开发人员就可以使用 CSS 来应用风格，而构架根据预定义的容器和独立的菜单和模型属性来决定它们的位置。

菜单控件由对 `setupWidget()` 函数的调用初始化，指定菜单类型、属性和模型。菜单类型来自 `Mojo.Menu.type`，它可以是 `appMenu`, `viewMenu` 或 `commandMenu` 三者之一。

程序菜单与命令菜单/视图菜单在属性上有一些差别，我们可以这样描述它们。模型主要是由 `items` 数组组成的，它为每个菜单项目和可选属性包含一个对象。除了 `items` 数组，还有一个简单的 `visible` 属性可以设置整个菜单的可见或不可见。如果该属性不存在，则菜单默认可见。

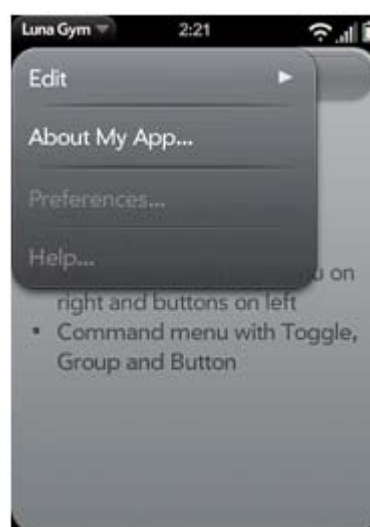
主要的选项都在 `items` 数组中，你可以把可选项目和分组放置在任何顶级菜单，你可以指定二级可选项目。项目可以包括一个标签和一个图标，图标可以指定构架的图标之一，也可以使用 `icon` 属性或者指定了图标的程序，开发人员可以在 `iconPath` 中找到它。

每一个项目都包括一个命令值，当这个项目被选中的时候，它由 `commander chain` 传递。这是一个相当重要的话题，我们要在这里简要地讲一下，但是，稍后你应该查阅下“Commander Chain”这一节，以便获得完整的描述。

4.2.1.1 程序菜单

当用户轻击状态栏的左边的时候，程序菜单就会显示在屏幕的左上角。它包括一些系统和程序定义的操作，并且大多数情况下，它的宽度和程序一样。图4-4是一个程序菜单的例子。

图4-4 程序菜单



程序菜单包括一些必须的项目：编辑（一个项目分组包括剪切、复制、粘贴）、首选项和帮助。后面的项目默认是不显示的。你可以把任意其他项目加入到菜单中，也可以加入操

作命令来启用首选项/帮助，在程序中进行操作。

4.2.1.2 回到新闻程序：添加一个程序菜单

现在，让我们给新闻程序添加一个程序菜单，使程序最初会指向“关于新闻...”这个项目。和我们先前的例子不同的是，我们会把程序菜单属性和模型声明为全局变量，并把 `handleCommand` 函数添加到新闻程序的舞台辅助。这样可以让所有新闻场景辅助中的程序菜单都可用。

```
StageAssistant.prototype.setup = function() {  
  //   Setup Application Menu with an About entry  
  //  
  newsMenuAttr = {omitDefaultItems: true};  
  newsMenuModel = {  
    visible: true,  
    items: [  
      {label: "About News...", command: 'do-aboutNews'},  
      Mojo.Menu.editItem,  
      Mojo.Menu.prefsItem,  
      Mojo.Menu.helpItem  
    ]  
  };  
  this.controller.pushScene("feedList");  
};  
//   handleCommand - Setup handlers for menus:  
//  
StageAssistant.prototype.handleCommand = function(event) {  
  var currentScene = this.controller.activeScene();  
  if(event.type == Mojo.Event.command) {  
    switch(event.command) {  
      case 'do-aboutNews':  
        currentScene.showAlertDialog({  
          onChoose: function(value) {},  
          title: "News - v1.0",  
          message: "Copyright 2008-2009, Palm Inc.",  
          choices:[  
            {label: "OK", value:""}  
          ]  
        });  
      break;  
    }  
  }  
};
```

对程序菜单来说，这些菜单属性是唯一的：

当你在场景中加入 RichTextEdit 控件的时候，richTextEditItems 可以设置为 true，它会把粗体、斜体、下划线加入到编辑菜单中。

当你想要启用首选项或帮助，或者想要禁用编辑的时候，omitDefaultItems 必须设置为 true。

如果你选择 omitDefaultItems，你必须把项目放置到自己的定义中去。如果你想覆盖一部分项目，你可以使用系统常量来替代那些不改变值的项目。在 newsMenuModel 中，默认项目是 Mojo.Menu.editItem、Mojo.Menu.prefsItem 和 Mojo.Menu.helpItem。

newsMenuAttr 声明该菜单可以覆盖默认选项，newsMenuModel 把“关于新闻”放置到菜单顶部并引用默认项目，由构架保证它们在菜单中，并加以操作。handleCommand 函数中，do-aboutNews 操作命令提供一个警告对话框，把它当做关于对话框。

当程序菜单命令被传递的时候，它们被舞台辅助处理，但是操作命令必须知道当前场景。本地变量 currentScene 就是 handleCommand 开始处用来激活场景控制器的。currentScene 应用场景辅助函数，比如 showAlertDialog 被应用到目前显示的所有场景。

这一切都是在 stage-assistant.js 中完成的，而长须菜单却是在场景中显示的，要设置和显示菜单控件，每个场景辅助的安装函数都要加入 setupWidget()调用：

```
// Setup Application Menu
this.controller.setupWidget(Mojo.Menu.appMenu, newsMenuAttr, newsMenuModel);
```

你可以在指定场景设置程序菜单之前定义场景指定程序菜单的属性或模型，并在场景中添加一个 handleCommand 函数来处理程序菜单的命令，这样即可覆盖程序宽的行为。如果在全局程序菜单中使用相同的命令，别忘了调用 Mojo.Event.stopPropagation 函数。

图4-5是程序菜单和关于对话框

图4-5新闻程序菜单和关于对话框



在舞台辅助中统一程序菜单声明恶化操作后,就很容易在所有的场景中提供一些列的菜单选项了。例如,我们可以在新闻程序中添加一个首选项场景。

4.2.1.3 回到新闻程序: 覆盖默认首选项

上一章,我们在 `feedlist-assistant.js` 中实现了 `ajax` 请求,它可以检索起始的源数据。现在我们要扩展它的功能,让它可以周期性地更新源,我们可以在首选项的场景中设置更新间隔/周期。

使用 `palm-generate` 创建一个首选项场景:

```
palm-generate -t new_scene -p "name=preferences" com.palm.app.news
```

场景视图文件，preferences-scene.html，看上去应该是这个样子的：

```
<div class="palm-page-header">
  <div class="header-icon news"></div>
  <div class="header-text">
    <span id="newsPrefsTitle" class="feedTitleSpan">News</span>
  </div>
</div>
<div class="palm-list">
  <div class="palm-row">
    <div id="feedCheckIntervalList" x-mojo-element="ListSelector"></div>
  </div>
</div>
```

头部是构架的风格类之一，.palm-page-header，首选项场景标准。你会注意到.header-icon style 和新闻程序的风格使得我们可以在头部添加一些 CSS，用来指定一个小小的新闻程序图标。图标必须加入到新闻程序的跟级 images 目录。

```
.palm-page-header .header-icon.news {
  background: url(..images/header-icon-news.png) no-repeat;
  font-weight: bold;
}
```

头部风格后面弄是几列风格类，紧接着是选择间隔设置的列选择控件声明。preferences-assistant.js 会安装列表选择器并使用列表选择器为选项添加监听器。feedIntervalHandler 更新全局变量，后面是 feedUpdateInterval。

```
function PreferencesAssistant() {
}
PreferencesAssistant.prototype.setup = function() {
  //      Setup list selector for feed interval
  //
  this.controller.setupWidget('feedCheckIntervalList',
    { label: "Interval",
      choices: [
        {label: "1 Minute",      value: 60000},
        {label: "5 Minutes",    value: 300000},
        {label: "15 Minutes",   value: 900000},
        {label: "1 Hour",       value: 3600000},
        {label: "4 Hours",      value: 14400000},
        {label: "1 Day",        value: 86400000}
      ]
    },
    this.feedIntervalModel = {
```

```

        value : feedUpdateInterval
    });
    this.controller.listen('feedCheckIntervalList', Mojo.Event.propertyChange,
        this.feedIntervalHandler.bindAsEventListener(this));
};
PreferencesAssistant.prototype.feedIntervalHandler = function(event) {
    feedUpdateInterval = this.feedIntervalModel.value;
};

```

`feedUpdateInterval` 用来设置更新的计时器，请在附录 D 中参考完整的新闻程序，开发人员可以在这里了解到我们对 `feedlist-assistant.js` 的更新操作。

写好了首选项场景的代码，我们可以返回到舞台辅助，改变 `newsMenuModel` 以覆盖默认的首选项命令：

```

//
//   Setup Application Menu with preferences entry
//
newsMenuAttr = {omitDefaultItems: true};
newsMenuModel = {
    visible: true,
    items: [
        {label: "About News...", command: 'do-aboutNews'},
        Mojo.Menu.editItem,
        { label: "Preferences...", command: 'do-newsPrefs' },
        Mojo.Menu.helpItem
    ]
};
this.controller.pushScene("feedList");
};

```

在 `handleCommand` 函数中为 `do-newsPrefs` 添加一个操作来传递首选项场景：

```

//   handleCommand - Setup handlers for menus:
//
StageAssistant.prototype.handleCommand = function(event) {
    var currentScene = Mojo.Controller.stageController.activeScene();
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
            case 'do-aboutNews':
                currentScene.showAlertDialog({
                    onChoose: function(value) {},
                    title: "News — v1.0",
                    message: "Copyright 2008-2009, Palm Inc.",

```

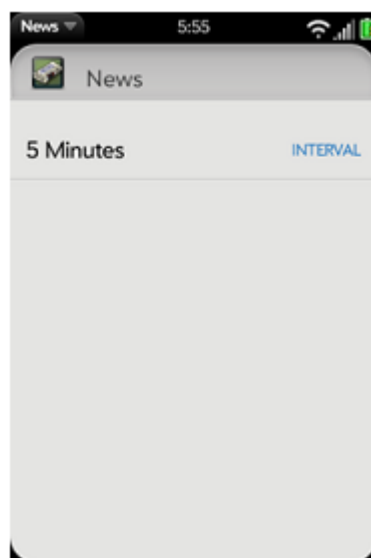


```
        choices:[
            {label:"OK", value:""}
        ]
    });
    break;
case 'do-newsPrefs':
    Mojo.Controller.stageController.pushScene("preferences");
    break;
}
}
};
```

当你现在运行程序的时候,你可以看到首选项已经启用,并且在新的场景中已经被选中。
图4-6是程序菜单和首选项场景。

图4-6带首选项场景的新闻程序菜单



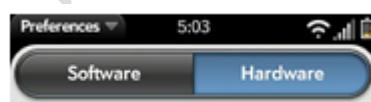


必须注意，我们不必修改任何场景辅助，首选项在任何场景中就都可用了。这个方式使得通过程序统一通用程序菜单操作变得简单！

4.2.1.4 视图菜单

视图菜单作为可变尺寸的按钮呈现项目，可以是单个的，也可以是一个个分组的。这些项目从屏幕左边到右边水平传递。按钮的宽度可以通过宽属性调整，构架也会调整自动调整两个按钮间的空间，如图4-7所示。使用分配或者空的列表项目来影响空间，以指定图层。

图4-7带按钮的视图菜单



通常来说，开发人员可以为操作按钮使用视图菜单，这些按钮可以附带子菜单或者显示头部。开发人员可以使用图4-7所示的头部信息来组合按钮或者联合操作按钮。

4.2.1.5 回到新闻程序：添加视图菜单

视图菜单允许开发人员为 `storyList` 场景头部应用风格，并为切换故事源提供一个简单的方法。我们会修改 `storyList-assistant.js` 来添加一个视图菜单，它包括下一个源和上一个源的菜单按钮，选中的时候由函数传递新场景。

首先，我们来加一个视图菜单。`feedMenuModel` 由三个菜单项目组成：

- `feedMenuPrev` 是一个本地变量，用来设置上一个菜单项目，当 `selectedFeedIndex` 指向源列表中的第一个源的时候，也可以设置为一个空的项目。

- feedMenuNext 是一个本地变量，用来设置下一个菜单，当 selectedFeedIndex 指向源列表中的最后一个源的时候，也可以设置为一个空的项目。
- 一个文字对象用来显示选中的源的标题。

安装函数开始是 feedMenuPrev 和 feedMenuNext 中的一些条件语句，用来处理第一个和最后一个源，然后视图菜单控件由 setupWidget()调用安装。

不指定任何可见属性（如标签和图标）或不分组的项目会被当做 dividers。菜单按钮的图层中，所有的多余空间都平均分配给每一个 dividers。如果没有 dividers，多余的空间都放置在菜单项目中，第一个和最后一个菜单项目永远对齐到场景的左边和右边。第一个源和最后一个源会在 feedMenuPrev 和 feedMenuNext 中新建 dividers 来保持头部的风格和格式。

菜单图标

Mojo 构架包括大量默认图标，开发人员可以在试图和命令行残带按钮中使用这些图标，或者提供自己的图标。

以.palm-menu-icon 为前缀的标准风格可以引用 icon 属性，或者定义自己的图片文件夹，通过 iconPath 属性引用它们。新的图例各使用它们中的一种，以便开发者可以观察这两种技术。

如果开发人员需要设计自己的图标，需要遵循一些规则：

- 使用支持8位 alpha 透明图层的 PNG-24
- 菜单图标是32x64 的 PNG 图片，由两部分组成，上部是普通状态，下部是按压时的状态。
- 每一个图标近似24x24像素，32x32帧以内。
- 开发人员可以试着先做一个单色的文字，应用 photoshop 的图层效果，尽管简单的白色图标也是可用的。

可以在构架的 images 目录找到一些图标的例子，这些图标都是 png 文件。

```
//
// 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 scene header with feed title and next/previous feed buttons
```

```
// If first feed, suppress Previous menu; if last feed, suppress Next menu
//
var feedMenuPrev = {};
var feedMenuNext = {};
if (this.feedIndex > 0) {
    feedMenuPrev = {
        icon: 'back',
        command: 'do-feedPrevious'
    };
} else {
    feedMenuPrev = {icon: "", command: "", label: ""};
}
if (this.feedIndex < feedList.length-1) {
    feedMenuNext = {
        iconPath: 'images/menu-icon-forward.png',
        command: 'do-feedNext'
    };
} else {
    feedMenuNext = {icon: "", command: "", label: ""};
}
// Define the model with next and previous menu items, and a title
// area for the feed title where the width is set to creating a
// style header
this.feedMenuModel =
{
    visible: true,
    items: [{
        items: [
            feedMenuPrev,
            { label: this.feed.title, width: 210 },
            feedMenuNext
        ]
    }]
};
// Setup the View Menu
this.controller.setupWidget(Mojo.Menu.viewMenu,
    { spacerHeight: 0, menuClass:'no-fade' },
    this.feedMenuModel);
// Setup Application Menu
this.controller.setupWidget(Mojo.Menu.appMenu, newsMenuAttr, newsMenuModel);
// Setup story list with standard news list
templates, and listener for story taps
//
```

```
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));
};
```

继续我们的例子，我们会在 `activate` 和 `readStory` 函数后面添加一个 `handleCommand` 函数，`handleCommand` 函数不变。

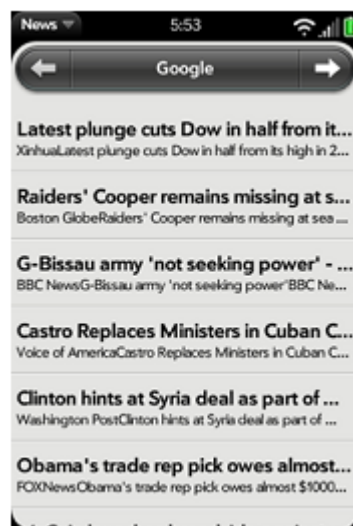
```
StoryListAssistant.prototype.activate = function() {
    // Unread count may have changed
    this.controller.modelChanged(this.storyModel);
};
// readStory - handler when user taps on displayed story,
// push story to the storyView scene
StoryListAssistant.prototype.readStory = function(event) {
    Mojo.Controller.stageController.pushScene("storyView", this.feed, event.index);
};
// -----
// Setup handlers for command menu
StoryListAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
            case 'do-feedNext':
                Mojo.Controller.stageController.swapScene("storyList", this.feedIndex+1);
                break;
            case 'do-feedPrevious':
                Mojo.Controller.stageController.swapScene("storyList", this.feedIndex-1);
                break;
        }
    }
};
```

`handleCommand` 函数被调用来执行上一个或下一个命令，并调用 `swapScene()` 函数来

传递下一个场景。我们在第二章中提到过，`swapScene()`函数和 `pushScene()`是很相似的，但是，它不会把旧的场景留在场景堆栈中，而是把它当做操作的一部分。同样，它的转变方式和 `pushScene()`或 `popScene()`都是不一样的，这对你来说可能应该是值得注意的，因为如此，你永远可以单独调用它来处理 `push` 和 `pop` 场景。

有了这些改变，新闻程序的故事列表场景看上去就像图4-8了。

图4-8新闻程序视图菜单和故事列表场景



4.2.1.6 命令菜单

命令菜单呈现在屏幕的底部，不过大多数时候和视图菜单是相似的。所有项目都包含可变尺寸的按钮，能够在水平层面上从左到右联合为分组。开发人员可以添加 `dividers` 使得项目置于屏幕的右边或者中间，也可以添加一个项目入口，并把禁用属性设为 `true`，这两个方法都可以覆盖位置。通常来说，开发人员会在以下情况使用命令菜单：操作按钮、动态按钮、包含进一步操作的子菜单。

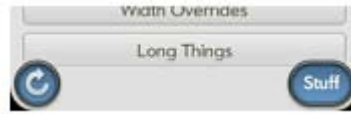
和视图菜单一样，按钮的宽度可以通过项目的宽属性来调整，构架页会自动调整按钮间的空间，如图4-9所示。

图4-9带按钮的命令菜单



图4-10显示的是你可以定义开关按钮或者在按钮上添加其他动态行为。

图4-10带开关的命令按钮



如果开发人员想把多个项目分为一个分组，必须包括一个 `items` 数组和 `toggleCmd`，`toggleCmd` 由构架设置，控制嵌套项目数组中已选的项目。开发人员可以为按钮分组，或者把可操作的按钮联合为一个开关组，如图4-11所示。

图4-11带分组的命令菜单



4.2.1.7 回到新闻程序：添加命令菜单

我们已经在故事视图中使用 HTML 按钮来设置故事间的后退、前进，这里，我们要用命令菜单替换他们。这里，我们要用程序和视图菜单讲解下。

在 `storyView-assistant.js` 中添加一个命令菜单，和视图菜单相似的是，下一个/上一个按钮通常都是分配来生成 `do-viewNext/do-viewPrevious` 命令的，除非当前故事是源中的第一个或者最后一个。安装函数的第一部分会根据正确的条目数量创建一个项目数组，然后调用 `setupWidget()` 函数来初始化菜单。我们已经把场景中的 HTML 按钮替换掉，就可以从安装函数（和场景的视图文件中的按钮生命）删除监听器。

注意：放入一个菜单中的项目的风格都是一样的。我们在分组的任意一边使用 `dividers`，强制分组居中。注意，程序菜单的分组显示上的不同，子项目是联合在扩展项目上的。视图菜单和命令菜单按钮分组显示为一个继承的视图元素。

```
// StoryViewAssistant(storyFeed, storyIndex)
//
// Passed a story element, displays that element in a
// full scene view and offers options for next story (right
// command menu button), previous story (left command menu button)
// and to launch story URL in the browser (view menu).
function StoryViewAssistant(storyFeed, storyIndex) {
    // Save the passed arguments for use in the scene.
    //
    this.storyFeed = storyFeed;
    this.storyIndex = storyIndex;
}
```

```

StoryViewAssistant.prototype.setup = function() {
    // Setup command menu for next and previous story as a menu group.
    // If first story, suppress Previous menu; if last story, suppress Next menu
    this.storyMenuAttr = {items: [{visible: false}, {items: [], {visible: false}}];
        if (this.storyIndex > 0)    {
            this.storyMenuAttr.items[1].items.push({icon:
"back", command: 'do-viewPrevious'});
        }    else    {
            this.storyMenuAttr.items[1].items.push({icon: "",
command: "", label: ""});
        }
        if (this.storyIndex < this.storyFeed.stories.length-1)    {
            this.storyMenuAttr.items[1].items.push({icon: "forward",
command: 'do-viewNext'});
        }
        else {
            this.storyMenuAttr.items[1].items.push({icon: "",
command: "", label: ""});
        }
        this.controller.setupWidget(Mojo.Menu.commandMenu, undefined, this.storyMenuAttr);
        // Setup Application Menu
        this.controller.setupWidget(Mojo.Menu.appMenu, newsMenuAttr, newsMenuModel);
    };
StoryViewAssistant.prototype.activate = function(event) {
    // Update story title in header and summary
    this.controller.update($("#storyViewTitle"),
        this.storyFeed.stories[this.storyIndex].title);
    this.controller.update($("#storyViewSummary"),
        this.storyFeed.stories[this.storyIndex].text);
    // Update unreadStyle string and unreadCount in case it's changed
    if (this.storyFeed.stories[this.storyIndex].unreadStyle === readStoryFormatting) {
        this.storyFeed.numUnRead--;
        this.storyFeed.stories[this.storyIndex].unreadStyle = "";
    }
};

```

激活函数依然没有改变，不过，我们要用操作命令替换命令按钮，如下：

```

// Handlers to go to next and previous stories
StoryViewAssistant.prototype.handleCommand = function(event) {
    if(event.type == Mojo.Event.command) {
        switch(event.command) {
            case 'do-viewNext':

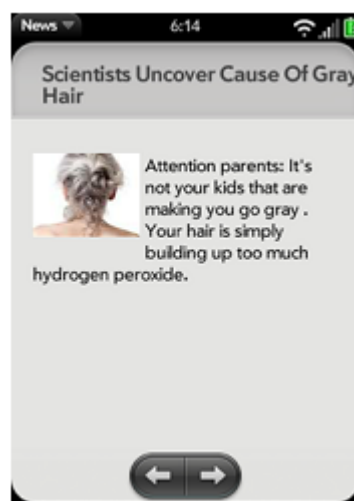
```



```
        Mojo.Controller.stageController.swapScene("storyView",
        this.storyFeed, this.storyIndex+1);
        break;
    case 'do-viewPrevious':
        Mojo.Controller.stageController.swapScene("storyView", this.storyFeed,
        this.storyIndex-1);
        break;
    }
}
};
```

运行程序，轻击一个源，然后轻击一个故事，看看结果。如图4-12

图4-12新闻程序命令菜单和故事视图场景



4.2.2 子菜单

弹出式子菜单可以用来为用户提供一个短暂的文本选择列表，通常另一个菜单类型或者场景中的一个 DOM 元素打开的时候会消失。它接受标准菜单模型和一些唯一的属性，不过和其他菜单类型不同的是，子菜单不使用 `commander chain` 来传递已选项目。因此，我们使用一个回调来操作选项。

一个模型列表会显示，其带有标签选择。当用户轻击其中一个的时候，`onChoose` 回调功能会被调用（在场景辅助的范围内），并把已选项目的命令属性作为一个参数。如果用户在弹出式菜单的外面轻击，它依然会解除，`onChoose` 函数会被调用，只不过是未定义的调用。

4.1.1.1 回到新闻程序：添加一个子菜单

当新闻程序的源列表中的未读项目被轻击的时候,我们会使用一个子菜单来呈现一些选项。每一个源都可以选择已标记/未标记/编辑。前两个选项把所有已选的故事标记为已标记/未标记,最后一个选项会打开添加源的对话框。

添加子菜单必须更改 `feedlist-assistant.js` 中的 `showFeed()` 函数,它可以检测列表项目右边的 `unreadCount` 图标上的轻击。如果是其他地方的轻击,故事列表场景会像以前一样被传递。

否则,子菜单会被创建。参数列表由 `onChoose` 开始,它指定了 `popupHandler` 来处理用户的菜单选项。其他的参数包括 `placeNear` 属性,用来定位子菜单附近的图标,还包括菜单项目的数组。开发人员会注意到,我们在 `popupHandler` 中把 `event.index` 值分配给 `this.popupIndex`。应用子菜单选项显示的操作的时候,我们需要引用被轻击的列表条目。

```
// showFeed - triggered by tapping a feed in the feedList.
// Detects taps on the unreadCount icon; anywhere else,
// the scene for the list view is pushed. If the icon is tapped,
// put up a submenu for the feedlist options
//
FeedListAssistant.prototype.showFeed = function(event) {
    var target = event.originalEvent.target.className;
    if (target !== "unreadCount") {
        this.clearTimers();
        Mojo.Controller.stageController.pushScene("storyList", event.index);
    }
    else {
        this.popupIndex = event.index;
        this.controller.popupSubmenu({
            onChoose: this.popupHandler,
            placeNear: event.target,
            items: [
                {label: 'All Unread', command: 'feed-unread'},
                {label: 'All Read', command: 'feed-read'},
                {label: 'Edit Feed', command: 'feed-edit'}
            ]
        });
    }
};
```

`popupHandler` 使用一个开关语句来调用适当的操作命令。它处理所有已选源中的故事的未阅读/已阅读操作,更新源中的 `numUnread`, 调用 `modelChanged` 来更新场景中的现实视图。一旦发出了指令,构架删除子菜单并把源列表场景返回到背景视图,然后操作返回。

```
// popupHandler - choose function for feedPopup
//
FeedListAssistant.prototype.popupHandler = function(command) {
```

```
var popupFeed=feedList[this.popupIndex];
switch(command) {
    case 'feed-unread':
        for (var i=0; i<popupFeed.stories.length; i++ ) {
            popupFeed.stories[i].unreadStyle = unreadStoryFormatting;
        }
        popupFeed.numUnRead = popupFeed.stories.length;
        this.controller.modelChanged(this.feedWgtModel);
        break;
    case 'feed-read':
        for (var j=0; j<popupFeed.stories.length; j++ ) {
            popupFeed.stories[j].unreadStyle = "";
        }
        popupFeed.numUnRead = 0;
        this.controller.modelChanged(this.feedWgtModel);
        break;
    case 'feed-edit':
        this.controller.showDialog({
            template: 'feedList/addFeed-dialog',
            assistant: new AddDialogAssistant(this, this.popupIndex)
        });
        break;
}
};
```

最后一个选项 `AddDialogAssistant` 是重复使用的。`this.popupIndex` 这个新的参数被添加到 `AddDialogAssistant` 调用中，来启用 `AddDialogAssistant`；`checkIt` 和 `checkOk` 用来检查编辑过的项目。这里不放出它们的改变，因为它们和子菜单没有直接关系，但是你也可以在复读 D 中查看完整的新闻程序代码，看看我们做了哪些改变。

图4-13是改变后的带源列表场景的子菜单。

图4-13带源列表场景的子菜单



4.3 Commander Chain

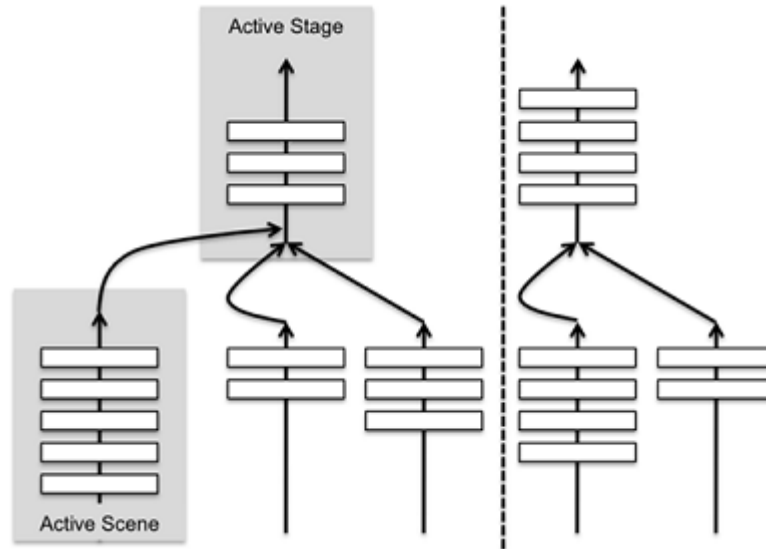
Mojo 构架为程序、舞台、场景控制器间的命令传送提供了一个模型，我们称之为 Commander Chain。它是一个操作数组，像堆栈一样排列。操作命令被放到这个链上，一一进行注册，然后按顺序传递命令。

命令初始化的时候，通过声明一个 `handleCommand` 函数被隐性地注册，不管是作为舞台辅助、场景辅助还是对话框。

命令通过调用舞台控制器或者场景控制器的 `pushCommander` 函数能被明确地注册，当场景辅助或者程序被关闭的时候，命令会被删除。

所谓“链”是一竖列的“链”，见图4-14。这是每个场景控制器的链，每个舞台都有一个场景控制器。命令从已经注册的场景控制器的链开始传递，当场景中的所有命令都被调用了以后，传递从剩余的链继续开始。每一个不活动的舞台控制器和场景控制器都有一个链，但是命令并不会传递到它们那里。。

图4-14命令链的传递



任何时候，任何命令都能通过调用 `Event.stopPropagation` 函数来终止传递。例如，一个场景打开一个模型对话框，它被隐性地添加到链中。它有机会操作一个回退事件，然后在场景返回前终止传递。如果不是这样，舞台控制器就会看到回退手势和弹出场景，这当然不是用户希望看到的。

命令永远能够通过调用 `StageController` 或 `SceneController` 中的 `removeCommander` 函数从链中删除自身，例如：

```
this.controller.removeCommander(this);
```

在链中传递有三种事件类型：

- `Mojo.Event.back`，这个事件代表一个回退手势。
- `Mojo.Event.command`，所有菜单命令都使用的事件。
- `Mojo.Event.commandEnable`，用来动态地启用一个菜单项目。

命令和命令启用事件都在菜单控件这节中讨论到了。前者在菜单命令被选中的时候会用到；后者在菜单被创建的时候会用到，当菜单项目包含一个 `commandEnable` 设置为 `true` 的属性的时候。如果任何命令试图禁止菜单命令的时候，可以调用一个 `stopPropagation` 函数。当程序菜单中文本区域被聚焦的时候，构架用这个函数来禁用编辑功能。

一个普通的命令链程序是由安装和舞台控制器中的程序菜单的操作统一而成的。程序菜单那一节使用新闻程序的时候已经讲过类似的例子。

4.4 概要

对话框和菜单可以满足大多数程序的基本需求，根据第二、第三、第四章学到的知识，开发人员足以写出一个有意义的程序来。在这章，我们讲到三个对话框功能和四种类型的菜

单，我们也在新闻程序中使用了它们。

下一章会讲到其他剩余的控件，在这里使用空间和用户界面模型创建一些例子是相当好的。根据前两章讲过的知识，开发人员可以创建一个全功能的程序，但是，更重要的是，这里讲到的概念将会用于书中余下的部分。

吹友吧内部交流专用