第 11 章

构建基于 HTML5 的生活轨迹 Web App

本章我们将通过构建 Web App 例子，介绍如何运用 Sencha Touch 搭建移动 Web 用户界面，并采用 HTML5 离线特性构建离线应用，HTML5 本地存储特性作为本地数据库，以及 Geolocation 和 Google 地图 API 接口实现记录当前地理位置信息。

11.1 项目背景

现在，越来越多的移动端应用程序都提供了地理定位的功能，并且也可以公开或选择性地向好友分享你的位置。然而，这些功能都是基于移动设备的原生客户端应用程序，它们都需要程序的安装、账户注册等才能正常使用地理位置分享功能。对于移动端的 Web 应用程序呢？Geolocation 技术就是为了解决 Web 端获取当前地理位置信息的。

本示例 Web App 应用程序，提供一种全新的 Web 应用体验方式，用户无须安装客户端应用程序。它的运行环境是基于支持 HTML5 标准的 Web 浏览器，无论用户当前网络是否在线，都可以通过 Web 应用程序记录事件或日记、地理定位（需要有网络状态下）、离线应用等功能。

11.1.1 功能介绍

基于 HTML5 的生活轨迹 Web 应用程序功能非常简单，其功能类似于记事本，但它比记事本多了一种地理定位的功能。

本实例的主要功能是提供给用户创建生活轨迹内容，并且允许获取当前用户所在地理位置的坐标信息。同时，用户还可以浏览生活轨迹列表，如果记录的生活轨迹内容保存着当前地理位置信息，浏览时便以地图的方式展示，否则以文本形式显示。

生活轨迹 Web App 应用程序使用的技术要点包括：Sencha Touch、Geolocation API、离线应用、LocalStorage 本地存储等。

Web App 既支持 IE 9、Chrome、Safari、FireFox、Opera 等桌面浏览器，也支持 iPhone、Android、BlackBerry、iPad 等智能移动设备。

1．Sencha Touch

Sencha Touch 是一套基于 HTML5 的富应用程序框架，主要提供创建移动 Web 应用程序所需的用户界面库，并且还可以通过业务逻辑实现各种界面之间的转换。

本实例将采用 Sencha Touch 的一项新特性：使用 MVC 开发模式编写 Web 应用程序。数据模型、数据展示及业务逻辑操作代码各自独立，因此，代码将变得更容易维护。

2．Geolocation

Web App 应用程序将主要使用 HTML5 的 Geolocation 技术获取用户当前的地理位置信息，并将读取的经纬度信息标记在 Google 地图上。

3．localStorage

对于数据的存取，我们采用 HTML5 标准的本地存储 LocalStorage 对象记录数据。在示例中，我们并没有采用 localStorage 对象提供的原生 API，而是采用由 Sencha Touch 封装的 LocalStorage 对象存取数据。

4．离线应用

本实例的最后，我们增加了一种新功能特性：在网络离线状态下也可访问 Web 应用程序。这个功能特性是采用 HTML5 标准中的离线应用，我们将大部分资源文件存储在用户的 Web 浏览器客户端中，同时使用 manifest 配置文件根据版本的变化而更新资源文件到本地。

由于例子中使用到 Google 地图 API 接口，而 API 不属于我们可以控制的范围，而且 Google 地图图片数据需要实时读取，并将地理位置信息标记在地图上，因此当用户的网络处于离线状态时，Web 应用程序将无法读取 Google 地图数据。

11.1.2　功能模块

本实例 Web App 应用程序的代码文件组织结构将按照 Sencha Touch 推荐使用的 MVC 模式文件目录结构，如图 11-1 所示。

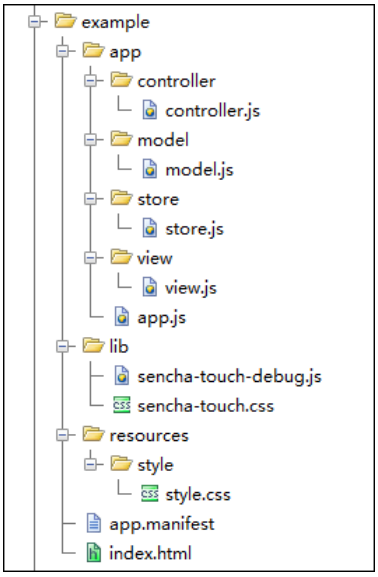

图 11-1　实例代码目录结构

现在，我们介绍一下图 11-1 中所示的目录结构的主要作用。

- example 目录是本实例 Web 应用程序的整个目录，目录下共有三个子目录：app、lib、resources。同时还包括两个主要文件：index.html 和 app.manifest。index.html 为整个 Web App 的首页；app.manifest 则是离线应用的配置文件。
- app 目录主要存放 Sencha Touch 的 MVC 应用程序 JavaScript 文件。这部分代码是整个 Web App 的核心代码，其中包括数据模型、视图界面及业务逻辑。app 目录下还包含一个 app.js，这个文件将是 Sencha Touch 应用程序的入口。
- controller 目录主要存放 Sencha Touch 应用程序的业务逻辑代码文件，这部分业

务代码还包括获取地理位置信息、获取地图数据等功能。

- model 目录主要存放 Sencha Touch 应用程序中数据模型 Model 类代码文件。
- store 目录主要存放 Sencha Touch 应用程序的数据 Store 对象文件。虽然这个目录不是 Sencha Touch 推荐使用，并且建议目录中的 Store 对象存放在 Model 目录内，但作者觉得在采用 MVC 模式开发 Web App 时 Store 对象作为数据操作对象，其功能类似于 JDBC（Java Data Base Connectivity，Java 数据库连接）的功能，最好将其独立分开。开发人员则可以根据实际情况而自定义此目录。
- view 目录主要存放 Sencha Touch 应用程序定义的视图界面文件。几乎所有的界面如 Panel 组件都是在这里定义的。
- lib 目录存放的是 Sencha Touch 的 JavaScript 源码库和 CSS 源码库。本实例采用的是一个 Sencha Touch 开发模式的 JavaScript 代码库，这个 JavaScirpt 代码库是未经过特殊工具的压缩，这方便我们在开发过程中调试编写的程序代码。
- resource 目录主要存放开发 Web App 应用程序时的自定义图片、CSS 样式文件等。

11.2 创建基本应用程序

在正式开始基于 HTML5 的 Web App 之前，我们需要准备 index.html，并导入 Sencha Touch 类库和样式文件。同时，由于用到 Sencha Touch 的 MVC 开发模式，我们也需要在 app.js 中创建一个 application 对象作为整个 Web App 的根节点对象。

11.2.1 创建首页

创建一个 HTML5 标准文档格式 HTML 文件，并将 JavaScrip 类库、自定义 JavaScript 文件、CSS 文件导入到文件，并命名为 index.html，如代码 11-1 所示。

在代码 11-1 中，关于 Sencha Touch 的外部资源文件导入的顺序，需要注意以下几点。

（1）由于 sencha-touch-debug.js 文件是 Sencha Touch 框架基础类库，所有基于该框架库开发的 JavaScript 文件，其导入的位置必须在 Sencha-Touch-debug.js 文件后面。

（2）app.js 是整个 Web App 的 JavaScript 开始文件。因此 app.js 导入的位置必须在 sencha-touch-debug.js 文件后面，同时也必须在 model.js、store.js、view.js、controller.js 文件前面。

（3）对于使用 MVC 模式中的 model.js、store.js、view.js、controller.js 这 4 个文件，一般情况下 controller 业务类都位于底层，其余文件导入的位置则根据实际情况而定。

代码 11-1　index.html 代码

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="UTF-8">
    <title>我的生活轨迹</title>
    <link rel="stylesheet"type="text/css"
href="lib/sencha-touch.css" />
<link rel="stylesheet" type="text/css"
href="resources/style/style.css" />
    <script type="text/javascript" src="lib/sencha-touch-debug.js">
</script>
<script type="text/javascript"
src="http://maps.google.com/maps/api/js?sensor=true">
</script>
<script type="text/javascript" src="app/app.js"></script>
<script type="text/javascript" src="app/model/model.js">
</script>
<script type="text/javascript" src="app/store/store.js">
</script>
<script type="text/javascript" src="app/view/view.js">
</script>
<script type="text/javascript"
src="app/controller/controller.js">
</script>
</head>
<body>
</body>
</html>
```

11.2.2　创建入口函数

1．app.js

我们在前面多次提及 app.js 这个文件，app.js 作为整个 Sencha Touch MVC 应用程序的入口，属于 Web App 的核心功能 JS 文件，如代码 11-2 所示。该文件内只有一个创建

Application 对象的方法，其中参数 name 设置“app”作为命名空间名称，同时 launch 参数设置应用程序的初始化函数，这个函数将是整个应用程序的入口函数。

代码 11-2　app.js 文件代码

```
Ext.regApplication({
    /* 定义应用程序名称，并作为命名空间 */
    name : 'app',
    /* 默认目标名称 */
    defaultTarget : 'viewport',
    /* 初始化函数 */
    launch : function(){
        this.viewport = new app.views.viewport();
    }
});
```

2．主视图 viewport

从代码 11-2 可以看到，我们在 launch 初始化函数内定义了一个 viewport 变量，并且将变量指向继承 Ext.Panel 的自定义组件对象实例。

app.views.viewport 是由 Sencha Touch 定义的命名空间变量格式，同时也是所有视图组件的父级对象，任何子视图以及其他组件对象都定义在其属性 items 内。由于我们是使用 Ext.Extend 方法继承 Panel 组件的方式创建一个新对象的，因此 app.views.viewport 只是一个对象，而非对象实例。创建实例仍然需要配合 new 关键字，如 new app.views.viewport()。

app.views.viewport 对象的定义在 view.js 文件内，代码如下：

```
app.views.viewport = Ext.extend(Ext.Panel,{
    id:'viewport',
    fullscreen:true,
    layout:'card',
      //默认显示第一个 item 项
    activeItem:0,
    items:[
        {
            xtype:'listPanel'
        },{
            xtype:'noteForm'
        },{
            xtype:'viewNote'
```

```
        }
    ],
    initComponent : function(){
         app.views.viewport.superclass.
                 initComponent.apply(this,arguments);
        }
});
```

其中 fullScreen、layout 两个属性作为父视图 viewport 是必须定义的，这两个属性用于设置自适应整个应用程序的界面大小及子视图的显示方式。

定义 id 属性主要为后面实现视图切换时可以通过 id 值获取该组件的实例对象。

在代码中，我们看到 items 属性指定了三种不同名称类型的 xtype，在这里我们先不介绍这些自定义 xtype 类型的组件，后续介绍 View 视图组件的时候会展示这部分代码。

11.3　设置 Model 数据模型

Web App 要实现在 Sencha Touch 中对数据的存取，首先需要创建一个 Model 实体数据模型及用于存取数据操作的 Store 对象。接下来我们将为你介绍如何实现这部分的代码。

11.3.1　创建 Model 实体类

现在我们开始编写 MVC 应用程序。首先在 model.js 中建立一个 model 实体类 note，并存储为命名空间变量 app.models.note，如代码 11-3 所示。

从代码 11-3 中可以看到，我们定义了 proxy 属性参数，并指定 type 类型为 localStorage，表示采用本地存储将数据存储到浏览器中。

代码 11-3　model.js 完整代码

```
app.models.note = Ext.regModel("note",{
    fields:[
       {name:'id',type:'int'},
       {name:'title',type:'string'},
       {name:'content',type:'string'},
       {name:'position',type:'string'},
       {name:'latitude',type:'string'},
       {name:'longitude',type:'string'}
    ],
    /* 使用 localStorage 代理 */
    proxy : {
```

```
        type:'localstorage',
        id:'noteStorage'
    }
});
```

11.3.2 设置 Store 对象

现在我们来定义 Store 组件对象，该对象的作用是数据存取对象。由于本实例只有一个 Model 数据模型，同时处理的业务需求也相对简单，因此我们只实例化一个 Store 对象，并定义到命名空间变量 app.stores.noteStore。

store.js 的完整代码如代码 11-4 所示。

代码 11-4 store.js 完整代码

```
app.stores.noteStore = new Ext.data.Store({
    model:'note',
    id:'noteStore'
});
```

11.4 创建 View 视图组件

上一节我们已经准备了存取数据最基本的组件对象：Model 对象和 Store 实例对象。现在，我们开始创建 Web App 视图界面组件。

11.4.1 列表视图

在 Web App 应用程序实例中，最主要的视图就是内容列表视图，同时也是所有功能模块的入口，它位于主视图 viewport 下的子视图。该视图实现的界面效果如图 11-2 所示。

图 11-2　列表视图

列表视图主要分成两部分：顶部的工具栏模块和列表模块。工具栏模块左右两侧分别有一个按钮，中间是一个标题。列表模块主要是显示生活轨迹内容清单。

定义列表视图的源代码位于 view.js 文件内，代码如下：

```
app.views.listPanel = Ext.extend(Ext.Panel,{
    id:'listPanel',
    layout:'card',
    dockedItems:[{
        xtype:'toolbar',
        dock:'top',
        title:'生活轨迹',
        items:[{
            xtype:'button',
            text:'添加',
            handler:function(){
                app.controllers.appController.showNoteForm();
            }
        },{
            xtype:'spacer'
        },{
            xtype:'button',
            text:'更多功能',
            handler:function(){
                app.controllers.appController
```

```
                .showNoteActionSheet();
            }
        }]
    }],
    items:[{
        xtype:'noteList'
    }],
    initComponent : function(){
         app.views.listPanel.superclass.initComponent
             .apply(this,arguments);
     }
});
Ext.reg('listPanel',app.views.listPanel);
```

11.4.2　列表组件

我们注意到列表视图代码中的 items 属性参数使用了自定义的 xtype 类型：noteList。这个 xtype 类型对象是通过继承 Ext.List 组件对象而创建的新组件对象，并使用 Ext.reg 方法命名 xtype 类型名称。

noteList 列表组件的源代码位于 view.js 文件内，代码如下：

```
app.views.noteList = Ext.extend(Ext.List,{
    store : app.stores.noteStore,
   cls:'noteList',
    itemTpl : '<p class="title">{title}</p><p>{content}</p>',
   onItemDisclosure:{
       scope:this,
       handler:function(record, btn, index){
       }
   },
        initComponent : function(){
            app.stores.noteStore.load();
            app.views.noteList.superclass
              .initComponent.apply(this,arguments);
            this.enableBubble('selectionchange');
        }
});
Ext.reg('noteList',app.views.noteList);
```

从代码中可以看到，在 initComponent 初始化组件函数内有以下一段代码：

```
app.stores.noteStore.load();
```

这段代码的作用是当加载列表组件时，组件自动加载列表视图的数据。因此本 Web App 实例在组件渲染完毕时，会自动加载数据，并立即显示在页面上。

11.4.3　表单视图

表单视图的功能是给用户提供标题、内容的输入以及获取当前所在的位置地理经纬度信息。该视图实现的界面效果如图 11-3 所示。

图 11-3　创建生活轨迹表单视图

表单视图的源代码位于 view.js 文件内，代码如下：

```
app.views.noteForm = Ext.extend(Ext.form.FormPanel,{
    id:'noteForm',
    scroll:'vertical',
    dockedItems:[{
        xtype:'toolbar',
        dock:'top',
        title:'创建',
        items:[{
            xtype:'button',
            text:'提交',
            handler:function(){
            }
        },{
            xtype:'spacer'
        },{
            xtype:'button',
            text:'返回',
            handler:function(){
            }
```

```
        }]
    }],
    items:[
        {
            xtype:'textfield',
            name:'title',
            labelWidth:'0%',
            maxLength:20,
            placeHolder:'请输入标题'
        },{
            xtype:'textareafield',
            name:'content',
            labelWidth:'0%',
            maxLength : 1024,
            maxRows : 10,
            placeHolder:'请输入生活轨迹内容'
        },{
            xtype:'togglefield',
            name:'isPosition',
            label:'是否获取你的当前位置',
            labelWidth:'65%',
            listeners:{
                change:function(slider,thumb,newValue,oldValue){
                }
            }
        },{
            xtype:'hiddenfield',
            id:'latitude',
            name:'latitude'
        },{
            xtype:'hiddenfield',
            id:'longitude',
            name:'longitude'
        }
    ],
    initComponent : function(){
         app.views.listPanel.superclass.initComponent
             .apply(this,arguments);
     }
});
Ext.reg('noteForm',app.views.noteForm);
```

11.4.4 浏览生活轨迹视图

浏览一条列表项的视图主要提供两种界面用于显示该列表项的内容。第一种视图风格是当列表项中没有地理位置信息时，视图就只显示文字内容；第二种视图风格是当列表项中含有地理位置信息时，视图将显示 Google 地图，并在中间位置显示该地理位置标记，当点击地图上的标记时会显示你记录过的生活轨迹内容。

Google 地图视图显示的效果界面如图 11-4 所示。

图 11-4 显示 Google 地图的视图

显示一条内容项的视图源代码位于 view.js 文件内，两种不同风格的视图效果将合并到一个视图组件源码，代码如下：

```
app.views.viewNote = Ext.extend(Ext.Panel,{
    id:'viewNote',
    tpl : new Ext.XTemplate(
        '<header><h1><em>标题：</em>{title}</h1></header>',
         '<p><em>内容：</em>{content}</p>'
    ),
```

```
    dockedItems:[{
        xtype:'toolbar',
        dock:'top',
        title:'生活轨迹',
        items:[{
            xtype:'button',
            text:'返回',
            handler:function(){
            }
        },{
            xtype:'spacer'
        },{
            xtype:'button',
            text:'删除',
            handler:function(){
            }
        }]
    }],
    items:[{
        xtype:'map',
        id:'map',
        mapOptions:{
            zoom : 14
        }
    },{
        xtype:'hiddenfield',
        id:'noteId'
    }],
    initComponent : function(){
         app.views.viewNote.superclass.initComponent
             .apply(this,arguments);
         }
});
```

11.4.5　Sheet 组件选择更多功能

在列表视图中，我们可以看到工具栏右侧有一个“更多功能”按钮。当单击该按钮时，Sencha Touch 会调用由 Ext.ActionSheet 组件创建的实例对象，该对象的作用是显示

更多的功能选择浮动层界面。由于本实例功能简单，现在只提供一个清除所有列表项功能的按钮，用户可以根据实际情况增添更多其他的功能按钮，该功能的界面效果如图 11-5 所示。

图 11-5　Sheet 视图组件

ActionSheet 视图组件的实例对象源代码位于 view.js 文件内，代码如下：

```
app.views.moreActionSheet = new Ext.ActionSheet({
    items : [
        {
            text:'清除所有数据',
            scope:this,
            handler:function(){
            }
        },{
            text:'返回',
            scope:this,
            handler:function(){
            }
        }
    ]
});
```

11.5　业务逻辑

在 11.4 节，我们完成了整个 Web App 应用程序的用户界面视图组件。接下来我们将要实现连接各个用户界面组件的业务逻辑功能。

11.5.1　定义 controller 类

我们先实现一个 controller 实例对象，并在该对象下实现各种业务逻辑功能，也就是 controller.js 代码结构。

代码如下：

```
app.controllers.appController =
        new Ext.regController('appController',{
    //在这里定义各种业务逻辑函数
});
```

11.5.2　实现视图之间的切换

在介绍 View 视图组件的时候，我们一共创建了三个页面视图：列表视图、表单视图、含 Google 地图的视图。那么我们如何实现这些视图之间的切换呢？其奥妙就在以下代码片段内。

1．切换列表视图函数

首先定义一个切换到列表视图功能的函数：showListPanel。

其源代码位于 controller.js 文件内，代码如下：

```
showListPanel:function(){
        Ext.getCmp("viewport").setActiveItem('listPanel',{
        type:'slide',
        direction:'left'
        });
}
```

接着在表单视图的工具栏右侧“返回”按钮的单击或触摸事件中添加 showListPanel

函数，因为它非常符合将表单视图切换回列表视图的业务需求。代码如下：

```
app.views.noteForm = Ext.extend(Ext.form.FormPanel,{
    ……
    dockedItems:[{
        xtype:'toolbar',
        dock:'top',
        title:'创建',
        items:[{
            xtype:'button',
            text:'提交',
            handler:function(){
                app.controllers.appController.saveNote();
            }
        },{
            xtype:'spacer'
        },{
            xtype:'button',
            text:'返回',
            handler:function(){
                app.controllers.appController.showListPanel();
            }
        }]
    ……
});
```

同理，当浏览一条生活轨迹内容视图时，其工具栏左侧的“返回”按钮也符合切换回列表视图的业务需求，因此在该视图的“返回”按钮中可以直接添加该业务逻辑函数，代码如下：

```
app.views.viewNote = Ext.extend(Ext.Panel,{
    ……
        items:[{
            xtype:'button',
            text:'返回',
            handler:function(){
                app.controllers.appController.showListPanel();
            }
        }
    ……
```

```
});
```

2. 切换表单视图函数

切换表单视图函数和切换列表视图函数的实现方法基本相同。

首先定义一个 showNoteForm 函数用于切换到表单视图功能，代码如下：

```
showNoteForm:function(){
        Ext.getCmp('viewport').setActiveItem('noteForm',{
            type:'slide',
       direction:'left'
        });
}
```

我们在列表视图的工具栏右侧提供一个“添加”按钮，该按钮的作用是切换到表单视图页面。同样地，我们在该按钮的单击或触摸事件中直接调用这个 showNoteForm 函数来处理这种业务需求，代码如下：

```
app.views.listPanel = Ext.extend(Ext.Panel,{
    ……
   dockedItems:[{
       xtype:'toolbar',
       dock:'top',
       title:'生活轨迹',
       items:[{
           xtype:'button',
           text:'添加',
           handler:function(){
               app.controllers.appController.showNoteForm();
           }
       }
    ……
});
```

3. 切换含地图的视图函数

同理，在 controller.js 内定义 showViewNote 函数。该函数的主要功能是显示一条内容页面的视图，代码如下：

```
showViewNote:function(){
   Ext.getCmp("viewport").setActiveItem('viewNote',{
```

```
        type:'slide',
        direction:'left'
    });
}
```

11.5.3　保存生活轨迹内容

在表单视图组件的工具栏右侧有一个“提交”按钮，该按钮的主要功能是提交表单视图的所有表单内容，并存储到 localStorage 对象。代码如下：

```
saveNote:function(){
        var form = Ext.getCmp("noteForm");
    var store = app.stores.noteStore;
    var last = store.last();
    var maxId = last==undefined?1:last.data.id+1;
    form.submit({
            waitMsg:'处理中...',
        success:function(){
            app.controllers.appController.showListPanel();
        }
    });
    var m = Ext.ModelMgr.create({id:maxId},'note');
    form.updateRecord(m,false);
    app.stores.noteStore.insert(maxId,m);
    app.stores.noteStore.sync();
    form.reset();
    app.controllers.appController.showListPanel();
}
```

11.5.4　实现 Geolocation 地理定位

如图 11-3 所示，获取地理定位信息是通过一个开关按钮控制的，在默认情况下，该开关按钮是关闭状态，用户需要自行开启并确认获取当前地理位置信息。

在这里，我们在代码中使用开关按钮的 change 事件，实时监听开关按钮的状态变化，以确定地理位置信息是否允许获取，代码如下：

```
{
            xtype:'togglefield',
        name:'isPosition',
```

```
            label:'是否获取你的当前位置',
            labelWidth:'65%',
            listeners:{
                change:function(slider,thumb,newValue,oldValue){
                //从关闭状态切换到开启状态时
                if(newValue==1 && oldValue==0){
                    app.controllers.appController
                        .getCurrentPosition();
                }
                //切换关闭状态
                 if(newValue == 0){
                    app.controllers.appController
                        .clearPosition();
                }
            }
        }
}
```

当开关按钮切换状态时触发 change 事件，事件函数内会调用 controller 实例对象的 getCurrentPosition 函数，该函数的代码如下：

```
getCurrentPosition:function(){
    var geo = new Ext.util.GeoLocation({
        autoUpdate: true,
        listeners: {
            locationupdate: function (geo) {
                Ext.getCmp('latitude')
                    .setValue(geo.coords.latitude);
                Ext.getCmp('longitude')
                    .setValue(geo.coords.longitude);
            },
            locationerror:function(geo,
                                   bTimeout,
                        bPermissionDenied,
                        bLocationUnavailable,
                        message){
            }
        }
    });
    geo.updateLocation();
}
```

clearPosition 函数的主要功能是当用户反选取消不获取位置信息时，程序需要将隐藏域中两个字段值置为空值，代码如下：

```
clearPosition:function(){
    Ext.getCmp('latitude').setValue('');
    Ext.getCmp('longitude').setValue('');
}
```

11.5.5　显示生活轨迹内容

本实例在显示生活轨迹内容视图的时候，采用了两种不同的视图显示类型。当单击列表项右侧图标时将触发 onItemDisclosure 事件，事件将在 handler 函数内调用 controller 实例对象的 showNote 函数，代码如下：

```
app.views.noteList = Ext.extend(Ext.List,{
        ……
   onItemDisclosure:{
       scope:this,
       handler:function(record, btn, index){
           app.controllers.appController
                 .showNote(record, btn, index);
       }
   }
        ……
});
```

showNote 函数的主要功能是判断当前生活轨迹内容是否存在地理位置信息，当不存在地理位置信息时，由视图组件定义的 tpl 模板转化成实际 HTML 代码并显示在视图组件可视区域。

showNote 函数代码位于 controller.js 文件内，代码如下：

```
showNote:function(record, btn, index){
    app.controllers.appController.showViewNote();
    var data = record.data;
    var viewNote = Ext.getCmp('viewNote');
    if(data.latitude == "" || data.longitude == ""){
        Ext.getCmp('viewNote').update(data);
        Ext.getCmp('map').hide();
    }else{
```

```
            app.controllers.appController.showNoteByMap(data);
        }
        Ext.getCmp('noteId').setValue(data.id);
    }
```

代码中通过判断已选中的列表项是否存在 latitude 和 longitude 两个属性值来判断显示哪一种视图显示方式。当两个值为空时，视图实例对象会调用 update 方法更新 tpl 模板，并转化成 HTML 代码显示在视图可见区域，同时隐藏地图组件。

11.5.6 显示 Google 地图

一般情况下，通过 Geolocation API 读取的地理位置信息，latitude 和 longitude 经纬度值是同时存在的，不可能出现其中一个属性为空而另外一个属性不为空的情况。因此无须判断两个值是否同时存在空置等业务逻辑。

在 showNote 函数中，当 latitude 和 longitude 两个属性值不为空时，将直接调用 controller 实例对象内的 showNoteByMap 函数。该函数的主要功能是将列表项的地理经纬度信息标记在 Google 地图上。

showNoteByMap 函数源代码位于 controller.js 文件内，代码如下：

```
showNoteByMap:function(data){
    var map = Ext.getCmp('map');
    map.show();
    var latlng = {latitude:data.latitude,
            longitude:data.longitude};
            //更新地图组件中心坐标位置
            map.update(latlng);
            //创建显示在地图中的信息窗口
            var infowindow = new google.maps.InfoWindow({
                content: '<p>'+data.title+'</p><p>'+data.content+'</p>'
            });
            //创建 google 地图标记的坐标位置对象
            var center = new google.maps
                    .LatLng(data.latitude, data.longitude);
            //创建显示在 Google 地图中的标记
            var marker = new google.maps.Marker({
                position: center,
                title : data.title,
                map: map.map
```

```
        });
        //监听鼠标移动到标记事件或触摸一下标记事件
        google.maps.event.addListener(marker,'mouseover',
            function(){
                infowindow.open(map,marker);
            }
        );
        //监听鼠标移开标记的事件或触摸非标记位置时的事件
        google.maps.event.addListener(marker, 'mouseout',
                function(){
                    infowindow.close();
            }
        );
}
```

代码中主要存在几个功能点：更新地图组件的坐标位置、创建显示地图信息窗口、创建 Google 地图标记、注册标记事件。

代码运行后的效果如图 11-4 所示，当点击或触摸图中的标记时，将会显示如图 11-6 所示的内容窗口。

图 11-6　显示 Google 地图

11.5.7 显示 Sheet 组件函数

本实例的列表视图中提供了一个选择更多功能的按钮，该按钮有别于其他切换视图的业务逻辑，它提供一种 Action Sheet 组件视图功能，用于给用户提供一种额外的区域让用户选择更多功能。

由于在此之前已经定义好了 Sheet 视图组件，现在创建一个函数用于显示该 Sheet，我们将其命名为 showNoteActionSheet，然后在函数内调用该实例对象的 show 方法将其显示在页面上。

showNoteActionSheet 函数源代码位于 controller.js 文件内，代码如下：

```
showNoteActionSheet:function(){
    app.views.moreActionSheet.show();
}
```

然后，在列表视图的“更多功能”按钮中添加单击或触摸事件函数调用 showNoteActionSheet 函数，代码如下：

```
{
    xtype:'button',
    text:'更多功能',
    handler:function(){
        app.controllers.appController.showNoteActionSheet();
    }
}
```

11.5.8 清除所有存储的列表函数

从 Sheet 组件中显示的视图来看，我们提供了一个清除所有列表项数据的功能按钮和“返回”按钮。两个按钮都在自己的单击或触摸事件中添加各自的业务逻辑函数，代码如下：

```
app.views.moreActionSheet = new Ext.ActionSheet({
    items : [
        {
            text:'清除所有数据',
            scope:this,
            handler:function(){
                app.controllers.appController.removeAllNote();
            }
        },{
```

```
            text:'返回',
            scope:this,
            handler:function(){
                app.views.moreActionSheet.hide();
            }
        }
    ]
});
```

清除所有列表项数据的业务逻辑是调用 controller 实例对象的 removeAllNote 函数；返回按钮则直接调用 Sheet 实例对象的 hide 方法隐藏。

removeAllNote 函数源代码位于 controller.js 文件内，代码如下：

```
removeAllNote:function(){
    Ext.Msg.confirm("确认","你确认要清除本地所有数据?",function(){
        var count = app.stores.noteStore.getCount();
        for(var i=0;i<count;i++){
            app.stores.noteStore.removeAt(0);
        }
        app.stores.noteStore.sync();
        app.views.moreActionSheet.hide();
    });
}
```

11.5.9　删除一条记录的函数

最后，本实例还提供单独删除一条记录的功能，该功能位于浏览生活轨迹视图的工具栏右侧，效果如图 11-4 所示。

controller 实例对象内的 deleteNote 函数代码如下：

```
deleteNote:function(){
    var noteId = Ext.getCmp('noteId').getValue();
    var store = app.stores.noteStore;
    var record = store.findRecord('id',noteId);
    store.remove(record);
    store.sync();
    app.controllers.appController.showListPanel();
}
```

在“删除”按钮的单击或触摸事件中调用 controller 实例对象的 deleteNote 函数，代码如下：

```
{
        xtype:'button',
        text:'删除',
        handler:function(){
            app.controllers.appController.deleteNote();
        }
}
```

11.6 缓存文件

至此，整个示例应用程序基本介绍完毕，不过 HTML5 为我们提供了一种非常实用的功能：离线应用。虽然该标准目前还不是很完善，但完全不影响我们在本例中使用它。

11.6.1 设置 manifest 文件内容

首先，我们创建一个 manifest 格式文件，并命名为 app.manifest，代码如下：

代码 11-5 app.manifest 文件完整代码

```
CACHE MANIFEST

#version 1.00

CACHE:
index.html
lib/sencha-touch-debug.js
app/app.js
app/model/model.js
app/controller/controller.js
app/store/store.js
app/view/view.js
resources/style/style.css
lib/sencha-touch.css

NETWORK:
http://maps.google.com/maps/api/js?sensor=true
http://maps.gstatic.com/intl/zh_cn/mapfiles/api-3/6/8/main.js
*
```

在使用离线应用特性时，需要注意如下两点。

1．更新 manifest 文件

当离线应用功能生效后，如果多次修改 app.manifest 文件中 CACHE 指定的文件，整个应用程序不会识别这些文件是否有修改。只有当 app.manifest 文件被修改过时，应用程序才会识别到 manifest 文件已更新并重新下载 CACHE 指定的文件。

作者推荐使用版本号的方案作为更新 CACHE 的文件，如上述代码中“#version 1.00”。当每次修改一次或多次文件后，我们都可以将 1.00 修改为 1.01 或 1.1 等自定义版本号。这样离线应用会识别到 manifest 已被更新并重新下载，这样的方案就不影响 manifest 文件内部定义的列表清单结构。

不过，当 manifest 版本被更新后，应用程序会被重新下载到本地，而此时整个应用程序依然使用上一版本的缓存文件。如果需要使用最新下载的版本文件，需要再重新刷新加载一次页面，新版本的功能才会正式生效。

2．服务器配置 manifest 格式文件

在本书介绍离线应用的时候提到过，一般情况下，如 Tomcat 或 Apache 等 Web 应用服务器默认不会识别 manifest 格式文件，如果要支持 manifest 格式文件，需要对这些应用服务器配置 mime 类型以支持 manifest 格式。

如果在使用离线应用过程中，发现开发人员工具界面的控制台出现如图 11-7 所示显示 Invalid manifest mime type 等错误提示，就表明服务器无法识别 manifest 格式文件，需要配置服务器支持 manifest 文件格式。

图 11-7　服务器无法识别 manifest 格式文件错误提示

11.6.2　设置 HTML 缓存文件

定义完 app.manifest 配置文件后，接下来我们要真正实现离线应用的功能。

现在将 app.manifest 文件部署到 HTML 代码中，代码效果如下：

```
<html manifest="app.manifest">
```

离线应用第一次下载或更新 CACHE 指定的文件时效果如图 11-8 所示。

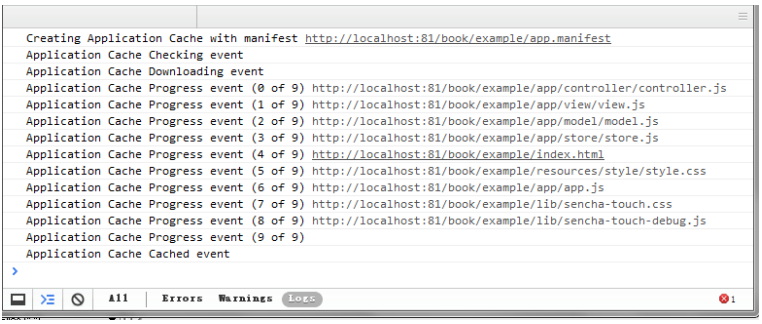

图 11-8　第一次下载或更新 CACHE 指定的文件效果图

11.7　后端服务器通信

本章的综合实例，大部分都是围绕前端业务进行的，并没有和后台服务器进行任何通信。在实际项目中，前端和后端之间的通信和数据交换非常频繁。本实例为了更好地展示 HTML5 特性的应用，没有介绍任何后端技术。

基于此，本实例仍然有很多实用功能扩展空间，并且有潜力成为一个真正有意义的 Web 应用程序。特别是实例中使用的 localStorage 对象，它的作用远远没有在本实例中得到很好的体现。

前端将所有业务逻辑处理的数据存储到 localStorage 对象，当 localStorage 对象发生变化时，可以通过 Ajax 异步请求方式，根据当前数据变化情况更新到后端服务器中，而此过程中完全不影响用户的正常使用。

当用户所在的网络不可用时，应用程序依然可以将数据存储到 localStorage 对象中，此时由于网络不可用，数据将不会自动更新到后端服务器。在网络可用状态下，程序会自动更新数据到后端服务器。

读者可以根据上述提到的功能特点，自行编写代码应用到这个实例中，让整个应用程序变得更强大。

11.8　本章小结

本章通过一个简单的实例实现开发一套基于移动端的 Web 应用程序。

示例中以 Sencha Touch 框架作为整个应用程序的基础库，利用了 Panel 视图组件、表单元素组件、Sheet 组件、地图组件、本地存储代理等特性构建一套 Web 应用程序界面。

同时，利用 HTML5 标准对移动 Web 应用程序的良好支持，使用 Geolocation 特性获取用户当前地理位置信息，并通过 Google 地图提供的 API 接口将位置信息标记到 Google 地图上，以提供更好的体验给用户。

最后，在示例应用程序中增加离线应用特性，可以使移动设备在没有网络状态的情况下依然可以使用应用程序的基本功能。