**基于JQM的移动记事本（1）**

来自于：<http://jorgeramon.me/>，学习之用

本文将介绍一个基于行为驱动模式开发(behavior-driven approach简称BDD)的框架Jasmine framework.(<https://github.com/pivotal/jasmine/wiki> )，在BDD开发中，将就每一个步骤都通过该开发框架来进行不断重构，以达到快速开发的目的。

　　首先看系统的整体设计，将完成如下的几个步骤：

　　1)、设计应用的总体架构

　　2)、创建一个基本的用户界面模型

　　3)、使用Jasmine开发框架定义应用的公共接口

　　4)、开始实现已定义的公共接口

**应用的总体架构**

　　我们打算记事本应用能提供如下功能：

　　1)、创建记事

　　2)、编辑记事内容

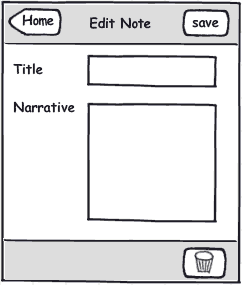
　　3)、删除记事内容

　　4)、将记事的内容保存在移动客户端中

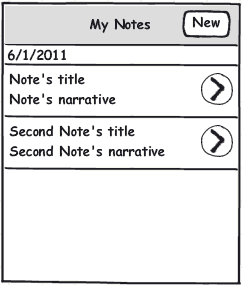
　　5)、查看所有已建立的记事列表

**主界面效果**

　　首先要实现的是一个可以增加和编辑记事的界面，将这个界面命名为Note Editor。界面看上去应该是如下图的样子(下图是个设计草稿图)：



　　此外，我们需要一个记事的列表的界面，以显示已经存在的记事，这个是我们启功应用程序时首先显示给用户的主界面，主界面如下所示：



　　接下来，看下如何使用BDD行为驱动的方式进行开发

**使用Jasmine Framework框架进行为驱动开发**

　　首先，我们将一边以实例的方式，一边让用户体验什么是行为驱动模式的开发。这种开发模式，能让我们先行定义和测试业务逻辑，可以跟表现层进行分离地设计。先来看下如何使用Jasmine framework框架。

　　Jasmine是一个允许开发者使用行为驱动的方式对Javascript进行测试的框架。这意味着开发者可以使用类似自然语言(易于理解)的方式，通过Jasmine编写测试用例，最后生成Javascript代码，这样明显降低了开发的难度，甚至让不是专业程序员的人也能读懂和编写。

　　下面快速举一个例子来说明Jasmine框架的使用，更复杂的用法，请参考其官方网站的手册，这个例子希望读者能理解，我们将在教程中稍后会用到。

   describe("Notes functions", function () {  
    it("Should return a NoteModel instance", function () {  
        var note = Notes.app.createNote();  
        expect(note instanceof Notes.model.NoteModel).toBeTruthy();  
    });  
});

　　在这个例子中，我们使用了多种Jasmine框架的特性。其中一个函数方法就是describe(),这个函数用来创建一个包含所有相关的测试的规格说明，我们称之为测试套件。

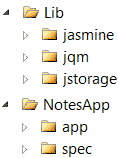
　　其中，一个测试规格说明是通过函数方法it()中声明的。当调用it()方法时，要传递一个字符串参数进去，比如这里，我们期待代码能创建并返回一个NoteModel的实例。

　　接下来，我们通过 var note = Notes.app.createNote();创建了NoteModel的实例，然后使用了expect()函数这个断言(ASSERT，在单元测试中我们称之为断言)，这里是断言判断note是否为Notes.model.NoteModel的一个实例，并使用toBeThruthy()这个匹配器去判断这个测试用例是否能正确通过测试。关于匹配器请参考(<https://github.com/pivotal/jasmine/wiki/Matchers>)。

　　小结一下，在it()这个函数中，开发者可以编写相关的代码建立测试用例，并可以多次调用expect()方法。

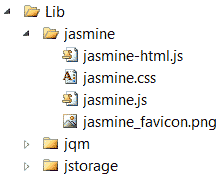
**Jasmine的使用步骤**

　　下面我们开始正式使用Jasmine 框架，首先我们来看下其目录结构。首先我们为项目创建一个名为NoteApp的文件夹。在这个主文件夹下，再创建一个名为app的目录，这个目录中将存放的是程序的应用逻辑和表示层的页面文件。创建一个名为spec的目录，这个目录中存放的是Jasmine的测试套件。除此之外，我们还创建一个lib的目录，这个目录保存的是项目中要用到的一些目录，整个目录架构如下图：

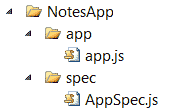


　　在上图中，读者可能奇怪jqm和jstorage两个文件夹存放的是什么，jqm存放的是jquery mobile的框架文件，而jstorage存放的是jstorage框架的文件，在接下来的学习中将会详细介绍。

　　接着，我们可以将下载到的jasmine框架的文件解压后，放到jasmine目录下，如下图：



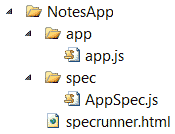
　　接下来，我们开始编写应用的第一个代码文件，一个是名为app.js,放在app目录下，另外一个是jasmine的测试程序，文件为Appsec.js，文件的保存位置如下图：



　　最后，我们将编写一个名为specrunner.html的文件，这个文件中将会运行调用jasmine的测试用例。这个文件实际在jasmine的下载包中存在，但我们要对其进行一些必要的修改，以使其引用到恰当的类库文件，代码如下：

      <!-- HEAD -->  
   <!-- Jasmine includes -->  
    css" rel="stylesheet" type="text/css" />  
<script type="text/javascript"  
src="../lib/jasmine/jasmine-html.js" type="text/javascript">  
</script>  
    <!-- Source files -->  
    <script src="app/App.js" type="text/javascript"></script>  
    <!-- Spec files -->  
    <script src="spec/AppSpec.js" type="text/javascript"></script>  
   <!-- BODY -->   
   <script type="text/javascript">    
    jasmine.getEnv().addReporter(new jasmine.TrivialReporter());    
    jasmine.getEnv().execute();    
</script>

　　最后，我们把这个文件放到恰当的位置，如下图：



**开始编写逻辑代码**

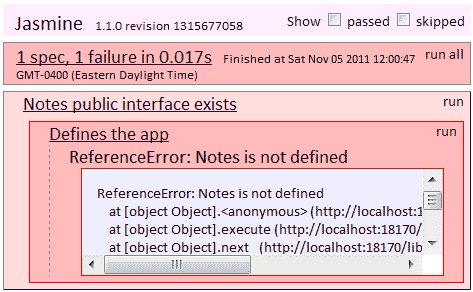
　　接下来，我们开始使用jasmine框架以BDD驱动方式编写主要的代码。BDD的方式需要我们不断编写和修改单元测试用例AppSec.js,让我们就从AppSec.js这个文件开始吧。

　　首先，我们必须断言命名空间是存在的，所以可以在AppSec.js文件中，编写如下代码：

   describe("Public interface exists", function () {  
    it("Defines the app", function () {  
        expect(Notes.app).toBeDefined();  
    });  
});

　　注意我们在一个名为Public interface exists的测试套件中，编写了测试说明用例。在这个测试套件中，将保存所有对业务逻辑的公共函数测试的用例，这里只是简单去断言应用的命名空间是否已经定义。

　　接下来我们开始进行测试，使用浏览器中打开specrunner.html这个文件，可以看到如下图的结果：



　　可以看到，出现了错误的提示：Notes is not defined。这是正确的，因为我们的代码中还没定义命名空间（？），接下来我们编写代码修正之，如下：

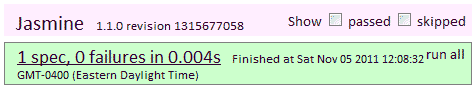
　　var Notes = Notes || {}

　　Notes.app = (function () {

　　return {}

　　})();

　　这里我们定义了应用的命名空间，再次运行jasmine测试框架，结果如下图：



　　现在我们可以总结出其步骤：先定义出行为，然后为行为编写测试用例，然后运行测试，看测试是否通过，不通过的话就修改代码，再运行测试直到其再次通过，这就是典型的BDD开发方法。

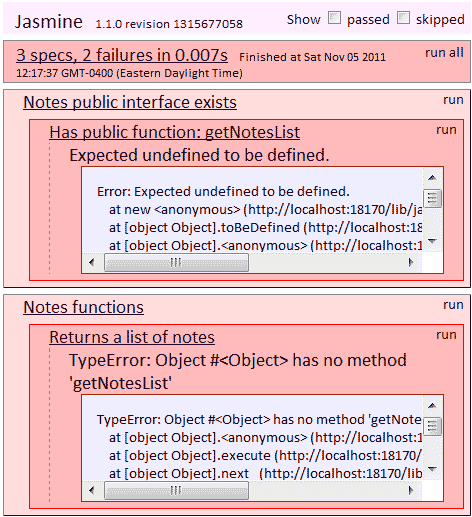
　　接下来，我们需要将一些记事的内容列表由业务逻辑端返回给前端的表现层，因此我们使用如下的jasmine的行为规格说明去定义，继续往测试套件中增加如下代码：

　　describe("Public interface exists", function () {  
　　it("Should have public interface to return notes list", function () {  
　　expect(Notes.app.getNotesList).toBeDefined();  
　　});  
　　});

　　这个测试用例断言测试是否已经定义了显示记事列表的方法getNotesList。再编写另外一个测试用例，稍微复杂点的，如下：

　　describe("Public interface implementation", function () {  
　　it("Should return notes list", function () {  
　　var notesList = Notes.app.getNotesList();  
　　expect(notesList instanceof Array).toBeTruthy();  
　　});  
　　});

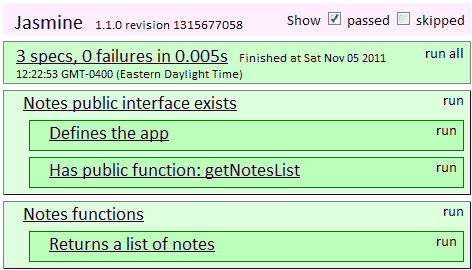
　　这个测试用例，主要用来断言测试返回的记事列表是否是一个数组。 注意我们现在新建立了一个测试套件,名为 Public interface implementation,在这个测试套件中,专门存放的是针对接口实现的测试用例。再次在浏览器中运行，可以看到如下图：



　　可以看到，两个测试用例都出错了，没关系，我们马上编写代码修正：

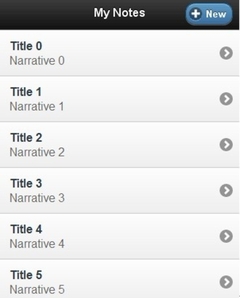
　　Notes.app = (function ()   
  
　　var notesList = [];  
　　function getNotesList() {  
　　return notesList;  
　　}  
　　return {  
　　getNotesList: getNotesList  
　　}  
　　})();

　　再次运行测试，结果如下图，这次我们通过了测试。



**基于JQM的移动记事本（2）**

将会看到如下的一个制作好的记事本界面的样子，如下图：



　　下面我们开始对单元测试用例中编写修改的代码进行不断的重构，以达到我们的目的，这样的一个好处是能够很高效并且不编写多余的代码。

**编写数据上下文模块**

　　我们在第一篇教程中，使用jasmine编写了测试套件，第一个测试套件是如下所示：

   describe("Public interface exists", function () {  
    it("Should have public interface to return notes list", function () {  
        expect(Notes.app.getNotesList).toBeDefined();  
    });  
});

　　第二个测试套件则为：

   describe("Public interface implementation", function () {  
    it("Should return notes list", function () {  
        var notesList = Notes.app.getNotesList();  
        expect(notesList instanceof Array).toBeTruthy();  
    });  
})

　　现在我们要对它们进行重构，尝试把它们整合为一个测试套件即可，如下代码所示：

  describe("Data Context tests", function () {  
    it(“Exists in the app”, function () {  
        expect(Notes.dataContext).toBeDefined();  
    });  
    it(“Returns notes Array”, function () {  
        var notesList = Notes.dataContext.getNotesList();  
        expect(notesList instanceof Array).toBeTruthy();  
    });  
});

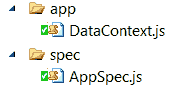
　　在第一个测试用例it()方法中，我们将应用app的实例参数改为dataContext,也就是凡是跟数据访问有关系的方法，都会放在这个类的实例中。在教程的第一篇中，app变量是如下的代码：

   Notes.app = (function () {  
    var notesList = [];  
    function getNotesList() {  
        return notesList;  
    }  
    return {  
        getNotesList: getNotesList  
    }  
})();

　　当我们更换名称后，代码如下：

Notes.dataContext = (function () {  
    var notesList = [];  
    function getNotesList() {  
        return notesList;  
    }  
    return {  
        getNotesList: getNotesList  
    }  
})();

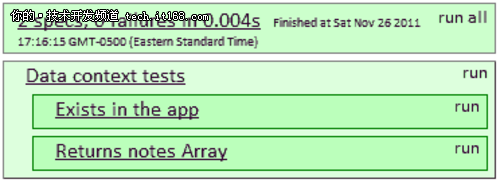
　　同时，我们将App.js更名为DataContext.js,在更改后，目录文件如下图所示：



　　当然，我们还要修改specrunner.html,将app.js修改为DataContext.js，如下：

<!-- App -->  
<script type="text/javascript" src="app/DataContext.js"></script>

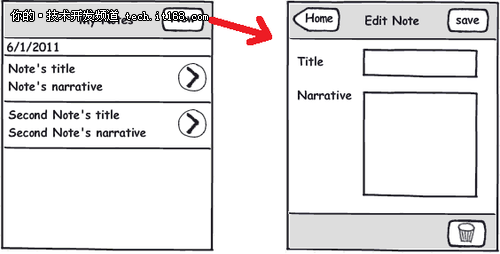
　　现在我们再次运行Jasmine,打开specrunner.html,然后运行，可以看到如下的结果：



　　这证明测试已经通过，简而言之，这里就是说我们将应用的模块名称改名了，叫做DataContext并且通过了单元测试。

**创建空白的笔记**

　　接下来，我们开始创建一个空白笔记的界面。界面的设计草图如下所示：



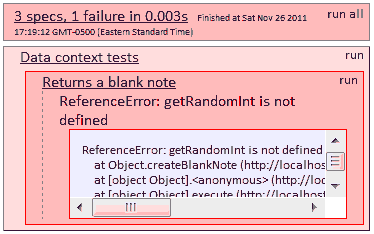
　　我们可以通过BDD测试描述这个行为，在测试用例中，添加如下代码：

it("Returns a blank note", function () {  
    var blankNote = Notes.dataContext.createBlankNote();  
    expect(blankNote.title.length === 0).toBeTruthy();  
    expect(blankNote.narrative.length === 0).toBeTruthy();  
});

　　如果我们现在运行这个测试，测试结果肯定不能通过，因为没有将createBlankNote方法添加到DataContext的测试模块中，修改后的代码如下：

Notes.dataContext = (function () {  
    var notesList = [];  
    function createBlankNote() {  
        var dateCreated = new Date();  
        var id = new String(dateCreated.getTime()) + new String(getRandomInt(0, 100));  
        var noteModel = new Notes.NoteModel({  
            id: id,  
            dateCreated: dateCreated,  
            title: "",  
            narrative: ""  
        });  
        return noteModel;  
    }  
    function getNotesList() {  
        return notesList;  
    }  
    return {  
        createBlankNote: createBlankNote,  
        getNotesList: getNotesList  
    };  
})();

　　在上面的代码中，会发现其中调用了getRandomInt这个助手类方法，它的作用是在创建新的记事时生成其id编号，由于我们还没编写getRandomint这个方法，所以运行测试时，可以看到是如下的出错界面：



　　接下来我们编写getRandomint的代码，如下：

function getRandomInt(min, max) {  
    return Math.floor(Math.random() \* (max - min + 1)) + min;  
}

　　再次运行测试后，依然发现有错误，这次的错误是说我们还没定义NoteModel类的实例，接下来我们在应用目录中创建NoteModel.js文件，代码如下：

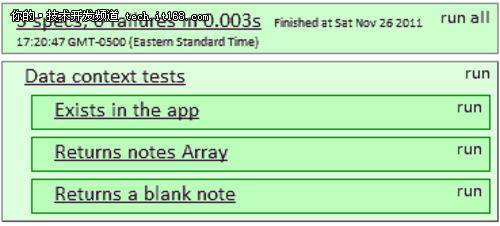
  Notes.NoteModel = function (id, dateCreated, title, narrative) {  
    this.id = id;  
    this.dateCreated = dateCreated;  
    this.title = title;  
    this.narrative = narrative;  
}

　　所有的跟记事相关的操作都必须使用到这个类的实例，包括将记事缓存在移动设备中(用到的是将该类的实例的一个序列化)。

　　我们同样要在specrunner.html中，加上对NoteModel.js的引用，如下：

<!-- App --><script type="text/javascript" src="app/DataContext.js"></script>  
<script type="text/javascript" src="app/NoteModel.js"></script>

　　再次运行测试，可以看到这次测试结果运行正确了，如下图：



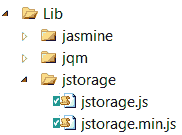
**对Local Storage的操作**

　　我们将使用HTML5新的特性localStorage API去保存记事内容。简单来说，HTML5的这一新特性localStorage是基于键值对操作的，其类型都为字符串。为了能将NoteModel类实例序列化，需要一个序列化机制将记事本数组转变为字符串，以方便保存在local storage中。同样，也要需要反序列化的机制将已序列化的记事重新转变为NoteModel实例，这是需要在用到NoteModel类时要用到。

　　为了方便操作，我们使用一个localStorage API的抽象层去实现，幸好有jStorage这个插件可以帮我们实现这个序列化和反序列化的操作。jStorage插件的地址如下可以下载：

　　http://www.jstorage.info/

　　jStorage插件允许我们去缓存数字、字符串和对象，在下载解包jStorage后，将相关的文件复制到如下图所示的目录中去，



　　同时，在dataContext模块中，我们将创建一个方法，从localStorage中获得保存过的记事对象，如下代码：

   function loadNotesFromLocalStorage() {  
    var storedNotes = $.jStorage.get(notesListStorageKey);  
    if (storedNotes !== null) {  
        notesList = storedNotes;  
    }  
}

　　同时我们也定义了notesListStorageKey这个变量，如下：

　　var notesList = [];

　　var notesListStorageKey = "Notes.NotesList";

　　其中的loadNotesFromLocalStorage()方法，通过使用了jStorage插件从本地存储中获得了已缓存的记事列表，而插件本身完成了从字符转变为数组实例的工作，并且赋值给数组变量notesList。

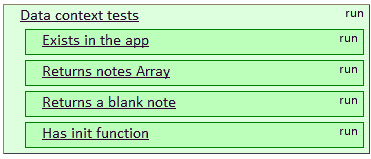
　　当应用运行的时候，要把已经缓存的记事内容呈现给呈现给用户，这就要求应用在运行时必须调用loadNotesFromLocalStorage()这个方法。为了在一开始调用这些相关的方法，我们再编写一个init()助手方法，首先同样是在AppSpec.js中进行驱动测试的代码编写：

it("Has init function", function () {  
    expect(Notes.dataContext.init).toBeDefined();  
});

　　接下来在DataContext.js中，添加如下代码：

function init() {  
    loadNotesFromLocalStorage();  
}  
.. // Other functions…  
return {  
    init: init,  
    createBlankNote: createBlankNote,  
    getNotesList: getNotesList  
};

　　运行上面的测试，可以看到如下图能通过所有的测试：



**测试获得记事列表**

　　接下来，我们看下测试如何将记事保存到localstorage中去，并且放到一个助手方法中去，命名为TestHelper.js文件，代码如下：

  Notes.testHelper = (function () {  
    function createDummyNotes() {  
        var notesListStorageKey = "Notes.NotesList";  
        var notesCount = 10;  
        var notes = [];  
        for (var i = 0; i < notesCount; i++) {  
            var note = Notes.dataContext.createBlankNote();  
            note.title = "Title " + i;  
            note.narrative = "Narrative " + i;  
            notes.push(note);  
        }  
        $.jStorage.set(notesListStorageKey, notes);  
    };  
    return {  
        createDummyNotes: createDummyNotes  
    }  
})();

　　在上面的代码中，将调用Notes.dataContext.createBlankNote()方法生成记事记录内容，并将其插入到notes数组中，最后通过jStorage插件，将notes数组保存到本地设备存储中去。

　　同时，在specrunner.html中，也要加入TestHelper.js的引用，如下：

<!-- App -->  
<script type="text/javascript" src="app/DataContext.js"></script>  
<script type="text/javascript" src="app/NoteModel.js"></script>  
<!-- Test Helper -->  
<script type="text/javascript" src="spec/TestHelper.js"></script>  
<!-- Spec -->  
<script type="text/javascript" src="spec/AppSpec.js"></script>

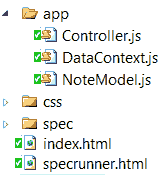
　　而在AppSpec.js中，我们可以增加一个测试断言如下：

it("Returns dummy notes saved in local storage", function () {  
    Notes.testHelper.createDummyNotes();  
      Notes.dataContext.init();  
    var notesList = Notes.dataContext.getNotesList();  
    expect(notesList.length > 0).toBeTruthy();  
});

　　在上面这个测试用例中，首先调用了testHelper助手类中的createDummyNotes创建了记事内容，并且调用了在init()方法，注意在init()方法中，也调用了loadNotesFromLocalStorage()方法，将记事从设备的localStorage中获取，再断言判断反序列化后的notesList是否为空。

**创建页面**

　　接下来，我们开始创建相关的页面，首先将设计首页index.html,其目录结构如下：

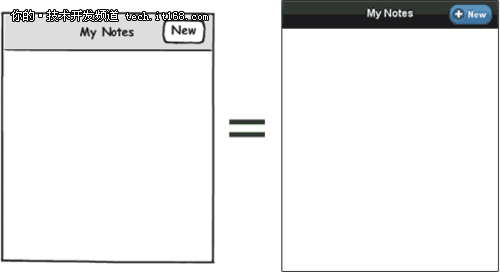


　　其中index.html代码如下：

<html>  
<head>  
    <title></title>  
    <meta name="viewport" content="width=device-width, initial-scale=1">  
    <link href="../../lib/jqm/jquery.mobile-1.0.min.css" rel="stylesheet"type="text/css" />  
    <link href="css/app.css" rel="stylesheet" type="text/css" />  
    <script src="../../lib/jqm/jquery-1.6.4.min.js" type="text/javascript"></script>  
    <script src="../../lib/jstorage/jstorage.min.js" type="text/javascript"></script>  
    <script src="app/DataContext.js" type="text/javascript"></script>  
    <script src="app/Controller.js" type="text/javascript"></script>  
    <script src="app/NoteModel.js" type="text/javascript"></script>  
    <script src="spec/TestHelper.js" type="text/javascript"></script>  
    <!—--- 这里添加相关的启动事件----->  
    <script src="../../Lib/jqm/jquery.mobile-1.0.min.js" type="text/javascript"></script>  
</head>  
<body>  
    <div data-role="page" id="notes-list-page" data-title="My Notes">  
        <div data-role="header" data-position="fixed">  
            <h1>  
                My Notes</h1>  
            <a href="#note-editor-page" class="ui-btn-right" data-theme="b" data-icon="plus">New</a>  
        </div>  
        <div data-role="content" id="notes-list-content">  
        </div>  
    </div>  
<</body>  
</html>

　　在上面的代码中，首先引用了jQuery Mobile的类库和相关的之前我们编写的各种js,注意我们留空了启动事件的部分暂时没编写，这部分的代码会在稍后进行编写，凡是希望应用在启动的时候进行相关初始化操作的代码，都可以在这里编写。

　　接下来代码中，就是使用jQuery mobile的模版，设置了页面的布局，其中分别用到了jQuery Mobile内置的样式模版定义了header部分和主体部分，如下图：



　　接下来，我们开始补充页面初始化时加载的脚本，如下代码：

<meta&nbsp;name="viewport"&nbsp;content="width=device-width,&nbsp;initial-scale=1">  
<link&nbsp;href="../../lib/jqm/jquery.mobile-1.0.min.css"&nbsp;rel="stylesheet"&nbsp;type="text/css"&nbsp;/>  
<link&nbsp;href="css/app.css"&nbsp;rel="stylesheet"&nbsp;type="text/css"&nbsp;/>  
<script&nbsp;src="../../lib/jqm/jquery-1.6.4.min.js"&nbsp;type="text/javascript"></script>  
<scriptsrc="../../lib/jstorage/jstorage.min.js" type="text/javascript"></script>  
<script src="app/DataContext.js" type="text/javascript"></script>  
<script src="app/Controller.js" type="text/javascript"></script>  
<script src="app/NoteModel.js" type="text/javascript"></script>  
<script src="spec/TestHelper.js" type="text/javascript"></script>  
<script type="text/javascript">  
$(document).bind("mobileinit",function(){  
//Notes.testHelper.createDummyNotes();  
Notes.controller.init();  
});  
</script>  
<scriptsrc="../../Lib/jqm/jquery.mobile-1.0.min.js" type="text/javascript"></script>

　　要注意的是，我们在加载jQuery Mobile框架前，编写了mobileinit事件的代码，这个事件会在jQuery Mobile框架执行前进行相关的事件处理(具体可以参考：<http://jquerymobile.com/test/docs/api/globalconfig.html> )。

　　在这个事件中会执行相关的函数调用，首先是调用TestHelper助手模块中的createDummyNotes()方法创建记事，接下来的的Notes.controller.init()是属于控制层的代码，我们稍后再编写。

　　通过以上过程的学习，你会发现我们的开发过程是遵守MVC架构的，其中的NoteModel类和DataContext数据模块可以被看作模型层，而jQuery Mobile则作为表现层的呈现，接下来开始编写控制层代码。

**创建控制层**

　　我们将控制层的文件命名为Control.js,先定义一个空的控制层文件如下：

　　Notes.controller = (function ($, dataContext) {

　　})(jQuery, Notes.dataContext);

　　现在，我们需要的是控制层将缓存在本地存储的记事列表罗列出来，在控制层中，将会调用init方法，这个init方法会在页面加载时，调用mobileinit方法会间接调用到。代码如下：

Notes.controller = (function ($, dataContext) {  
    function init() {  
        dataContext.init();  
        var d = $(document);  
        d.bind("pagechange", onPageChange);  
    }  
    return {  
        init: init  
    }  
})(jQuery, Notes.dataContext);

　　在控制层中的init()方法首先触发了dataContext中的init()方法，这很容易理解，因为我们必须获得数据访问的上下文，接下来可能大家会认为在控制层的init()方法中会列出在缓存中的记事列表，这个想法没错，但我们会通过在jQuery Mobile的pagechange事件中(相关详细介绍见http://jquerymobile.com/demos/1.0/docs/pages/page-scripting.html)进行绑定，这样有个好处，就是不单在应用启动的时候显示记事的列表，而且是当用户从编辑记事等其他页面返回主页时也能显示这些记事列表。下面我们往控制层中增加这个onPageChange()方法如下：

Notes.controller = (function ($, dataContext) {  
    var notesListSelector = "#notes-list-content";  
var noNotesCachedMsg = "</pre>  
<div>No notes cached</div>  
<pre>";  
    var notesListPageId = "notes-list-page";  
    var currentNote = null;  
    function init() {  
        dataContext.init();  
        var d = $(document);  
        d.bind("pagechange", onPageChange);  
    }  
    function onPageChange(event, data) {  
        var toPageId = data.toPage.attr("id");  
        switch (toPageId) {  
            case notesListPageId:  
                renderNotesList();  
                break;  
        }  
    }  
    return {  
        init: init  
    }  
})(jQuery, Notes.dataContext);

　　处理的过程其实很简单，就是在onPageChange方法中，通过检查即将要跳转的页面的id属性(通过data.toPage方法)，如果是等于“notes-list-page”，则调用renderNotesList()方法即可。

　　接下来，我们编写renderNotesList()方法，代码如下：

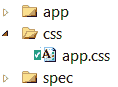
function renderNotesList() {  
var notesList = dataContext.getNotesList();  
var view = $(notesListSelector);  
view.empty();  
if (notesList.length === 0) {  
$(noNotesCachedMsg).appendTo(view);  
} else {  
var notesCount = notesList.length;  
var note;  
var ul = $("<ul id=\"notes-list\" data-role=\"listview\"></ul>").appendTo(view);  
for (var i = 0; i < notesCount; i++) {  
note = notesList[i];  
$("<li>"  
+ "<a data-url=\"index.html#note-editor-page?noteId=" + note.id + "\" href=\"index.html#note-editor-page?noteId=" + note.id + "\">"  
+ "<div>" + note.title + "</div>"  
+ "<div class=\"list-item-narrative\">" + note.narrative + "</div>"  
+ "</a>"  
+ "</li>").appendTo(ul);  
}  
ul.listview();  
}  
};

　　在上面的代码中，首先通过data context的getNotesList()方法，获得了记事列表，并且将view区域(是个div)的内容清除，然后判断记事内容是否为空，如果为空的话则显示没有记事内容的提示信息，否则使用一个循环，将每个记事的内容增添到要显示的区域中，注意这里是通过字符串拼接的方式，构造出一个需要符合jQuery Mobile显示格式的字符串。最后还需要调用jQueryMobile的listview()方法才能生成一个完成的listview列表。

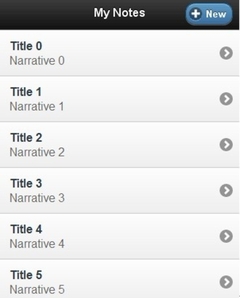
　　并且注意到，在列表中我们使用到了list-item-narrative的CSS样式文件，这需要我们定义如下：

.list-item-narrative  
{  
    color: #666666;  
    font-weight: normal;  
}

　　并且将这个CSS文件保存在css目录下，命名为app.css，如下图：



　　最后，我们在手机浏览器上运行index.html，可以看到如下的界面：



**小结**

　　在这一讲中，我们继续使用单元测试驱动的方法，初步构建了模型层、控制层和表示层三层的需要的文件，读者可以从中体会到MVC的开发模式在移动开发中的应用，在下一讲中，我们将开始设计新增、编辑和删除记事文件。

**基于JQM的移动记事本（3）**

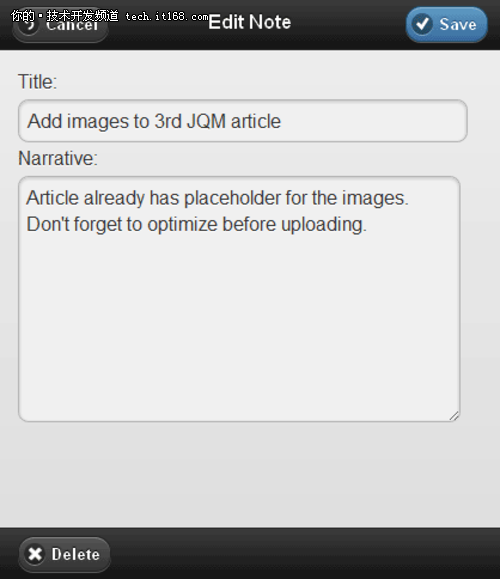
在上一讲中，我们学习了如何通过jasmine的BDD驱动开发框架，一边编写单元测试用例一边按照MVC的架构编写了数据模型层，控制层和表现层的基本代码。在本文中，将继续介绍如下的几个方面内容。

　　1、如何在记事列表页面和记事编辑页面之间通过URL和参数传递信息。

　　2、如何在记事的编辑页面中加载一个记事内容进行编辑。

　　3、如何保存或者编辑一个记事内容。

　　我们预期完成本讲后，能设计出如下功能的界面：



**步骤一、重构**

　　下面我们开始重构之旅。之前我们的数据访问上下文在调用jStorage插件时，是直接引用了jQuery的类库的。我们现在进行一点修改，当访问数据模块dataContext时，将jQuery作为参数进行传递，如下：

　　Notes.dataContext = (function ($) {

　　// ……….

　　} (jQuery));

　　除此以外，我们将之前在程序中对在localstorage中保存的键值的名称去掉，原来是这样的：

　　var notesListStorageKey = "Notes.NotesList";

　　我们则可以通过init方法将这个值以参数的形式传递到dataContext中，如下所示：

　　var init = function (storageKey) {

　　notesListStorageKey = storageKey;

　　loadNotesFromLocalStorage();

　　};

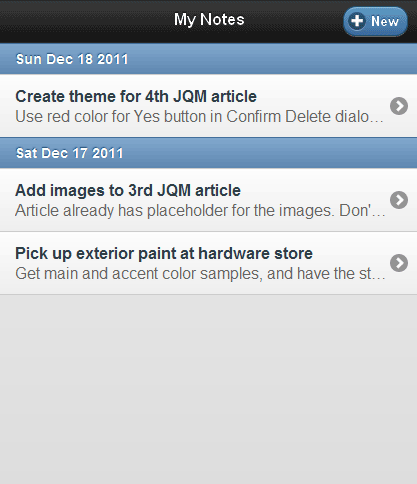
　　当然，这个测试用例也很简单，我们稍微修改一下就可以了，注意修改的是

　　AppSpec.js文件，如下代码所示：

describe("Data Context tests", function () {  
    var notesListStorageKey = "Notes.NotesListTest";  
    // 省略其他部分  
    it("Returns dummy notes saved in local storage", function () {  
        Notes.testHelper.createDummyNotes();  
         //从localstorage中加载记事  
        Notes.dataContext.init(notesListStorageKey);  
        var notesList = Notes.dataContext.getNotesList();  
        expect(notesList.length > 0).toBeTruthy();  
    });  
});

**按日期对记事进行分组**

　　为了能让用户更方便地在记事列表中寻找记事,我们将记事按日期进行分组,效果如下图:



　　代码如下：

var renderNotesList = function () {  
    var notesList = dataContext.getNotesList();  
    var view = $(notesListSelector);  
    view.empty();  
    if (notesList.length === 0) {  
        $(noNotesCachedMsg).appendTo(view);  
    } else {  
        var liArray = [],  
            notesCount = notesList.length,  
            note,  
            dateGroup,  
            noteDate,  
            i;  
        var ul = $("<ul id=\"notes-list\" data-role=\"listview\"></ul>").appendTo(view);  
        for (i = 0; i < notesCount; i += 1) {  
            note = notesList[i];  
            noteDate = (new Date(note.dateCreated)).toDateString();  
            if (dateGroup !== noteDate) {  
                liArray.push("<li data-role=\"list-divider\">" + noteDate + "</li>");  
                dateGroup = noteDate;  
            }  
            liArray.push("<li>"  
                + "<a data-url=\"index.html#note-editor-page?noteId=" + note.id + "\" href=\"index.html#note-editor-page?noteId=" + note.id + "\">"  
                + "<div  class=\"list-item-title\">" + note.title + "</div>"  
                + "<div class=\"list-item-narrative\">" + note.narrative + "</div>"  
                + "</a>"  
                + "</li>");  
        }  
        var listItems = liArray.join("");  
        $(listItems).appendTo(ul);  
        ul.listview();  
    }  
};

　　如果学习过第2讲的教程，应该对上面循环产生记事的过程不陌生。但注意到的是，其中使用了变量noteDate保存了每个记事的日期，当判断出每个记事的所属日期不同的时候，则重新生成一个分隔项(其data-role为list-divider)，而其内容为另外一个新的日期(详见

liArray.push("<li data-role=\"list-divider\">" + noteDate + "</li>");）。

此外，请注意学习如何为每个记事生成编辑页面的链接，即：

   <a data-url=\"index.html#note-editor-page?noteId=" + note.id + "\" href=\"index.html#note-editor-page?noteId=" + note.id + "\">"

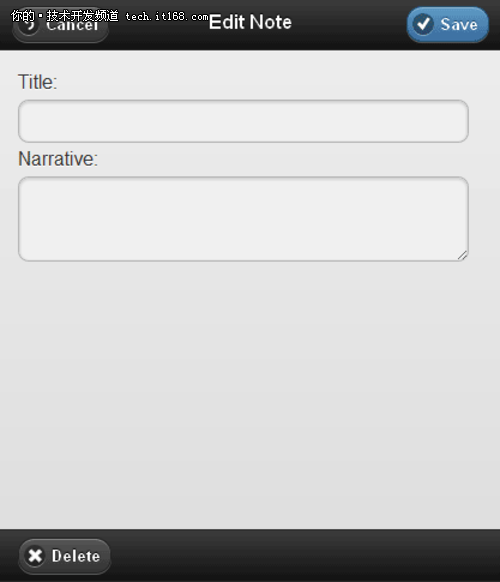
　　这个note-editor-page就是我们要新增或者编辑记事的页面，下面开始学习如何设计

**新增加记事页面设计**

　　在新增加的记事页面中，注意我们是在index.html中增加相关的代码，并且注意其页面的data-role属性为page,代码如下：

  <div data-role="page" id="note-editor-page" data-title="Edit Note">  
    <div data-role="header" data-position="fixed">  
        <a href="#notes-list-page" data-icon="back" data-rel="back">Cancel</a>  
        <h1>  
            Edit Note</h1>  
        <a id="save-note-button" href="" data-theme="b" data-icon="check">Save</a>  
    </div>  
    <div data-role="content">  
        <form action="" method="post" id="note-editor-form">  
        <label for="note-title-editor">  
            Title:</label>  
        <input type="text" name="note-title-editor" id="note-title-editor" value="" />  
        <label for="note-narrative-editor">  
            Narrative:</label>  
        <textarea name="note-narrative-editor" id="note-narrative-editor"></textarea>  
        </form>  
    </div>  
    <div data-role="footer" data-position="fixed" class="ui-bar">  
        <a id="delete-note-button" data-icon="delete" data-transition="slideup" data-rel="dialog">Delete</a>  
    </div>  
</div>

　　其页面效果如下图：



　　其中的cancel按键用于返回记事列表用。

**装载记事内容到记事页面**

　　下面，我们看下如何装载记事到记事编辑器中。当用户在记事列表中点某个记事时，我们分为下面几个步骤进行设计：

　　在notesList数组中，根据在链接中的li元素中的id去找出对应的记事。

　　在编辑界面的标题和内容文本框中，将读取到的记事的标题和内容放置到相应的文本框中。

　　最后让记事编辑器页面处于激活状态

　　而如果用户是点新建记事按钮，则执行相类似的步骤，只不过不用先根据notedid在记事列表中找出对应的记事而已，而是直接获得用户提交的记事标题和内容再予以保存。下面先看下如何实现装载已有的记事内容到记事编辑页中。

　　首先，我们要在控制器的声明部分中，定义如下的页面变量：

　　var noteEditorPageId = "note-editor-page";

　　接下来，我们在onPageChange()方法中编写相关的事件处理代码，专门用于处理在记事列表页面跳转到记事编辑页面：

var onPageChange = function (event, data) {  
    var toPageId = data.toPage.attr("id");  
    var fromPageId = null;  
    if (data.options.fromPage) {  
        fromPageId = data.options.fromPage.attr("id");  
    }  
    switch (toPageId) {  
        case notesListPageId:  
            resetCurrentNote();  
            renderNotesList();  
            break;  
       case noteEditorPageId:  
            if (fromPageId === notesListPageId) {  
                renderSelectedNote(data);  
            }  
            break;  
    }  
};

　　下面解析下上面的代码，其中我们定义了fromPageId这个参数，今后我们简称为源页面，而toPageId则称之为目标页面。其中fromPageId的值会判断是否在记事编辑界面中加载记事内容：

　　if (fromPageId === notesListPageId) {  
　　renderSelectedNote(data);  
　　}

　　这段代码判断了如果源页面来自记事列表页面，则通过renderSelectedNote方法读取具体的某个记事内容(这个方法稍后编写)。要注意的是，在应用启动后，pagechange 事件就发生了，但这个时候源页面为空，所以必须通过下面的代码，首先获得页面的id,如下：

　　if (data.options.fromPage) {  
　　fromPageId = data.options.fromPage.attr("id");  
　　}

　　接下来，我们编写核心的从记事列表中装载记事的代码，即renderSelectedNote方法：

var renderSelectedNote = function (data) {  
    var u = $.mobile.path.parseUrl(data.options.fromPage.context.URL);  
    var re = "^#" + noteEditorPageId;  
    if (u.hash.search(re) !== -1) {  
        var queryStringObj = queryStringToObject(data.options.queryString);  
        var titleEditor = $(noteTitleEditorSel);  
        var narrativeEditor = $(noteNarrativeEditorSel);  
        var noteId = queryStringObj["noteId"];  
        if (typeof noteId !== "undefined") {  
                  var notesList = dataContext.getNotesList();  
            var notesCount = notesList.length;  
            var note;  
            for (var i = 0; i < notesCount; i++) {  
                note = notesList[i];  
                if (noteId === note.id) {  
                    titleEditor.val(note.title);  
                    narrativeEditor.val(note.narrative);  
                    currentNote = note;  
                }  
            }  
        } else {  
         //新建一个记事，设置标题文本框和记事文本框初始内容为空  
            titleEditor.val("");  
            narrativeEditor.val("");  
        }  
        titleEditor.focus();  
    }  
};

　　在上面的代码中，首先判断来源页面是否为记事列表页面，其中使用了如下的通过正则表达式的方法判断：

var u = $.mobile.path.parseUrl(data.options.fromPage.context.URL);  
    var re = "^#" + noteEditorPageId;  
    if (u.hash.search(re) !== -1) {  
    …    }

　　这里通过了jQuery Mobile的parseUrl方法(见http://jquerymobile.com/test/docs/api/methods.html)首先获得了来源的URL，存放到变量u中，并且通过正则表达式的判断方法进行判断。接着我们创建一个对象，用来存放相对应的来源URL，这通过一个助手类queryStringToObjec

　　t来实现，代码如下：

    var queryStringToObject = function (queryString) {  
    var queryStringObj = {};  
    var e;  
var a = /\+/g;    
var r = /([^&;=]+)=?([^&;]\*)/g;  
    var d = function (s) { return decodeURIComponent(s.replace(a, " ")); };  
    e = r.exec(queryString);  
    while (e) {  
        queryStringObj[d(e[1])] = d(e[2]);  
        e = r.exec(queryString);  
    }  
    return queryStringObj;  
};

　　然后在使用时，通过var queryStringObj = queryStringToObject(data.options.queryString);进行调用，获得的是上一个页面传进来的字符串参数对象，再通过var noteId = queryStringObj["noteId"];获得了要查看编辑的记事的notedId。再来分析以下这段代码：

  if (typeof noteId !== "undefined") {  
       var notesList = dataContext.getNotesList();  
    var notesCount = notesList.length;  
    var note;  
    for (var i = 0; i < notesCount; i++) {  
        note = notesList[i];  
        if (noteId === note.id) {  
            currentNote = note;  
            titleEditor.val(currentNote.title);  
            narrativeEditor.val(currentNote.narrative);  
        }  
    }  
} else {  
        titleEditor.val("");  
    narrativeEditor.val("");  
}

　　这里首先判断notedId是否已定义，如果不为空且已定义，则证明为加载编辑记事，通过dataContext数据模块加载记事列表，然后通过循环找出notedId为要查看记事id的记事

　　，注意这里使用currentNode变量保存了要查看的notes对象，这样就为后面增加和删除提供了方便，不用再次在noteList数组中再寻找。

　　运行后，当点记事列表中的某个记事时，则会出现具体的记事内容，如下图：



**新增加记事内容**

　　接下来我们开始设计新增记事的界面和代码，首先定义一个保存按钮的标识，如下：

　　var saveNoteButtonSel = "#save-note-button";

　　并在控制层中的init()方法中如下编写代码：

var init = function () {  
    dataContext.init("Notes.NotesList");  
    var d = $(document);  
    d.bind("pagebeforechange", onPageBeforeChange);  
    d.bind("pagechange", onPageChange);  
    d.delegate(saveNoteButtonSel, "tap", onSaveNoteButtonTapped);  
};

　　这里通过委托定义了用户在点新增按钮时，将触发onSaveNoteButtonTapped事件，代码如下：

     var onSaveNoteButtonTapped = function () {  
       var titleEditor = $(noteTitleEditorSel);  
    var narrativeEditor = $(noteNarrativeEditorSel);  
    var tempNote = dataContext.createBlankNote();  
    tempNote.title = titleEditor.val();  
    tempNote.narrative = narrativeEditor.val();  
    if (tempNote.isValid()) {  
        if (null !== currentNote) {  
            currentNote.title = tempNote.title;  
            currentNote.narrative = tempNote.narrative;  
        } else {  
            currentNote = tempNote;  
        }  
        dataContext.saveNote(currentNote);  
        returnToNotesListPage();  
    } else {  
      //告知用户保存失败，此处代码省略，留待下一讲完成    }  
};

　　在上面的代码中，首先调用了NoteModel的isValid方法，用来判断要新增的记事是否已经完整添写了标题和内容，代码如下：

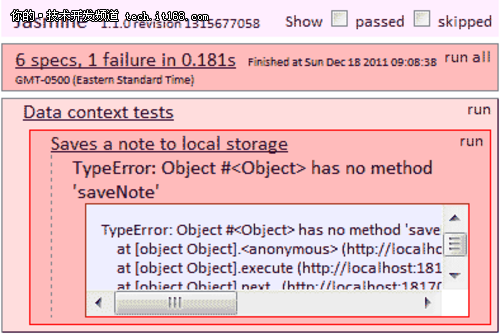
Notes.NoteModel.prototype.isValid = function () {  
    "use strict";  
    if (this.title && this.title.length > 0) {  
        return true;  
    }  
    return false;  
};

　　接下来，我们通过观察currentNote变量的值，去判断是否在编辑记事还是新增记事。首先通过dataContext.createBlankNote()创建一个临时的记事对象tempNote,然后在判断输入合法后，判断是否是更新还是新增记事。如果currentNote变量不为空(证明是更新一个已存在的记事)，则将currentNote的标题和内容简介设置为tempNote的标题和内容简介,否则是新增一个空的记事让currentNote设置为等于tempNote。再调用dataContext的saveNote方法，

　　这个方法还没定义，但我们先在AppSec文件中编写相关的测试用例。

   it("Saves a note to local storage", function () {  
    //确认在测试前localstorage被清空  
    $.jStorage.deleteKey(notesListStorageKey);  
    var notesList = $.jStorage.get(notesListStorageKey);  
    expect(notesList).toBeNull();  
     // 创建一个记事  
    var dateCreated = new Date();  
    var id = dateCreated.getTime().toString();  
    var noteModel = new Notes.NoteModel({  
        id: id,  
        dateCreated: dateCreated,  
        title: ""  
    });  
    Notes.dataContext.init(notesListStorageKey);  
    Notes.dataContext.saveNote(noteModel);  
    notesList = $.jStorage.get(notesListStorageKey);  
    var expectedNote = notesList[0];  
expect(expectedNote instanceof Notes.NoteModel).toBeTruthy();  
    // 清除notesListStorageKey)  
    $.jStorage.deleteKey(notesListStorageKey);  
});

测试后结果如图所示：



编写代码如下：

var saveNote = function (noteModel) {  
    var found = false;  
    var i;  
    for (i = 0; i < notesList.length; i += 1) {  
        if (notesList[i].id === noteModel.id) {  
            notesList[i] = noteModel;  
            found = true;  
            i = notesList.length;  
        }  
    }  
    if (!found) {  
        notesList.splice(0, 0, noteModel);  
    }  
    saveNotesToLocalStorage();  
};

　　由于我们使用的是数组，因此跟传统的数据库保存有点不同，这里可能显得有点复杂。我们遍历整个notesList数组，如果是找到跟要编辑的记事对象，则直接设置found(找到标志)为true,并且覆盖原对象数组的内容，如果found为false,则找不到对象为新增，将新增的对象放置到数组头部。

　　再定义saveNotesToLocalStorage方法如下：

　　var saveNotesToLocalStorage = function () {

　　$.jStorage.set(notesListStorageKey, notesList);

　　};

　　将记事列表保存到local storage中去。

　　为了能在编辑或者新增记事后，能返回到记事列表页，特增加了

　　returnToNotesListPage方法，如下：

　　var returnToNotesListPage = function () {

　　$.mobile.changePage("#" + notesListPageId,

　　{ transition: "slide", reverse: true });

　　};

　　最后，为了能在新增或者删除记事后，都能将currentNote变量重新设置，需要增加方法resetCurrentNote，代码为：

　　var resetCurrentNote = function () {

　　currentNote = null;

　　}

　　这样，我们就完成了新增一个记事及编辑一个记事内容的工作，注意到我们在保存记事时，如果遇到异常情况，我们本讲没处理，将留待下一讲进行讲授。

**基于JQM的移动记事本（4）**

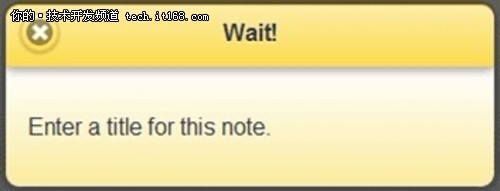
在本讲中，将会实现如下的几个目标：

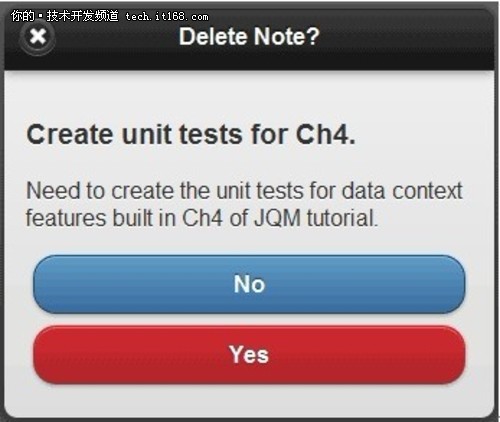
　　1、当用户在新增记事时，如果记事内容不完整，系统应给予用户友好的提示。

　　2、实现删除按钮的功能。

　　3、创建自定义的样式，改变系统默认的对话框的样式。

　　我们能期望达到如下的效果：



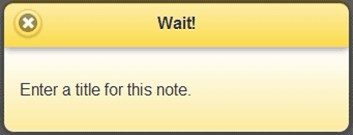


**验证输入记事的合法性**

　　在上一讲中，当我们要新保存新的记事时，只是针对标题和内容在保存前进行了是否为空的校验，回忆下相关代码如下：

　　Notes.NoteModel.prototype.isValid = function () {  
　　"use strict";  
　　if (this.title && this.title.length > 0) {  
　　return true;  
　　}  
　　return false;  
　　};

　　并且在onSaveNoteButtonTapped的方法中，我们上一讲并没有对校验失败时，应该如何提示通知用户，所以这里我们使用jQuery mobile框架中的对话框，其中有一个标题和内容文本提示即可，如下图：



**如何设计jQuery Mobile对话框**

　　在jQuery Mobile中，使用对话框很简单，只需要添加data-role属性为dialog即可，在index.html中添加如下代码：

<!--Invalid Note dialog-->  
<div id="invalid-note-dialog" data-role="dialog" data-title="Invalid Note" data-theme="e">  
<div data-role="header" data-theme="e">  
<h1>Wait!</h1>  
</div>  
<div data-role="content">Enter a title for this note.</div>  
</div>

　　接下来，我们还要在控制层中添加对话框的标识，

var invalidNoteDlgSel = "#invalid-note-dialog";  
   我们将修改onSaveNoteButtonTapped事件的代码如下：  
  var onSaveNoteButtonTapped = function () {  
    // 校验记事  
    var titleEditor = $(noteTitleEditorSel);  
    var narrativeEditor = $(noteNarrativeEditorSel);  
    var tempNote = dataContext.createBlankNote();  
    tempNote.title = titleEditor.val();  
    tempNote.narrative = narrativeEditor.val();  
    if (tempNote.isValid()) {  
        if (null !== currentNote) {  
            currentNote.title = tempNote.title;  
            currentNote.narrative = tempNote.narrative;  
        } else {  
            currentNote = tempNote;  
        }  
        dataContext.saveNote(currentNote);  
        returnToNotesListPage();  
    } else {  
        $.mobile.changePage(invalidNoteDlgSel, defaultDlgTrsn);  
    }  
};

　　这里，请注意$.mobile.changePage(invalidNoteDlgSel, defaultDlgTrsn);

　　这句。在这里，我们使用了 $mobile.changePage方法，这个方法能动态地触发页面改变。这个方法有两个参数，第一个参数是要出现的页面的id(这里就是对话框的id),而第二个参数则是页面切换的效果，这里使用的过渡切换效果如下定义：

　　var defaultDlgTrsn = { transition: "slideup" };

**mobileinit事件的放置位置**

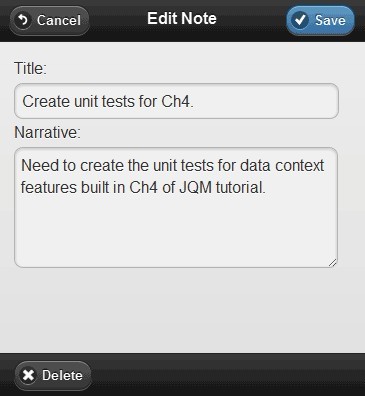
　　在上一讲中，我们在index.html中编写了mobileinit事件的代码，但这个习惯不是很好，我们不希望在html页中出现跟逻辑相关的代码，因此我们重构一下，将mobileinit事件的代码放到Controller.js这个文件中进行处理，代码如下：

　　Notes.controller = (function ($, dataContext, document) {  
　　//省略了详细的contoller代码，具体见附件  
　　} (jQuery, Notes.dataContext, document));  
　　$(document).bind("mobileinit", function () {  
　　Notes.controller.init();  
　　});

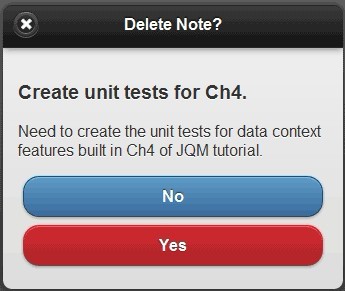
　　现在，index.html代码中，则只是引用了相关的JS文件了，不会再象上一讲中，在index.html中混杂了js代码。

**设计删除记事功能**

　　接下来，我们要设计删除某一个记事的功能，设计好的界面如下图：



　　当用户按delete按钮后，弹出如下界面：



　　当用户选No时，则返回查看记事具体内容的界面，如果选择的是YES，则删除一条记事。

　　首先我们在index.html中，设计删除信息提示的界面，如下：

<div id="confirm-delete-note-dialog" data-role="dialog" data-title="Delete Note">  
<div data-role="header">  
<h1>Delete Note?</h1>  
</div>  
<div data-role="content">  
<div id="delete-note-content-placeholder"></div>  
<a id="cancel-delete-note-button" data-role="button" data-theme="b" data-rel="back">No</a>  
<a id="ok-to-delete-note-button" data-role="button" data-theme="f">Yes</a></div>  
</div>

　　注意我们使用了data-role=”dialog”属性，这个之前谈到的一样，让框架输出一个对话框风格的页面。之后在Yes和No的按钮中，注意使用了data-role=”button”的方法，并指出了每个按钮的data-theme的风格样式，其中b和f都是jQuery Mobile内置的CSS样式，当然用户可以自己修改。

　　跟之前一样，我们要在控制层中控制这个删除界面的行为，同样首先定义删除界面的标识如下：

　　var confirmDeleteNoteDlgSel = "#confirm-delete-note-dialog";

　　再将在编辑记事页面中的删除按钮以及删除界面中的确认按钮、在删除确认界面中的一个DIV都定义一个别名标识，如下：

　　var deleteNoteButtonSel = "#delete-note-button",

　　deleteNoteContentPlaceholderSel = "#delete-note-content-placeholder",

　　okToDeleteNoteButtonSel = "#ok-to-delete-note-button";

　　接下来，我们编写删除按钮的基本事件框架如下：

　　var init = function () {  
　　// 其余省略的代码  
　　d.delegate(deleteNoteButtonSel, "tap", onDeleteNoteButtonTapped);  
　　};  
　　其中onDeleteNoteButtonTapped代码如下：  
　　var onDeleteNoteButtonTapped = function () {  
　　if (currentNote) {  
　　var noteContentPlaceholder = $(deleteNoteContentPlaceholderSel);  
　　noteContentPlaceholder.empty();  
　　$("  
　　" + currentNote.title + "  
　　" + currentNote.narrative + "  
　　").appendTo(noteContentPlaceholder);  
　　$.mobile.changePage(confirmDeleteNoteDlgSel, defaultDlgTrsn);  
　　}  
　　};

　　在onDeleteNotebuttonTapped()方法中，首先我们读取了当前即将要被删除的记事，然后将其添加到placeholder这个div中，并且再使用changePage方法让当前界面成为焦点界面，这个之前的论述是一样的。

　　接下来，再分别看下当用户点NO或YES时，如何进行事件的处理。当用户点NO按钮时，

　　页面的代码为：

<a id="cancel-delete-note-button" data-role="button" data-theme="b" data-rel="back">No</a>

　　其中的data-rel=”back”是内置的功能，能自动回退到上一个页面，而这正是我们所需要的。

　　而在YES按钮的事件处理中，当用户点YES后，需要删除记事后再带用户回到记事列表，并且刷新此时的记事列表。同样在控制层中的init方法中加入如下代码：

    var init = function () {  
    //省略其他代码  
    d.delegate(okToDeleteNoteButtonSel, "tap", onOKToDeleteNoteButtonTapped);  
};  
接下来定义onOKToDeleteNoteButtonTapped方法，如下：  
  var onOKToDeleteNoteButtonTapped = function () {  
    dataContext.deleteNote(currentNote);  
    returnToNotesListPage();  
};

　　这个很容易理解，其中调用了dataContext中的deleteNote方法删除一个真正的记事，然后再调用returnToNotesListPage()返回记事列表。在编写deleteNote方法前，先继续编写单元测试用例如下：

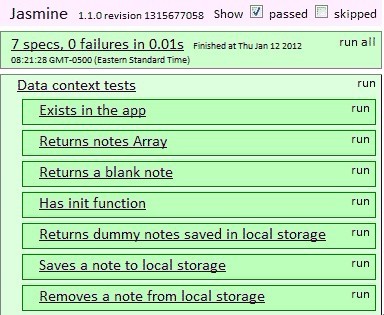
     it("Removes a note from local storage", function () {  
        // 创建记事  
        var dateCreated = new Date();  
        var id = new String(dateCreated.getTime());  
        var noteModel = new Notes.NoteModel({  
            id: id,  
            dateCreated: dateCreated,  
            title: "",  
            narrative: ""  
        });  
        // 清空记事列表  
        var notesList = [];  
        // 添加到localstorage本地存储  
        notesList.push(noteModel);  
        $.jStorage.set(notesListStorageKey, notesList);  
        notesList = $.jStorage.get(notesListStorageKey);  
        expect(notesList.length).toEqual(1);  
        // 删除记事  
        Notes.dataContext.init(notesListStorageKey);  
        Notes.dataContext.deleteNote(noteModel);  
        // 删除后应该返回空的列表，断言判断  
        notesList = $.jStorage.get(notesListStorageKey);  
        expect(notesList.length).toEqual(0);  
        // 清除相关资源  
        $.jStorage.deleteKey(notesListStorageKey);  
    });  
    运行测试后当然结果是失败的，因为没有编写deleteNote方法。因此编写代码如下：  
var deleteNote = function (noteModel) {  
  
    var i;  
    for (i = 0; i < notesList.length; i += 1) {  
        if (notesList[i].id === noteModel.id) {  
            notesList.splice(i, 1);  
            i = notesList.length;  
        }  
    }  
    saveNotesToLocalStorage();  
};

　　这里就是通过循环遍历数组的方法，根据notedId找出要删除的元素，然后通过数组的splice方法进行删除，删除后记得调用saveNotesToLocalStorage();

　　方法进行保存。为了方便在模块的接口处进行调用，我们添加到controller.js中：

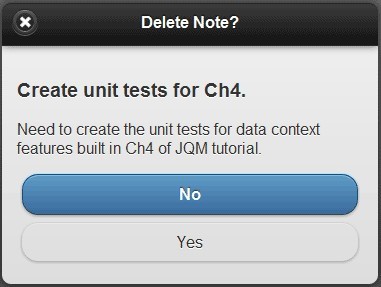
  var pub = {  
    init: init,  
    createBlankNote: createBlankNote,  
    getNotesList: getNotesList,  
    saveNote: saveNote,  
    deleteNote: deleteNote  
};

　　再运行测试，可以看到顺利通过。如下图：



**更换Jquery Mobile的自定义样式**

　　在完成上面的工作后，可以看到默认的删除界面的按钮样式，如果不经过改变，是如下的样子的：



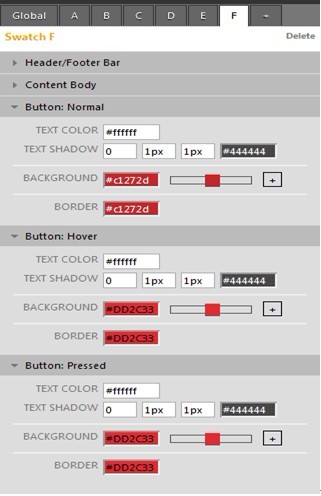
　　如果要改变其样式，这就要需要使用jQuery Mobile的自定义样式。回忆一下之前的代码，在按钮中：

  <a id="ok-to-delete-note-button" data-role="button" data-theme="f">Yes</a>

　　其中的data-theme=“f”，要求我们定义一个新的名为f的样式，这个可以通过

　　jQuery Mobile提供的在线样式编辑器实现，地址为：<http://jquerymobile.com/themeroller/>

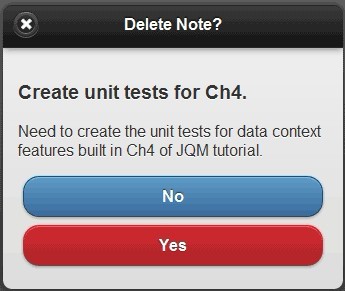
　　(需要在chrome,firefox下使用)。该设计器如下图：



　　用户可以根据自己的喜好，对所有的按钮，文字，超链接进行样式的自定义，十分方便，在定义好后，可以使用其download的功能，把样式导出下载，就可以把样式增加到我们的app.css中，如下代码：

.ui-btn-up-f {  
    border: 1px solid #c1272d /\*{f-bup-border}\*/;  
    background: #c1272d /\*{f-bup-background-color}\*/;  
    font-weight: bold;  
    color: #ffffff /\*{f-bup-color}\*/;  
    text-shadow:  0  /\*{f-bup-shadow-x}\*/  1px  /\*{f-bup-shadow-y}\*/  1px  /\*{f-bup-shadow-radius}\*/ #444444 /\*{f-bup-shadow-color}\*/;  
    background-image: -webkit-gradient(linear, left top, left bottom, from( #D42A31 /\*{f-bup-background-start}\*/), to( #AD2328 /\*{f-bup-background-end}\*/)); /\* Saf4+, Chrome \*/  
    background-image: -webkit-linear-gradient(top, #D42A31 /\*{f-bup-background-start}\*/, #AD2328 /\*{f-bup-background-end}\*/); /\* Chrome 10+, Saf5.1+ \*/  
    background-image:    -moz-linear-gradient(top, #D42A31 /\*{f-bup-background-start}\*/, #AD2328 /\*{f-bup-background-end}\*/); /\* FF3.6 \*/  
    background-image:     -ms-linear-gradient(top, #D42A31 /\*{f-bup-background-start}\*/, #AD2328 /\*{f-bup-background-end}\*/); /\* IE10 \*/  
    background-image:      -o-linear-gradient(top, #D42A31 /\*{f-bup-background-start}\*/, #AD2328 /\*{f-bup-background-end}\*/); /\* Opera 11.10+ \*/  
    background-image:         linear-gradient(top, #D42A31 /\*{f-bup-background-start}\*/, #AD2328 /\*{f-bup-background-end}\*/);  
}  
.ui-btn-up-f a.ui-link-inherit {  
    color: #ffffff /\*{f-bup-color}\*/;  
}  
  
.ui-btn-hover-f {  
    border: 1px solid #DD2C33 /\*{f-bhover-border}\*/;  
    background: #DD2C33 /\*{f-bhover-background-color}\*/;  
    font-weight: bold;  
    color: #ffffff /\*{f-bhover-color}\*/;  
    text-shadow:  0  /\*{f-bhover-shadow-x}\*/  1px  /\*{f-bhover-shadow-y}\*/  1px  /\*{f-bhover-shadow-radius}\*/ #444444 /\*{f-bhover-shadow-color}\*/;  
    background-image: -webkit-gradient(linear, left top, left bottom, from( #F33038 /\*{f-bhover-background-start}\*/), to( #C6272D /\*{f-bhover-background-end}\*/)); /\* Saf4+, Chrome \*/  
    background-image: -webkit-linear-gradient(top, #F33038 /\*{f-bhover-background-start}\*/, #C6272D /\*{f-bhover-background-end}\*/); /\* Chrome 10+, Saf5.1+ \*/  
    background-image:    -moz-linear-gradient(top, #F33038 /\*{f-bhover-background-start}\*/, #C6272D /\*{f-bhover-background-end}\*/); /\* FF3.6 \*/  
    background-image:     -ms-linear-gradient(top, #F33038 /\*{f-bhover-background-start}\*/, #C6272D /\*{f-bhover-background-end}\*/); /\* IE10 \*/  
    background-image:      -o-linear-gradient(top, #F33038 /\*{f-bhover-background-start}\*/, #C6272D /\*{f-bhover-background-end}\*/); /\* Opera 11.10+ \*/  
    background-image:         linear-gradient(top, #F33038 /\*{f-bhover-background-start}\*/, #C6272D /\*{f-bhover-background-end}\*/);  
}  
.ui-btn-hover-f a.ui-link-inherit {  
    color: #ffffff /\*{f-bhover-color}\*/;  
}  
.ui-btn-down-f {  
    border: 1px solid #DD2C33 /\*{f-bdown-border}\*/;  
    background: #DD2C33 /\*{f-bdown-background-color}\*/;  
    font-weight: bold;  
    color: #ffffff /\*{f-bdown-color}\*/;  
    text-shadow:  0  /\*{f-bdown-shadow-x}\*/  1px  /\*{f-bdown-shadow-y}\*/  1px  /\*{f-bdown-shadow-radius}\*/ #444444 /\*{f-bdown-shadow-color}\*/;  
    background-image: -webkit-gradient(linear, left top, left bottom, from( #C6272D /\*{f-bdown-background-start}\*/), to( #F33038 /\*{f-bdown-background-end}\*/)); /\* Saf4+, Chrome \*/  
    background-image: -webkit-linear-gradient(top, #C6272D /\*{f-bdown-background-start}\*/, #F33038 /\*{f-bdown-background-end}\*/); /\* Chrome 10+, Saf5.1+ \*/  
    background-image:    -moz-linear-gradient(top, #C6272D /\*{f-bdown-background-start}\*/, #F33038 /\*{f-bdown-background-end}\*/); /\* FF3.6 \*/  
    background-image:     -ms-linear-gradient(top, #C6272D /\*{f-bdown-background-start}\*/, #F33038 /\*{f-bdown-background-end}\*/); /\* IE10 \*/  
    background-image:      -o-linear-gradient(top, #C6272D /\*{f-bdown-background-start}\*/, #F33038 /\*{f-bdown-background-end}\*/); /\* Opera 11.10+ \*/  
    background-image:         linear-gradient(top, #C6272D /\*{f-bdown-background-start}\*/, #F33038 /\*{f-bdown-background-end}\*/);  
}  
.ui-btn-down-f a.ui-link-inherit {  
    color: #ffffff /\*{f-bdown-color}\*/;  
}

　　　最后，可以看到效果为下图，实现了样式的变更



**小结**

　　在本文中，实现了对记事的删除的相关操作，介绍了如何在jQuery Mobile中实现对话框的功能。