MVC

单页应用程序：使用 ASP.NET 构建响应迅速的现代 Web 应用程序

Mike Wasson

下载代码示例

单页应用程序 (SPA) 是加载单个 HTML 页面并在用户与应用程序交互时动态更新该页面的 Web 应用程序。

SPA 使用 AJAX 和 HTML5 创建流畅且响应迅速的 Web 应用程序，不会经常进行页面重载。 但是，这意味着许多工作在客户端的 JavaScript 中进行。 传统的 ASP.NET 开发人员可能难以适应这一巨变。 幸运的是，可以借助许多开放源代码 JavaScript 框架来简化创建 SPA 的任务。

在本文中，我将演示如何创建一个简单的 SPA 应用程序。 在此过程中，我将介绍一些构建 SPA 的基本概念，包括“模型-视图-控制器”(MVC) 和“模型-视图-视图模型”(MVVM) 模式、数据绑定和路由。

关于示例应用程序

我创建的示例应用程序是简单的电影数据库，如**图 1** 所示。 页面最左列显示影片类型列表。 单击某个类型可显示该类型的电影列表。 单击某个条目旁的 Edit 按钮可以更改该条目。 进行编辑之后，可以单击 Save 以将更新提交给服务器，或单击 Cancel 以撤销更改。

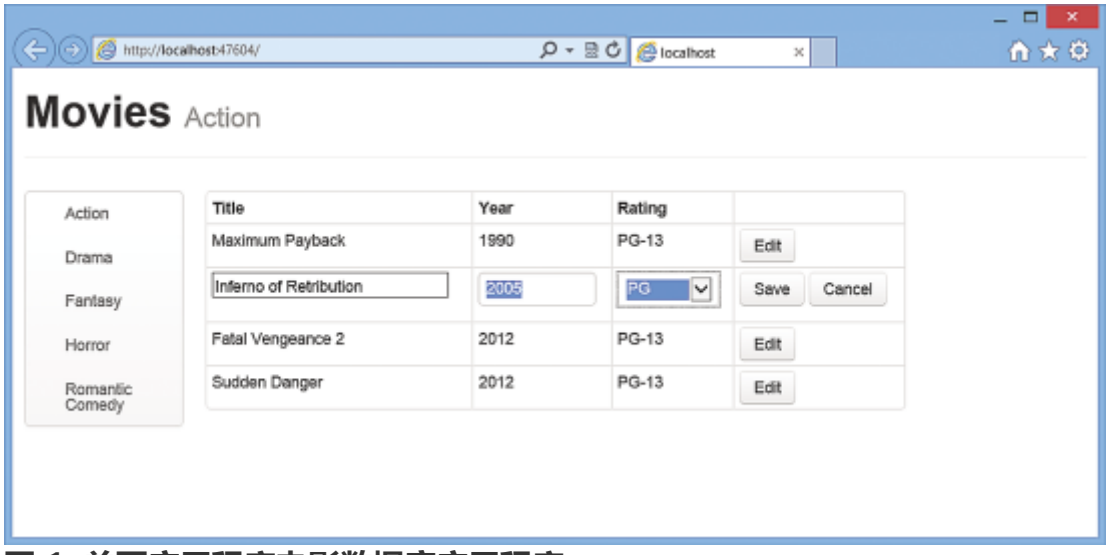

图 1. 单页应用程序电影数据库应用程序

我创建了两个不同版本的应用程序，一个版本使用 Knockout.js 库，另一个使用 Ember.js 库。 这两个库具有不同的方法，因此将其进行比较具有指导意义。 在两种情况下，客户端应用程序的 JavaScript 行数均少于 150。 在服务器端，我使用 ASP.NET Web API 向客户端提供 JSON。 您可以在 github.com/MikeWasson/MoviesSPA 上找到这两个应用程序版本的源代码。

（注意：我使用 Visual Studio 2013 的候选发布 [RC] 版本创建应用程序。 某些内容可能与交付厂商 [RTM] 版本不同，但应该不会影响代码。 ）

背景

在传统 Web 应用程序中，每次应用程序调用服务器时，服务器都会呈现新的 HTML 页面。 这会在浏览器中触发页面刷新。 如果您曾经编写过 Web 窗体应用程序或 PHP 应用程序，那么此页面生命周期在您看来应该十分熟悉。

在 SPA 中，第一个页面加载之后，与服务器之间的所有交互都通过 AJAX 调用进行。 这些 AJAX 调用通常以 JSON 格式返回数据（而不是标记）。 应用程序使用 JSON 数据动态更新页面，而无需重载页面。 **图 2** 说明了两种方法之间的差异。

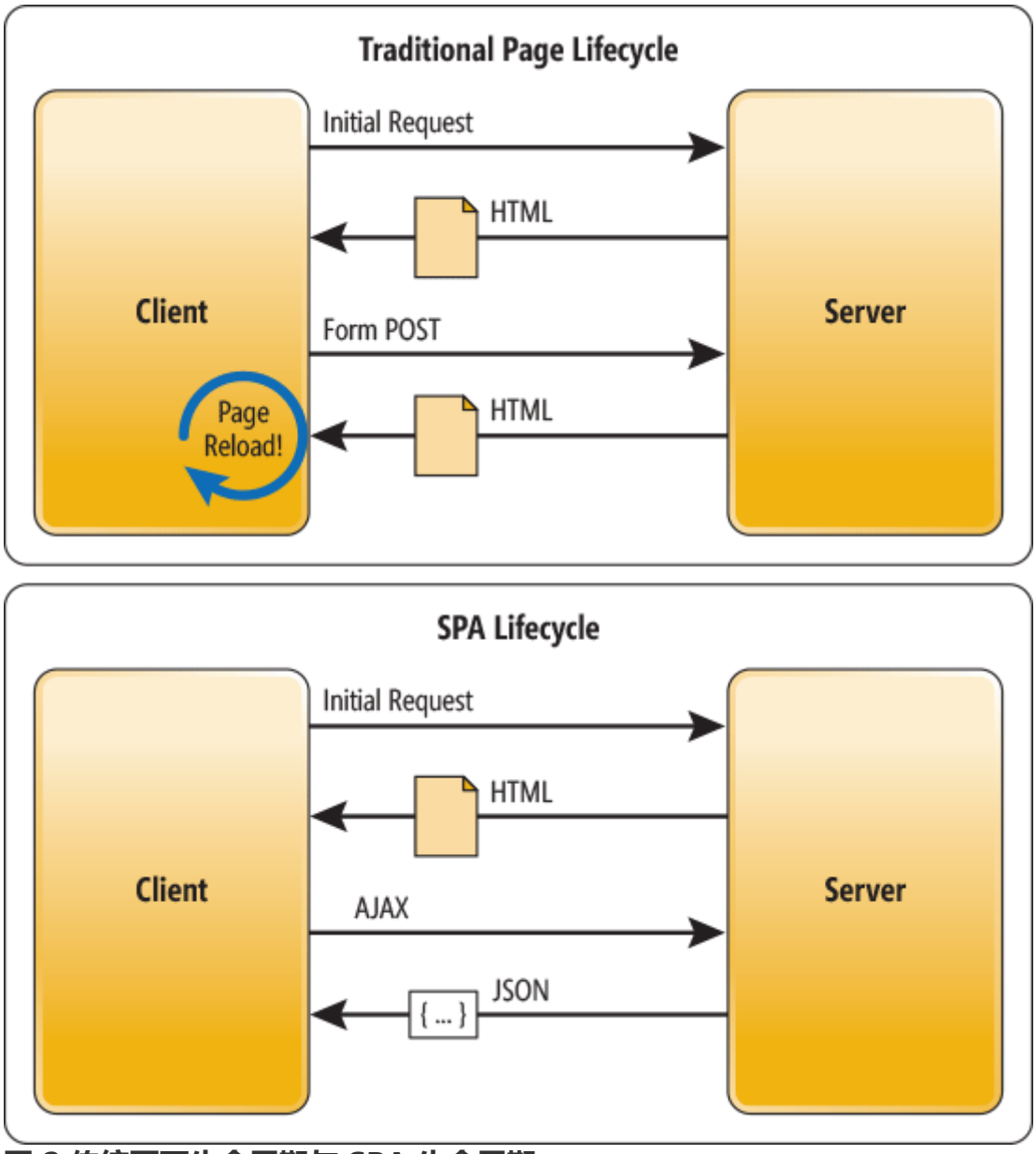

图 2 传统页面生命周期与 SPA 生命周期

SPA 的一个好处显而易见：应用程序更加流畅且响应迅速，不会出现重载和重新呈现页面时的不和谐效果。 另一个好处可能不那么明显，涉及到您如何构建 Web 应用程序。

将应用程序数据作为 JSON 进行发送会在呈现（HTML 标记）与应用程序逻辑（AJAX 请求以及 JSON 响应）之间形成分离。

此分离使每一层的设计和发展演变更加轻松。 可以在构建合理的 SPA 中更改 HTML 标记，而无需涉及实现应用程序逻辑的代码（至少理想情况是这样）。 在我稍后讨论数据绑定时，您将看到实际的效果。

在纯 SPA 中，所有 UI 交互都通过 JavaScript 和 CSS 在客户端进行。 初始页面加载之后，服务器将完全充当服务层。 客户端只需了解要发送的 HTTP 请求即可。 它并不关心服务器如何在后端实现内容。

借助这种体系结构，客户端和服务可以实现相互独立。 您可以替换运行服务的整个后端，只要您不更改 API，便不会破坏客户端。 反之亦然 — 您可以替换整个客户端应用程序，而不必更改服务层。 例如，您可以编写使用服务的本机移动客户端。

创建 Visual Studio 项目

Visual Studio 2013 具有单一的 ASP.NET Web 应用程序项目类型。 通过项目向导可以选择要包含在项目中的 ASP.NET 组件。 我从空白模板开始，然后通过在“为以下对象添加文件夹和核心引用:”下选中 Web API 来将 ASP.NET Web API 添加到项目中，如**图 3** 所示。

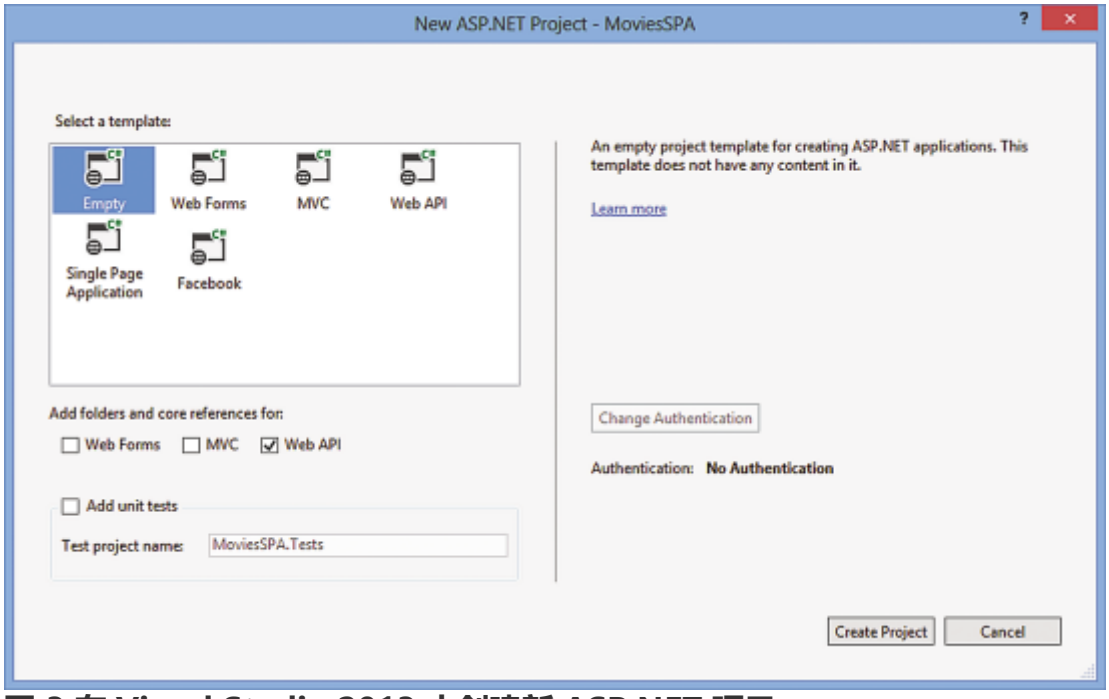

图 3 在 Visual Studio 2013 中创建新 ASP.NET 项目

新项目具有 Web API 所需的所有库，以及一些 Web API 配置代码。 我没有依赖任何 Web 窗体或 ASP.NET MVC。

在**图 3** 中可注意到，Visual Studio 2013 包含一个单页应用程序模板。 此模板会在 Knockout.js 基础上安装一个框架 SPA。 它支持使用成员资格数据库或外部身份验证提供程序登录。 我没有在我的应用程序中使用该模板，因为我想从零开始演示一个更简单的示例。 不过，该 SPA 模板是很好的资源，尤其适合要向应用程序添加身份验证的情况。

创建服务层

我使用 ASP.NET Web API 为应用程序创建了一个简单的 REST API。 在这里我不会详细介绍 Web API — 您可以在 asp. net/web-api 上了解更多信息。

首先，我创建了一个表示电影的 Movie 类。 此类有两个用途：

- 告知实体框架 (EF) 如何创建用于存储电影数据的数据库表。
- 告知 Web API 如何设置 JSON 负载的格式。

您不必为两者使用相同的模型。 例如，您可能希望数据库架构看上去与 JSON 负载有所不同。 对于此应用程序，我尽量使操作简单化：

```
namespace MoviesSPA.Models
{
  public class Movie
  {
    public int ID { get; set; }
    public string Title { get; set; }
    public int Year { get; set; }
    public string Genre { get; set; }
    public string Rating { get; set; }
  }
}
```

接下来，我使用 Visual Studio 基架创建将 EF 用作数据层的 Web API 控制器。 若要使用该基架，请右键单击解决方案资源管理器中的“控制器”文件夹，然后选择“添加”|“新基架项目”。 在“添加基架”向导中，选择“具有操作的 Web API 2 控制器，使用实体框架”，如**图 4** 所示。

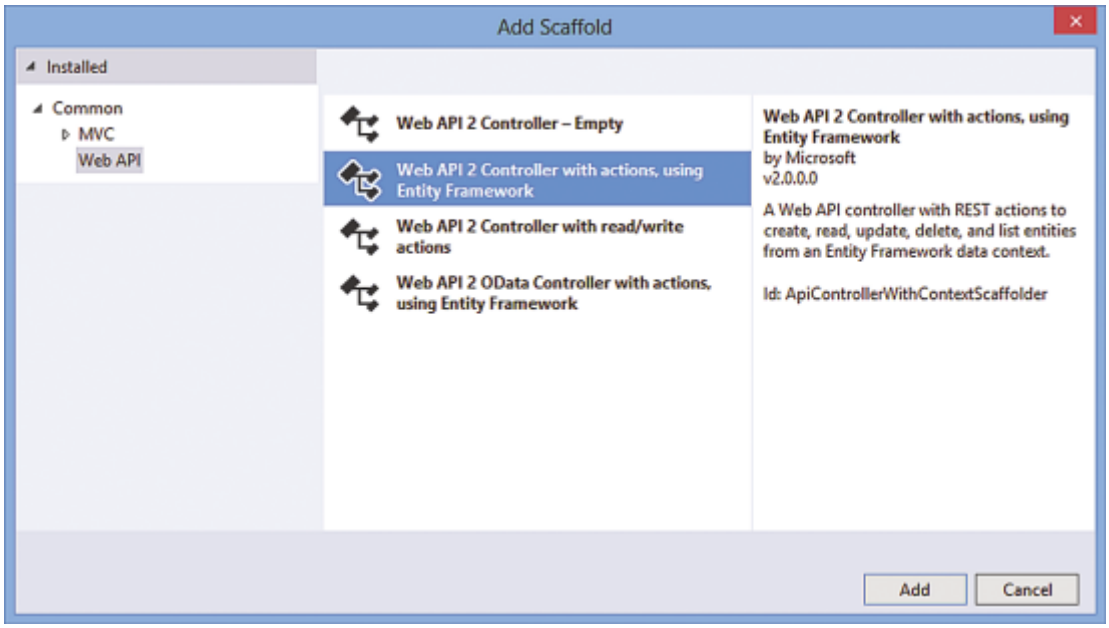

图 4 添加 Web API 控制器

图 5 显示“添加控制器”向导。 我将该控制器命名为 MoviesController。 名称十分重要，因为 REST API 的 URI 基于控制器名称。 我还选中“使用异步控制器操作”以利用 EF 6 中新的异步功能。 我为模型选择了 Movie 类并选择“新建数据上下文”以创建新 EF 数据上下文。

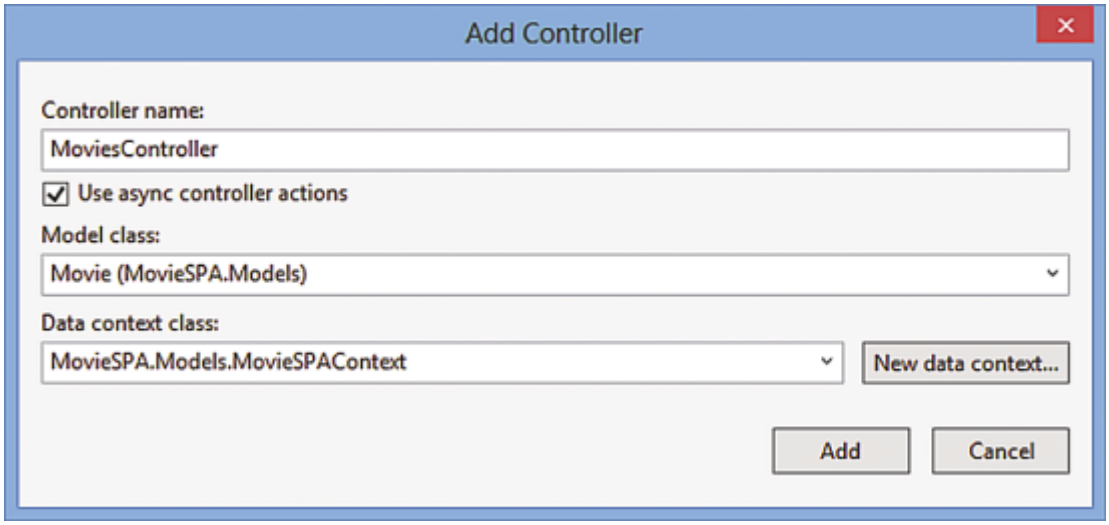

图 5“添加控制器”向导

该向导添加两个文件：

- MoviesController.cs 定义为应用程序实现 REST API 的 Web API 控制器。
- MovieSPAContext.cs 主要作为 EF 粘合，提供用于查询基础数据库的方法。

图 6 显示基架创建的默认 REST API。

图 6 Web API 基架创建的默认 REST API

HTTP 动词	URI	说明
GET	/api/movies	获取所有电影的列表
GET	/api/movies/{id}	获取 ID 等于 {id} 的电影
PUT	/api/movies/{id}	更新 ID 等于 {id} 的电影
POST	/api/movies	向数据库添加新电影
DELETE	/api/movies/{id}	从数据库中删除电影

大括号中的值是占位符。 例如，若要获取 ID 等于 5 的电影，URI 为 /api/movies/5。

我通过添加一个查找指定类型中所有电影的方法扩展了此 API：

```
public class MoviesController : ApiController
{
  public IQueryable<Movie> GetMoviesByGenre(string genre)
  {
    return db.Movies.Where(m =>
      m.Genre.Equals(genre, StringComparison.OrdinalIgnoreCase));
  }
  // Other code not shown
```

客户端将类型置于 URI 的查询字符串中。 例如，若要获取 Drama 类型中的所有电影，客户端会向 /api/movies?genre=drama 发送 GET 请求。 Web API 自动将查询参数绑定到 GetMoviesByGenre 方法中的 genre 参数。

创建 Web 客户端

到目前为止，我刚刚创建了一个 REST API。 如果向 /api/movies?genre=drama 发送 GET 请求，则原始 HTTP 响应如下所示：

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Date: Tue, 10 Sep 2013 15:20:59 GMT
Content-Length: 240
[{"ID":5,"Title":"Forgotten Doors","Year":2009,"Genre":"Drama","Ra
Blue Moon June","Year":1998,"Genre":"Drama","Rating":"PG-13"},{"ID
```

现在我需要编写一个对此执行有意义的操作的客户端应用程序。 基本工作流为：

- UI 触发 AJAX 请求
- 更新 HTML 以显示响应负载
- 处理 AJAX 错误

可以手工为所有这些内容进行编码。 例如，这里是一些创建电影标题列表的 jQuery 代码：

```
$.getJSON(url)
  .done(function (data) {
    // On success, "data" contains a list of movies
    var ul = $("<ul></ul>")
    $.each(data, function (key, item) {
      // Add a list item
      $('<li>', { text: item.Title }).appendTo(ul);
    });
  $('#movies').html(ul);
});
```

此代码有一些问题。 它将应用程序逻辑与表示逻辑混合在一起，并紧密绑定到 HTML。 而且其编写十分繁琐。 您没有专注于应用程序，而是将时间花费在编写事件处理程序和代码以操作 DOM 方面上。

解决方案构建于 JavaScript 框架之上。 幸运的是，您可以从许多开放源代码 JavaScript 框架进行选择。 一些较常用的框架包括 Backbone、Angular、Ember、Knockout、Dojo 和 JavaScriptMVC。 大多数人使用 MVC 或 MVVM 模式的变体，因此了解这两种模式可能会有所帮助。

MVC 和 MVVM 模式

MVC 模式可追溯到二十世纪八十年代及早期的图形 UI。 MVC 的目标是将代码分为三个单独的责任因素，如**图 7** 所示。 以下是它们的用途：

- 模型表示域数据和业务逻辑。
- 视图显示模型。
- 控制器接收用户输入并更新模型。

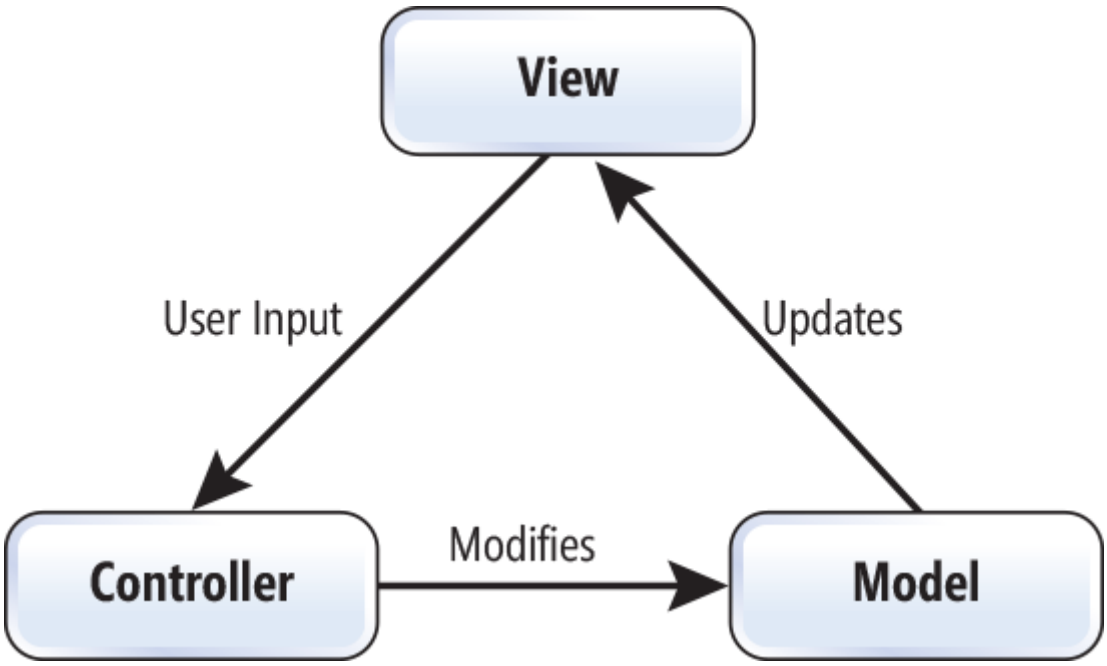

图 7 MVC 模式

MVC 的一个最新变体是 MVVM 模式（请参见**图 8**）。 在 MVVM 中：

- 模型仍表示域数据。
- 视图模型是视图的抽象表示形式。
- 视图显示视图模型并将用户输入发送到视图模型。

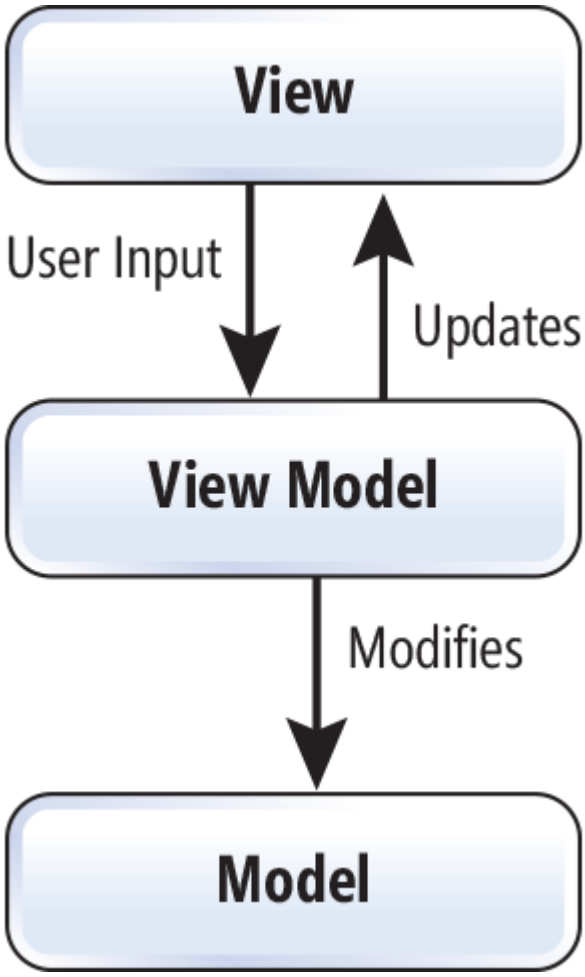

图 8 MVVM 模式

在 JavaScript MVVM 框架中，视图是标记，视图模型是代码。

MVC 具有许多变体，有关 MVC 的文献资料通常比较混乱且相互矛盾。 对于从 Smalltalk-76 开始并且仍在现代 Web 应用程序中使用的设计模式，这可能没什么好令人惊讶的。 因此，了解理论固然不错，不过主要是要了解所使用的特定 MVC 框架。

使用 Knockout.js 构建 Web 客户端

对于第一个应用程序版本，我使用了 Knockout.js 库。 Knockout 遵循 MVVM 模式，使用数据绑定将视图与视图模型连接在一起。

若要创建数据绑定，可向 HTML 元素添加特殊的数据绑定属性。 例如，以下标记将 span 元素绑定到视图模型上名为 genre 的属性。 每当 genre 的值更改时，Knockout 便会自动更新 HTML：

```
<h1><span data-bind="text: genre"></span></h1>
```

绑定还可以在另一个方向上发挥作用 — 例如，如果用户向文本框中输入文本，则 Knockout 会更新视图模型中的对应属性。

很棒的一点是数据绑定是声明性的。 您不必将视图模型绑定到 HTML 页面元素。 只需添加数据绑定属性，由 Knockout 执行其余工作。

我首先创建了一个 HTML 页面，该页面具有基本布局，没有数据绑定，如**图 9** 所示。

（注意：我使用 Bootstrap 库设置应用程序样式，因此实际应用程序具有许多额外的 元素和 CSS 类，用于控制格式设置。 为清晰起见，我未在代码示例中包含这些内容。）

图 9 初始 HTML 布局

```
<!DOCTYPE html>
<html>
<head>
  <title>Movies SPA</title>
</head>
<body>
  <ul>
    <li><a href="#"><!-- Genre --></a></li>
  </ul>
  <table>
    <thead>
      <tr><th>Title</th><th>Year</th><th>Rating</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td><!-- Title --></td>
        <td><!-- Year --></td>
        <td><!-- Rating --></td></tr>
    </tbody>
  </table>
  <p><!-- Error message --></p>
  <p>No records found.</p>
```

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

创建视图模型

可观察量是 Knockout 数据绑定系统的核心。 可观察量是存储值并且可以在值更改时通知订阅者的对象。 以下代码将电影的 JSON 表示形式转换为具有可观察量的等效对象：

```
function movie(data) {
  var self = this;
  data = data || {};
  // Data from model
  self.ID = data.ID;
  self.Title = ko.observable(data.Title);
  self.Year = ko.observable(data.Year);
  self.Rating = ko.observable(data.Rating);
  self.Genre = ko.observable(data.Genre);
};
```

图 10 显示视图模型的初始实现。 此版本仅支持获取电影列表。 我稍后将添加编辑功能。 视图模型包含用于电影列表、错误字符串和当前类型的可观察量。

图 10 视图模型

```
var ViewModel = function () {
  var self = this;
  // View model observables
  self.movies = ko.observableArray();
  self.error = ko.observable();
  self.genre = ko.observable();  // Genre the user is currently browsin
  // Available genres
  self.genres = ['Action', 'Drama', 'Fantasy', 'Horror', 'Romantic Com
  // Adds a JSON array of movies to the view model
  function addMovies(data) {
    var mapped = ko.utils.arrayMap(data, function (item) {
      return new movie(item);
    });
    self.movies(mapped);
  }
  // Callback for error responses from the server
  function onError(error) {
    self.error('Error: ' + error.status + '' + error.statusText);
  }
  // Fetches a list of movies by genre and updates the view model
  self.getByGenre = function (genre) {
    self.error(''); // Clear the error
    self.genre(genre);
    app.service.byGenre(genre).then(addMovies, onError);
  };
  // Initialize the app by getting the first genre
```

```
  self.getByGenre(self.genres[0]);
}
// Create the view model instance and pass it to Knockout
ko.applyBindings(new ViewModel());
```

请注意，电影是 observableArray。 顾名思义，observableArray 充当在数组内容更改时通知订阅者的数组。

getByGenre 函数向服务器发出针对电影列表的 AJAX 请求，然后使用结果填充 self.movies 数组。

使用 REST API 时，最棘手的部分之一是处理 HTTP 的异步特性。 jQuery ajax 函数返回实现 Promises API 的对象。 可以使用 Promise 对象的 then 方法设置一个在 AJAX 调用成功完成时调用的回调，以及另一个在 AJAX 调用失败时调用的回调：

```
app.service.byGenre(genre).then(addMovies, onError);
```

数据绑定

现在，我有一个视图模型，可以将 HTML 数据绑定到该模型。 对于在屏幕左侧出现的类型列表，我使用以下数据绑定：

```
<ul data-bind="foreach: genres">
  <li><a href="#"><span data-bind="text: $data"></span></a></li>
</ul>
```

数据绑定属性包含一个或多个绑定声明，其中每个绑定的形式为“绑定:表达式”。在此示例中，foreach 绑定告知 Knockout 在视图模型的 genres 数组内容中循环。 对于该数组中的每个项，Knockout 都创建一个新 <li> 元素。 中的 text 绑定设置等于数组项的值（在此例中为类型名称）的 span 文本。

现在，单击类型名称不会执行任何操作，因此我添加了一个 click 绑定以处理单击事件：

```
<li><a href="#" data-bind="click: $parent.getByGenre">
  <span data-bind="text: $data"></span></a></li>
```

这将单击事件绑定到视图模型上的 getByGenre 函数。 我需要在此处使用 $parent，因为此绑定在 foreach 的上下文中进行。 默认情况下，foreach 中的绑定引用循环中的当前项。

为了显示电影列表，我向表添加了绑定，如**图 11** 所示。

图 11 向表添加绑定以显示电影列表

```
<table data-bind="visible: movies().length > 0">
  <thead>
    <tr><th>Title</th><th>Year</th><th>Rating</th><th></th></tr>
  </thead>
  <tbody data-bind="foreach: movies">
```

```
    <tr>
      <td><span data-bind="text: Title"></span></td>
      <td><span data-bind="text: Year"></span></td>
      <td><span data-bind="text: Rating"></span></td>
      <td><!-- Edit button will go here --></td>
    </tr>
  </tbody>
</table>
```

在**图 11** 中，foreach 绑定在 movie 对象数组中循环。 在 foreach 中，text 绑定引用当前对象的属性。

<table> 元素上的 visible 绑定控制是否呈现表。 如果 movies 数组为空，则会隐藏表。

最后，这里是用于错误消息和“No records found”消息的绑定（请注意，可以将复杂表达式置于绑定中）：

```
<p data-bind="visible: error, text: error"></p>
<p data-bind="visible: !error() && movies().length == 0">No records foun
```

使记录可编辑

此应用程序的最后一个部分是使用户可以编辑表中的记录。 这涉及到一些功能：

- 在查看模式（纯文本）与编辑模式（输入控件）之间切换。
- 将更新提交到服务器。
- 允许用户取消编辑并恢复为原始数据。

为了跟踪查看/编辑模式，我向 movie 对象添加了一个布尔标志作为可观察量：

```
function movie(data) {
  // Other properties not shown
  self.editing = ko.observable(false);
};
```

我希望电影表在 editing 属性为 false 时显示文本，而在 editing 为 true 时切换到输入控件。 为实现此目的，我使用了 Knockout if 和 ifnot 绑定，如**图 12** 所示。 通过“<!-- ko -->”语法可以包含 if 和 ifnot 绑定，而无需将其置于 HTML 容器元素中。

图 12 实现电影记录的编辑

```
<tr>
  <!-- ko if: editing -->
  <td><input data-bind="value: Title" /></td>
  <td><input type="number" class="input-small" data-bind="value: Y
  <td><select class="input-small"
    data-bind="options: $parent.ratings, value: Rating"></select><
  <td>
    <button class="btn" data-bind="click: $parent.save">Save</butt
```

```
    <button class="btn" data-bind="click: $parent.cancel">Cancel</
  </td>
  <!-- /ko -->
  <!-- ko ifnot: editing -->
  <td><span data-bind="text: Title"></span></td>
  <td><span data-bind="text: Year"></span></td>
  <td><span data-bind="text: Rating"></span></td>
  <td><button class="btn" data-bind="click: $parent.edit">Edit</bu
  <!-- /ko -->
</tr>
```

value 绑定设置输入控件的值。 这是双向绑定，因此当用户在文本字段中键入某些内容或更改下拉菜单选择时，更改会自动传播到视图模型。

我将按钮单击处理程序绑定到视图模型上名为 save、cancel 和 edit 的函数。

edit 函数非常简单。 只需将 editing 标志设置为 true：

```
self.edit = function (item) {
  item.editing(true);
};
```

Save 和 cancel 稍微复杂一点。 为了支持 cancel，我需要一种在编辑过程中缓存原始值的方法。 幸运的是，通过 Knockout 可以方便地扩展可观察量的行为。 **图 13** 中的代码向 observable 类添加了一个 store 函数。 对可观察量调用 store 函数将向可观察量赋予两种新功能：恢复和提交。

图 13 使用恢复和提交扩展 ko.observable

现在我可以调用 store 函数以向模型添加此功能：

```
function movie(data) {
  // ...
  // New code:
  self.Title = ko.observable(data.Title).store();
  self.Year = ko.observable(data.Year).store();
  self.Rating = ko.observable(data.Rating).store();
  self.Genre = ko.observable(data.Genre).store();
};
```

图 14 显示视图模型上的 save 和 cancel 函数。

图 14 添加 Save 和 Cancel 函数

```
self.cancel = function (item) {
  revertChanges(item);
  item.editing(false);
};
self.save = function (item) {
  app.service.update(item).then(
```

```
      function () {
        commitChanges(item);
      },
      function (error) {
        onError(error);
        revertChanges(item);
      }).always(function () {
        item.editing(false);
    });
  }
  function commitChanges(item) {
    for (var prop in item) {
      if (item.hasOwnProperty(prop) && item[prop].commit) {
        item[prop].commit();
      }
    }
  }
  function revertChanges(item) {
    for (var prop in item) {
      if (item.hasOwnProperty(prop) && item[prop].revert) {
        item[prop].revert();
      }
    }
  }
```

使用 Ember 构建 Web 客户端

为进行比较，我使用 Ember.js 库编写了另一个版本的应用程序。

Ember 应用程序从路由表开始，该表定义用户如何在应用程序中导航：

```
window.App = Ember.Application.create();
App.Router.map(function () {
  this.route('about');
  this.resource('genres', function () {
    this.route('movies', { path: '/:genre_name' });
  });
});
```

第一行代码创建一个 Ember 应用程序。 Router.map 调用创建三个路由。 每个路由对应于一个 URI 或 URI 模式：

```
/#/about
/#/genres
/#/genres/genre_name
```

对于每个路由，使用 Handlebars 模板库创建一个 HTML 模板。

Ember 具有用于整个应用程序的顶级模板。 此模板会对每个路由进行呈现。 **图 15** 显示我的应用程序的应用程序模板。 如您所见，该模板基本上是 HTML，使用

type="text/x-handlebars" 置于脚本标记中。该模板在双大括号内包含特殊的 Handlebars 标记：{{ }}. 此标记的用途与 Knockout 中的数据绑定属性类似。 例如，{{#linkTo}} 创建指向路由的链接。

图 15 应用程序级别 Handlebars 模板

```
ko.observable.fn.store = function () {
  var self = this;
  var oldValue = self();
  var observable = ko.computed({
    read: function () {
      return self();
    },
    write: function (value) {
      oldValue = self();
      self(value);
    }
  });
  this.revert = function () {
    self(oldValue);
  }
  this.commit = function () {
    oldValue = self();
  }
  return this;
}
<script type="text/x-handlebars" data-template-name="application">
  <div class="container">
    <div class="page-header">
      <h1>Movies</h1>
    </div>
    <div class="well">
      <div class="navbar navbar-static-top">
        <div class="navbar-inner">
          <ul class="nav nav-tabs">
            <li>{{#linkTo 'genres'}}Genres{{/linkTo}} </li>
            <li>{{#linkTo 'about'}}About{{/linkTo}} </li>
          </ul>
        </div>
      </div>
    </div>
    <div class="container">
      <div class="row">{{outlet}}</div>
    </div>
  </div>
  <div class="container"><p>&copy;2013 Mike Wasson</p></div>
</script>
```

现在假设用户导航到 /#/about。 这会调用“about”路由。 Ember 首先呈现顶级应用程序模板。 随后在应用程序模板的 {{outlet}} 中呈现 about 模板。 下面是 about 模板：

```
<script type="text/x-handlebars" data-template-name="about">
  <h2>Movies App</h2>
  <h3>About this app...</h3>
</script>
```

图 16 显示如何在应用程序模板中呈现 about 模板。

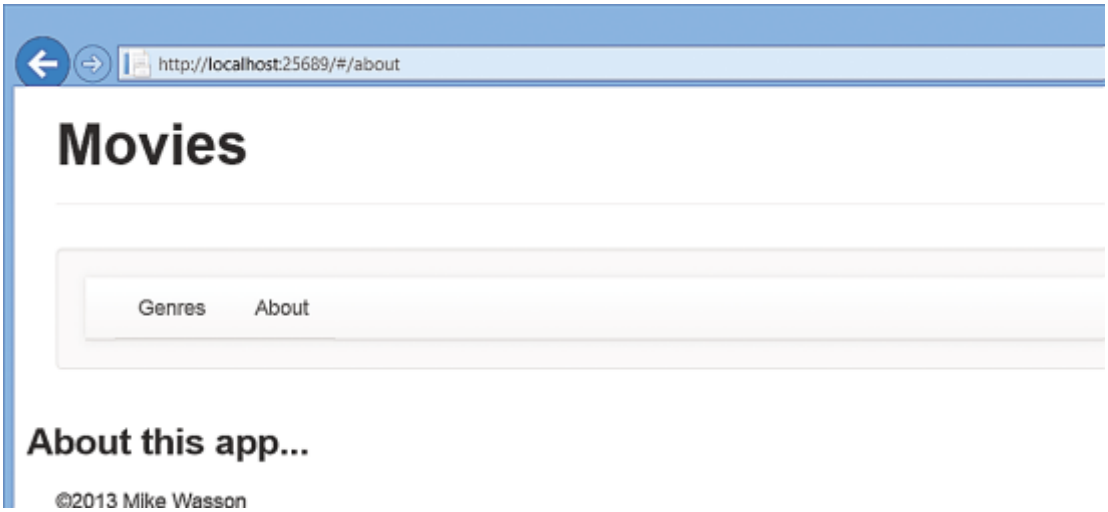

图 16 呈现 About 模板

因为每个路由都有自己的 URI，所有浏览器历史记录会保留。 用户可以使用“后退”按钮进行导航。 用户还可以刷新页面而不会丢失上下文，或是对相同页面添加书签和重载。

Ember 控制器和模型

在 Ember 中，每个路由都有一个模型和一个控制器。 模型包含域数据。 控制器充当模型的代理并存储视图的任何应用程序状态数据。 （这并不与 MVC 的经典定义完全相符。 在某些方面，控制器更类似于视图模型。 ）

下面介绍我定义 movie 模型的方式：

```
App.Movie = DS.Model.extend({
  Title: DS.attr(),
  Genre: DS.attr(),
  Year: DS.attr(),
  Rating: DS.attr(),
});
```

控制器从 Ember.ObjectController 派生，如**图 17** 所示。

图 17 Movie 控制器从 Ember.ObjectController 派生

```
App.MovieController = Ember.ObjectController.extend({
  isEditing: false,
  actions: {
    edit: function () {
      this.set('isEditing', true);
    },
    save: function () {
```

```
        this.content.save();
        this.set('isEditing', false);
      },
      cancel: function () {
        this.set('isEditing', false);
        this.content.rollback();
      }
    }
});
```

此处有一些有趣的事情。 首先，我未在 controller 类中指定模型。 默认情况下，路由自动在控制器上设置模型。 其次，save 和 cancel 函数使用内置到 DS.Model 类中的事务功能。 若要恢复编辑，只需对模型调用 rollback 函数。

Ember 使用许多命名约定来连接不同的组件。 genres 路由与 GenresController 通信，后者呈现 genres 模板。 事实上，如果您未定义 GenresController 对象，则 Ember 将自动创建一个。 不过，您可以重写默认值。

在我的应用程序中，我通过实现 renderTemplate 挂钩，将 genres/movies 路由配置为使用不同的控制器。 这样，几个路由便可以共享同一控制器（请参见**图 18**）。

图 18 几个路由可以共享同一控制器

```
App.GenresMoviesRoute = Ember.Route.extend({
  serialize: function (model) {
    return { genre_name: model.get('name') };
  },
  renderTemplate: function () {
    this.render({ controller: 'movies' });
  },
  afterModel: function (genre) {
    var controller = this.controllerFor('movies');
    var store = controller.store;
    return store.findQuery('movie', { genre: genre.get('name') })
    .then(function (data) {
      controller.set('model', data);
  });
  }
});
```

Ember 的一大优点在于您可以使用非常少的代码来完成工作。 我的示例应用程序大约有 110 行 JavaScript。 这比 Knockout 版本要短，并且我可以轻松地获得浏览器历史记录。 另一方面，Ember 也是非常“固执”的框架。 如果您不按照“Ember 方式”编写代码，则可能会遇到一些障碍。 选择框架时，应考虑框架的功能集和整体设计是否符合需要和编码样式。

了解更多

在本文中，我演示了 JavaScript 框架如何使 SPA 创建更加简单。 在此过程中，我介绍了这些库的一些常用功能，包括数据绑定、路由以及 MVC 和 MVVM 模式。 可以在 asp.net/single-page-application 上了解有关使用 ASP.NET 构建 SPA 的更多信息。

Mike Wasson *是 Microsoft 的一名程序员兼作家。多年来，他一直负责撰写 Win32 多媒体 API 的文档。他目前正在撰写 ASP.NET 的相关内容（以 Web API 为主）。您可以通过 mwasson@microsoft.com 与他联系。*

衷心感谢以下技术专家对本文的审阅：Xinyang Qiu (Microsoft)
Xinyang Qiu 是 Microsoft ASP.NET 团队中从事测试的高级软件设计工程师，经常在 blogs.msdn.com/b/webdev 上发表博客。 他很乐意回答 ASP.NET 问题或请专家回答您的问题。 可通过电子邮件 xinqiu@microsoft.com 与他联系。

关注我们 页面有帮助吗？ 是 否 注册 MSDN 时事通讯

开发人员中心

Windows

Office

Visual Studio

Microsoft Azure

更多...

学习资源

Microsoft 虚拟学院

第 9 频道

MSDN 杂志

计划

BizSpark（针对新创企业）

创新杯

社区

论坛

博客

Codeplex

支持

自助支持