Backbone 教程

一. 前言
二. Hello Backbonejs
三. Backbonejs 中的 Model 实践
四. Backbonejs 中的 Collections 实践
五. Backbonejs 中的 Router 实践
六. Backbonejs 中的 View 实践
七. 实战演练：todos 分析（一）
八. 实战演练：todos 分析（二）View 的应用
九. 实战演练：todos 分析（三）总结
十. 后端环境搭建：web.py 的使用
十一. 实战演练：扩展 todos 到 Server 端（backbonejs+webpy）
十二. 前后端实战演练：Web 聊天室-功能分析
十三. 前后端实战演练：Web 聊天室-详细设计
十四. 前后端实战演练：Web 聊天室-服务端开发
十五. 前后端实战演练：Web 聊天室-前端开发
十六. 引入 requirejs
十七. 补充异常处理
十八. 定制 Backbonejs
十九. 再次总结的说
二十. Backbonejs 相关资源

第二版前言

一年前写的东西，当时写完之后工作中用不到也就不再看了，谁想后来越来越多的人通过搜索 backbonejs 来到我的博客，有表示感谢的，有吐槽的，有帮忙改 bug 的。在之前接触时只是为了能让代码运行，因此很多实例没有考虑版本问题。在 backbonejs 更新到 1.0.0 之后，我的很多代码实例都出现了问题，于是我不得不去更新其中的东西。但是这样一点点的更新总是不能覆盖全面，随想不如在搞个第二版，把之前的坑填上，然后再补充点我新学到的东西，尽可能的让有机会看到这本电子书的人有点收获。

说起这一年，学习了 angularjs，还没有来得及实践。用 backbonejs 实践了我博客的手机版:m.the5fire.com 功能，比较简陋。

在 qq 群里聊天时@wrongway 兄调侃说 backbonejs 属于远古时期的框架了，关于这一点，相比于 angularjs 确实显的有些落后。在我看来，两个的思路或者哲学并不相同，backbonejs 就如其官网所说提供了一个 web 应用的结构，而 angularjs 则是对 html 的增强。

大概就闲扯这么多，总之主要的目的是填上一版的一些坑，然后或许会留下新的坑。

从去年 8 月份开始到现在，最初的目的算是完成了。但是随着不断的和正在学习 Backbonejs 的人交流发现，有些人虽然已经掌握了 Backbonejs 基本模型、视图以及其他模块的使用，但是却无法把已掌握的内容转化到实际项目中。

另外大家也看厌了到处在分析的 Todos 这个实例，希望看到真实点的项目。虽然对大家看待优秀项目代码的态度不太认同，但我很理解初学者的这种心情——不要给我 Demo，我要真实的东西。

但真实的东西往往需要太多其他方面的知识，比如我在最后的项目中用到了 socket.io 来实现实时的交流。这些东西都算是超出 Backbonejs 这个框架内容之外的。

要学习一个东西，和结识一个新的朋友一样。如果你总是站在远处想看风景一般看着他/她，那他/她对你来说只能是风景。对于 Backbonejs 来说也一样，不要停留在看完书、觉得会用的阶段，拉起 Ta 的手，一起来做一些东西。

基于此，最后的那个项目是打算放到线上来运行的，也让读完本书的读者能有一个地方体验一下你在本书看到的东西，最终是如何转化为被用户可用的网站的。

第一章 Hello Backbonejs

1.1 基础概念

Backbone，英文意思是：勇气， 脊骨，但是在程序里面，尤其是在 Backbone 后面加上后缀 js 之后，它就变成了一个框架，一个 js 库。

Backbone.js，不知道作者是以什么样的目的来对其命名的，可能是希望这个库会成为 web 端开发中脊梁骨。

好了，八卦完了开始正题。

Backbone.js 提供了一套 web 开发的框架，通过 Models 进行 key-value 绑定及自定义事件处理，通过 Collections 提供一套丰富的 API 用于枚举功能，通过 Views 来进行事件处理及与现有的 Application 通过 RESTful JSON 接口进行交互.它是基于 jQuery 和 underscore 的一个前端 js 框架。

整体上来说，Backbone.js 是一个 web 端 javascript 的 MVC 框架，算是轻量级的框架。它能让你像写 Java（后端）代码组织 js 代码，定义类，类的属性以及方法。更重要的是它能够优雅的把原本无逻辑的 javascript 代码进行组织，并且提供数据和逻辑相互分离的方法，减少代码开发过程中的数据和逻辑混乱。

在 Backbonejs 有几个重要的概念，先介绍一下:Model，Collection，View，Router。其中 Model 是根据现实数据建立的抽象，比如人(People)；Collection 是 Model 的一个集合，比如一群人；View 是对 Model 和 Collection 中数据的展示，把数据渲染（Render）到页面上；Router 是对路由的处理，就像传统网站通过 url 现实不同的页面，在单页面应用（SPA）中通过 Router 来控制前面说的 View 的展示。

通过 Backbone，你可以把你的数据当作 Models，通过 Models 你可以创建数据，进行数据验证，销毁或者保存到服务器上。当界面上的操作引起 model 中属性的变化时，model 会触发 change 的事件。那些用来显示 model 状态的 views 会接受到 model 触发 change 的消息，进而发出对应的响应，并且重新渲染新的数据到界面。在一个完整的 Backbone 应用中，你不需要写那些胶水代码来从 DOM 中通过特殊的 id 来获取节点，或者手工的更新 HTML 页面，因为在 model 发生变化时，views 会很简单的进行自我更新。

上面是一个简单的介绍，关于 backbone 我看完他的介绍和简单的教程之后，第一印象是它为前端开发制定了一套自己的规则，在这个规则下，我们可以像使用 django 组织 python 代码一样的组织 js 代码，它很优雅，能够使前端和 server 的交互变得简单。

在查 backbone 资料的时候，发现没有很系统的中文入门资料和更多的实例，所以我打算自己边学边实践边写，争取能让大家通过一系列文章能快速的用上 Backbone.js。

关于 backbone 的更多介绍参看这个：

http://documentcloud.github.com/backbone/

http://backbonetutorials.com/

1.2 backbone 的应用范围：

它虽然是轻量级框架，但是框架这东西也不是随便什么地方都能用的，不然就会出现杀鸡用牛刀，费力不讨好的结果。那么适用在哪些地方呢？

根据我的理解，以及 Backbone 的功能，如果单个网页上有非常复杂的业务逻辑，那么用它很合适，它可以很容易的操作 DOM 和组织 js 代码。

豆瓣的阿尔法城是一个极好的例子——纯单页、复杂的前端逻辑。

当然，除了我自己分析的应用范围之外，在 Backbone 的文档上看到了很多使用它的外国站点，有很多，说明 Backbonejs 还是很易用的。

稍稍列一下国内用到 Backbonejs 的站点：

1. 豆瓣阿尔法城 链接：http://alphatown.com/
2. 豆瓣阅读 链接：http://read.douban.com/ 主要用在图书的正文页
3. 百度开发者中心 链接：http://developer.baidu.com/
4. 手机搜狐直播间 链接：http://zhibo.m.sohu.com/
5. OATOS 企业网盘 链接：http://app.oatos.com

1.3 学以致用

现在，我们就要开始学习 Backbonejs 了，我假设你没有看过我的第一版，那一版有很多很多问题，在博客上也有很多人反馈。但是如果你把那一版看明白了，这新版的教程你可以粗略的浏览一遍，不过后面新补充的实践是要自己写出来、跑起来的。

先说我们为什么要学习这新的东西呢？简单说来是为了掌握更加先进的工具。那为什么要掌握先进的工具呢？简单来说就是为了让我们能够以更合理、优雅的方式完成工作，反应到代码上就是让代码变得可维护，易扩展。如果从复杂的方向来说的话，这俩话题都够我写好几天的博客了。

学以致用，最直接有效的就是用起来，光学是没用的，尤其是编程这样的实践科学。新手最常犯的一个错误就是喜欢不停的去看书，看过了就以为会了，然后就开始疯狂的学下一本。殊不知看懂和写出来能运行是两种完全不同的状态。因此建议新手——编程新手还是踏踏实实的把代码都敲了，执行了，成功了才是。

下面直接给一个简单的 Demo 出来，用到了 Backbonejs 的三个主要模块：Views，Collection，Model。通过执行这个例子，了解这个例子的运行过程，快速对要做的东西有一个感觉，然后再逐步击破。

1.4 完整 DEMO

这个 demo 的主要功能是点击页面上得“新手报到”按钮，弹出对话框，输入内容之后，把内容拼上固定的字符串显示到页面上。事件触发的逻辑是： click 触发 checkIn 方法，然后 checkIn 构造 World 对象放到已经初始化 worlds 这个 collection 中。

来看完整的代码:

```
<!DOCTYPE html><html><head>
    <title>the5fire.com-backbone.js-Hello World</title></head><body>
    <button id="check">新手报到</button>
    <ul id="world-list">
    </ul>
    <a href="http://www.the5fire.com">更多教程</a>
    <script
src="http://the5fireblog.b0.upaiyun.com/staticfile/jquery-1.10.2.js"></script>
    <script
src="http://the5fireblog.b0.upaiyun.com/staticfile/underscore.js"></script>
    <script
src="http://the5fireblog.b0.upaiyun.com/staticfile/backbone.js"></script>
    <script>
    (function ($) {
        World = Backbone.Model.extend({
            //创建一个 World 的对象，拥有 name 属性
            name: null
        });
```

```
        Worlds = Backbone.Collection.extend({
            //World对象的集合
            initialize: function (models, options) {
                    this.bind("add", options.view.addOneWorld);
            }
        });

        AppView = Backbone.View.extend({
            el: $("body"),
            initialize: function () {
                //构造函数，实例化一个World集合类，并且以字典方式传入AppView
的对象
                this.worlds = new Worlds(null, { view : this })
            },
            events: {
                "click #check":  "checkIn",    //事件绑定，绑定Dom中id为check
的元素
            },
            checkIn: function () {
                var world_name = prompt("请问，您是哪星人?");
                if(world_name == "") world_name = '未知';
                var world = new World({ name: world_name });
                this.worlds.add(world);
            },
            addOneWorld: function(model) {
                $("#world-list").append("<li>这里是来自 <b>" + model.get('name')
+ "</b> 星球的问候：hello world! </li>");
            }
        });
        //实例化AppView
        var appview = new AppView;
    })(jQuery);
    </script></body></html>
```

这里面涉及到 backbone 的三个部分，View、Model、Collection，其中 Model 代表一个数据模型，Collection 是模型的一个集合，而 View 是用来处理页面以及简单的页面逻辑的。

动手把代码放到你的编辑器中吧，成功执行，然后修改某个地方，再次尝试。

第二章 Backbonejs 中的 Model 实践

上一章主要是通过简单的代码对 Backbonejs 做了一个概括的展示，这一章开始从 Model 层说起，详细解释 Backbonejs 中的 Model 这个东西。

对于 Model 这一部分，其官网是这么说的：“Model 是 js 应用的核心，包括基础的数据以及围绕着这些数据的逻辑：数据转换、验证、属性计算和访问控制”。这句话基本上高度概括了 Model 在一个项目中的作用。实际上，不仅仅是 js 应用，在任何以数据收集和处理的项目中 Model 都是很重要的一块内容。

Model 这个概念在我的印象中是来自于 MVC 这个东西，Model 在其中的作用，除了是对业务中实体对象的抽象，另外的作用就是做持久化，所谓持久化就是把数据存储到磁盘上——文件形式、数据库形式。在 web 端也有对应的操作，比如存入 LocalStorage，或者 Cookie。

在 web 端，Model 还有一个重要的功能就是和服务器端进行数据交互，就像是服务器端的程序需要和数据库交互一样。因此 Model 应该是携带数据流窜于各个模块之间的东西。

下面让我们通过一个一个的实例来逐步了解 Model。

先定义一个页面结构，实践时须在注释的地方填上各小节的代码

```
<!DOCTYPE
html><html><head><title>the5fire-backbone-model</title></head><body></body><scr
ipt
src="http://the5fireblog.b0.upaiyun.com/staticfile/jquery-1.10.2.js"></script><
script
src="http://the5fireblog.b0.upaiyun.com/staticfile/underscore.js"></script><scr
ipt
src="http://the5fireblog.b0.upaiyun.com/staticfile/backbone.js"></script><scrip
t>(function ($) {
    /**     *此处填充代码下面练习代码      **/})(jQuery);</script></html>
```

2.1 最简单的对象

```
var Man = Backbone.Model.extend({
    initialize: function(){
        alert('Hey, you create me!');
    }});var man = new Man;
```

这个确实很简单了，只是定义了一个最基础的 Model，只是实现了 initialize 这个初始化方法，也称构造函数。这个函数会在 Model 被实例化时调用。

2.2 对象属性赋值的两种方法

第一种，直接定义，设置默认值。

```
var Man = Backbone.Model.extend({
    initialize: function(){
        alert('Hey, you create me!');
    },
    defaults: {
        name:'张三',
        age: '38'
    }});
var man = new Man;alert(man.get('name'));
```

第二种，赋值时定义

```
var Man = Backbone.Model.extend({
    initialize: function(){
        alert('Hey, you create me!');
    });var man = new
Man;man.set({name:'the5fire',age:'10'});alert(man.get('name'));
```

从这个对象的取值方式可以知道，属性在一个 Model 是以字典（或者类似字典）的方式存在的,第一种设定默认值的方式,只不过是实现了 Backbone 的 defaults 这个方法，或者是给 defaults 进行了赋值。

2.3 对象中的方法

```
var Man = Backbone.Model.extend({
    initialize: function(){
        alert('Hey, you create me!');
    },
    defaults: {
        name:'张三',
        age: '38'
    },
    aboutMe: function(){
        return '我叫' + this.get('name') + ',今年' + this.get('age') + '岁';
    });var man = new Man;alert(man.aboutMe());
```

也是比较简单，只是增加了一个新的属性，值是一个 function。说到这，不知道你是否发现，在所有的定义或者赋值操作中，都是通过字典的方式来完成的，

比如 extend Backbone 的 Model，以及定义方法，定义默认值。方法的调用和其他的语言一样，直接 . 即可，参数的定义和传递也一样。

2.4 监听对象中属性的变化

假设你有在对象的某个属性发生变化时去处理一些业务的话，下面的示例会有帮助。依然是定义那个类，不同的是我们在构造函数中绑定了 name 属性的 change 事件。这样当 name 发生变化是，就会触发这个 function。

```
var Man = Backbone.Model.extend({
    initialize: function(){
        alert('Hey, you create me!');
        //初始化时绑定监听
        this.bind("change:name",function(){
            var name = this.get("name");
            alert("你改变了 name 属性为：" + name);
        });
    },
    defaults: {
        name:'张三',
        age: '38'
    },
    aboutMe: function(){
        return '我叫' + this.get('name') + ',今年' + this.get('age') + '岁';
    }});var man = new Man;man.set({name:'the5fire'})  //触发绑定的 change 事件，alert。man.set({name:'the5fire.com'})  //触发绑定的 change 事件，alert。
```

2.5 为对象添加验证规则，以及错误提示

```
var Man = Backbone.Model.extend({
    initialize: function(){
        alert('Hey, you create me!');
        //初始化时绑定监听，change 事件会先于 validate 发生
        this.bind("change:name",function(){
            var name = this.get("name");
            alert("你改变了 name 属性为：" + name);
        });
        this.bind("invalid",function(model,error){
            alert(error);
        });
    },
    defaults: {
```

```
        name:'张三',
        age: '38'
    },
    validate:function(attributes){
        if(attributes.name == '') {
            return "name 不能为空！";
        }
    },
    aboutMe: function(){
        return '我叫' + this.get('name') + ',今年' + this.get('age') + '岁';
    }});var man = new Man;// 这种方式添加错误处理也行// man.on('invalid',
function(model, error){//          alert(error);// });
man.set({name:''}); //默认 set 时不进行验证//man.set({name:''},
{'validate':true});  //手动触发验证, set 时会触发man.save(); //save 时触发验证。
根据验证规则，弹出错误提示。
```

2.6 和服务器进行交互，对象的保存和获取

首先需要声明的是，这个例子需要后端配合，可以在 code 目录中找到对应的 py 文件，需要 webpy 和 mako 这两个库。 这里需要为对象定义一个 url 属性，调用 save 方法时会 post 对象的所有属性到 server 端，调用 fetch 方法是又会发送 get 请求到 server 端。接受数据和发送数据均为 json 格式:

```
var Man = Backbone.Model.extend({
    url:'/man/',
    initialize: function(){
        alert('Hey, you create me!');
        //初始化时绑定监听
        this.bind("change:name",function(){
            var name = this.get("name");
            alert("你改变了 name 属性为: " + name);
        });
        this.bind("error",function(model,error){
            alert(error);
        });
    },
    defaults: {
        name:'张三',
        age: '38'
    },
    validate:function(attributes){
        if(attributes.name == '') {
            return "name 不能为空！";
```

```
        }
    },
    aboutMe: function(){
        return '我叫' + this.get('name') + ',今年' + this.get('age') + '岁';
    }});var man = new Man;;man.set({name:'the5fire'});man.save();  //会发送 POST 到模型对应的 url,数据格式为 json{"name":"the5fire","age":38}//然后接着就是从服务器端获取数据使用方法 fetch([options])var man1 = new Man;//第一种情况，如果直接使用 fetch 方法，那么他会发送 get 请求到你 model 的 url 中，
    //你在服务器端可以通过判断是 get 还是 post 来进行对应的操作。man1.fetch();//第二种情况，在 fetch 中加入参数，如下：man1.fetch({url:'/man/'});//这样，就会发送 get 请求到/getmans/这个 url 中，//服务器返回的结果样式应该是对应的 json 格式数据，同 save 时 POST 过去的格式。
//不过接受服务器端返回的数据方法是这样的：man1.fetch({url:'/man/',
    success:function(model,response){
        alert('success');
        //model 为获取到的数据
        alert(model.get('name'));
    },error:function(){
        //当返回格式不正确或者是非 json 数据时，会执行此方法
        alert('error');
    }});
```

还有一点值得一提的是关于 url 和 urlRoot 的事情了，如果你设置了 url，那么你的 CRUD 都会发送对应请求到这个 url 上，但是这样有一个问题，就是 delete 请求，发送了请求，但是却没有发送任何数据，那么你在服务器端就不知道应该删除哪个对象（记录），所以这里又一个 urlRoot 的概念，你设置了 urlRoot 之后，你发送 PUT 和 DELETE 请求的时候，其请求的 url 地址就是：/baseurl/[model.id]，这样你就可以在服务器端通过对 url 后面值的提取更新或者删除对应的对象（记录）

补充一点，就是关于服务器的异步操作都是通过 Backbone.sync 这个方法来完成的，调用这个方法的时候会自动的传递一个参数过去，根据参数向服务器端发送对应的请求。比如你 save，backbone 会判断你的这个对象是不是新的，如果是新创建的则参数为 create，如果是已存在的对象只是进行了改变，那么参数就为 update，如果你调用 fetch 方法，那参数就是 read，如果是 destory，那么参数就是 delete。也就是所谓的 CRUD ("create", "read", "update", or "delete")，而这四种参数对应的请求类型为 POST，GET，PUT，DELETE。你可以在服务器根据这个 request 类型，来做出相应的 CRUD 操作。

关于 Backbone.sync 在后面会有如何自定义这一部分的章节。

上面服务器端的代码在 code 下可以找到，基于 webpy 和 mako 的。

第三章 Backbonejs 中的 Collections 实践

上一节介绍了 model 的使用，model 算是对现实中某一物体的抽象，比如你可以定义一本书的 model，具有书名（title）还有书页（page_num)等属性。仅仅用一个 Model 是不足以呈现现实世界的内容，因此基于 Model，这节我们来看 collection。collection 是 model 对象的一个有序的集合，也可以理解为是 model 的容器。概念理解起来十分简单，在通过几个例子来看一下，会觉得更容易理解。

3.1 关于 book 和 bookshelf 的例子

```
var Book = Backbone.Model.extend({

    defaults : {
        title:'default'
    },

    initialize: function(){
        //alert('Hey, you create me!');
    }
});
var BookShelf = Backbone.Collection.extend({
    model : Book});
var book1 = new Book({title : 'book1'});var book2 = new Book({title : 'book2'});var
book3 = new Book({title : 'book3'});
//var bookShelf = new BookShelf([book1, book2, book3]); //注意这里面是数组,或者
使用 add
var bookShelf = new BookShelf;
bookShelf.add(book1);bookShelf.add(book2);bookShelf.add(book3);bookShelf.remove
(book3);

//基于 underscore 这个 js 库，还可以使用 each 的方法获取 collection 中的数据
bookShelf.each(function(book){
    alert(book.get('title'));});
```

很容易理解吧。

3.2 使用 fetch 从服务器端获取数据

首先要在上面的的 Bookshelf 中定义 url，注意 collection 中并没有 urlRoot 这个属性。或者你直接在 fetch 方法中定义 url 的值，如下：

```
bookShelf.url = '/books/'; //注意这里bookShelf.fetch({
    success:function(collection, response, options){
        collection.each(function(book){
            alert(book.get('title'));
        });
    },error:function(collection, response, options){
        alert('error');
    }});
```

其中也定义了两个接受返回值的方法，具体含义我想很容易理解，返回正确格式(json)的数据，就会调用 success 方法，错误格式的数据就会调用 error 方法，当然 error 方法也看添加和 success 方法一样的形参。

对应的 BookShelf 的返回格式如下：
[{'title':'book0'},{'title':'book1'}.....]

3.3 reset 方法

这个方法的时候是要和上面的 fetch 进行配合的，collection 在 fetch 到数据之后，默认情况会调用 set 方法(set 方法会触发 collection 的 add 方法)，但是可以通过参数{reset: true}来手动触发 reset。这时你就需要在 collection 中定义 reset 方法或者是绑定 reset 方法。这里使用绑定演示：

```
var showAllBooks = function(){
    bookShelf.each(function(book){
        //将 book 数据渲染到页面的操作。
        document.writeln(book.get('title'));
    });}
bookShelf.bind('reset',showAllBooks);bookShelf.url = '/books/'; //注意这里
bookShelf.fetch({
    reset: true,    // 需要主动传递 reset，才会触发 reset
    success:function(collection, response, options){
        collection.each(function(book){
            alert(book.get('title'));
        });
    },error:function(collection, response, options){
        alert('error');
    }});
```

绑定的步骤要在 fetch 之前进行。

3.4 发送数据到 Server 端

创建数据，其实就是调用 collection 的 create 方法，POST 对应的 Model 对象（json 数据）到配置好的 url 上。之后会返回一个 model 的实例，如下面代码中的 onebook。

```
var NewBooks = Backbone.Collection.extend({
    model: Book,
    url: '/books/'});
var books = new NewBooks;
var onebook = books.create({
    title: "I'm coming",});
```

完整代码可以在 code 中找到，服务器端的代码后面会介绍。

第四章 Backbonejs 中的 Router 实践

前面介绍了 Model 和 Collection，基本上属于程序中静态的数据部分。这一节介绍 Backbone 中的 router，属于动态的部分，见名知意，router——路由的意思，显然是能够控制 url 指向哪个函数的。具体是怎么做的一会通过几个实例来看看。

在现在的单页应用中，所有的操作、内容都在一个页面上呈现，这意味着浏览器的 url 始终要定位到当前页面。那么一个页面中的左右的操作总不能都通过事件监听来完成，尤其是对于需要切换页面的场景以及需要分享、收藏固定链接的情况。因此就有了 router，通过 hash 的方式（即#page）来完成。不过随着浏览器发展，大多数的浏览器已经可以通过 history api 来操控 url 的改变，可以直接使用 /page 来完成之前需要 hash 来完成的操作，这种方式看起来更为直观一些。下面提供过几个 demo 来切实体会一番。

4.1 一个简单的例子

```
var AppRouter = Backbone.Router.extend({
    routes: {
        "*actions" : "defaultRoute"
    },
```

```
    defaultRoute : function(actions){
        alert(actions);
    }});
var app_router = new AppRouter;
Backbone.history.start();
```

需要通过调用 Backbone.history.start()方法来初始化这个 Router。

在页面上需要有这样的 a 标签：

```
<a href="#actions">testActions</a>
```

点击该链接时，便会触发 defaultRouter 这个方法。

4.2 这个 routes 映射要怎么传参数

看下面例子，立马你就知道了

```
var AppRouter = Backbone.Router.extend({

    routes: {
        "posts/:id" : "getPost",
        "*actions" : "defaultRoute"
    },

    getPost: function(id) {
        alert(id);
    },

    defaultRoute : function(actions){
        alert(actions);
    }});
var app_router = new AppRouter;Backbone.history.start();
```

对应的页面上应该有一个超链接：

```
<a href="#/posts/120">Post 120</a>
```

从上面已经可以看到匹配#标签之后内容的方法，有两种：一种是用“:”来把#后面的对应的位置作为参数；还有一种是“*”，它可以匹配所有的 url，下面再来演练一下。

```
var AppRouter = Backbone.Router.extend({
```

```
    routes: {
        "posts/:id" : "getPost",
        //下面对应的链接为<a href="#/download/user/images/hey.gif">downloa
gif</a>
        "download/*path": "downloadFile",
        //下面对应的链接为<a href="#/dashboard/graph">Load Route/Action View</a>
        ":route/:action": "loadView",
        "*actions" : "defaultRoute"
    },

    getPost: function(id) {
        alert(id);
    },

    defaultRoute : function(actions){
        alert(actions);
    },

    downloadFile: function( path ){
        alert(path); // user/images/hey.gif
    },

    loadView: function( route, action ){
        alert(route + "_" + action); // dashboard_graph
    }
});
var app_router = new AppRouter;Backbone.history.start();
```

4.3 手动触发 router

上面的例子都是通过页面点击触发 router 到对应的方法上，在实际的使用中，还存在一种场景就是需要在某一个逻辑中触发某一个事件，就像是 jQuery 中得 trigger 一样，下面的代码展示怎么手动触发 router。

```
routes: {
    "posts/:id" : "getPost",
    "manual": "manual",
    "*actions": "defaultRoute",},// 省略部分代码loadView: function( route,
action ){
    alert(route + "_" + action); // dashboard_graph},manual: function() {
    alert("call manual");
    app_router.navigate("/posts/" + 404, {trigger: true, replace: true});}
```

对应着在页面添加一个 a 标签： `<a href="#/manual">manual</a>` 然后点击这个链接，便会触发 posts/:id 对应的方法。

这里需要解释的是 navigate 后面的两个参数。trigger 表示触发事件，如果为 false，则只是 url 变化，并不会触发事件，replace 表示 url 替换，而不是前进到这个 url，意味着启用该参数，浏览器的 history 不会记录这个变动。

完整代码依然在 `code` 中可以找到。

第五章 Backbonejs 中的 View 实践

前面介绍了存放数据的 Model 和 Collection 以及对用户行为进行路由分发的 Router（针对链接）。这一节终于可以往页面上放点东西来玩玩了。这节就介绍了 Backbone 中得 View 这个模块。Backbone 的 View 是用来显示你的 model 中的数据到页面的，同时它也可用来监听 DOM 上的事件然后做出响应。但是这里要提一句的是，相比于 Angularjs 中 model 变化之后页面数据自动变化的特性，Backbone 要手动来处理。至于这两种方式的对比，各有优劣，可以暂时不关心。

下面依然是通过几个示例来介绍下 view 的功能,首先给出页面的基本模板：

```
<!DOCTYPE html><html><head>
    <title>the5fire-backbone-view</title></head><body>
    <div id="search_container"></div>

    <script type="text/template" id="search_template">
        <label><%= search_label %></label>
        <input type="text" id="search_input" />
        <input type="button" id="search_button" value="Search" />
    </script><script
src="http://the5fireblog.b0.upaiyun.com/staticfile/jquery-1.10.2.js"></script><
script
src="http://the5fireblog.b0.upaiyun.com/staticfile/underscore.js"></script><scr
ipt
src="http://the5fireblog.b0.upaiyun.com/staticfile/backbone.js"></script><scrip
t>(function ($) {
    //此处添加下面的试验代码})(jQuery);</script></body></html>
```

5.1 一个简单的 view

```
var SearchView = Backbone.View.extend({
    initialize: function(){
```

```
        alert('init a SearchView');
    }});var searchView = new SearchView();
```

是不是觉得很没有技术含量，所有的模块定义都一样。

5.2 el 属性

这个属性用来引用 DOM 中的某个元素，每一个 Backbone 的 view 都会有这么个属性，如果没有显示声明，Backbone 会默认的构造一个，表示一个空的 div 元素。el 标签可以在定义 view 的时候在属性中声明，也可以在实例化 view 的时候通过参数传递。

```
var SearchView = Backbone.View.extend({
    initialize: function(){
        alert('init a SearchView');
    }});
var searchView = new SearchView({el: $("#search_container")});
```

这段代码简单的演示了在实例化的时候传递 el 属性给 View。下面我们来看看模板的渲染。

```
var SearchView = Backbone.View.extend({
    initialize: function(){
    },
    render: function(context) {
        //使用 underscore 这个库，来编译模板
        var template = _.template($("#search_template").html(), context);
        //加载模板到对应的 el 属性中
        $(this.el).html(template);
    }});var searchView = new SearchView({el:
$("#search_container")});searchView.render({search_label: "搜索渲染"});  //这个
reander 的方法可以放到 view 的构造函数中,这样初始化时就会自动渲染
```

运行页面之后，会发现 script 模板中的 html 代码已经添加到了我们定义的 div 中。

这里面需要注意的是在模板中定义的所有变量必须在 render 的时候传递参数过去，不然就会报错。 关于 el 还有一个东西叫做$el,这个东西是对 view 中元素的缓存。

5.3 再来看 view 中 event 的使用

页面上的操作除了可以由之前的 router 来处理之外，在一个 view 中定义元素，还以可以使用 event 来进行事件绑定。这里要注意的是在 view 中定义的 dom 元素是指你 el 标签所定义的那一部分 dom 节点，event 进行事件绑定时会在该节点范围内查找。

来，继续看代码。

```
var SearchView = Backbone.View.extend({
    el: "#search_container",

    initialize: function(){
        this.render({search_label: "搜索按钮"});
    },
    render: function(context) {
        //使用 underscore 这个库，来编译模板
        var template = _.template($("#search_template").html(), context);
        //加载模板到对应的 el 属性中
        $(this.el).html(template);
    },

    events:{  //就是在这里绑定的
        'click input[type=button]' :'doSearch'  //定义类型为 button 的 input 标签的点击事件，触发函数 doSearch

    },

    doSearch: function(event){
        alert("search for " + $("#search_input").val());
    }
});
var searchView = new SearchView();
```

自己运行下，是不是比写
$("input[type=button]").bind('click',function(){})好看多了。

5.4 View 中的模板

上面已经简单的演示了模板的用法，如果你用过 django 模板的话，你会发现模板差不多都是那么回事。上面只是简单的单个变量的渲染，那么逻辑部分怎么处理呢，下面来看下。

把最开始定义的模板中的内容换成下面这个。

```
<ul><% _.each(labels, function(name) { %>
    <% if(name != "label2") {%>
    <li><%= name %></li>
    <% } %><% }); %></ul>
```

下面是 js 代码

```
var SearchView = Backbone.View.extend({
    el: "#search_container",

    initialize: function(){
        var labels = ['label1', 'label2', 'label3'];
        this.render({labels: labels});
    },

    render: function(context) {
        //使用 underscore 这个库，来编译模板
        var template = _.template($("#search_template").html(), context);
        //加载模板到对应的 el 属性中
        $(this.el).html(template);
    },
});
var searchView = new SearchView();
```

再次运行，有木有觉得还不错，模板中使用的就基本的 js 语法。

总结一下，关于 view 中的东西就介绍这么多，文档上还有几个其他的属性，不过大体用法都一致。在以后的实践中用到在介绍。

第六章 实战演练：todos 分析（一）

经过前面的几篇文章，Backbone 中的 Model, Collection，Router，View，都简单的介绍了一下，我觉得看完这几篇文章，差不多就能开始使用 Backbone 来做东西了，所有的项目无外乎对这几个模块的使用。不过对于实际项目经验少些的同学，要拿起来用估计会有些麻烦。因此这里就先找个现成的案例分析一下。

6.1 大家都来分析 todos

关于 Backbonejs 实例分析的文章网上真是一搜一大把，之所以这么多，第一是这东西需求简单，不用花时间到理解情景中；第二是代码就是官方的案例，顺手可得，也省得去找了，自己琢磨一个不得花时间吗。

于是就有人问了，丫们都在分析 todos，能不能有点新意呢。这问题要我说，如果你真的能把 todos 搞明白了，那其他的也就不用去看了。不管是看谁的分析，把这个搞明白的。所有的项目大体思路都差不多。尤其是对于这样的 MVC 的模型，就是往对应的模块里填东西。因此，不管有多少人都在分析这玩意，自己弄懂了才是应该关心的。

话虽如此，不同于网络上的绝大部分的分析的是，the5fire 除了分析这个，还是对其进行了扩充，另外在后面也会有真实的案例。但我也是从这些案例的代码中汲取的营养。

补充一下，新版的 todos 代码相较于之前简直清晰太多，完全可以当做一个前端的范本来学习、模仿。

6.2 获取代码

todos 的代码这里下载：https://github.com/jashkenas/backbone/ ，建议自己 clone 一份到本本地。线上的地址是：
http://backbonejs.org/examples/todos/index.html

clone 下来之后可以在 example 中找到 todos 文件夹，文件结构如下：：

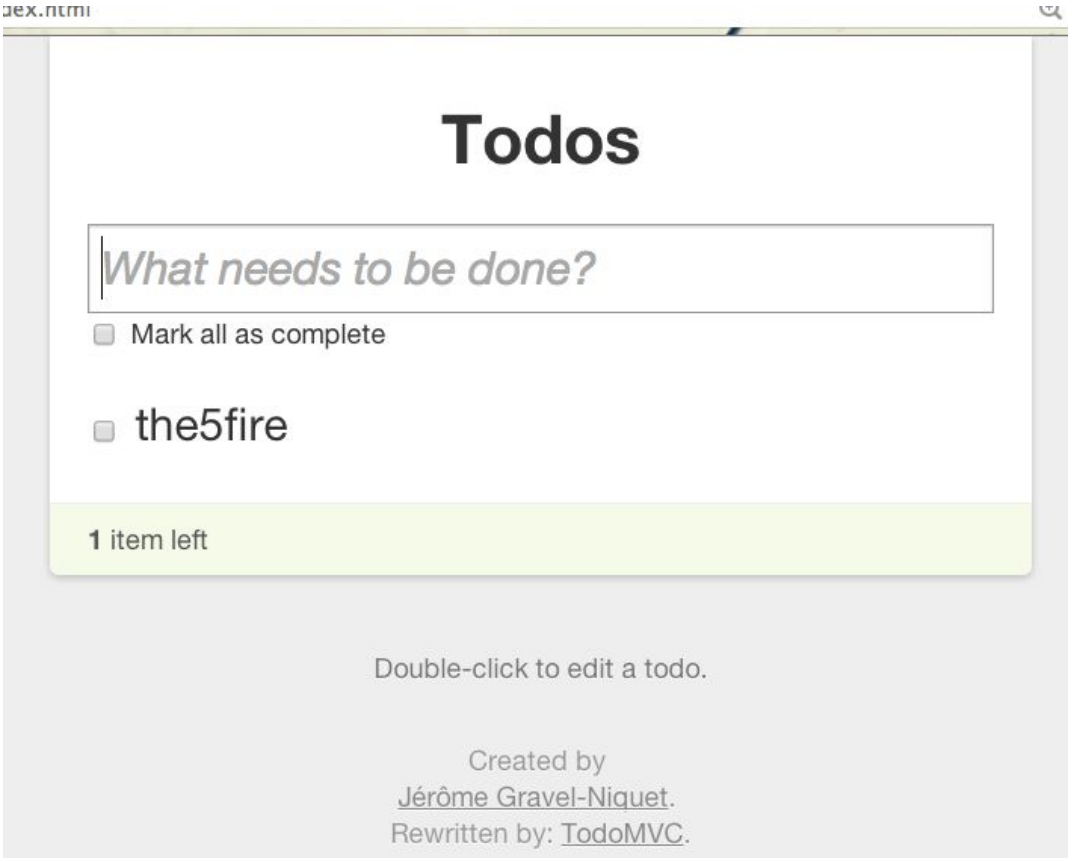
```
examples
├── backbone.localStorage.js
└── todos
    ├── destroy.png
    ├── index.html
    ├── todos.css
    └── todos.js

1 directory, 5 files
```

用浏览器打开 index.html 文件，推荐使用 chome 浏览器，就可以看到和官网一样的界面了。关键代码都在 todos.js 这个文件里。

6.3 功能分析

首先来分析下页面上有哪些功能：



从这个界面我们可以总结出来,下面这些功能:

```
* 任务管理
    添加任务
    修改任务
    删除任务
* 统计
    任务总计
    已完成数目
```

总体上就这几个功能。

这个项目仅仅是在 web 端运行的，没有服务器进行支持，因此在项目中使用了一个叫做 backbone-localstorage 的 js 库，用来把数据存储到前端。

6.4 从模型下手

因为 Backbone 为 MVC 模式，根据对这种模式的使用经验，我们从模型开始分析。首先我们来看 Model 部分的代码:

```
/***基本的 Todo 模型，属性为：content,order,done。**/var Todo =
Backbone.Model.extend({
```

```
    // 设置默认的属性
    defaults: {
        title: "empty todo...",
        order: Todos.nextOrder(),
        done: false
    },

    // 设置任务完成状态
    toggle: function() {
        this.save({done: !this.get("done")});
    }});
```

这段代码是很好理解的，不过我依然是画蛇添足的加上了一些注释。这个 Todo 显然就是对应页面上的每一个任务条目。那么显然应该有一个 collection 来统治（管理）所有的任务，所以再来看 collection：

```
/***Todo 的一个集合，数据通过 localStorage 存储在本地。**/
var TodoList = Backbone.Collection.extend({

    // 设置 Collection 的模型为 Todo
    model: Todo,
    //存储到浏览器，以 todos-backbone 命名的空间中
    //此函数为 Backbone 插件提供 地址：
https://github.com/jeromegn/Backbone.localStorage
    localStorage: new Backbone.LocalStorage("todos-backbone"),

    //获取所有已经完成的任务数组
    done: function() {
        return this.where({done: true});
    },

    //获取任务列表中未完成的任务数组
    //这里的 where 在之前是没有的，但是语法上更清晰了
    //参考文档：http://backbonejs.org/#Collection-where
    remaining: function() {
        return this.where({done: false});
    },

    //获得下一个任务的排序序号，通过数据库中的记录数加 1 实现。
    nextOrder: function() {
        if (!this.length) return 1;

        return this.last().get('order') + 1;  // last 获取 collection 中最后一个
元素
```

```
    },

    //Backbone 内置属性，指明 collection 的排序规则。
    comparator: 'order'});
```

collection 的主要功能有以下几个：：

```
1、获取完成的任务;
2、获取未完成的任务;
3、获取下一个要插入数据的序号;
4、按序存放 Todo 对象。
```

如果你看过第一版的话，这里 Backbone 新的属性和方法(comparator 和 where)用起来更加符合语义了。

这篇文章先分析到这里，下篇文章继续分析。

第七章 实战演练：todos 分析（二）View 的应用

在上一篇文章中我们把 todos 这个实例的数据模型进行了简单的分析，有关于数据模型的操作也都知道了。接着我们来看剩下的两个 view 的模型，以及它们对页面的操作。

7.1 为什么要两个 view

首先要分析下，这俩 view 是用来干嘛的。有人可能会问了，这里不就是一个页面吗？一个 view 掌控全局不就完了？

我觉得这就是新手和老手的主要区别之一，喜欢在一个方法里面搞定一切，喜欢把东西都拧到一块去，觉得这样看起来容易。熟不知，这样的代码对于日后的扩展会造成很大的麻烦。因此我们需要学习下优秀的设计，从好的代码中汲取营养。

这里面的精华就是，将对数据的操作和对页面的操作进行分离，也就是现在代码里面 TodoView 和 AppView。前者的作用是对把 Model 中的数据渲染到模板中;后者是对已经渲染好的数据进行处理。两者各有分工，TodoView 可以看做是加工后的数据，这个数据就是待使用的 html 数据。

7.2 TodoView 的代码分析

TodoView 是和 Model 一对一的关系，在页面上一个 View 也就展示为一个 item。除此之外，每个 view 上还有其他的功能，比如编辑模式，展示模式，还有对用户的输入的监听。详细还是来看下代码：

```
// 首先是创建一个全局的 Todo 的 collection 对象
var Todos = new TodoList;
// 先来看 TodoView，作用是控制任务列表var TodoView = Backbone.View.extend({

    //下面这个标签的作用是，把 template 模板中获取到的 html 代码放到这标签中。
    tagName:  "li",

    // 获取一个任务条目的模板,缓存到这个属性上。
    template: _.template($('#item-template').html()),

    // 为每一个任务条目绑定事件
    events: {
        "click .toggle"   : "toggleDone",
        "dblclick .view"  : "edit",
        "click a.destroy" : "clear",
        "keypress .edit"  : "updateOnEnter",
        "blur .edit"      : "close"
    },

    //在初始化时设置对 model 的 change 事件的监听
    //设置对 model 的 destroy 的监听，保证页面数据和 model 数据一致
    initialize: function() {
        this.listenTo(this.model, 'change', this.render);
        //这个 remove 是 view 的中的方法，用来清除页面中的 dom
        this.listenTo(this.model, 'destroy', this.remove);
    },

    // 渲染 todo 中的数据到 item-template 中，然后返回对自己的引用 this
    render: function() {
        this.$el.html(this.template(this.model.toJSON()));
        this.$el.toggleClass('done', this.model.get('done'));
        this.input = this.$('.edit');
        return this;
    },

    // 控制任务完成或者未完成
    toggleDone: function() {
        this.model.toggle();
    },
```

```
    // 修改任务条目的样式
    edit: function() {
        $(this.el).addClass("editing");
        this.input.focus();
    },

    // 关闭编辑模式，并把修改内容同步到 Model 和界面
    close: function() {
        var value = this.input.val();
        if (!value) { //无值内容直接从页面清除
            this.clear();
        } else {
            this.model.save({title: value});
            this.$el.removeClass("editing");
        }
    },

    // 按下回车之后，关闭编辑模式
    updateOnEnter: function(e) {
      if (e.keyCode == 13) this.close();
    },

    // 移除对应条目，以及对应的数据对象
    clear: function() {
        this.model.destroy();
    }});
```

7.3 AppView 的代码分析

再来看 AppView，功能是显示所有任务列表，显示整体的列表状态（如：完成多少，未完成多少）

```
//以及任务的添加。主要是整体上的一个控制var AppView = Backbone.View.extend({

    //绑定页面上主要的 DOM 节点
    el: $("#todoapp"),

    // 在底部显示的统计数据模板
    statsTemplate: _.template($('#stats-template').html()),

    // 绑定 dom 节点上的事件
    events: {
        "keypress #new-todo":  "createOnEnter",
```

```
        "click #clear-completed": "clearCompleted",
        "click #toggle-all": "toggleAllComplete"
    },

    //在初始化过程中，绑定事件到 Todos 上，
    //当任务列表改变时会触发对应的事件。
    //最后从 localStorage 中 fetch 数据到 Todos 中。
    initialize: function() {
        this.input = this.$("#new-todo");
        this.allCheckbox = this.$("#toggle-all")[0];

        this.listenTo(Todos, 'add', this.addOne);
        this.listenTo(Todos, 'reset', this.addAll);
        this.listenTo(Todos, 'all', this.render);

        this.footer = this.$('footer');
        this.main = $('#main');

        Todos.fetch();
    },

    // 更改当前任务列表的状态
    render: function() {
        var done = Todos.done().length;
        var remaining = Todos.remaining().length;

        if (Todos.length) {
            this.main.show();
            this.footer.show();
            this.footer.html(this.statsTemplate({done: done, remaining:
remaining}));
        } else {
            this.main.hide();
            this.footer.hide();
        }

        //根据剩余多少未完成确定标记全部完成的 checkbox 的显示
        this.allCheckbox.checked = !remaining;
    },

    // 添加一个任务到页面 id 为 todo-list 的 div/ul 中
    addOne: function(todo) {
        var view = new TodoView({model: todo});
        this.$("#todo-list").append(view.render().el);
```

```
    },

    // 把 Todos 中的所有数据渲染到页面,页面加载的时候用到
    addAll: function() {
        Todos.each(this.addOne, this);
    },

    //生成一个新 Todo 的所有属性的字典
    newAttributes: function() {
        return {
            content: this.input.val(),
            order:   Todos.nextOrder(),
            done:    false
        };
    },

    //创建一个任务的方法，使用 backbone.collection 的 create 方法。
    //将数据保存到 localStorage,这是一个 html5 的 js 库。
    //需要浏览器支持 html5 才能用。
    createOnEnter: function(e) {
        if (e.keyCode != 13) return;
        if (!this.input.val()) return;

        //创建一个对象之后会在 backbone 中动态调用 Todos 的 add 方法，该方法已绑定
addOne。
        Todos.create({title: this.input.val()});
        this.input.val('');
    },

    //去掉所有已经完成的任务
    clearCompleted: function() {
        // 调用 underscore.js 中的 invoke 方法,对过滤出来的 todos 调用 destroy 方法
        _.invoke(Todos.done(), 'destroy');
        return false;
    },

    //处理页面点击标记全部完成按钮
    //处理逻辑：如果标记全部按钮已选，则所有都完成，如果未选，则所有的都未完成。
    toggleAllComplete: function () {
        var done = this.allCheckbox.checked;
        Todos.each(function (todo) { todo.save({'done': done}); });
    }});
```

通过上面的代码，以及其中的注释，我们认识里面每个方法的作用。下面来看最重要的，页面部分。

7.4 页面模板分析

在前几篇的 view 介绍中我们已经认识过了简单的模板使用，以及变量参数的传递，如：

```
<script type="text/template" id="search_template">

        <label><%= search_label %></label>
        <input type="text" id="search_input" />
        <input type="button" id="search_button" value="Search" />
</script>
```

既然能定义变量，那么就能使用语法，如同 django 模板，那来看下带有语法的模板，也是上面的两个 view 用到的模板，我想这个是很好理解的。

```
<script type="text/template" id="item-template">
    <div class="view">
        <input class="toggle" type="checkbox" <%= done ? 'checked="checked"' : '' %> />
        <label><%- title %></label>
        <a class="destroy"></a>
    </div>
    <input class="edit" type="text" value="<%- title %>" /></script>

<script type="text/template" id="stats-template">
    <% if (done) { %>
        <a id="clear-completed">Clear <%= done %> completed <%= done == 1 ? 'item' : 'items' %></a>
    <% } %>
    <div class="todo-count"><b><%= remaining %></b> <%= remaining == 1 ? 'item' : 'items' %> left</div></script>
```

简单的语法，上面的那个对应 TodoView。有木有觉得比之前的那一版简洁太多了，有木有！！啥叫代码的美感，对比一下就知道了。

这一篇文章就先到此为止，文章中我们了解到在 todos 这个实例中，view 的使用，以及具体的 TodoView 和 AppView 中各个函数的作用，这意味着所有的肉和菜都已经放到你碗里了，下面就是如何吃下去的问题了。

下一篇我们一起来学习 todos 的整个流程。

第八章 实战演练：todos 分析（三）总结

在前两篇文章中，我们已经对这个 todos 的功能、数据模型以及各个模块的实现细节进行了分析，这篇文章我们要对前面的分析进行一个整合。

首先让我们来回顾一下我们分析的流程：1. 先对页面功能进行了分析；2. 然后又分析了数据模型；3. 最后又对 view 的功能和代码进行了详解。你是不是觉得这个分析里面少了点什么？没错，就知道经验丰富的你已经看出来了，这里面少了对于流程的分析。这篇文章就对整体流程进行分析。

所以从我的分析中可以看的出来，我是先对各个原材料进行分析，然后再整体的分析（当然前提是我是理解流程的），这并不是分析代码的唯一方法，有时我也会采用跟着流程分析代码的方法。当然还有很多其他的分析方法，大家都有自己的套路嘛。

下面简单的说说流程分析的方法。记得多年前在学 vb 的时候，分析一个完整项目代码的时候，习惯从程序的入口点开始分析。虽然 web 网站和桌面软件的实现不同，但是大致思路是一样的（同时也有网站即软件的说法，在 RESTful 架构中）。所以我们要先找到网站的入口点所在。

和桌面应用项目的分析一样，网站的入口点就在于网页加载的时候。对于 todos，自然就是在页面加载完之后执行的操作了，然后就看到下面的代码。

首先是对 AppView 的一个实例化：

```
var App = new AppView;
```

实例化，自然就会调用构造函数：

```
//在初始化过程中，绑定事件到 Todos 上，//当任务列表改变时会触发对应的事件。//最后从 localStorage 中 fetch 数据到 Todos 中。initialize: function() {
    this.input = this.$("#new-todo");
    this.allCheckbox = this.$("#toggle-all")[0];

    this.listenTo(Todos, 'add', this.addOne);
    this.listenTo(Todos, 'reset', this.addAll);
    this.listenTo(Todos, 'all', this.render);
```

```
    this.footer = this.$('footer');
    this.main = $('#main');

    Todos.fetch();},
```

注意其中的 Todos.fetch()方法，前面说过，这个项目是在客户端保存数据，所以使用 fetch 方法并不会发送请求到服务器。另外在前面关于 collection 的单独讲解中我们也介绍了 fetch 执行完成之后，会调用 set（默认）或者 reset（需要手动设置{reset: true} ）。所以在没有指明 fetch 的 reset 参数的情况下，backbonejs 的 Collection 中的 set 方法会遍历 Todos 的内容并且调用 add 方法。

在 initialize 中我们绑定了 add 到 addOne 上，因此在 fetch 的时候会 Backbonejs 会帮我们调用 addOne（其实也是在 collection 的 set 方法中）。和 collection 中的 set 类似的，我们可以自定义 reset 方法，自行来处理 fetch 到得数据，但是需要在 fetch 时手动添加 reset 参数。

PS: 感谢网友指正

这里先来看下我们绑定到 reset 上的 addAll 方法是如何处理 fetch 过来的数据的:

```
// 添加一个任务到页面 id 为 todo-list 的 div/ul 中addOne: function(todo) {
    var view = new TodoView({model: todo});
    this.$("#todo-list").append(view.render().el);},
// 把 Todos 中的所有数据渲染到页面,页面加载的时候用到addAll: function() {
    Todos.each(this.addOne, this);},
```

在 addAll 中调用 addOne 方法，关于 Todos.each 很好理解，就是语法糖（简化的 for 循环）。关于 addOne 方法的细节下面介绍。

然后再来看添加任务的流程，一个良好的代码命名风格始终是让人满心欢喜的。因此很显然，添加一个任务，自然就是 addOne,其实你看 events 中的绑定也能知道，先看一下绑定：

```
// 绑定 dom 节点上的事件events: {
    "keypress #new-todo":  "createOnEnter",
    "click #clear-completed": "clearCompleted",
    "click #toggle-all": "toggleAllComplete"},
```

这里并没有 addOne 方法的绑定，但是却有 createOnEnter，语意其实一样的。来看主线，createOnEnter 这个方法：

```
//创建一个任务的方法，使用 backbone.collection 的 create 方法。将数据保存到 localStorage,这是一个 html5 的 js 库。需要浏览器支持 html5 才能用。createOnEnter: function(e) {
    if (e.keyCode != 13) return;
    if (!this.input.val()) return;

    //创建一个对象之后会在 backbone 中动态调用 Todos 的 add 方法，该方法已绑定 addOne。
    Todos.create({title: this.input.val()});
    this.input.val('');},
```

注释已写明,Todos.create 会调用 addOne 这个方法。由此顺理成章的来到 addOne 里面：

```
//添加一个任务到页面 id 为 todo-list 的 div/ul 中 addOne: function(todo) {
    var view = new TodoView({model: todo});
    this.$("#todo-list").append(view.render().el);},
```

在里面实例化了一个 TodoView 类，前面我们说过，这个类是主管各个任务的显示的。具体代码就不细说了。

有了添加再来看更新，关于单个任务的操作，我们直接找 TodoView 就 ok 了。所以直接找到

```
// 为每一个任务条目绑定事件 events: {
    "click .toggle"   : "toggleDone",
    "dblclick .view"  : "edit",
    "click a.destroy" : "clear",
    "keypress .edit"  : "updateOnEnter",
    "blur .edit"      : "close"},
```

其中的 edit 事件的绑定就是更新的一个开头，而 updateOnEnter 就是更新的具体动作。所以只要搞清楚这俩方法的作用一切就明了了。这里同样不用细说。

在往后还有删除一条记录以及清楚已有记录的功能，根据上面的分析过程，我想大家都很容易的去‘顺藤模瓜’。

关于 Todos 的分析到此就算完成了。

在下一篇文章中我们将一起来学习通过 web.py 来搭建 web 服务器，以及简单的数据库的使用。

第九章 后端环境搭建：web.py 的使用

前面都是前端的一些内容，但是要想做出一个能用的东西，始终是不能脱离后端的。因此这一节主要介绍如何使用 python 的一个 web 框架 webpy。我想读我这个教程的同学大多都是前端，对后端没有什么感觉。因此关于后端的介绍以能用为主，不涉及太多的后端的东西。

9.1 python 是什么

简单来说 Python 和 JavaScript 一样，是一个动态语言，运行在服务器端。语法类似于程序伪码，或者说类似于自然语言。过多的语法和关键字就不再介绍。只需要记住 Python 是用缩进来判断语法块的，不像 js 用大括号。

9.2 webpy 是什么

和 Backbonejs 是 js 的一个框架一样，webpy 是 python 的一个简单的 web 开发的框架。可以通过简单的几行代码启动一个 web 服务（虽然只是输出 helloworld ^_^）。用它可以简单的满足咱们的开发需求。

因为是基于 Python 的框架，因此需要先安装 Python 环境，具体怎么装就不细说了，到 http://python.org/download/ 安装 python2.7.6 这个版本。

之后安装 webpy 官网的说明，通过命令安装 webpy：

```
pip install web.py
或者
easy_install web.py

注意：linux 下非 root 用户需要 sudo
```

9.3 来一个 Helloworld

安装好之后，直接把 webpy 网站上的那段代码，贴到的用编辑器打开的文件中，保存为 server.py。webpy 网站代码如下：

```
import web
urls = (
    '/', 'index')app = web.application(urls, globals())
class index:
    def GET(self):
        return 'Hello, World!'
```

```
if __name__ == "__main__":
    app.run()
```

然后在 server.py 的同目录下执行：

```
python server.py
```

之后命令行会输出：

```
http://0.0.0.0:8080/
```

这个提示，现在你在浏览器访问 http://127.0.0.1:8080 ，就会看到熟悉的 helloworld，是不是超级简单。

9.4 简单构建 api 接口

在上面代码的基础上，按照前面 backboneModel 的定义，我们需要一个 todo 这模型的对应的链接，这个链接应该返回 json 格式的数据。并且能够支持 post、put、get、delete 这四个请求。现在来看接口部分的代码：

```
#添加 todo 相关的 urlsurls = (
    '/', 'index',  #返回首页
    '/todo/(\d+)/', 'todo',  # 处理前端 todo 的请求,操作对应的 todo
    '/todo/', 'todos',  # 处理前端 todo 的整体请求,主要是获取所有的 todo 数据)

#添加接口的处理代码class todo:
    def GET(self, todo_id=None):
        context = {
            "title": "下午 3 点,coding",
            "order": 0,
            "done": False,
        }
        return json.dumps(context)
#处理整体的请求class todos:
    def GET(self):
        result = []
        result.append({
            "title": "下午 3 点,coding",
            "order": 0,
            "done": False,
        })
        return json.dumps(result)
```

添加完这部分代码之后，启动 server.py。访问 http://localhost:8080/todo/ 就能看到数据了，这里只是实现了 get 方法，其他的方法在下一篇中介绍。

9.5 加入数据库 sqlite

关于数据存储部分，我们使用 sqlite 数据库。sqlite 的好处就是不需要安装即可使用。这样可以省去在数据库安装方面的折腾。

sqlite 的介绍就不多说了，感兴趣的同学想必已经在查 sqlite 相关的东西了。这里只是演示在 webpy 中如何操作 sqlite。

具体依然看代码:

```
#使用 sqlite3 操作数据库 import sqlite3conn = sqlite3.connect('todos.db')
#把 todo 改为这样: class todo:
    def GET(self, todo_id=None):
        cur = conn.cursor()
        cur.execute(sql_query + ' where id=?', (todo_id, ))
        todo = cur.fetchone()
        cur.close()

        # 先用这种比较傻的方式
        context = {
            "id": todo[0],
            "title": todo[1],
            "order": todo[2],
            "done": todo[3],
        }
        return json.dumps(context)
class todos:
    def GET(self):
        result = []
        cur = conn.cursor()
        cur.execute(sql_query)
        todos = cur.fetchall()
        cur.close()

        for todo in todos:
            result.append({
                "id": todo[0],
                "title": todo[1],
                "order": todo[2],
```

```
                "done": todo[3],
            })
        return json.dumps(result)
```

完整代码可以在 code 文件夹找到。使用时，先运行 init_sqlite.py 这个文件，会帮你创建一个 sqlite 的数据库，并且插入一条数据，然后运行 server.py 就可以在浏览器访问 http://localhost:8080/todo/ 或者 http://localhost:8080/todo/1/ 看到输出数据了。

9.6 总结

这里打算用 webpy+sqlite 来完成后台主要是想到这个东西比 Django+Mysql 那一套搭建起来比较容易。有兴趣看 Django 后台搭建的可以看这篇文章： django 开发环境搭建及使用 。

这里没有使用 webpy 自带的 db 模块进行数据的操作，主要是文档和案例都不全，并且源码看起来挺绕。用 Python 自带的模块显然操作起来有点笨拙，之后会对这个数据操作部分进行简单的封装。

第十章 实战演练：扩展 todos 到 Server 端（backbonejs+webpy）

上一节简单介绍了怎么使用 webpy 搭建一个后端的接口服务，这一节就来简单实现一下。

10.1 项目结构

首先还是来看下项目的结构，然后再一步一步的分析，结构如下：

```
src
├── index.html
├── init_sqlite.py
├── models.py
├── server.py
├── static
│   ├── backbone.js
│   ├── destroy.png
│   ├── jquery.js
│   ├── json2.js
```

```
|   ├── todos.css
|   ├── todos.js
|   └── underscore.js
└── todos.db
```

以上结构可以分为四个部分：模板、静态资源、后端逻辑处理、后端数据处理，其实最后两个都属于后端部分。

因为模板和静态资源和之前没有太大差异，因此合并在一起介绍。首先来看后端的接口。

10.2 后端接口

相对于前端的各种 model、collection 和 view，后端显得比较简单。只需要提供可访问的接口，并且根据 POST、PUT、DELETE、GET 这四种操作完成对数据库的 CRUD 即可（Create,Read,Update,Delete）。

先来看 models 中的代码，这里对 todo 表的操作进行了简单的封装：

```
#coding:utf-8import web
db = web.database(dbn='sqlite', db="todos.db")
class Todos(object):
    @staticmethod
    def get_by_id(id):
        return db.select('todos', where="id=$id", vars=locals())

    @staticmethod
    def get_all():
        return db.select('todos')

    @staticmethod
    def create(**kwargs):
        db.insert('todos', **kwargs)

    @staticmethod
    def update(**kwargs):
        db.update('todos', where="id=$id", vars={"id": kwargs.pop('id')},
**kwargs)

    @staticmethod
    def delete(id):
        db.delete('todos', where="id=$id", vars=locals())
```

代码很简单，从方法的命名上就知道要完成的功能是什么，这里不得不说一句，任何语言中好的变量或方法的命名，胜过大段的注释。

model 部分没有具体的业务逻辑，只是针对数据库进行 CRUD 操作。下面来看给浏览器提供接口的部分。

server 部分，提供了前端浏览器需要访问的接口，同时也提供了页面初始加载时的渲染页面的功能。server.py 的代码如下：

```
#coding:utf-8import json
import webfrom models import Todos
urls = (
    '/', 'index',  #返回首页
    '/todo', 'todo',  #  处理 POST 请求
    '/todo/(\d*)', 'todo',  # 处理前端 todo 的请求,对指定记录进行操作
    '/todos/', 'todos',  # 处理前端 todo 的请求，返回所有数据)
app = web.application(urls, globals())
render = web.template.render('')
# 首页class index:
    def GET(self):
        # 渲染首页到浏览器
        return render.index()
class todo:
    def GET(self, todo_id=None):
        result = None
        itertodo = Todos.get_by_id(id=todo_id)
        for todo in itertodo:
            result = {
                "id": todo.id,
                "title": todo.title,
                "order": todo._order,
                "done": todo.done == 1,
            }
        return json.dumps(result)

    def POST(self):
        data = web.data()
        todo = json.loads(data)
        # 转换成_order, order 是数据库关键字, sqlite3 报错
        todo['_order'] = todo.pop('order')
        Todos.create(**todo)

    def PUT(self, todo_id=None):
        data = web.data()
```

```
        todo = json.loads(data)
        todo['_order'] = todo.pop('order')
        Todos.update(**todo)

    def DELETE(self, todo_id=None):
        Todos.delete(id=todo_id)

class todos:
    def GET(self):
        todos = []
        itertodos = Todos.get_all()
        for todo in itertodos:
            todos.append({
                "id": todo.id,
                "title": todo.title,
                "order": todo._order,
                "done": todo.done == 1,
            })
        return json.dumps(todos)
if __name__ == "__main__":
    app.run()
```

相对于 model.py 来说，这里做了些数据转换的操作，如前端 backbone 通过 ajax 发过来的数据需要转换之后才能存入数据库，而从数据库取出的数据也要稍加处理才能符合前端 todos.js 中定义的 model 的要求。

在这个 server 中，提供了三个四个 url，依次功能为：首页加载、单个 todo 创建、单个 todo 查询修改和删除、查询全部。分成四个也主要是依据所选框架 webpy 的特性。

在 url 之后，是对应一个具体的 class，url 接受到的请求将有对应的 class 来处理，比如说 /todo 这个 url，对应的处理请求的 class 就是 todo。另外对应浏览器端发过来的 POST、GET、PUT、DELETE 请求，class 对应的也是相应的方法。这也是选 webpy 的一个原因。

说我了后端提供的接口，以及如何进行处理的原理。我们来看如何修改前端的代码，才能让数据发送到后端来。

10.3 修改 todos，发送数据到后端

这个部分改动比较小，就不贴代码了。有需要的可以到 code 中看。

之前的数据是存在 localstorage 中，是因为引用了 localStorage.js 文件，并且在 collection 中声明了 localStorage: new Backbone.LocalStorage("todos-backbone") 。

在修改的时候有三个地方需要修改，第一是 model 的定义，部分代码：

```
var Todo = Backbone.Model.extend({
    urlRoot: '/todo',
    ......
```

第二个就是 collection 的修改，去掉了 localStorage 的声明，并添加 url：

```
var TodoList = Backbone.Collection.extend({
    url: '/todos/',
    ......
```

这样就搞定了。

10.4 Demo 的使用

在 code 中，如果想要把我的 demo 在本地运行的话，需要首先运行下 python init_sqlite.py 来初始化 sqlite3 的数据库，运行完之后会在本地生成一个 todos.db 的数据库文件。

之后，就可以通过运行 python server.py ，然后访问命令行提示的网址就可以使用了。

最后稍稍总结一下，我觉得到这一章为止，对技术比较认真、比较有追求的同学应该知道怎么通过 backbonejs 和 webpy 把前后端连起来了。所有的这些文章只是为了帮你打开一扇门，或者仅仅只是一盏灯，具体你的业务逻辑还是需要通过自己的思考来解决。妄图让别人帮你实现业务逻辑的人都是切实的不思上进的菜鸟。

另外，关于这个 Todos 的案例，是你在打算把 Backbonejs 应用于实践时必须要参考和思考的。虽然到网上搜罗一下 Backbonejs 项目实例 比思考要省心，但是别人的始终是别人的，你不转化成自己的，始终无法灵活运用。借此告诫那些觉得这个 Todo 案例没啥用的同学们。

第十一章 前后端实战演练：Web 聊天室-功能分析

上面的一个简单的项目完成之后，对 python 感兴趣的应该已经把 web.py 这个东西熟悉的差不多了，说不定也像我这样把项目放到服务器上跑了起来。对于没有动手去做的同学，我只能表示很遗憾，作为观众的你一定体会不到参与的乐趣，当然也不会有切身的收获。

11.1 项目目标

相比于之前那个项目，这个项目的目标是按照专业前端的方法搭建一个我自己凭空想出来的需求。最后把这个需求完成，然后注册域名放到网络上，可以真实使用。这样也可以持续改进。

上线其实只是一个开始，希望有兴趣参与的同学可以主动参与进来，体会下实际的开发过程。

11.2 功能需求

这个项目的名字叫做聊天室，那肯定是在线聊天用的了，因为不是朝着 Demo 方向做的，因此需要多用户管理。

按照这个需求分析下功能大体有:

```
1. 多用户管理
    1.1 用户注册
    1.2 多用户登陆
2. 话题管理
    2.1 创建话题
    2.2 浏览话题
3. 消息管理
    3.1 发送消息
    3.2 删除消息
    3.3 回复消息
    3.4 浏览消息
```

大概这么几个功能

11.3 技术选型

因为是关于 Backbonejs 的教程，因此必然是基于 Backbonejs 的。技术选型暂定如下:

```
/* 前端技术 */
backbone.js
bootstrap.css
requirejs

/* 后端技术 */
web.py
sqlite3
```

暂时先估计用到这些，说不定真实情况用的更多。

第十二章 前后端实战演练：Web 聊天室-详细设计

上一章简单的介绍了这个聊天室的功能和要使用的技术，这一章的主要目的是为下一章的实现做准备。 任何一个项目从需求到最后的实现，都是要经历这么个过程的，不论这个过程是否显示的存在你自己活着团队的项目历程中。拿到需求后立马开始写代码的同学要么是大牛——（对付各种需求已经有相当多的经验，在理解需求的同时已然对项目进行了分析和设计），要么就是小白——（在这个阶段，对写代码充满了激情，渴望让自己的手指在键盘上得到释放，结果大多会走偏或者自己写到混乱）。

因此需要这么个过程，对已有的需求再次思考、规划

12.1 实体(Model)设计

所谓的实体，就是在项目中数据存放和被传输的对象，从定义上来说就是客观存在的事物。那么在这个项目中有哪些实体存在呢？

从功能上分析，只有三个实体：用户，话题，消息。这三个实体也就是项目中的三个 Model，剩下的所有业务都是围绕它们来运转的。

那么这三个实体中都应该存放什么样的数据呢？根据需求简单的列一下:

```
用户(user):
    id
    username
    password
    registed_time
话题(topic):
    id
    title
    created_time
    owner
消息(message):
    id
    content
    topic_id
    user_id
    created_time
```

12.2 接口设计

因为这个项目的重点是在前端，因此后端只是提供一个接口，先把需要的哪些接口整理清楚了，剩下的就好办了。

那么，需要哪些接口呢？

依然是根据功能来： **用户管理** 1. 用户注册接口:

```
/user/ [POST]
```

1.

    获取用户列表接口:

2.

```
/user/ [GET]
```

3.
4.

    获取单个用户接口:

5.

```
/user/<id>/  [GET]
```

6.
7.

    用户登录:

8.

```
/login/ [POST]
```

9.
10.

    用户登出:

11.

```
/logout/ [GET]
```

12.

话题管理 1. 话题列表:

```
/topic/ [GET]
```

1.

    创建话题:

2.

```
/topic/ [POST]
```

3.
4.

    查看具体话题(相对于进入消息列表):

5.

```
/topic/<topic_id>/  [GET]
```

6.

消息管理 1. 发送/回复消息:

```
/message/ [POST]
```

1.

删除消息:

2.

```
/message/<message_id>/ [DELETE]
```

3.
4.

浏览消息:

5.

```
/message/ [GET]
```

6.

大体就这么些个接口，url 后面表示的是 HTTP 的方法。

这样分析完之后，服务器的工作算是清晰了，下面在来分析页面上的工作。

12.3 页面设计

因为是单页应用，所有也就一个页面，但是这一个页面也是由多个视图来组成的——这里的视图可以理解为桌面程序的那种窗口。

为了有的放矢，这里我就先画几个草图，来看一下我们这个项目最终的结果可能是什么:

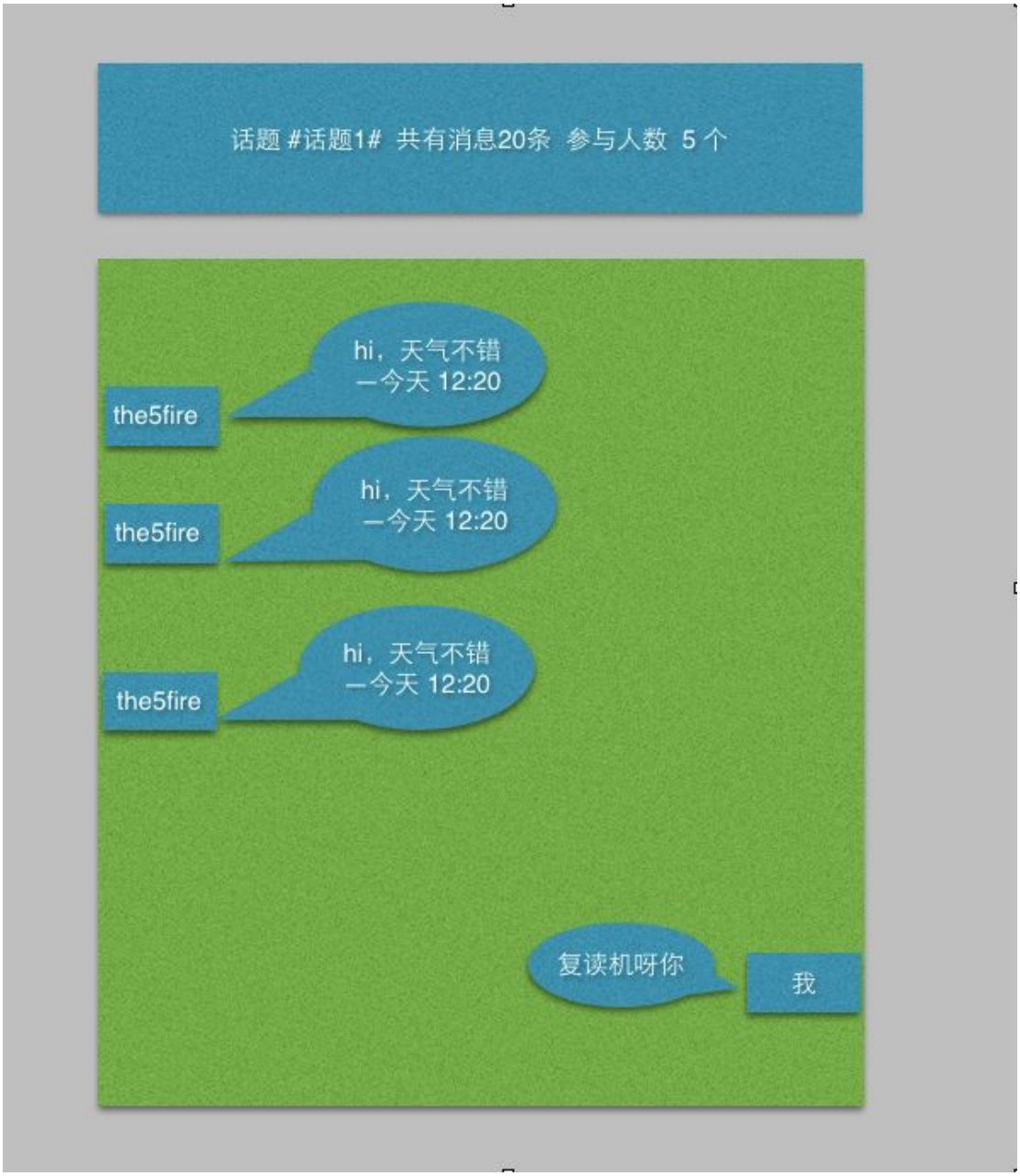
登录 这个是直接截得现成的图。

注册界面和这个一样

话题界面


共有 5 个话题
话题1
创建者：the5fire
话题1
创建者：the5fire
话题1
创建者：the5fire
话题1
创建者：the5fire
话题1
创建者：the5fire
添加话题

消息界面



12.4 view 的设计

这里说的 view，是指 backbonejs 中的 view，按照之前的经验来说，大概需要划分的 view 和功能分别为：

1. 话题 view(topic_view):
2.

```
功能:
    根据话题接口获取所有话题数据，然后渲染模板
```

3.
4.

    消息 view(message_view):

5.

```
功能:
    根据消息接口返回的数据，渲染模板
```

6.
7.

    登录 view(login_view):

8.

```
功能:
    展示登录页面，发送用户数据到服务器端
```

9.
10.

    整体 view（main_view):

11.

```
功能：
    负责其他 view 的切换
```

12.

12.5 总结

这一章主要是对功能做了更进一步的处理，目的就是能够更好的开始编码。在详细设计完成之后具体实现就变得有迹可循了。

到目前为止，这个项目我也是边写边做，现在还没开始写代码。因此这篇文章的分析可能在之后还需要改动。但，即便如此，这个过程也是需要存在的，因为软件开发本身就是一个不断迭代的过程，你不可能一拍脑袋便出来一个完美方案，设计一个可行的方案，然后持续迭代才是最好的实践。

第十三章 前后端实战演练：Web 聊天室-服务器端开发

在上一章中简单的进行了在开发中经常要经过的一个步骤——详细设计，在详细设计中定义了数据模型，有哪些接口，以及页面的长相。比较简单，并没有对服务器端 Python 项目的结构进行设计，也没有对前端文件的组织进行规划。但是这个详细设计的目的是达到了，在我个人的实际经验中，项目的开发很少有按照软件工程中的瀑布模型来做的。大部分都是明确阶段目标，然后开发，然后再次明确，再次开发，不断迭代。

因此说到前篇文章的目的也就是指导后端开发提供一个什么样的接口，输出什么样的数据结构。也是指导前端应该怎么获取数据。在日常工作中，设计到前后端甚至是服务器和客户端的这种模式也是一样，两方人员在项目初期只要协定好接口和数据结构就行，随着项目的进行不断的调整。当然这种情况是说内部人员之间的沟通，如果是和外部人员的沟通情况就不一样了。

回到正题，后端的开发主要功能是提供接口，要提供哪些接口一定定义好了，模型也建立好了，现在需要做的就是搭一个项目，实现接口。

13.1 项目结构

项目使用了 webpy 这个微型的 python 框架，项目结构是依然按照之前对 todos 进行服务器端扩展时的结构， onlinetodos

```
.
├── __init__.py
├── handlers.py
├── init_sqlite.py
├── models.py
├── server.py
├── static
│   ├── css
│   │   ├── body.css
│   │   └── semantic.min.css
```

```
|    ├──── img
|    |    └──── bg.jpg
|    └──── js
|        ├──── backbone.js
|        ├──── jquery.js
|        ├──── json2.js
|        └──── underscore.js
├──── templates
|    └──── index.html
└──── wechat.db
```

可以先忽略其中的静态文件的部分，下一章实现的时候会进行调整。只说后端的实现，主要分为三个部分：server 部分、handler 部分、和 models 部分，也就是上面对应的名字，这些名字本身就说明了这部分的功能。server 主要是启动一个 web 服务器，其中进行了 url 的定义，对应 url 接受到的请求会传递到 handlers 中对应的类方法中，在方法中会调用 Models 中的 Model 类来获取数据，然后再返回给客户端（浏览器）。

13.2 server 部分详解

这部分功能上已经介绍了，这里贴出代码详细介绍：

```
#!/usr/bin/env python#coding:utf-8import webfrom web.httpserver import
StaticMiddleware
urls = (
    '/', 'IndexHandler',  # 返回首页
    '/topic', 'TopicHandler',
    '/topic/(\d+)', 'TopicHandler',
    '/message', 'MessageHandler',
    '/user', 'UserHandler',
    '/user/(\d+)', 'UserHandler',
    '/login', 'LoginHandler',
    '/logout', 'LogoutHandler',)
app = web.application(urls, globals())application =
app.wsgifunc(StaticMiddleware)
if web.config.get('_session') is None:
    session = web.session.Session(
        app,
        web.session.DiskStore('sessions'),
        initializer={'login': False, 'user': None}
    )
    web.config._session = session
from handlers import (  # NOQA
```

```
    IndexHandler, RegisteHandler,
    LoginHandler, LogoutHandler,
    TopicHandler, MessageHandler)

def main():
    app.run()
if __name__ == "__main__":
    main()
```

这里首先是定义了 url 对应的 handlers 中的类，然后通过 webpy 的静态文件 Middleware 来处理静态文件的请求，接着初始化了项目的 session。最后从 handlers 中引入所有用到的 Handler。这里需要注意的是，handlers 的引入需要在 session 定义的下面，因为 handlers 中需要用到 session。

13.3 handler 中的逻辑

这里面主要逻辑是处理来自浏览器对相应的 url 的请求，因为项目需要处理用户登录，因此要引入前面定义的 session 来保存用户的状态。

来看代码：

```
#coding:utf-8import copyimport jsonimport hashlibimport sqlite3from datetime
import datetime
import web
from models import Message, User, Topic
session = web.config._session
CACHE_USER = {}

def sha1(data):
    return hashlib.sha1(data).hexdigest()

def bad_request(message):
    raise web.BadRequest(message=message)

# 首页class IndexHandler:
    def GET(self):
        render = web.template.render('templates/')
        return render.index()

class UserHandler:
    def GET(self):
        # 获取当前登录的用户数据
        user = session.user
```

```
        return json.dumps(user)

    def POST(self):
        data = web.data()
        data = json.loads(data)
        username = data.get("username")
        password = data.get("password")
        password_repeat = data.get("password_repeat")

        if password != password_repeat:
            return bad_request('两次密码输入不一致')

        user_data = {
            "username": username,
            "password": sha1(password),
            "registed_time": datetime.now(),
        }

        try:
            user_id = User.create(**user_data)
        except sqlite3.IntegrityError:
            return bad_request('用户名已存在!')

        user = User.get_by_id(user_id)
        session.login = True
        session.user = user

        result = {
            'id': user_id,
            'username': username,
        }
        return json.dumps(result)

class LoginHandler:
    def POST(self):
        data = web.data()
        data = json.loads(data)
        username = data.get("username")
        password = data.get("password")
        user = User.get_by_username_password(
            username=username,
            password=sha1(password)
        )
        if not user:
```

```
            return bad_request('用户名或密码错误！')

        session.login = True
        session.user = user
        result = {
            'id': user.get('id'),
            'username': user.get('username'),
        }
        return json.dumps(result)

class LogoutHandler:
    def GET(self):
        session.login = False
        session.user = None
        session.kill()
        return web.tempredirect('/#login')

class TopicHandler:
    def GET(self, pk=None):
        if pk:
            topic = Topic.get_by_id(pk)
            return json.dumps(topic)

        topics = Topic.get_all()
        result = []
        for t in topics:
            topic = dict(t)
            try:
                user = CACHE_USER[t.owner_id]
            except KeyError:
                user = User.get_by_id(t.owner_id)
                CACHE_USER[t.owner_id] = user
            topic['owner_name'] = user.username
            result.append(topic)
        return json.dumps(result)

    def POST(self):
        if not session.user or not session.user.id:
            return bad_request('请先登录！')

        data = web.data()
        data = json.loads(data)

        topic_data = {
```

```
            "title": data.get('title'),
            "owner_id": session.user.id,
            "created_time": datetime.now(),
        }

        try:
            topic_id = Topic.create(**topic_data)
        except sqlite3.IntegrityError:
            return bad_request('你已创建过该名称!')

        result = {
            "id": topic_id,
            "title": topic_data.get('title'),
            "owner_id": session.user.id,
            "owner_name": session.user.username,
            "created_time": str(topic_data.get('created_time')),
        }
        return json.dumps(result)

    def PUT(self, obj_id=None):
        pass

    def DELETE(self, obj_id=None):
        pass

class MessageHandler:
    def GET(self):
        topic_id = web.input().get('topic_id')
        if topic_id:
            messages = Message.get_by_topic(topic_id) or []
        else:
            messages = Message.get_all()

        result = []
        current_user_id = session.user.id
        for m in messages:
            try:
                user = CACHE_USER[m.user_id]
            except KeyError:
                user = User.get_by_id(m.user_id)
                CACHE_USER[m.user_id] = user
            message = dict(m)
            message['user_name'] = user.username
            message['is_mine'] = (current_user_id == user.id)
```

```
            result.append(message)
        return json.dumps(result)

    def POST(self):
        data = web.data()
        data = json.loads(data)
        if not (session.user and session.user.id):
            return bad_request("请先登录！")

        message_data = {
            "content": data.get("content"),
            "topic_id": data.get("topic_id"),
            "user_id": session.user.id,
            "created_time": datetime.now(),
        }
        m_id = Message.create(**message_data)
        result = {
            "id": m_id,
            "content": message_data.get("content"),
            "topic_id": message_data.get("topic_id"),
            "user_id": session.user.id,
            "user_name": session.user.username,
            "created_time": str(message_data.get("created_time")),
            "is_mine": True,
        }
        return json.dumps(result)
```

别看代码这么多，所有的具体的 Handler 的处理逻辑都是一样的——接受 post 请求，验证用户状态，存储；或者是接受 get 请求，调用 Model 获取数据，组织成 json，然后返回。相当简单了，对吧。

13.4 models 中的实现

这部分功能就是现实数据库的增删改查，行为基本一致，因此提出一个基础类来完成基本的操作。如果基础类满足不了需求，需要在各子类中实现自己的逻辑。

来看下实现代码：

```
#coding:utf-8import web
db = web.database(dbn='sqlite', db="wechat.db")

class DBManage(object):
    @classmethod
```

```
    def table(cls):
        return cls.__name__.lower()

    @classmethod
    def get_by_id(cls, id):
        itertodo = db.select(cls.table(), where="id=$id", vars=locals())
        return next(iter(itertodo), None)


    @classmethod
    def get_all(cls):
        # inspect.ismethod(cls.get_all)
        return db.select(cls.table())

    @classmethod
    def create(cls, **kwargs):
        return db.insert(cls.table(), **kwargs)

    @classmethod
    def update(cls, **kwargs):
        db.update(cls.table(), where="id=$id", vars={"id": kwargs.pop('id')},
**kwargs)

    @classmethod
    def delete(cls, id):
        db.delete(cls.table(), where="id=$id", vars=locals())

class User(DBManage):
    id = None
    username = None
    password = None
    registed_time = None

    @classmethod
    def get_by_username_password(cls, username, password):
        itertodo = db.select(cls.table(), where="username=$username and
password=$password", vars=locals())
        return next(iter(itertodo), None)

class Topic(DBManage):
    id = None
    title = None
    created_time = None
    owner = None
```

```
class Message(DBManage):
    id = None
    content = None
    top_id = None
    user_id = None
    reply_to = None

    @classmethod
    def get_by_topic(cls, topic_id):
        return db.select(cls.table(), where="topic_id=$topic_id", vars=locals())
```

在操作的同时还是定义了模型的属性，不过目前并没有用的上，如果打算进一步抽象的话是要用到的。

13.5 总结

整个后端的实现并不复杂，只是简单的数据库 CRUD 操作，也没有进行更深一步的抽象，不过满足接口需求就好，等前端实现的时候可能需要调整。

这个项目已经托管在 github 上了：wechat ，欢迎围观以及贡献代码。

第十四章 前后端实战演练：Web 聊天室-前端开发

在上一章的 服务器端开发 中我们定义了模型，实现了几个实体增删改查得功能，也提供了前端访问数据的接口。但在前端的实现过程中又对接口进行了调整，以更符合前端的使用。在真实的开发中也是如此，定义的接口合不合适只有在开发时才知道。

目前代码并没有进行模块的划分，在单 js 文件(chat.js)中实现了所有逻辑。下一步会进行通过 seajs 或者 requirejs 来进行模块管理。

关于前端样式的设计和开发并不在这个系列的计划中，因此就不多做介绍了，只是基于 semantic 进行了简单的设计，有兴趣的可以自己去看： wechat 项目 。

14.1 前端文件结构

前端的结构和前面的项目结构一样，只是添加了 chat.js 和自定义样式的 chat.css 文件，我们所有的代码都在这个文件中编写。

```
├─── static
|    ├─── css
|    |    ├─── body.css
|    |    ├─── chat.css
|    |    └─── semantic.min.css
|    ├─── fonts
|    |    ├─── basic.icons.svg
|    |    ├─── basic.icons.woff
|    |    ├─── icons.svg
|    |    └─── icons.woff
|    ├─── img
|    |    └─── bg.jpg
|    └─── js
|         ├─── backbone.js
|         ├─── chat.js
|         ├─── jquery.js
|         ├─── json2.js
|         └─── underscore.js
├─── templates
|    └─── index.html
```

14.2 Model 和 Collection 定义

我们还是先来定义 Model 的实现，前端的 Model 应该和后端的 Model 定义一样，不然数据传递就会有问题。因为在后端已经明确定义了 Model 有哪些属性，这里的定义就简单多了。当然这也是动态语言的优势——动态的添加属性。

我们要定义三个 Model 和两个 Collection，因为 User 这个对象在前端只会存在一份，不需要定义集合。来看具体实现：

```
var User = Backbone.Model.extend({
    urlRoot: '/user',});
var Topic = Backbone.Model.extend({
    urlRoot: '/topic',});
var Message = Backbone.Model.extend({
    urlRoot: '/message',});
var Topics = Backbone.Collection.extend({
    url: '/topic',
    model: Topic,});
var Messages = Backbone.Collection.extend({
```

```
    url: '/message',
    model: Message,});
```

我们之定义了基本的属性，这些属性保证了我们可以可以直接通过 collection 或者 model 获取到后端的数据。

14.3 视图和模板的定义

定义了基本的 Model 之后，就相当于是有了数据的获取方式，下一步就是如何显示这些数据了。因此就需要用 Backbonejs 中的 view 和 template 来定义我们的具体显示了。

首先来定义 view:

```
var TopicView = Backbone.View.extend({
    tagName:  "div class='column'",
    templ: _.template($('#topic-template').html()),

    // 渲染列表页模板
    render: function() {
      $(this.el).html(this.templ(this.model.toJSON()));
      return this;
    },});
var MessageView = Backbone.View.extend({
    tagName:  "div class='comment'",
    templ: _.template($('#message-template').html()),

    // 渲染列表页模板
    render: function() {
      $(this.el).html(this.templ(this.model.toJSON()));
      return this;
    },});
var UserView = Backbone.View.extend({
    el: "#user_info",
    username: $('#username'),

    show: function(username) {
        this.username.html(username);
        this.$el.show();
    },});
```

根据定义的三个 Model，定义了把数据渲染到模板的方式，对应的模块是什么样的呢，我们来看下:

```
<script type="text/template" id="topic-template">
    <a href="#topic/<%= id %>">
        <div class="column">
            <div class="ui segment">
                <h3><%= title %></h3>
                <p>
                创建者：<%= owner_name %>
                </p>
                <p>
                创建时间：<%= created_time %>
                </p>
            </div>
        </div>
    </a></script>
<script type="text/template" id="message-template">
    <div class="content <% if(is_mine) { %> right <% } %>" data="<%= id %>">
        <a class="author"><%= user_name %></a>
        <br/>
        <div class="metadata">
            <span class="date"><%= created_time %></span>
        </div>
        <div class="text" style="min-width:55px">
            <div class="ui pointing label large <% if(is_mine) { %> right <% } %>">
                <p><%= content %></p>
            </div>
        </div>
    </div></script>
```

这里并没有定义 user 的模板，因为目前对 user 只是做了简单的展现，即仅在顶部栏上加了一个用户名，通过：user_name 这个 Dom 节点的 id 添加数据。

到目前已经介绍了所有的基础数据：从 model 到 collection，到用来显示数据的 view，再到定义的页面模板 template。每部分的数据都可以单独的从后台获取，并且渲染。好了，材料都准备好了就差什么了？当然是流程。不过还有一个东西得先说一下，这些数据被塞到页面之后到底长成什么样还不知道。因此得先来看下页面结构。

下面先来看看上面的那些数据最终要被填充到页面的什么部位，然后再来说流程的事。

14.4 页面结构

这里还是从代码上说事，但是最终效果图已经在 wechat 的 readme 中贴出来了，你可以跳过去看看长相先。

欣赏完外表，来看看内部的骨架，这里只贴主要代码。

顶部的固定栏：

```
<!-- Top Bar  --><div class="ui fixed transparent inverted main menu">
    <div class="container">
        <div class="title item">
            <b>We Chat</b> 在线聊天系统
        </div>

        <div class="right menu">
            <div class="title item">
                Backbonejs 交流群：308466740
            </div>
        </div>
        <div id="user_info" class="right menu hide">
            <div class="title item">
                <i class="icon user"></i>
                <label id="username">the5fire</lable>
            </div>
            <a class="popup icon github item" href="/logout" title="退出登录">
                退出登录
            </a>
        </div>
    </div></div>
```

登陆注册的代码，纯静态代码：

```
<div id="wrapper" style="display: block; z-index: 998;">
    <div class="container">
        <div id="login" class="ui two column relaxed grid">
            <div class="column">
                <div class="ui fluid form segment">
                    <h3 class="ui header">登录</h3>
                    <div class="field">
                        <label>用户名</label>
                        <input id="login_username" placeholder="用户名"
type="text">
                    </div>
                    <div class="field">
                        <label>密码</label>
```

```
                    <input id="login_pwd" type="password">
                </div>
                <div class="ui blue login_submit button">登录</div>
            </div>
        </div>
        <div class="column">
            <div class="ui fluid form segment">
                <h3 class="ui header">注册</h3>
                <div class="field">
                    <label>用户名</label>
                    <input id="reg_username" placeholder="用户名"
type="text">
                </div>
                <div class="field">
                    <label>密码</label>
                    <input id="reg_pwd" type="password">
                </div>
                <div class="field">
                    <label>重复密码</label>
                    <input id="reg_pwd_repeat" type="password">
                </div>
                <div class="inline field">
                    <div class="ui checkbox">
                        <input type="checkbox" id="terms">
                        <label for="terms">我同意 the5fire's WeChat 网的服务
条款。</label>
                    </div>
                </div>
                <div class="ui blue registe_submit button">注册</div>
            </div>
        </div>
    </div>
</div></div>
```

用来展示话题和消息的内容区域:

```
<!-- Content --><div id="main" class="main container">

    <!-- Topic List -->
    <div id="topic_section">
        <div id="topic_list" class="ui three column grid">
            <!-- 这里放 topic 列表 -->
        </div>
        <div id="topic_form" class="ui error form segment">
            <div class="two fields">
```

```
                <div class="field">
                    <label>新建 Topic</label>
                    <input id="topic_title" placeholder="topic" type="text">
                </div>
            </div>
            <div class="ui blue submit_topic button">Add</div>
        </div>
    </div>

    <!-- Message -->
    <div  id="message_section" class="ui column grid hide" style="display:none">
        <div class="column">
            <div class="circular ui button"><a href="#index">返回列表</a></div>
            <div class="ui piled blue segment">
                <h2 class="ui header">
                    #<i id="message_head"></i># <!-- 用来放 topic name -->
                </h2>
                <div id="message_list" class="ui comments">
                    <!-- comments 列表 -->
                </div>
                <div class="ui reply form">
                    <div class="field">
                        <input type="text" id="comment"/>
                    </div>
                    <div id="submit" data="" class="ui fluid blue labeled submit
icon button">
                        <i class="icon edit"></i> 我也来说一句!
                    </div>
                </div>
            </div>
        </div>
    </div></div>
```

页面布局大概介绍了一下，如果你熟悉 html，并且也看了我上面链接里给的最终效果， 上面的这些理解上面的这些代码应该很 Easy 了。如果不熟悉的也没问题，只要关注于我写了注释的地方就行了，这些地方就是上面我们定义的那些模板被渲染好之后的归宿。

14.5 view 管理和 router 管理

上面占了点篇幅介绍了页面的布局，以便对我们数据最终的处理有一个感觉。

有了数据，也有了最后数据的去处，最后当然要说流程了。所谓的流程就是说我要怎么把 Model 渲染好的模板给塞到对于的页面 div 节点中，我要怎么来控制不同 Model 的展示。毕竟是 SPA(单页应用)，也只有这一个页面来供数据的展示。因此需要在一个页面上切换的展示不同的视图。

这里我们是通过 Backbone 的 Route 和 View 来做。Route 用来做路由分发（也就是 URI 的匹配，比如：#index 匹配到首页）。另外不同于上面用来把 Model 数据传到 Template 中的 View，这里的 View 是用来管理其他具体 View 和 Collection 的，可以比喻为管家 View，就是用来控制这个视图什么时候显示，那个 Collection 的数据什么时候获取。

但是，需要注意，这个 View 需要被 Route 来控制，也就是通过路由控制（根据 URI），因此 View 在具备上述功能的情况下也要提供接口（方法）给 Route。

上面介绍了一堆，仿佛说不太清晰，没关系，Talk is cheap, Show you my code。

先来看 View 管家-AppView，主要功能就是获取 Topic 和 Message 的数据到 Collection 中，调用 Model 对应的 View 把数据填到模板中，然后把最终拼好的数据放到上面介绍的页面对应 div 中。

```
var AppView = Backbone.View.extend({
    el: "#main",
    topic_list: $("#topic_list"),
    topic_section: $("#topic_section"),
    message_section: $("#message_section"),
    message_list: $("#message_list"),
    message_head: $("#message_head"),

    events: {
        'click .submit': 'saveMessage', // 发送消息
        'click .submit_topic': 'saveTopic',  // 新建主题
        'keypress #comment': 'saveMessageEvent', // 键盘事件
    },

    initialize: function() {
        _.bindAll(this, 'addTopic', 'addMessage');

        topics.bind('add', this.addTopic);

        // 定义消息列表池，每个 topic 有自己的 message collection
        // 这样保证每个主题下得消息不冲突
        this.message_pool = {};
```

```
        this.message_list_div = document.getElementById('message_list');
    },

    addTopic: function(topic) {
        var view = new TopicView({model: topic});
        this.topic_list.append(view.render().el);
    },

    addMessage: function(message) {
        var view = new MessageView({model: message});
        this.message_list.append(view.render().el);
    },

    saveMessageEvent: function(evt) {
        if (evt.keyCode == 13) {
            this.saveMessage(evt);
        }
    },
    saveMessage: function(evt) {
        var comment_box = $('#comment')
        var content = comment_box.val();
        if (content == '') {
            alert('内容不能为空');
            return false;
        }
        var topic_id = comment_box.attr('topic_id');
        var message = new Message({
            content: content,
            topic_id: topic_id,
        });
        self = this;
        var messages = this.message_pool[topic_id];
        message.save(null, {
            success: function(model, response, options){
                comment_box.val('');
                // 重新获取，看服务器端是否有更新
                // 比较丑陋的更新机制
                messages.fetch({
                    data: {topic_id: topic_id},
                    success: function(){

self.message_list.scrollTop(self.message_list_div.scrollHeight);
                        messages.add(response);
```

```
                },
            });
        },
    });
},

saveTopic: function(evt) {
    var topic_title = $('#topic_title');
    if (topic_title.val() == '') {
        alert('主题不能为空！');
        return false
    }
    var topic = new Topic({
        title: topic_title.val(),
    });
    self = this;
    topic.save(null, {
        success: function(model, response, options){
            topics.add(response);
            topic_title.val('');
        },
    });
},

showTopic: function(){
    // 获取所有主题
    topics.fetch();
    this.topic_section.show();
    this.message_section.hide();
    this.message_list.html('');
},

initMessage: function(topic_id) {
    // 初始化消息集合，并放到消息池中
    var messages = new Messages;
    messages.bind('add', this.addMessage);
    this.message_pool[topic_id] = messages;
},

showMessage: function(topic_id) {
    this.initMessage(topic_id);

    this.message_section.show();
    this.topic_section.hide();
```

```
        this.showMessageHead(topic_id);
        $('#comment').attr('topic_id', topic_id);

        var messages = this.message_pool[topic_id];
        messages.fetch({
            data: {topic_id: topic_id},
            success: function(resp) {
                self.message_list.scrollTop(self.message_list_div.scrollHeight)
            }
        });
    },

    showMessageHead: function(topic_id) {
        var topic = new Topic({id: topic_id});
        self = this;
        topic.fetch({
            success: function(resp, model, options){
                self.message_head.html(model.title);
            }
        });
    },});
```

上面是所有数据视图的展示的逻辑控制部分，虽然代码很多，但没有复杂逻辑，很直观。这里只是 Topic 和 Message 的展示。但是这些所有的数据都是需要用户登录之后才能看到的，那么用户登录和注册部分的逻辑在哪呢？在上面的页面布局部分已经展示了登录注册的页面，下面展示下具体逻辑。

登录注册-LoginView:

```
    var LoginView = Backbone.View.extend({
     el: "#login",
     wrapper: $('#wrapper'),

     events: {
        'keypress #login_pwd': 'loginEvent',
        'click .login_submit': 'login',
        'keypress #reg_pwd_repeat': 'registeEvent',
        'click .registe_submit': 'registe',
     },

     hide: function() {
        this.wrapper.hide();
     },
```

```
show: function() {
    this.wrapper.show();
},

loginEvent: function(evt) {
    if (evt.keyCode == 13) {
        this.login(evt);
    }
},

login: function(evt){
    var username_input = $('#login_username');
    var pwd_input = $('#login_pwd');
    var u = new User({
        username: username_input.val(),
        password: pwd_input.val(),
    });
    u.save(null, {
        url: '/login',
        success: function(model, resp, options){
            g_user = resp;
            // 跳转到 index
            appRouter.navigate('index', {trigger: true});
        }
    });
},

registeEvent: function(evt) {
    if (evt.keyCode == 13) {
        this.registe(evt);
    }
},

registe: function(evt){
    var reg_username_input = $('#reg_username');
    var reg_pwd_input = $('#reg_pwd');
    var reg_pwd_repeat_input = $('#reg_pwd_repeat');
    var u = new User({
        username: reg_username_input.val(),
        password: reg_pwd_input.val(),
        password_repeat: reg_pwd_repeat_input.val(),
    });
    u.save(null, {
```

```
            success: function(model, resp, options){
                g_user = resp;
                // 跳转到 index
                appRouter.navigate('index', {trigger: true});
            }
        });
    },});
```

这里的 View 的主要功能是：注册（保存 user 数据到后台），登录（发送用户请求到后台,成功则跳到首页），事件监听和处理。很基础的功能。

从上面两部分我们知道了如何控制不同 Model 对应视图的展示，也知道了如何处理用户登录。下面再来看些 Route 部分是如何把 url 匹配到对应的方法上的。

路由部分代码-AppRouter:

```
var AppRouter = Backbone.Router.extend({
    routes: {
        "login": "login",
        "index": "index",
        "topic/:id" : "topic",
    },

    initialize: function(){
        // 初始化项目，显示首页
        this.appView = new AppView();
        this.loginView = new LoginView();
        this.userView = new UserView();
        this.indexFlag = false;
    },

    login: function(){
        this.loginView.show();
    },

    index: function(){
        if (g_user && g_user.id != undefined) {
            this.appView.showTopic();
            this.userView.show(g_user.username);
            this.loginView.hide();
            this.indexFlag = true;  // 标志已经到达主页了
        }
    },
```

```
    topic: function(topic_id) {
        if (g_user && g_user.id != undefined) {
            this.appView.showMessage(topic_id);
            this.userView.show(g_user.username);
            this.loginView.hide();
            this.indexFlag = true;  // 标志已经到达主页了
        }
    },});
```

这里设定了三条路由:login,index,topic,分别对应这个登录视图(LoginView),主题和 Message 的视图（由 AppView 管理）。

在不同的路由中的逻辑大致一样，就是根据当前的条件决定是否现实视图。 比如 index 中的 if (g_user && g_user.id != undefined) { 就是判断当前环境中是否有 g_user 这个对象（这个对象是用来存放已登录用户数据的，后面会介绍),根据这个对象判断是否用户已经登录，进而决定是否现实首页——topic 列表页。

14.6 启动

当所有的逻辑都定义好之后，页面加载完毕首先要做的就是启动整个流程，怎么启动呢？按照我们的项目结构：AppRouter 管理 AppView 和 LoginView，AppView 管理 TopicView 和 MessageView，因此，只需要启动 AppRouter 即可。

启动代码如下:

```
var appRouter = new AppRouter();var g_user = new User();g_user.fetch({
    success: function(model, resp, options){
        g_user = resp;
        Backbone.history.start({pustState: true});

        if(g_user === null || g_user.id === undefined) {
            // 跳转到登录页面
            appRouter.navigate('login', {trigger: true});
        } else if (appRouter.indexFlag == false){
            // 跳转到首页
            appRouter.navigate('index', {trigger: true});
        }
    },}); // 获取当前用户
```

就是这一小段代码，程序可以正常运行了。这段代码中的逻辑是：声明一个全局的 appRouter 和 g_user，然后获取当前用户（服务器端会通过 session 保存对

应浏览器的信息），之后根据获取到得用户状态做进一步操作（到登录页面或是到首页）。

这里需要注意的是，这段代码只有在页面加载（刷新或重新访问）的时候才会执行。

好了，到此为止整个项目已经介绍完毕了，不知道你是否看懂，或者这么问，我是否把这个项目讲明白了？

14.7 总结

这一篇看起篇幅很长，其实都是代码。而这些代码只有当你真正打算做这么个东西的时候才会主动去理解，因为那些走马观花的人会选择性的忽略代码。

最后还是补充一下整个流程，其实整个项目开始做的时候，项目的设计者就应该有一个具体的需求和用户使用的场景。对于这个项目我自己设想的用户使用流程：

用户打开浏览器，看到登录和注册页面——》输入用户名、密码进行登录（注册）操作——》展示主题列表视图，并显示用户名在顶部——》用户创建并进入某一主题（显示消息列表视图）——》用户发送消息，消息保存的同时获取服务器端的消息到当前视图。

另外一定要说的是，项目没有进行太多优化和代码的精简，还有很多改进的地方。在我写代码的这些年中我始终坚信并践行的一件事就是——获取知识最好的方法就是实践。因此如果你想掌握这个 Backbone 这个工具，最佳的方式是开始一个项目，并持续的做下去。或者参与一个项目，持续改善项目。

我在边写边实践中写了 WeChat 这个项目，并且已经部署上线，相信会是一个好的开始，因为我没打算把它仅仅作为一个 Demo 来用。 本文涉及的所有代码均在该项目的 basic-version 分支可以看到。

第十五章 引入 requirejs

前面花了四章的时间完成了项目（ wechat ）的开发，并且也放到了线上。这篇来说说模块化的事情。

15.1 模块化的概念

对于通常的网站来说，一般我们不会吧所有的 js 都写到一个文件中，因为当一个文件中的代码行数太多的话会导致维护性变差，因此我们常常会根据业务（页面）来组织 js 文件，比如全站都用到的功能，我就写一个 base.js，只是在首页会用到的功能，就写一个 index.js。这样的话我更改首页的逻辑只需要更改 index.js 文件，不需要考虑太多的不相关业务逻辑。当然还有很重要的一点是按需加载，在非 index.js 页面我就不需要引入 index.js。

那么对于单页应用（SPA）来说要怎么做呢，只有一个页面，按照传统的写法，即便是分开多个文件来写，也得全部放到<script>标签中，由浏览器统一加载。如果你有后端开发经验的话，你会意识到，是不是我们可以像写后端程序（比如 Python）那样，定义不同的包、模块。在另外的模块中按需加载（import）呢？

答案当然是可以。

在前端也有模块化这样的规范，不过是有两套：AMD 和 CMD。关于这俩规范的对比可以参考知乎上的问答 AMD 和 CMD 的区别有哪些 。

按照 AMD 和 CMD 实现的两个可以用来做模块化的是库分别是：require.js 和 sea.js。从本章的题目可以知道我们这里主要把 require.js 引入我们的项目。对于这两库我都做了一个简单的 Demo，再看下面长篇代码之前，可以先感受下： require.js Demo 和 sea.js Demo 。

15.2 简单使用 require.js

要使用 require.js 其实非常简单，主要有三个部分：1. 页面引入 require.js；2. 定义模块；3. 加载模块。我们以上面提到我做的那个 demo 为例：

首先 - 页面引入

```
<!DOCTYPE html><html><head>
    <title>the5fire.com-backbone.js-Hello World</title></head><body>
    <button id="check">新手报到- requirejs 版</button>
    <ul id="world-list">
    </ul>
    <a href="http://www.the5fire.com">更多教程</a>
    <script data-main="static/main.js"
src="static/lib/require.js"></script></body></html>
```

上面的 script 的 data-main 定义了入口文件，我们把配置项也放到了入口文件中。 来看下入口文件：

```
require.config({
    baseUrl: 'static/',
    shim: {
        underscore: {
            exports: '_'
        },
    },
    paths: {
        jquery: 'lib/jquery',
        underscore: 'lib/underscore',
        backbone: 'lib/backbone'
    }});
require(['jquery', 'backbone', 'js/app'], function($, Backbone, AppView) {
    var appView = new AppView();});
```

上面 baseUrl 部分指明了所有要加载模块的根路径，shim 是指那些非 AMD 规范的库，paths 相当于你 js 文件的别名，方便引入。

后面的 require 就是入口了，加载完 main.js 后会执行这部分代码，这部分代码的意思是，加载 jquery 、 backbone 、js/app （这个也可通过 paths 来定义别名），并把加载的内容传递到后面的 function 的参数中。o

来看看 js/app 的定义。

定义模块

```
// app.jsdefine(['jquery', 'backbone'], function($, Backbone) {
    var AppView = Backbone.View.extend({
        // blabla..bla
    });
    return AppView;});
// 或者这种方式define(function(require, exports, module) {
    var $ = require('jquery');
    var Backbone = require('backbone');

    var AppView = Backbone.View.extend({
        // blabla..bla
    });
    return AppView;});
```

这两种方式均可，最后需要返回你想暴露外面的对象。这个对象（AppView）会在其他模块中 require('js/app') 时加载，就像上面一样。

15.3 拆分文件

上一篇中我们写了一个很长的 chat.js 的文件，这个文件包含了所有的业务逻辑。这里我们就一步步来把这个文件按照 require.js 的定义拆分成模块。

上一篇是把 chat.js 文件分开来讲的，这里先来感受下整体代码：

```
$(function(){
    var User = Backbone.Model.extend({
        urlRoot: '/user',
    });

    var Topic = Backbone.Model.extend({
        urlRoot: '/topic',
    });

    var Message = Backbone.Model.extend({
        urlRoot: '/message',
    });

    var Topics = Backbone.Collection.extend({
        url: '/topic',
        model: Topic,
    });

    var Messages = Backbone.Collection.extend({
        url: '/message',
        model: Message,
    });

    var topics = new Topics;

    var TopicView = Backbone.View.extend({
        tagName:  "div class='column'",
        templ: _.template($('#topic-template').html()),

        // 渲染列表页模板
        render: function() {
        $(this.el).html(this.templ(this.model.toJSON()));
        return this;
        },
    });
```

```
var MessageView = Backbone.View.extend({
    tagName:  "div class='comment'",
    templ: _.template($('#message-template').html()),

    // 渲染列表页模板
    render: function() {
    $(this.el).html(this.templ(this.model.toJSON()));
    return this;
    },
});

var UserView = Backbone.View.extend({
    el: "#user_info",
    username: $('#username'),

    show: function(username) {
        this.username.html(username);
        this.$el.show();
    },
});

var AppView = Backbone.View.extend({
    el: "#main",
    topic_list: $("#topic_list"),
    topic_section: $("#topic_section"),
    message_section: $("#message_section"),
    message_list: $("#message_list"),
    message_head: $("#message_head"),

    events: {
        'click .submit': 'saveMessage',
        'click .submit_topic': 'saveTopic',
        'keypress #comment': 'saveMessageEvent',
    },

    initialize: function() {
        _.bindAll(this, 'addTopic', 'addMessage');

        topics.bind('add', this.addTopic);

        // 定义消息列表池，每个 topic 有自己的 message collection
        // 这样保证每个主题下得消息不冲突
        this.message_pool = {};
```

```
            this.message_list_div = document.getElementById('message_list');
        },

        addTopic: function(topic) {
            var view = new TopicView({model: topic});
            this.topic_list.append(view.render().el);
        },

        addMessage: function(message) {
            var view = new MessageView({model: message});
            this.message_list.append(view.render().el);
        },

        saveMessageEvent: function(evt) {
            if (evt.keyCode == 13) {
                this.saveMessage(evt);
            }
        },
        saveMessage: function(evt) {
            var comment_box = $('#comment')
            var content = comment_box.val();
            if (content == '') {
                alert('内容不能为空');
                return false;
            }
            var topic_id = comment_box.attr('topic_id');
            var message = new Message({
                content: content,
                topic_id: topic_id,
            });
            self = this;
            var messages = this.message_pool[topic_id];
            message.save(null, {
                success: function(model, response, options){
                    comment_box.val('');
                    // 重新获取，看服务器端是否有更新
                    // 比较丑陋的更新机制
                    messages.fetch({
                        data: {topic_id: topic_id},
                        success: function(){

self.message_list.scrollTop(self.message_list_div.scrollHeight);
                            messages.add(response);
                        },
```

```
                });
            },
        });
    },

    saveTopic: function(evt) {
        var topic_title = $('#topic_title');
        if (topic_title.val() == '') {
            alert('主题不能为空！');
            return false
        }
        var topic = new Topic({
            title: topic_title.val(),
        });
        self = this;
        topic.save(null, {
            success: function(model, response, options){
                topics.add(response);
                topic_title.val('');
            },
        });
    },

    showTopic: function(){
        topics.fetch();
        this.topic_section.show();
        this.message_section.hide();
        this.message_list.html('');
    },

    initMessage: function(topic_id) {
        var messages = new Messages;
        messages.bind('add', this.addMessage);
        this.message_pool[topic_id] = messages;
    },

    showMessage: function(topic_id) {
        this.initMessage(topic_id);

        this.message_section.show();
        this.topic_section.hide();

        this.showMessageHead(topic_id);
        $('#comment').attr('topic_id', topic_id);
```

```
            var messages = this.message_pool[topic_id];
            messages.fetch({
                data: {topic_id: topic_id},
                success: function(resp) {

self.message_list.scrollTop(self.message_list_div.scrollHeight)
                }
            });
        },

        showMessageHead: function(topic_id) {
            var topic = new Topic({id: topic_id});
            self = this;
            topic.fetch({
                success: function(resp, model, options){
                    self.message_head.html(model.title);
                }
            });
        },
    });


    var LoginView = Backbone.View.extend({
        el: "#login",
        wrapper: $('#wrapper'),

        events: {
            'keypress #login_pwd': 'loginEvent',
            'click .login_submit': 'login',
            'keypress #reg_pwd_repeat': 'registeEvent',
            'click .registe_submit': 'registe',
        },

        hide: function() {
            this.wrapper.hide();
        },

        show: function() {
            this.wrapper.show();
        },

        loginEvent: function(evt) {
            if (evt.keyCode == 13) {
```

```
            this.login(evt);
        }
    },

    login: function(evt){
        var username_input = $('#login_username');
        var pwd_input = $('#login_pwd');
        var u = new User({
            username: username_input.val(),
            password: pwd_input.val(),
        });
        u.save(null, {
            url: '/login',
            success: function(model, resp, options){
                g_user = resp;
                // 跳转到 index
                appRouter.navigate('index', {trigger: true});
            }
        });
    },

    registeEvent: function(evt) {
        if (evt.keyCode == 13) {
            this.registe(evt);
        }
    },

    registe: function(evt){
        var reg_username_input = $('#reg_username');
        var reg_pwd_input = $('#reg_pwd');
        var reg_pwd_repeat_input = $('#reg_pwd_repeat');
        var u = new User({
            username: reg_username_input.val(),
            password: reg_pwd_input.val(),
            password_repeat: reg_pwd_repeat_input.val(),
        });
        u.save(null, {
            success: function(model, resp, options){
                g_user = resp;
                // 跳转到 index
                appRouter.navigate('index', {trigger: true});
            }
        });
    },
```

```
});

var AppRouter = Backbone.Router.extend({
    routes: {
        "login": "login",
        "index": "index",
        "topic/:id" : "topic",
    },

    initialize: function(){
        // 初始化项目，显示首页
        this.appView = new AppView();
        this.loginView = new LoginView();
        this.userView = new UserView();
        this.indexFlag = false;
    },

    login: function(){
        this.loginView.show();
    },

    index: function(){
        if (g_user && g_user.id != undefined) {
            this.appView.showTopic();
            this.userView.show(g_user.username);
            this.loginView.hide();
            this.indexFlag = true;  // 标志已经到达主页了
        }
    },

    topic: function(topic_id) {
        if (g_user && g_user.id != undefined) {
            this.appView.showMessage(topic_id);
            this.userView.show(g_user.username);
            this.loginView.hide();
            this.indexFlag = true;  // 标志已经到达主页了
        }
    },
});

var appRouter = new AppRouter();
var g_user = new User;
g_user.fetch({
    success: function(model, resp, options){
```

```
            g_user = resp;
            Backbone.history.start({pustState: true});

            if(g_user === null || g_user.id === undefined) {
                // 跳转到登录页面
                appRouter.navigate('login', {trigger: true});
            } else if (appRouter.indexFlag == false){
                // 跳转到首页
                appRouter.navigate('index', {trigger: true});
            }
        },
    }); // 获取当前用户});
```

上面三百多行的代码其实只是做了最基本的实现，按照上篇文章的介绍，我们根据 User，Topic，Message，AppView，AppRouter 来拆分。当然你也可以通过类似后端的常用的结构：Model， View，Router 来拆分。

User 的拆分

这个模块我打算定义用户相关的所有内容，包括数据获取，页面渲染，还有登录状态，于是有了这个代码：

```
// user.jsdefine(function(require, exports, module) {
    var $ = require('jquery');
    var Backbone = require('backbone');
    var _ = require('underscore');

    var User = Backbone.Model.extend({
        urlRoot: '/user',
    });

    var LoginView = Backbone.View.extend({
        el: "#login",
        wrapper: $('#wrapper'),

        initialize: function(appRouter) {
            this.appRouter = appRouter;
        },

        events: {
            'keypress #login_pwd': 'loginEvent',
            'click .login_submit': 'login',
            'keypress #reg_pwd_repeat': 'registeEvent',
            'click .registe_submit': 'registe',
```

```
    },

    hide: function() {
        this.wrapper.hide();
    },

    show: function() {
        this.wrapper.show();
    },

    loginEvent: function(evt) {
        if (evt.keyCode == 13) {
            this.login(evt);
        }
    },

    login: function(evt){
        var username_input = $('#login_username');
        var pwd_input = $('#login_pwd');
        var u = new User({
            username: username_input.val(),
            password: pwd_input.val(),
        });
        var self = this;
        u.save(null, {
            url: '/login',
            success: function(model, resp, options){
                self.appRouter.g_user = resp;
                // 跳转到 index
                self.appRouter.navigate('index', {trigger: true});
            }
        });
    },

    registeEvent: function(evt) {
        if (evt.keyCode == 13) {
            this.registe(evt);
        }
    },

    registe: function(evt){
        var reg_username_input = $('#reg_username');
        var reg_pwd_input = $('#reg_pwd');
        var reg_pwd_repeat_input = $('#reg_pwd_repeat');
```

```
            var u = new User({
                username: reg_username_input.val(),
                password: reg_pwd_input.val(),
                password_repeat: reg_pwd_repeat_input.val(),
            });
            var self = this;
            u.save(null, {
                success: function(model, resp, options){
                    self.appRouter.g_user = resp;
                    // 跳转到 index
                    self.appRouter.navigate('index', {trigger: true});
                }
            });
        },
    });

    var UserView = Backbone.View.extend({
        el: "#user_info",
        username: $('#username'),

        show: function(username) {
            this.username.html(username);
            this.$el.show();
        },
    });

    module.exports = {
        "User": User,
        "UserView": UserView,
        "LoginView": LoginView,
    };});
```

通过 define 的形式定义了 User 这个模块，最后通过 module.exports 暴露给外面 User，UserView 和 LoginView。

Topic 模块

同 User 一样，我们在这个模块定义 Topic 的 Model、Collection 和 View，来完成 topic 数据的获取也最终渲染。

```
//topic.jsdefine(function(require, exports, module) {
    var $ = require('jquery');
    var Backbone = require('backbone');
    var _ = require('underscore');
```

```
var Topic = Backbone.Model.extend({
    urlRoot: '/topic',
});

var Topics = Backbone.Collection.extend({
    url: '/topic',
    model: Topic,
});

var TopicView = Backbone.View.extend({
    tagName:  "div class='column'",
    templ: _.template($('#topic-template').html()),

    // 渲染列表页模板
    render: function() {
    $(this.el).html(this.templ(this.model.toJSON()));
    return this;
    },
});

module.exports = {
    "Topic": Topic,
    "Topics": Topics,
    "TopicView": TopicView,
}});
```

一样的，这个模块也对外暴露了 Topic、Topics、TopicView 的内容。

message 模块

```
//message.jsdefine(function(require, exports, module) {
    var $ = require('jquery');
    var Backbone = require('backbone');
    var _ = require('underscore');

    var Message = Backbone.Model.extend({
        urlRoot: '/message',
    });

    var Messages = Backbone.Collection.extend({
        url: '/message',
        model: Message,
    });
```

```
    var MessageView = Backbone.View.extend({
        tagName:  "div class='comment'",
        templ: _.template($('#message-template').html()),

        // 渲染列表页模板
        render: function() {
        $(this.el).html(this.templ(this.model.toJSON()));
        return this;
        },
    });
    module.exports = {
        "Messages": Messages,
        "Message": Message,
        "MessageView": MessageView,
    }});
```

最后也是对外暴露了 Message、Messages 和 MessageView 数据。

AppView 模块

上面定义的都是些基础模块，这个模块我们之前也说过，可以称为“管家 View”，因为它是专门用来管理其他模块的。

```
//appview.jsdefine(function(require, exports, module) {
    var $ = require('jquery');
    var _ = require('underscore');
    var Backbone = require('backbone');
    var TopicModule = require('topic');
    var MessageModule = require('message');

    var Topics = TopicModule.Topics;
    var TopicView = TopicModule.TopicView;
    var Topic = TopicModule.Topic;

    var Message = MessageModule.Message;
    var Messages = MessageModule.Messages;
    var MessageView = MessageModule.MessageView;

    var topics = new Topics();

    var AppView = Backbone.View.extend({
        el: "#main",
        topic_list: $("#topic_list"),
        topic_section: $("#topic_section"),
```

```
        message_section: $("#message_section"),
        message_list: $("#message_list"),
        message_head: $("#message_head"),

        events: {
            'click .submit': 'saveMessage',
            'click .submit_topic': 'saveTopic',
            'keypress #comment': 'saveMessageEvent',
        },

        initialize: function() {
            _.bindAll(this, 'addTopic', 'addMessage');

            topics.bind('add', this.addTopic);

            // 定义消息列表池，每个 topic 有自己的 message collection
            // 这样保证每个主题下得消息不冲突
            this.message_pool = {};

            this.message_list_div = document.getElementById('message_list');
        },

        addTopic: function(topic) {
            var view = new TopicView({model: topic});
            this.topic_list.append(view.render().el);
        },

        addMessage: function(message) {
            var view = new MessageView({model: message});
            this.message_list.append(view.render().el);
            self.message_list.scrollTop(self.message_list_div.scrollHeight);
        },

        saveMessageEvent: function(evt) {
            if (evt.keyCode == 13) {
                this.saveMessage(evt);
            }
        },
        saveMessage: function(evt) {
            var comment_box = $('#comment')
            var content = comment_box.val();
            if (content == '') {
                alert('内容不能为空');
                return false;
```

```
            }
            var topic_id = comment_box.attr('topic_id');
            var message = new Message({
                content: content,
                topic_id: topic_id,
            });
            var messages = this.message_pool[topic_id];
            message.save(null, {
                success: function(model, response, options){
                    comment_box.val('');
                    // 重新获取，看服务器端是否有更新
                    // 比较丑陋的更新机制
                    messages.fetch({
                        data: {topic_id: topic_id},
                        success: function(){

self.message_list.scrollTop(self.message_list_div.scrollHeight);
                            messages.add(response);
                        },
                    });
                },
            });
        },

        saveTopic: function(evt) {
            var topic_title = $('#topic_title');
            if (topic_title.val() == '') {
                alert('主题不能为空！');
                return false
            }
            var topic = new Topic({
                title: topic_title.val(),
            });
            self = this;
            topic.save(null, {
                success: function(model, response, options){
                    topics.add(response);
                    topic_title.val('');
                },
            });
        },

        showTopic: function(){
            topics.fetch();
```

```
            this.topic_section.show();
            this.message_section.hide();
            this.message_list.html('');

            this.goOut()
        },

        initMessage: function(topic_id) {
            var messages = new Messages;
            messages.bind('add', this.addMessage);
            this.message_pool[topic_id] = messages;
        },

        showMessage: function(topic_id) {
            this.initMessage(topic_id);

            this.message_section.show();
            this.topic_section.hide();

            this.showMessageHead(topic_id);
            $('#comment').attr('topic_id', topic_id);

            var messages = this.message_pool[topic_id];
            messages.fetch({
                data: {topic_id: topic_id},
                success: function(resp) {

self.message_list.scrollTop(self.message_list_div.scrollHeight)
                }
            });
        },

        showMessageHead: function(topic_id) {
            var topic = new Topic({id: topic_id});
            self = this;
            topic.fetch({
                success: function(resp, model, options){
                    self.message_head.html(model.title);
                }
            });
        },

    });
    return AppView;});
```

不同于上面三个基础模块，这个模块只需要对外暴露 AppView 即可（貌似也就只有这一个东西）。

AppRouter 模块

下面就是用来做路由的 AppRouter 模块，这里只是定义了 AppRouter，没有做初始化的操作，初始化的操作我们放到 app.js 这个模块中，app.js 也是项目运行的主模块。

```
// approuter.jsdefine(function(require, exports, module) {
    var $ = require('jquery');
    var _ = require('underscore');
    var Backbone = require('backbone');
    var AppView = require('appview');
    var UserModule = require('user');
    var LoginView = UserModule.LoginView;
    var UserView = UserModule.UserView;

    var AppRouter = Backbone.Router.extend({
        routes: {
            "login": "login",
            "index": "index",
            "topic/:id" : "topic",
        },

        initialize: function(g_user){
            // 设置全局用户
            this.g_user = g_user;
            // 初始化项目，显示首页
            this.appView = new AppView();
            this.loginView = new LoginView(this);
            this.userView = new UserView();
            this.indexFlag = false;

        },

        login: function(){
            this.loginView.show();
        },

        index: function(){
            if (this.g_user && this.g_user.id != undefined) {
                this.appView.showTopic();
                this.userView.show(this.g_user.username);
```

```
                this.loginView.hide();
                this.indexFlag = true;  // 标志已经到达主页了
            }
        },

        topic: function(topic_id) {
            if (this.g_user && this.g_user.id != undefined) {
                this.appView.showMessage(topic_id);
                this.userView.show(this.g_user.username);
                this.loginView.hide();
                this.indexFlag = true;  // 标志已经到达主页了
            }
        },
    });

    return AppRouter;});
```

同样，对外暴露 AppRouter，主要供 app.js 这个主模块使用。

app 模块

最后，让我们来看下所有 js 的入口：

```
// app.jsdefine(function(require) {
    var $ = require('jquery');
    var _ = require('underscore');
    var Backbone = require('backbone');
    var AppRouter = require('approuter');
    var UserModule = require('user');

    var User = UserModule.User;

    var g_user = new User();
    var appRouter = new AppRouter(g_user);
    g_user.fetch({
        success: function(model, resp, options){
            g_user = resp;
            Backbone.history.start({pustState: true});

            if(g_user === null || g_user.id === undefined) {
                // 跳转到登录页面
                appRouter.navigate('login', {trigger: true});
            } else if (appRouter.indexFlag == false){
                // 跳转到首页
                appRouter.navigate('index', {trigger: true});
```

```
            }
        },
    }); // 获取当前用户});
```

这个模块中，我们通过 require 引入 Approuter，引入 User 模块。需要注意的是，不同于之前一个文件中所有的模块可以共享对象的实例（如：g_user, appRouter），这里需要通过参数传递的方式把这个各个模块都需要的对象传递过去。同时 AppRouter 和 User 也是整个页面生存期的唯一实例。因此我们把 User 对象作为 AppRouter 的一个属性。在上面的 AppRouter 定义中，我们又吧 AppRouter 的实例传递到了 LoginView 中，因为 LoginView 需要对 url 进行变换。

总结

好了，我们总结下模块拆分的结构，还是来看下项目中 js 的文件结构：

```
└─── js
    ├─── app.js
    ├─── approuter.js
    ├─── appview.js
    ├─── backbone.js
    ├─── jquery.js
    ├─── json2.js
    ├─── message.js
    ├─── require.js
    ├─── topic.js
    ├─── underscore.js
    └─── user.js
```

15.4 用 require.js 加载

上面定义了项目需要的所有模块，知道了 app.js 相当于程序的入口，那么要怎么在页面开始呢？

就像一开始介绍的 require.js 的用法一样，只需要在 index.html 中加入一个 js 引用，和一段定义即可：

```
// index.html<script data-main="/static/js/app.js"
src="/static/js/require.js"></script><script>require.config({
    baseUrl: '/static',
    shim: {
        underscore: {
            exports: '_'
```

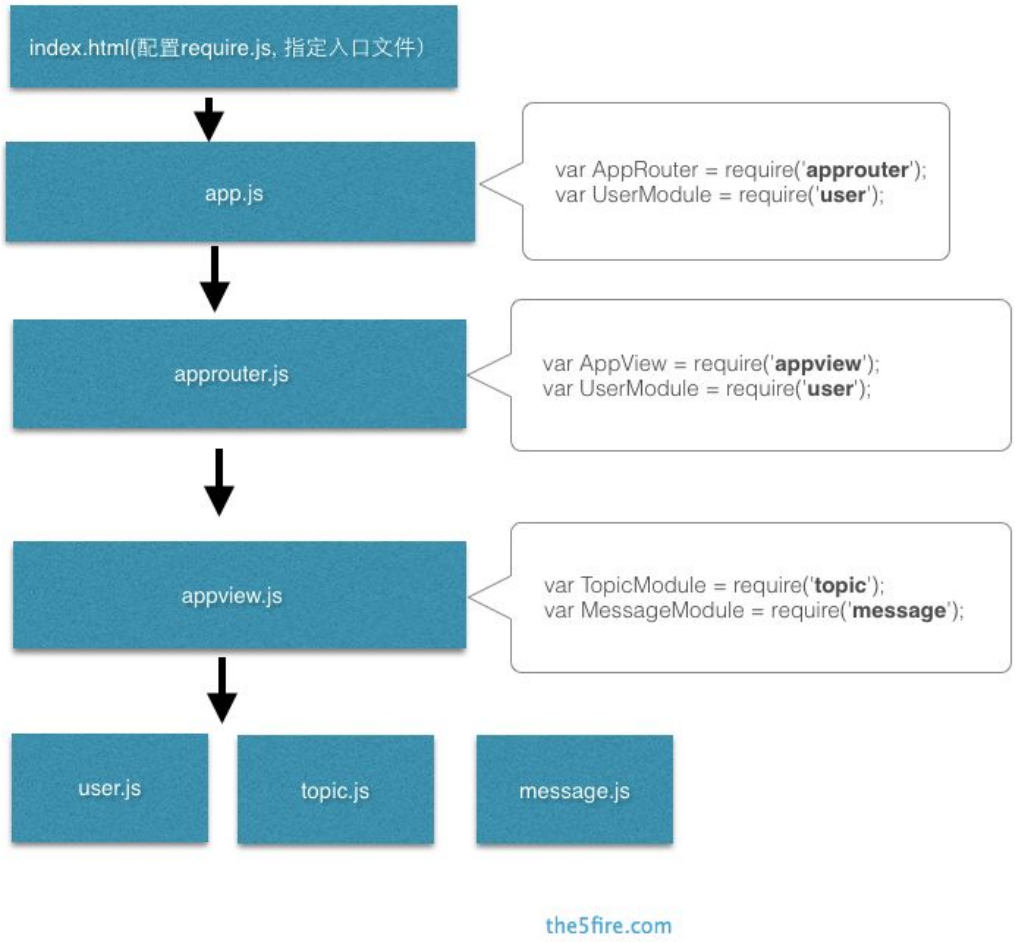
```
        },
    },
    paths: {
        "jquery": "js/jquery",
        "underscore": "js/underscore",
        "backbone": "js/backbone",

        "user": "js/user",
        "message": "js/message",
        "topic": "js/topic",
        "appview": "js/appview",
        "approuter": "js/approuter",
        "app": "js/app",
    });</script>
```

需要解释的是上面的那个 shim 的定义。因为 underscore 并不没有对 AMD 这样的模块规范进行处理，因此需要进行模块化处理，有两种方式：1.修改 underscore 的源码，加上 define(function(require, exports, module) 这样的定义；2. 采用 requirejs 提供的 shim 来进行处理。

15.5 捋捋结构

上面把文件拆分了一下，但是没有把 template 从页面提取出来。有兴趣的可以自己尝试下。最后我们来整理一下项目的结构。



具体的代码也可以到 wechat 中去看，在 requirejs 这个分支，代码中添加了 socketio，但是对上面的介绍没有影响。

第十六章 补充异常处理

忘了之前定这么个题目是要表达什么内容，就简单介绍下 wechat 项目中的错误处理。

16.1 error 的使用

说这个错误处理其实很简单，有过 JavaScript 经验的同学应该看到 Backbonejs 中定义的回调函数选项中的 error 参数就知道怎么写了。

需要处理错误的场景都是在客户端和服务器端通信时，在 wechat 中主要是 save 和 fetch 时。有一段代码展示下就是:

```
registe: function(evt){
    var reg_username_input = $('#reg_username');
    var reg_pwd_input = $('#reg_pwd');
    var reg_pwd_repeat_input = $('#reg_pwd_repeat');
    var u = new User({
        username: reg_username_input.val(),
        password: reg_pwd_input.val(),
        password_repeat: reg_pwd_repeat_input.val(),
    });
    u.save(null, {
        success: function(model, resp, options){
            g_user = resp;
            // 跳转到 index
            appRouter.navigate('index', {trigger: true});
        },
        error: function(model, resp, options) {
            alert(resp.responseText);
        }
    });},
```

这是用户注册时的代码，在 save 的参数的第二个参数部分添加了 error 的处理，具体功能就是 alert 出服务器端传回来的 response 的内容。

那么这么错误是什么时候触发的呢？正常情况下是触发 success 对应的 function，那么什么是正常呢？正常和错误在 Backbone.js 中是通过返回的 HTTP 状态码来区分的（应该说是 jQuery 或者 zepto 这样下一层处理 ajax 的库），jQuery 中错误判断的代码是这样的：`if ( status >= 200 && status < 300 || status === 304 ) {` 。

因此，对应着服务端的处理就是返回非 20x 和 304 的错误就行，一般客户端的错误都会返回 400（40x 系列）这样的错误，服务器端的错误一般都是 500 以上的错误。

对应上面的的错误，服务器端在使用 web.py 框架要这么处理：

```
def POST(self):
    data = web.data()
    data = json.loads(data)
    username = data.get("username")
    password = data.get("password")
    password_repeat = data.get("password_repeat")

    if password != password_repeat:
```

```
        # 会返回 HTTP400 的错误，内容是 message 的内容
        raise web.BadRequest(message='两次密码输入不一致')

    user_data = {
        "username": username,
        "password": sha1(password),
        "registed_time": datetime.now(),
    }

    try:
        user_id = User.create(**user_data)
    except sqlite3.IntegrityError:
        raise web.BadRequest(message="用户名已存在!")

    user = User.get_by_id(user_id)
    session.login = True
    session.user = user

    result = {
        'id': user_id,
        'username': username,
    }
    return json.dumps(result)
```

这样就 ok 了。看起来都是基本的东西。

参考： HTTP 状态码

第十七章 定制 Backbone.js

这里说的定制 Backbone.js，主要是定制 Backbone 中的 sync 部分，也就是最后和服务器端通信的部分。

17.1 三个级别的定制

首先得说，在 Backbone 里面和后端能通信的对象也就两个——Model 和 Collection。这俩的主要工作就是从服务器拉取数据，保存到实例中，或者把实例中的属性发送到服务器端。

上面两中类型的对象都是基于 Backbone.sync 来进行通信的，同时也可以定义各自的 sync 方法，类似这样：

Model 级别的

```
var Message = Backbone.Model.extend({
    urlRoot: '/message',
    # 这么写, 打印要操作的实体, 调用系统的 sync
    sync: function(method, model, options){
        console.log(model);
        return Backbone.sync(method, model, options);
    },});
```

Collection 级别的

```
var Messages = Backbone.Collection.extend({
    url: '/message',
    model: Message,
    # 这么写, 打印要操作的实体, 调用系统的 sync
    sync: function(method, collection, options){
        console.log(collection);
        return Backbone.sync(method, collection, options);
    },});
```

当然，也会存在这样的需求，要修改全局的 sync：

```
var old_sync = Backbone.sync;Backbone.sync = function(method, model, options) {
    console.log(model);
    return old_sync(method, model, options);}
```

所谓定制，是 Backbonejs 给开发者提供的可以被重写的接口。因此定制过程也得符合 Backbone 对 sync 的定义。

17.2 简单实例，用 socketio 通信

在 wechat 这个项目中为了保证聊天的实时性，我引入了 socketio，后端使用 gevent-socketio，（socketio 这个东西不打算写，和主题关系不大，有需求的可以题 issue）。

实时聊天的需求主要是在 Message 上，用户 A 发送一个请求，在同一聊天室内的其他用户应该立马能看见。之前有两个版本：

> 第一版是发送 message 时会调用 messages 的 fetch 方法，也就是用户只有发言才能看到别人发的聊天内容，代码如下：

```
message.save(null, {
    success: function(model, response, options){
        comment_box.val('');
        // 重新获取，看服务器端是否有更新
        // 比较丑陋的更新机制
        messages.fetch({
            data: {topic_id: topic_id},
            success: function(){

self.message_list.scrollTop(self.message_list_div.scrollHeight);
                messages.add(response);
            },
        });
    },});
```

- 第二版是引入 socketio 之后，在 save 完数据之后，通过 socket 把数据再次发送到服务器上的 socket 端，服务器再挨个发送数据到各个客户端，代码如下：

```
message.save(null, {
    success: function(model, response, options){
        comment_box.val('');
        messages.add(response);
        // 发送成功之后，通过 socket 再次发送
        // FIXME: 最后可通过 socket 直接通信并保存
        socket.emit('message', response);
    },});
// 对应着有一个 socket 监听，监听服务器发来的消息// 监听 message 事件，添加对话到 messages 中socket.on('message', function(response) {
    messages.add(response);});
```

可以看得出来，上面的第二版算是比较合适了，但是还是有些别扭，数据要重复发送。因此终于到了需要定制的时刻了。

上面说了，有三种级别的定制。根据我的需求，只需要定制 Model 级别的就可以了，怎么定制呢？

和一开头的示例代码类似：

```
var Message = Backbone.Model.extend({
    urlRoot: '/message',

    sync: function(method, model, options){
        if (method === 'create') {
```

```
            socket.emit('message', model.attributes);
            $('#comment').val('');
        } else {
            return Backbone.sync(method, model, options);
        };
    },});
// 对应着上面的那个 message.save 后的一堆东西都可以去掉了，直接message.save();
```

这样就好了，客户端只需要发送一次数据。但要记得在服务器端的监听 message 的接口上添加保存 message 的逻辑。

好了，定制就介绍这么多。关于上面提到的代码想了解上下文的，可以到我的 wechat 这个项目的 master 分支查看。

第十八章 再次总结的说

终于又到了写着一篇的时候了，从去年(2013)8 月份决定再更新一版 Backbonejs 入门教程。原因在前言中已经介绍过了，主要是填一些坑。

看看上一版的那个总结 16、总结的说 的时间——2012.4.18。再看看这一篇的时间——2014.4.15，差不多刚好两年的时间。之前的状态是刚接触前端框架方面的东西，像是在迷雾中行走，摸摸探探的总算知道这是条什么样的路。

现在的这一本更多的是希望从一个实用的角度出发，在长期的实践开发中总结出来的经验是：如果你学了一个东西，最终没有把它用到实际应用中，那么你学习这个东西最终产生的价值就抵不上你所花得成本。软件开发这东西，大部分情况下你只有切身的经历了技术对最终产品的作用，你才会对如何运用这一技术有些感觉，不然的话真的是纸上谈兵。

因此在这一版中我一开始写的时候就确定了一个最终的目标，那就是要做一个上线的系统，能够让互联网用户访问得到。另外还希望所有用心看了这一系列教程的 Backbonejs 初学者能够把学到的东西运用进来。

在最开始确定要不要刨这么一个坑的时候我犹豫了很久。这个框架在实际的工作中并没有用到，这意味着我对 Backbonejs 的再次学习和整理对工作上的事情没有太大帮助，虽然我会据此同我们的前端同学进行交流。

时间成本是最大的成本，在后端的世界里也有很多东西值得我去探索。犹豫几天之后我写了这篇文章 心存恐惧便无自由 。我觉得应该去做一些自己想做的事情，心里不受任何约束的。

一些牢骚完之后，再说些技术的东西。

Backbonejs 学习最大的难点在于你需要有以一个前端 MVC 框架的概念。一旦有了这么一个概念之后，你就会意识到，所有的前端框架只是填充了 MVC 中的各个部分，然后提供给你一个工具，让你可以按这种方式来组织你的代码。

因此在学习过程中，你也只需要掌握里面的几个概念：Model、Collection、View、Router，剩下的东西就和框架无关了，都是你软件开发经验的运用。新手的话建议去参透官网的那个 todos 案例，关于这个案例的事情这里不再赘述了。

好了，就写这么多。这个持久的工程总算是到了尾声了。

基于这个 wechat 这个项目，我创建了 http://bb-js.org 这样一个可实时交流的社区，也是读者在练习完 Backbonejs 可以参与社区开发中的一个项目。有兴趣的读者可以在读完本书，向我申请，申请条件是你得了解所有 wechat 项目的代码，并且知道如何添加新功能。

Backbonejs 相关资源

这次没有参考太多资源，主要还是官方的东西：

官网： http://backbonejs.org/

能让你大概认识 backbone.js 是什么以及怎么用的网站： http://backbonetutorials.com/

源码： https://github.com/jashkenas/backbone/blob/master/backbone.js

todos 代码： http://backbonejs.org/examples/todos/index.html

另外还有几篇中文的博客也不错： http://weakfi.iteye.com/blog/1391990

http://blog.csdn.net/soasme/article/details/6581029

http://www.cnblogs.com/nuysoft/archive/2012/03/19/2404274.html

当你对这个东西有一个感觉之后，去看最本质的东西才是王道！