# 1. 前言

　　本文的目的，是帮助大家快速理解掌握Backbone的使用，但它并不是API，因此我不会将每一个方法都详细地描述，但是我会告诉你如何学习它们。

　　这是一篇稍稍较长的技术文章，因为我想将我所了解的东西尽可能详细地分享给大家。但它还不至于出版为一本书籍，因为Backbone其实非常简单。当然，这并不表示Backbone提供的功能有限，相反，它能用简单的实现构建出复杂的应用，这正是Backbone的强大之处。

　　在正式学习Backbone之前，你需要准备一些东西：

　　首先，你需要获取Backbone框架源文件：<http://documentcloud.github.com/backbone/>

　　Backbone依赖于Underscore框架的基础方法，因此，你同时需要下载Underscore框架的源文件：<http://documentcloud.github.com/underscore/>

　　在Backbone中，对DOM和事件的操作依赖于第三方库（如jQuery或Zepto），你只需要选择其中一个下载：

　　jQuery：<http://jquery.com/>

　　Zepto：<http://zeptojs.com/>

　　看起来好像挺麻烦的样子，但Backbone的目的是在于使用简单的框架来构建复杂的应用，因此麻烦并不表示它很复杂。

　　在后续章节，我将详细介绍它们之间的关系和依赖，但在此之前，你可以新建一个HTML页面，并将这些框架引入进来，就像这样：

**Html代码  [收藏代码](javascript:void())**

1. **<script** type="text/javascript" src="jquery/jquery-1.8.2.min.js"**></script>**
2. **<script** type="text/javascript" src="underscore/underscore-min.js"**></script>**
3. **<script** type="text/javascript" src="backbone/backbone-min.js"**></script>**

　　至此，你已经准备好了运行Backbone必须的环境。

　　在学习过程中，你可以参考我的另一篇文章《Backbone及Underscore源码分析》：<http://yujianshenbing.iteye.com/blog/1731340>，它有助于你快速了解Backbone和Underscore的内部实现。

　　如果学习过程中有任何问题，或者你有更好的建议，欢迎联系我一起交流：

　　邮箱/新浪微博：[wubing@188.com](mailto:wubing@188.com) [御剑神兵](http://weibo.com/u/3101694513)

# 2. 什么是Backbone?

　　Backbone是一个轻量级的前端MVC框架，用于结构化管理页面中的大量JS，建立与服务器、视图间的无缝连接，为构建复杂的应用提供基础框架。

下面我先简单地阐述下Backbone的主要特点及特性：

## 2.1 轻量级

　　Backbone的源码只有1000行左右（去注释和空行后），文件大小只有16KB，加上依赖库Underscore，也仅有29KB。

　　你只需要花一点时间，就能轻松了解Backbone内部实现；或编写少量代码，来重载Backbone的部分机制；如果你想在Backbone的基础上做二次开发，也并不是一件复杂的事情。

## 2.2 MVC结构化

　　Backbone可以轻松将页面中的数据、逻辑、视图解耦，依照Backbone进行代码结构组织，你可以将项目中的数据交互、业务逻辑、用户界面等工作，分配给多个同事同时开发，并能有序地组织到一起。同时，这对于大型和复杂项目的维护开发非常有帮助。

## 2.3 继承机制

　　在Backbone中，模块是可以被继承的，你可以通过面向对象的方式将应用中的数据模型、集合、视图有序地组织，让整个架构更加清晰；也可以方便地重载和扩展自定义方法。

## 2.4 建立与服务器的无缝连接

　　在Backbone中内置了一套与服务器数据的交互规则（如果你了解REST架构，就能够轻松地理解它们），而数据的同步工作会在Model中自动进行，前端开发人员只需对客户端数据的进行操作，Backbone会自动将操作的数据同步到服务器。

　　这是件非常有趣的事情，因为服务器数据接口对前端开发者来说是透明的，他们不需要再关心如何和服务器交互。

　　然而服务器提供的数据接口也需要兼容Backbone的规则，对于一个新的项目来说，我们可以尝试使用这套规则来构建接口。但如果你的项目中已经有一套稳定的接口，你可能会担心接口改造的风险。

　　没关系，我们可以通过重载Backbone.sync方法来适配现有的数据接口，针对不同的客户端环境，我们还可以实现不同的数据交互方式。例如：用户通过PC浏览器使用服务时，数据会实时同步到服务器；而用户通过移动终端使用服务时，考虑到网络环境问题，我们可以先将数据同步到本地数据库，在合适的时候再同步到服务器。而这些只需要你重载一个方法就可以实现。

## 2.5 界面事件管理

　　在MVC中，我们希望能将界面展现和业务逻辑完全分离，但对于用户产生的交互事件（如单击事件），我们却常常通过类似jQuery中的bind方法进行获取和绑定。

　　Backbone中的视图帮助我们将用户事件和执行方法有序的组织起来，这只需要我们声明一个简单的表达式，例如：

**Js代码  [收藏代码](javascript:void())**

1. events: {
2. // 单击id为”save”的元素时，执行视图的add方法
3. 'click #save': 'add',
4. 'mousedown .button': 'show',
5. 'mouseover .button': 'hide'
6. }

　　在表达式中，事件名称可以是任意的DOM事件（如click、mouseover、keypress等），元素可以是jQuery支持的任意选择器（如标签选择器、id选择器、class选择器等）。

　　视图会自动将表达式中的事件绑定到选择器元素，当元素的事件被触发后，视图会自动调用表达式中绑定的方法。

## 2.6 轻量级模板解析

　　模板解析是Underscore中提供的一个方法。为什么我要在介绍Backbone特性时引入Underscore中的方法？因为该方法能帮助我们更好地分离视图结构和逻辑，且Underscore是Backbone必须依赖的库。

　　模板解析方法能允许我们在HTML结构中混合嵌入JS代码，就像在JSP页面中嵌入JAVA代码一样：

**Js代码  [收藏代码](javascript:void())**

1. <ul>
2. <% **for**(**var** i = 0; i < len; i++) { %>
3. <li><%=data[i].title%></li>
4. <% } %>
5. </li>

　　通过模板解析，我们不需要在动态生成HTML结构时拼接字符串，更重要的是，我们可以将视图中的HTML结构独立管理（例如：不同的状态可能会显示不同的HTML结构，我们可以定义多个单独的模板文件，按需加载和渲染即可）。

## 2.7 自定义事件管理

　　在Backbone中，你可以使用on或off方法绑定和移除自定义事件。在任何地方，你都可以使用trigger方法触发这些绑定的事件，所有绑定过该事件的方法都会被执行，如：

**Js代码  [收藏代码](javascript:void())**

1. **var** model = **new** Backbone.Model();
2. // 在model对象中向自定义事件custom绑定两个函数
3. model.on('custom', **function**(p1, p2) {
4. // todo
5. });
6. model.on('custom', **function**(p1, p2) {
7. // todo
8. });
9. // 触发custom事件，将调用上面绑定的两个函数
10. model.trigger('custom', 'value1', 'value2');
11. // 移除custom事件中绑定的所有方法
12. model.off('custom');
13. // 触发custom事件，但不会执行任何函数，已经事件中的函数已经在上一步被移除
14. model.trigger('custom');

　　如果你熟悉jQuery，你会发现它们与jQuery中的bind、unbind和trigger方法非常类似。

　　另外，Backbone支持一个特殊事件”all”，当在一个对象中绑定了名为”all”的事件后，该对象在触发任何事件时，都会同时触发”all”事件中绑定的方法。有时这种方法会非常有用，例如我们可以通过”all”事件监听对象状态的变化。

## 2.8 路由器

　　在单页应用中，我们通过JavaScript来控制界面的切换和展现，并通过AJAX从服务器获取数据。

　　可能产生的问题是，当用户希望返回到上一步操作时，他可能会习惯性地使用浏览器“返回”和“前进”按钮，而结果却是整个页面都被切换了，因为用户并不知道他正处于同一个页面中。

　　对于这个问题，我们常常通过Hash（锚点）的方式来记录用户的当前位置，并通过onhashchange事件来监听用户的“前进”和“返回”动作，但我们发现一些低版本的浏览器（例如IE6）并不支持onhashchange事件。

　　Backbone提供了路由控制功能，通过Backbone提供的路由器，我们能通过一个简单的表达式将路由地址和事件函数绑定在一起，例如：

**Js代码  [收藏代码](javascript:void())**

1. **var** CustomRouter = Backbone.Router.extend({
2. routes : {
3. '' : 'index', // 当URL Hash在根目录时执行index方法：url#
4. 'list' : 'getList', // 当URL Hash在list节点时执行getList方法：url#list
5. 'detail/:id' : 'query', // 当URL Hash在detail节点时执行query方法，并将detail后的数据作为参数传递给query方法：url#list/1001
6. '\*error' : 'showError' // 当URL Hash不匹配以上规则时, 执行error方法
7. },
8. index : **function**() {
9. alert('index');
10. },
11. getList : **function**() {
12. alert('getList');
13. },
14. query : **function**(id) {
15. alert('query id: ' + id);
16. },
17. showError : **function**(error) {
18. alert('error hash: ' + error);
19. },
20. });
22. **var** custom = **new** CustomRouter();
23. Backbone.history.start();

　　请尝试将这段代码复制到你的页面中，并依次访问以下地址（其中URL表示你的页面地址）：

* URL
* URL#list
* URL#detail/1001
* URL#hash1
* URL#hash2

　　请再试着使用浏览器的“返回”和“前进”按钮来回切换刚刚输入的地址。

　　你可以看到，当URL Hash发生变化时，会执行所绑定的方法，当遇到没有定义的Hash时，都会执行showError方法，并将未定义的Hash传递给该方法。

　　Backbone默认会通过Hash的方式来记录地址的变化，对于不支持onhashchange的低版本浏览器，会通过setInterval心跳监听Hash的变化，因此你不必担心浏览器的兼容性问题。

　　对于支持HTML5 pushState特性的浏览器，Backbone还允许你通过pushState来创建个性化的URL，但是这需要你的Web服务器做一些适配。

# 3. Backbone的适用性

　　Backbone并不像jQuery那样具有非常强的适用性，如果你正准备构建一个大型或复杂的单页Web应用，那么Backbone再适合不过。

　　如果想将Backbone应用到你的网站页面中，且页面中并没有复杂的逻辑和结构，那么这只会让你的页面更加繁琐和难以维护。

　　如果你的项目并不复杂，但你却深深喜欢它的某个特性(可能是数据模型、视图管理或路由器)，那么你可以将这部分源码从Backbone中抽取出来，因为在Backbone中，各模块间的依赖并不是很强，你能轻易的获取并使用其中的某一个模块。

# 4. 依赖库

　　你不能独立使用Backbone，因为它的基础函数、DOM操作、AJAX都依赖于第三方库。

## 4.1 Underscore

　　（必选）

　　Underscore是一个用于提高开发效率的基础函数库，它封装了对集合、数组、对象、函数的常用操作，就像jQuery封装DOM对象一样，你能通过Underscore轻易地访问和操作JavaScript内部对象。

　　Underscore还提供了一些非常实用的函数方法，如：函数节流、模板解析等。

　　关于Underscore中一些主要的方法，我会在下一章详细介绍，但在此之前你必须了解：Underscore是Backbone必须依赖的库，因为在Backbone中许多实现都是基于Underscore。

## 4.2 jQuery和Zepto

　　（可选）

　　相信你对jQuery一定不会陌生，它是一个跨浏览器的DOM和AJAX框架。

　　而对于Zepto你可以理解为“移动版的jQuery”，因为它更小、更快、更适合在移动终端设备的浏览器上运行，它与jQuery语法相同，因此你能像使用jQuery那样使用它。

　　Zepto目前仅支持Webkit内核的浏览器，因此它能兼容IOS、Adnroid、塞班、黑莓和Meego等大部分移动系统，而对于Windows Phone或Firefox OS，它暂时还不支持。

　　因为jQuery和Zepto语法相同，因此对于Backbone来说，你无论是使用jQuery还是Zepto，都没有问题（当然，你不可能两个同时都用到）。

　　在Backbone中，DOM选择器、DOM事件和AJAX，都使用了jQuery的方法。这里之所以所它们是可选的，是假设你没有用到Backbone中的视图和AJAX数据同步功能，那么就不需要导入它们。

　　如果你不想使用jQuery或Zepto，而是使用其它的、或自定义库，只要你的库中实现了与jQuery语法相同的DOM选择器、事件管理和AJAX方法，那么就不会任何问题。

Backbone允许你通过setDomLibrary方法动态配置需要使用的第三方库，这种情况常常用于：

　　你的自定义库虽然包含了和jQuery相同语法的方法，但全局变量并不是$，而且你想保持现有的命名。这时你可以通过setDomLibrary方法将其设置为Backbone内部引用的对象。

　　你希望通过检查用户的环境，来决定更适合使用哪一个库。例如：如果用户使用PC浏览器访问，则载入jQuery，如果用户通过移动终端访问，则载入Zepto。

5. Underscore.js

　　Underscore封装了常用的JavaScript对象操作方法，用于提高开发效率。它本身与我们介绍的主题“Backbone”没有半毛钱的关系，因此你可以完全不理会“Backbone”的概念来学习它，或将它单独运用到任何一个页面。（另外，Underscore还可以被使用在Node.js运行环境。）

　　在学习Underscore之前，你应该先保存它的API地址，因为你将在以后经常访问它：

<http://documentcloud.github.com/underscore/>

　　从API中，你已经可以看出，Underscore没有任何复杂的结构和流程，它仅仅提供了一系列常用的函数。如果你将API中的方法从头至尾用一遍，你就会对它非常了解。

　　尽管如此，但我觉得还是有必要将一些重要的方法拿出来与大家讨论，它们十分重要，却在API中描述地还不够清楚。

## 5.1 Underscore对象封装

　　Underscore并没有在原生的JavaScript对象原型中进行扩展，而是像jQuery一样，将数据封装在一个自定义对象中（下文中称“Underscore对象”）。

　　你可以通过调用一个Underscore对象的value()方法来获取原生的JavaScript数据，例如：

**Js代码  [收藏代码](javascript:void())**

1. // 定义一个JavaScript内置对象
2. **var** jsData = {
3. name : 'data'
4. }
6. // 通过\_()方法将对象创建为一个Underscore对象
7. // underscoreData对象的原型中包含了Underscore中定义的所有方法，你可以任意使用
8. **var** underscoreData = \_(jsData);
10. // 通过value方法获取原生数据, 即jsData
11. underscoreData.value();

## 5.2 优先调用JavaScript 1.6内置方法

　　Underscore中有许多方法在JavaScript1.6中已经被纳入规范，因此在Underscore对象内部，会优先调用宿主环境提供的内置方法（如果宿主环境已经实现了这些方法），以此提高函数的执行效率。

　　而对于不支持JavaScript 1.6的宿主环境，Underscore会通过自己的方式实现，而对开发者来说，这些完全是透明的。

　　这里所说的宿主环境，可能是Node.js运行环境，或客户端浏览器。

## 5.3 改变命名空间

　　Underscore默认使用\_（下划线）来访问和创建对象，但这个名字可能不符合我们的命名规范，或容易引起命名冲突。

　　我们可以通过noConflict()方法来改变Underscore的命名，并恢复\_（下划线）变量之前的值，例如：

**Js代码  [收藏代码](javascript:void())**

1. <script type="text/javascript">
2. **var** \_ = '自定义变量';
3. </script>
4. <script type="text/javascript" src="underscore/underscore-min.js"></script>
5. <script type="text/javascript">
6. // Underscore对象
7. console.dir(\_);
8. // 将Underscore对象重命名为us, 后面都通过us来访问和创建Underscore对象
9. **var** us = \_.noConflict();
10. // 输出"自定义变量"
11. console.dir(\_);
12. </script>

## 5.4 链式操作

　　还记得我们在jQuery中是如何进行链接操作吗？例如：

**Js代码  [收藏代码](javascript:void())**

1. $('a')
2. .css('position', 'relative')
3. .attr('href', '#')
4. .show();

　　Underscore同样支持链式操作，但你需要先调用chain()方法进行声明：

**Js代码  [收藏代码](javascript:void())**

1. **var** arr = [10, 20, 30];
2. \_(arr)
3. .chain()
4. .map(**function**(item){ **return** item++; })
5. .first()
6. .value();

　　如果调用了chain()方法，Underscore会将所调用的方法封装在一个闭包内，并将返回值封装为一个Underscore对象并返回：

**Js代码  [收藏代码](javascript:void())**

1. // 这是Underscore中实现链式操作的关键函数，它将返回值封装为一个新的Underscore对象，并再次调用chain()方法，为方法链中的下一个函数提供支持。
2. **var** result = **function**(obj, chain) {
3. **return** chain ? \_(obj).chain() : obj;
4. }

## 5.5 扩展Underscore

　　我们可以通过mixin()方法轻松地向Underscore中扩展自定义方法，例如：

**Js代码  [收藏代码](javascript:void())**

1. \_.mixin({
2. method1: **function**(object) {
3. // todo
4. },
5. method2: **function**(arr) {
6. // todo
7. },
8. method3: **function**(fn) {
9. // todo
10. }
11. });

　　这些方法被追加到Underscore的原型对象中，所有创建的Underscore对象都可以使用这些方法，它们享有和其它方法同样的环境。

## 5.6 遍历集合

　　each()和map()方法是最常用用到的两个方法，它们用于迭代一个集合（数组或对象），并依次处理集合中的每一个元素，例如：

**Js代码  [收藏代码](javascript:void())**

1. **var** arr = [1, 2, 3];
3. \_(arr).map(**function**(item, i) {
4. arr[i] = item + 1;
5. });
7. **var** obj = {
8. first : 1,
9. second : 2
10. }
12. \_(obj).each(**function**(value, key) {
13. **return** obj[key] = value + 1;
14. });

　　map()方法与each()方法的作用、参数相同，但它会将每次迭代函数返回的结果记录到一个新的数组并返回。

## 5.7 函数节流

　　函数节流是指控制一个函数的执行频率或间隔（就像控制水流的闸门一样），Underscore提供了debounce()和throttle()两个方法用于函数节流。

　　为了更清楚地描述这两个方法，假设我们需要实现两个需求：

**需求1**：当用户在文本框输入搜索条件时，自动查询匹配的关键字并提示给用户（就像在Tmall输入搜索关键字时那样）

　　首先分析第1个需求，我们可以绑定文本框的keypress事件，当输入框内容发生变化时，查询匹配关键字并展示。假设我想查询“windows phone”，它包含13个字符，而我输入完成只花了1秒钟（好像有点快，就意思意思吧），那么在这1秒内，调用了13次查询方法。这是一件非常恐怖的事情，如果Tmall也这样实现，我担心它会不会在光棍节到来之前就挂掉了（当然，它并没有这么脆弱，但这绝对不是最好的方案）

　　更好的方法是，我们希望用户已经输入完成，或者正在等待提示（也许他懒得再输入后面的内容）的时候，再查询匹配关键字。

　　最后我们发现，在我们期望的这两种情况下，用户会暂时停止输入，于是我们决定在用户暂停输入200毫秒后再进行查询（如果用户在不断地输入内容，那么我们认为他可能很明确自己想要的关键字，所以等一等再提示他）

　　这时，利用Underscore中的debounce()函数，我们可以轻松实现这个需求：

**Js代码  [收藏代码](javascript:void())**

1. <input type="text" id="search" name="search" />
2. <script type="text/javascript">
3. **var** query = \_(**function**() {
4. // 在这里进行查询操作
5. }).debounce(200);
7. $('#search').bind('keypress', query);
8. </script>

　　你能看到，我们的代码非常简洁，节流控制在debounce()方法中已经被实现，我们只告诉它当query函数在200毫秒内没有被调用过的话，就执行我们的查询操作，然后再将query函数绑定到输入框的keypress事件。

　　query函数是怎么来的？我们在调用debounce()方法时，会传递一个执行查询操作的函数和一个时间（毫秒数），debounce()方法会根据我们传递的时间对函数进行节流控制，并返回一个新的函数（即query函数），我们可以放心大胆地调用query函数，而debounce()方法会按要求帮我们做好控制。

**需求2**：当用户拖动浏览器滚动条时，调用服务器接口检查是否有新的内容

　　再来分析第2个需求，我们可以将查询方法绑定到window.onscroll事件，但这显然不是一个好的做法，因为用户拖动一次滚动条可能会触发几十次甚至上百次onscroll事件。

　　我们是否可以使用上面的debounce()方法来进行节流控制？当用户拖动滚动条完毕后，再查询新的内容？但这与需求不符，用户希望在拖动的过程中也能看到新内容的变化。

　　因此我们决定这样做：用户在拖动时，每两次查询的间隔不少于500毫秒，如果用户拖动了1秒钟，这可能会触发200次onscroll事件，但我们最多只进行2次查询。

　　利用Underscore中的throttle()方法，我们也可以轻松实现这个需求：

**Js代码  [收藏代码](javascript:void())**

1. <script type="text/javascript">
2. **var** query = \_(**function**() {
3. // 在这里进行查询操作
4. }).throttle(500);
6. $(window).bind('scroll', query);
7. </script>

　　代码仍然十分简洁，因为在throttle()方法内部，已经为我们实现的所有控制。

　　你可能已经发现，debounce()和throttle()两个方法非常相似（包括调用方式和返回值），作用却又有不同。

　　它们都是用于函数节流，控制函数不被频繁地调用，节省客户端及服务器资源。

* debounce()方法关注函数执行的间隔，即函数两次的调用时间不能小于指定时间。
* throttle()方法更关注函数的执行频率，即在指定频率内函数只会被调用一次。

## 5.8 模板解析

　　Underscore提供了一个轻量级的模板解析函数，它可以帮助我们有效地组织页面结构和逻辑。

　　我将通过一个例子来介绍它：

**Js代码  [收藏代码](javascript:void())**

1. <!-- 用于显示渲染后的标签 -->
2. <ul id="element"></ul>
4. <!-- 定义模板，将模板内容放到一个script标签中 -->
5. <script type="text/template" id="tpl">
6. <% **for**(**var** i = 0; i < list.length; i++) { %>
7. <% **var** item = list[i] %>
8. <li>
9. <span><%=item.firstName%> <%=item.lastName%></span>
10. <span><%-item.city%></span>
11. </li>
12. <% } %>
13. </script>
14. <script type="text/javascript" src="underscore/underscore-min.js"></script>
15. <script type="text/javascript">
16. // 获取渲染元素和模板内容
17. **var** element = $('#element'),
18. tpl = $('#tpl').html();
20. // 创建数据, 这些数据可能是你从服务器获取的
21. **var** data = {
22. list: [
23. {firstName: '<a href="#">Zhang</a>', lastName: 'San', city: 'Shanghai'},
24. {firstName: 'Li', lastName: 'Si', city: '<a href="#">Beijing</a>'},
25. {firstName: 'Wang', lastName: 'Wu', city: 'Guangzhou'},
26. {firstName: 'Zhao', lastName: 'Liu', city: 'Shenzhen'}
27. ]
28. }
30. // 解析模板, 返回解析后的内容
31. **var** html = \_.template(tpl, data);
32. // 将解析后的内容填充到渲染元素
33. element.html(html);
34. </script>

　　在本例中，我们将模板内容放到一个<script>标签中，你可能已经注意到标签的type是text/template而不是text/javascript，因为它无法作为JavaScript脚本直接运行。

　　我也建议你将模板内容放在<script>中，因为如果你将它们写在一个<div>或其它标签中，它们可能会被添加到DOM树中进行解析（即使你隐藏了这个标签也无法避免）。

　　\_.template模板函数只能解析3种模板标签（这比Smarty、JSTL要简单得多）：

　　<%  %>：用于包含JavaScript代码，这些代码将在渲染数据时被执行。

　　<%= %>：用于输出数据，可以是一个变量、某个对象的属性、或函数调用（将输出函数的返回值）。

　　<%- %>：用于输出数据，同时会将数据中包含的HTML字符转换为实体形式（例如它会将双引号转换为&quot;形式），用于避免XSS攻击。

　　当我们希望将数据中的HTML作为文本显示出来时，常常会使用<%- %>标签。

　　Underscore还允许你修改这3种标签的形式，如果我们想使用{% %}、{%= %}、{%- %}作为标签，可以通过修改templateSettings来实现，就像这样：

**Js代码  [收藏代码](javascript:void())**

1. \_.templateSettings = {
2. evaluate : /\{%([\s\S]+?)\%\}/g,
3. interpolate : /\{%=([\s\S]+?)\%\}/g,
4. escape : /\{%-([\s\S]+?)%\}/g
5. }

　　在本例中，我们将模板内容和需要填充的数据传递给template方法，它会按以下顺序进行处理：

* 将模板内容解析为可执行的JavaScript（解析模板标签）
* 通过with语句将解析后的JavaScript作用域修改为我们传递的数据对象，这使我们能够直接在模板中通过变量形式访问数据对象的属性
* 执行解析后的JavaScript（将数据填充到模板）
* 返回执行后的结果

　　我们经常会遇到一种情况：多次调用template方法将数据渲染到同一个模板。

　　假设我们有一个分页列表，列表中的每一条数据都通过模板渲染，当用户进入下一页，我们会获取下一页的数据并重新渲染，实际上每次渲染的模板都是同一个，但刚才描述的template所有处理过程总会被执行。

　　其实Underscore的template方法提供了一种更高效的调用方式，我们将上面代码中的最后两句修改为：

**Js代码  [收藏代码](javascript:void())**

1. // 解析模板, 返回解析后的内容
2. **var** render = \_.template(tpl);
3. **var** html = render(data);
4. // 将解析后的内容填充到渲染元素
5. element.html(html);

　　你会发现细微的差别：我们在调用template方法时只传递了模板内容，而没有传递数据，此时template方法会解析模板内容，生成解析后的可执行JavaScript代码，并返回一个函数，而函数体就是解析后的JavaScript，因此当我们调用该函数渲染数据时，就省去了模板解析的动作。

你应该将返回的函数存储起来（就像我将它存储在render变量中一样），再通过调用该函数来渲染数据，特别是在同一个模板可能会被多次渲染的情况下，这样做能提高执行效率（具体提升多少，应该根据你的模板长度和复杂度而定，但无论如何，这都是一个良好的习惯）。

# 6. Events (事件管理)

　　从这一章开始，我将正式介绍Backbone的内容，过程中会有许多例子和代码，你应该将这些代码复制到你的页面，并查看它们的运行效果。

　　我介绍的第一个模块是Backbone.Events（事件管理），因为我在介绍任何其它的模块时都会提到它，你可以想象到它的重要性。

　　Events是Backbone中所有其它模块的基类，无论是Model、Collection、View还是Router和History，都继承了Events中的方法。

　　Events为其它的模块提供了事件管理相关的方法，包括on、off和trigger.

　　假如你熟悉jQuery，你可以跳过本节，因为它们与jQuery中的bind、unbind和trigger方法非常类似，但我并不建议你这样做，因为它们之间仍有不同。

　　我们无法直接实例化一个Events对象，因为它必须要依赖于Backbone中的某一个模块类（或子类）才能使用，例如：

**Js代码  [收藏代码](javascript:void())**

1. // 创建一个Model的实例
2. **var** m = **new** Backbone.Model();
4. // 将监听函数绑定到m对象的自定义事件custom中
5. m.on('custom', **function**(index) {
6. // 监听函数接收并显示参数index
7. alert(index);
8. });
10. // 循环触发m对象的custom事件
11. **for**(**var** i = 0; i < 3; i++) {
12. m.trigger('custom', i);
13. }
15. // 从m对象的custom事件中移除已绑定的所有监听函数
16. m.off('custom');
18. // 试着再次触发custom事件
19. m.trigger('custom', i);

　　我们创建了一个Model的实例，Model是Backbone中的数据模型类，目前你只需要知道它继承了Events类中的方法。

　　我们通过on方法给m对象的自定义事件custom绑定了一个监听函数，监听函数会接收一个参数index并将它显示出来。接着我们在一个循环中调用m对象的trigger方法触发custom事件，结果正如你看到的：监听函数接收了参数i并将其显示出来。最后我们通过off方法移除m对象custom事件中已绑定的监听函数，当试着再次触发custom事件时，不会再执行任何动作。

　　（在本例中，我们创建了Model类的实例，这仅仅是为了让例子显得更简单，而实际上我们常常都会继承Model而不是直接实例化）

　　通过上面的例子中，我们可以为Events中的方法这样定义：

* on：将一个函数绑定到对象的某个事件中
* off：移除对象某个事件中已绑定的所有函数（实际上它还可以只移除指定的函数，或移除对象中已绑定的所有事件，这与调用off方法时传递的参数有关，请参考API文档）
* trigger：触发对象的某个事件

　　这看起来和jQuery中的事件方法非常类似（当然，jQuery还支持DOM事件），接下来我们再看另一个例子：

**Js代码  [收藏代码](javascript:void())**

1. // 创建一个Model的实例
2. **var** m = **new** Backbone.Model();
4. // 将监听函数绑定到m对象的all事件中
5. m.on('all', **function**() {
6. alert('all');
7. });
8. // 将监听函数绑定到m对象的自定义事件show中
9. m.on('show', **function**() {
10. alert('show title');
11. });
12. // 将另一个监听函数绑定到m对象的自定义事件show中
13. m.on('show', **function**() {
14. alert('show content');
15. });
16. // 将监听函数绑定到m对象的自定义事件hide中
17. m.on('hide', **function**() {
18. alert('hide');
19. });
21. // 触发m对象的show事件和hide事件
22. m.trigger('show');
23. m.trigger('hide');

　　执行显示的结果分别为：

　　show title

　　show content

　　all

　　hide

　　all

　　在本例中，我们为m对象的all事件绑定了监听函数，all是Backbone中一个特殊的事件，它在当前对象触发任何事件之后，都会被自动触发。（你也可以通过trigger直接触发all事件）

　　你需要注意监听函数的调用顺序，all事件总会在其它事件中的监听函数都执行完毕之后触发，同一个事件中如果绑定了多个监听函数，那它们将按照函数绑定时的顺序依次调用。

　　在Backbone内部，会为每个对象添加一个\_callbacks私有属性，用于记录该对象中绑定的事件和监听函数列表。Events类的on、off和trigger方法都围绕每个对象的\_callbacks列表在操作。

　　（你最好能在每阅读一个章节时，同时浏览Backbone对应的源码，这能帮助你加深记忆和理解，请参考：<http://yujianshenbing.iteye.com/blog/1731340>）

# 7. Model (数据模型)

　　Model是Backbone中所有数据模型的基类，用于封装原始数据，并提供对数据进行操作的方法，我们一般通过继承的方式来扩展和使用它。

　　如果你做过数据库开发，可能对ORM（对象关系映射）不会陌生，而Backbone中的Model就像是映射出来的一个数据对象，它可以对应到数据库中的某一条记录，并通过操作对象，将数据自动同步到服务器数据库。（下一节即将介绍的Collection就像映射出的一个数据集合，它可以对应到数据库中的某一张或多张关联表）。

## 7.1 创建数据模型

　　我们先通过一段代码来看看如何创建数据模型：

// 定义Book模型类

var Book = Backbone.Model.extend({

defaults : {

name : 'unknown',

author : 'unknown',

price : 0

}

});

// 实例化模型对象

var javabook = new Book({

name : 'Thinking in Java',

author : 'Bruce Eckel',

price : 395.70

});

　　我们通过Model.extend方法，定义一个自己的模型类Book。

　　Backbone模块类（包括子类）都包含一个extend静态方法用于实现继承。给extend方法传递的第一个参数应该是一个对象，对象中的属性和方法将被添加到子类，我们可以通过extend方法扩展子类或重载父类的方法。

　　从Backbone模块类继承的子类，都包含一个\_\_super\_\_静态属性，这是一个指向父类原型对象的引用，例如：

var Book = Backbone.Model.extend({

constructor: function() {

Book.\_\_super\_\_.constructor.call(this)

}

});

　　在这个例子中，我们重载了Model类的构造函数，但我们希望在子类被实例化时，调用父类的构造函数，因此我们可以通过引用Book.\_\_super\_\_.constructor来调用它。

　　实际上我们一般并不会重载模块类的constructor方法，因为在Backbone中所有的模块类都提供了一个initialize方法，用于避免在子类中重载模块类的构造函数，当模块类的构造函数执行完成后会自动调用initialize方法。

　　回到本节的第一个例子，我们在定义Book类的时候，传递了一个defaults参数，它用于定义模型数据的默认状态，虽然我们在创建Book实例后再添加它们，但为每个数据模型定义属性列表和默认值，是一个好的编码习惯。

　　最后，我们通过new关键字，创建了一个Book的实例，并向它的构造函数中传递了一系列初始化数据，它们将覆盖defaults中定义的默认状态。

## 7.2 初始化和读取数据

　　在我们定义好一个模型类之后，可以通过new关键字实例化该模型的对象。

　　如果模型类在定义时设置了defaults默认数据，这些数据将被复制到每一个实例化的对象中，如：

// 定义Book模型类

var Book = Backbone.Model.extend({

defaults : {

name : 'unknown',

author : 'unknown',

price : 0

}

});

// 实例化模型对象

var javabook = new Book();

　　上面的代码创建了一个Book实例javabook，它包含了模型类在定义时的默认数据。

　　我们将实例化的代码稍作修改：

// 实例化模型对象

var javabook = new Book({

name : 'Thinking in Java'

});

// 通过get和escape()方法获取模型中的数据

var name = javabook.get('name');

var author = javabook.escape('author');

var price = javabook.get('price');

// 在控制台输出模型中的数据name

console.log(name); // 输出Thinking in Java

console.log(author); // 输出unknown

console.log(price); // 输出0

　　我们在实例化对象时传递了初始数据，它将覆盖Book类定义时defaults中的默认数据，这一点很容易理解。

　　上面的例子中我们通过get()和escape()方法获取模型中的数据，它们的区别在于：

* get()方法用于直接返回数据
* escape()方法先将数据中包含的HTML字符转换为实体形式（例如它会将双引号转换为&quot;形式）再返回，用于避免XSS攻击。

　　模型将原始数据存放在对象的attributes属性中，因此我们也可以通过javabook.attributes属性直接读取和操作这些数据，例如：

// 在控制台直接输出对象的attributes属性

console.dir(javabook.attributes);

// 控制台输出结果

// {

// author: 'unknown',

// name: 'Thinking in Java',

// price: 0

// }

　　但通常并不会这样做，因为模型中数据状态的变化会触发一系列事件、同步等动作，直接操作attributes中的数据可能导致对象状态异常。更安全的做法是：通过get()或escape()方法读取数据，通过set()等方法操作数据。

## 7.3 修改数据

　　我们通常可以调用模型对象的set()方法，来修改模型中的数据，例如：

// 实例化模型对象

var javabook = new Book();

// 通过set方法设置模型数据

javabook.set('name', 'Java7入门经典');

javabook.set('author', 'Ivor Horton');

javabook.set('price', 88.50);

// 获取数据并将数据输出到控制台

var name = javabook.get('name');

var author = javabook.get('author');

var price = javabook.get('price');

console.log(name); // 输出Java7入门经典

console.log(author); // 输出Ivor Horton

console.log(price); // 输出88.50

set()方法也允许同时设置多个属性，例如：

javabook.set({

name : 'Java7入门经典',

author : 'Ivor Horton',

price : 88.50

});

　　当调用set()方法修改模型中的数据时，会触发一系列事件，我们常常通过监

听这些事件，来动态调整界面中数据的显示，我们先来看一个例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : 'unknown',
5. author : 'unknown',
6. price : 0
7. }
8. });
10. // 实例化模型对象
11. **var** javabook = **new** Book();
13. // 监听模型"change"事件
14. javabook.on('change', **function**(model) {
15. console.log('change事件被触发');
16. });
17. // 监听模型"change:name"事件
18. javabook.on('change:name', **function**(model, value) {
19. console.log('change:name事件被触发');
20. });
21. // 监听模型"change:author"事件
22. javabook.on('change:author', **function**(model, value) {
23. console.log('change:author事件被触发');
24. });
25. // 通过set()方法设置数据
26. javabook.set({
27. name : 'Thinking in Java',
28. author : 'unknown',
29. price : 395.70
30. });
32. // 控制台输出结果:
33. // change:name事件被触发
34. // change事件被触发

　　在本例中，我们监听了模型对象的change事件，该事件在模型对象的任何数据发生改变时被触发，change事件触发时，会将当前模型作为参数传递给监听函数。

　　我们还监听了change:name和change:author两个属性事件，属性事件是当模型中对应属性的数据发生改变时被触发，属性事件按照“change:属性名”来命名，因此它并不固定。属性事件触发时，会将当前模型和最新的数据作为参数传递给监听函数。

　　本例执行后在控制台的输出结果为：

　　“change:name事件被触发

　　change事件被触发”

　　从结果中看，并没有触发我们监听的change:author事件，因为在调用set()方法时，它会在内部检查新的数据较上一次是否发生变化，只有发生变化的数据才会被设置和触发监听事件。

　　另一个细节是，我们先监听了change事件，然后监听了属性事件，但事件在触发时，总是会先触发属性事件，然后再触发change事件。

　　Backbone允许我们在修改模型数据时获取上一个状态的数据，这常常用于数据比较和数据回滚。

　　例如在下面的例子中，我们希望当price价格被改变时，提示用户价格的变化情况：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : 'unknown',
5. author : 'unknown',
6. price : 0
7. }
8. });
10. // 实例化模型对象
11. **var** javabook = **new** Book();
13. // 监听"change:price"事件
14. javabook.on('change:price', **function**(model, value) {
15. **var** price = model.get('price');
17. **if**(price < value) {
18. console.log('价格上涨了' + (value - price) + '元.');
19. } **else** **if**(price > value) {
20. console.log('价格下降了' + (value - price) + '元.');
21. } **else** {
22. console.log('价格没有发生变化.');
23. }
24. });
25. // 设置新的价格
26. javabook.set('price', 50);
28. // 控制台输出结果:
29. // 价格没有发生变化.

　　我们通过监听change:price事件来监听价格的变化，并希望将最新的价格和当前（上一次）的价格作比较，但控制台的输出结果却是“价格没有发生变化.”。这是因为当change事件或属性事件被触发时，模型中的数据已经被修改，因此通过get()方法获取到的是模型中最新的数据。

　　这时，我们可以通过previous()和previousAttributes()方法来获取数据被修改之前的状态。

　　我们将代码稍作修改，只需要修改监听事件的函数即可：

**Js代码  [收藏代码](javascript:void())**

1. // 监听"change:price"事件
2. javabook.on('change:price', **function**(model, value) {
3. **var** price = model.previous('price');
5. **if**(price < value) {
6. console.log('价格上涨了' + (value - price) + '元.');
7. } **else** **if**(price > value) {
8. console.log('价格下降了' + (value - price) + '元.');
9. } **else** {
10. console.log('价格没有发生变化.');
11. }
12. });

　　我们将get()方法修改为previous()方法，用来获取价格在修改之前的状态，此时控制台输出的结果为：“价格上涨了50元.”

* previous()方法接收一个属性名，并返回该属性在修改之前的状态；
* previousAttributes()方法返回一个对象，该对象包含上一个状态的所有数据。

　　需要注意的是，previous()和previousAttributes()方法只能在数据修改过程中调用（即在模型的change事件和属性事件中调用），比如下面的例子就是错误的调用方法：

**Js代码  [收藏代码](javascript:void())**

1. // 设置新的价格
2. javabook.set('price', 50);
4. **var** prevPrice = javabook.previous('price');
5. **var** newPrice = javabook.get('price');
7. **if**(prevPrice < newPrice) {
8. console.log('价格上涨了' + (newPrice - prevPrice) + '元.');
9. } **else** **if**(prevPrice > newPrice) {
10. console.log('价格下降了' + (newPrice - prevPrice) + '元.');
11. } **else** {
12. console.log('价格没有发生变化.');
13. }
15. // 控制台输出结果:
16. // 价格没有发生变化.

　　控制台输出的结果是“价格没有发生变化.”，因为在set()方法被调用完毕后，模型的上一个状态也会被新数据替换。

　　（有一种特殊情况是当我们使用了silent配置时，上面的代码可以得到我们想要的结果，关于silent配置将在后面“数据验证”章节中介绍）

## 7.4 数据验证

　　Backbone模型提供了一套数据验证机制，确保我们在模型中存储的数据都是通过验证的，我们通过下面的例子来说明这套验证机制：

　　执行这段代码，你会在控制台看到这段信息：“书籍价格不应低于1元.”

　　在定义模型类时，我们可以添加一个validate方法，该方法会在模型中的数据发生改变之前被自动调用（就像我们通过set()方法修改数据时一样）。

　　validate方法接收一个参数，表示需要进行验证的数据集合，如果validate方法没有任何返回值（即undefined），则表示验证通过；如果验证不通过，我们常常会返回一个错误字符串或自定义对象。但实际上，当你返回一个false也会被认为验证通过，因为Backbone内部会将validate的返回值转换为布尔类型，如果为false则认为验证通过，反之则认为不通过（虽然这听起来有些别扭）。

　　当validate验证不通过时，会触发error事件，并将模型对象和validate方法的返回值传递给error事件的监听函数（就像例子中的那样）。

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. validate : **function**(data) {
4. **if**(data.price < 1) {
5. **return** '书籍价格不应低于1元.';
6. }
7. }
8. });
10. **var** javabook = **new** Book();
12. // 监听error事件，当验证失败时触发
13. javabook.on('error', **function**(model, error) {
14. console.log(error);
15. });
17. javabook.set('price', 0);

　　上面的例子中，我们监听了javabook对象的error事件，用于在验证不通过时提示用户。但在某个场景下，我希望以另一种方式提示用户，我可以在error监听函数中判断是否处于这种场景下，然后作出不同的提示，但这显然不是最好的办法。

　　因此，Backbone提供了另一种方式对error事件进行覆盖，来看看这个例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. validate : **function**(data) {
4. **if**(data.price < 1) {
5. **return** '书籍价格不应低于1元.';
6. }
7. **return** **false**;
8. }
9. });
11. **var** javabook = **new** Book({
12. price : 50
13. });
15. // 监听error事件，当验证失败时触发
16. javabook.on('error', **function**(model, error) {
17. console.log(error);
18. });
20. // 在调用set()方法时，传递了一个配置对象，包含自定义的error处理方法
21. javabook.set('price', 0, {
22. error : **function**(model, error) {
23. console.log('自定义错误：' + error);
24. }
25. });

　　在这段代码中，我们在调用set()方法时，传递了第三个参数，它是一个用于描述配置信息的对象，我们设定了一个error函数。当validate方法验证失败时，会优先调用配置中传递的error函数，如果没有传递error函数，则会触发error事件。

　　如果我们不希望对数据进行验证，可以在调用set()方法时传递一个silent配置，silent配置用于忽略验证规则，并且它不会触发change和error等事件。

　　在上面例子的基础上，我们修改一下set()方法的调用方式：

**Js代码  [收藏代码](javascript:void())**

1. javabook.set('price', 0, {
2. silent : **true**
3. });

　　我们设置了一个非法的数据，同时指定了silent配置，结果并没有触发error事件，因为它并没有调用validate方法对数据进行验证。

　　我们再将set()方法的调用代码修改一下：

**Js代码  [收藏代码](javascript:void())**

1. javabook.set('price', 0, {
2. silent : **true**
3. });
5. javabook.set('name', 'Thinking in Java');

　　执行这段代码，控制会再次输出“书籍价格不应低于1元.”。

　　从结果中可以看出，silent配置仅仅是忽略对本次数据的验证，而在下一次数据发生变化时，会重新执行validate方法进行验证。

　　使用silent配置设置数据时，模型也不会触发change事件，这这意味着数据在业务层面并没有真正被设置到模型中，这有利于我们及时对错误的数据进行恢复和回滚。

　　我们来看一个数据回滚的例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend();
4. // price的初始值为50
5. javabook = **new** Book({
6. price : 50
7. });
9. // 我们设置一个非法数据并绕过验证
10. javabook.set('price', 0.2, {
11. silent : **true**
12. });
14. // 检查数据非法，回滚数据
15. **if**(javabook.get('price') < 1) {
16. **var** value = javabook.previous('price');
17. javabook.set('price', value);
18. }

　　在本例中，我们在实例化Book模型时设置price的初始化数据为50，然后将price设置为一个非法数据并绕过验证，最后调用previous()方法获取数据的上一个状态，并重新设置。

　　值得注意的是，在上一节中我们提到，previous()方法只有在change事件的监听函数中才能获取数据的上一个状态。而这里我并没有这样做，因为当我们使用silent配置时，设置的数据仅仅是被临时存放在模型，这种情况下Backbone允许我们随时调用previous()和previousAttributes()方法获取数据的上一个状态。

　　（数据的上一个状态会在下一次调用set()方法将数据设置到模型中后自动清除）

## 7.5 删除数据

　　Backbone中删除模型数据的操作相对简单，我们常常用unset()和clear()方法来删除模型中的数据：

* unset()方法用于删除对象中指定的属性和数据
* clear()方法用于删除模型中所有的属性和数据

　　我们来看一个unset()方法的例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend();
4. // 实例化模型对象
5. **var** javabook = **new** Book({
6. name : 'Java7入门经典',
7. author : 'Ivor Horton',
8. price : 88.50
9. });
11. // 输出: Java7入门经典
12. console.log(javabook.get('name'));
14. // 删除对象name属性
15. javabook.unset('name');
17. // 输出: undefined
18. console.log(javabook.get('name'));
19. 当我们对模型的name属性执行unset()方法后，模型内部会使用**delete**关键字将name属性从对象中删除。
21. clear()方法与unset()方法执行过程类似，但clear()方法会删除模型中的所有数据，例如：
22. // 定义Book模型类
23. **var** Book = Backbone.Model.extend();
25. // 实例化模型对象
26. **var** javabook = **new** Book({
27. name : 'Java7入门经典',
28. author : 'Ivor Horton',
29. price : 88.50
30. });
32. // 删除对象name属性
33. javabook.clear();
35. // 以下均输出: undefined
36. console.log(javabook.get('name'));
37. console.log(javabook.get('author'));
38. console.log(javabook.get('price'));

　　在调用unset()和clear()方法清除模型数据时，会触发change事件，我们也同样可以在change事件的监听函数中通过previous()和previousAttributes()方法获取数据的上一个状态。

## 7.6 将模型数据同步到服务器

　　Backbone提供了与服务器数据的无缝连接，我们只需要操作本地Model对象，Backbone就会按照规则自动将数据同步到服务器。

　　如果需要使用Backbone默认的数据同步特性，请确定你的服务器数据接口已经支持了REST架构。在REST架构中，客户端会通过请求头中的Request Method告诉服务器我们将要进行的操作（包括create、read、update和delete，它们对应的Request Method分别为POST、GET、PUT和DELETE），而对于没有良好支持REST发送方式的浏览器，Backbone会使用另外一些方法来实现，这在本节中会详细讨论。

　　在讨论数据同步相关方法之前，你需要先了解一些Backbone中与数据同步息息相关的内容：

**数据标识：**

　　设想一下，如果我们需要通过服务器接口删除一条数据，仅仅在报文头中通过Request Method标识告诉服务器进行delete操作是不够的，更重要的是还需要告诉服务器删除哪一条数据，这需要我们传递给服务器一个数据的唯一标识（例如记录id）。

　　Backbone中每一个模型对象都有一个唯一标识，默认名称为id，你可以通过idAttribute属性来修改它的名称。

　　id应该由服务器端创建并保存在数据库中，在与服务器的每一次交互中，模型会自动在URL后面加上id，而对于客户端新建的模型，在保存时不会在URL后加上id标识（我们可以通过模型的isNew()方法来检查，该模型对象是否是由客户端新建的）。

**URL规则：**

　　Backbone默认使用PATHINFO的方式来访问服务器接口。

　　例如：我们在删除一个模型数据时，模型会在报文头的Request Method中声明delete操作，并在服务器接口后自动加上模型id，格式类似于http://urlRoot/10001，其中urlRoot是我们设置的服务器接口地址，而10001是模型id。请注意它是通过URL路径的方式自动追加到接口地址后的，因此服务器也必须要支持PATHINFO的解析方式。

　　使用PATHINFO方式，因为它更直观，更利于SEO，还可以避免与Backbone中的路由器发生混淆（关于路由器将在后面的章节中介绍）。

　　如果我们希望让Backbone自动与服务器接口进行交互，首先应该配置模型的URL，Backbone支持3种方式的URL配置：

　　第一种是urlRoot方式：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service'
4. });
6. // 创建实例
7. **var** javabook = **new** Book({
8. id : 1001,
9. name : 'Thinking in Java',
10. author : 'Bruce Eckel',
11. price : 395.70
12. });
14. // 保存数据
15. javabook.save();

　　在这个例子中，我们创建了一个Book模型的实例，并调用save()方法将数据保存到服务器。（可能你对save()方法还不太了解，但这并不重要，因为我们马上就会讲到它，现在你只需知道我们用它将模型中的数据保存到服务器）

　　你可以抓包查看请求记录，你能看到请求的接口地址为：http://localhost/service/1001

　　其中localhost是我的主机名，因为我在本地搭建了一个Web服务器环境。

　　service是该模型的接口地址，是我们在定义Book类时设置的urlRoot。

　　1001是模型的唯一标识（id），我们之前说过，模型的id应该是由服务器返回的，对应到数据库中的某一条记录，但这里为了能直观的测试，我们假设已经从服务器端拿到了数据，且它的id为1001。

　　这段内容很容易理解，接下来，我们将save()方法换成destroy()方法（该方法用于将模型中的数据从服务器删除）：

**Js代码  [收藏代码](javascript:void())**

1. // 删除数据
2. javabook.destroy();

　　你能看到请求的接口地址仍然为：http://localhost/service/1001。这并不奇怪，如果你细心观察，会发现两次请求头中的Request Method参数分别为PUT和DELETE，服务器接口会根据它来判断你所做的操作。

　　如果你的浏览器不支持REST发送方式，你可能会看到Request Method始终是POST类型，且在Form Data中会多出一个\_method参数，PUT和DELETE操作名被放在了这个\_method参数中。这是Backbone为了适配低版本浏览器而设计的另一种方法，你的服务器接口也必须同时支持这种方式。

　　我们再来看第二种URL方式的例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service',
4. url : '/javaservice'
5. });
7. // 创建实例
8. **var** javabook = **new** Book({
9. id : 1001,
10. name : 'Thinking in Java',
11. author : 'Bruce Eckel',
12. price : 395.70
13. });
15. // 保存数据
16. javabook.save();

　　在这个例子中，我们在定义Book类时，新增了参数url，执行这段代码，你会发现请求的接口地址为http://localhost/javaservice。它没有再使用urlRoot定义的参数，也没有将模型的id追加到接口地址中，urlRoot和url参数我们一般只会同时定义一个，它们的区别在于：

　　urlRoot参数表示服务器接口地址的根目录，我们无法直接访问它，只能通过连接模型id来组成一个最终的接口地址

　　url参数表示服务器的接口地址是已知的，我们无需让Backbone自动连接模型id（这可能是在url本身已经设置了模型id，或者不需要传递模型id）

　　如果同时设置了urlRoot和url参数，url参数的优先级会高于urlRoot。

　　（另一个细节是，url参数不一定是固定的字符串，也可以是一个函数，最终使用的接口地址是这个函数的返回值。）

　　最后一种URL方式的例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service',
4. url : '/javaservice'
5. });
7. // 创建实例
8. **var** javabook = **new** Book({
9. id : 1001,
10. name : 'Thinking in Java',
11. author : 'Bruce Eckel',
12. price : 395.70
13. });
15. // 保存数据
16. javabook.save(**null**, {
17. url: '/myservice'
18. });

　　在这个例子中，我们在调用save()方法的时候传递了一个配置对象，它包含一个url配置项，最终抓包看到的请求地址是http://localhost/myservice。因此你可以得知，通过调用方法时传递的url参数优先级会高于模型定义时配置的url和urlRoot参数。

　　在Backbone中，模型对象提供了3个方法用于和服务器保持数据同步：

* save()方法：在服务器创建或修改数据
* fetch()方法：从服务器获取数据
* destroy()方法：从服务器移除数据

　　下面我们将依次介绍这些方法的使用：

**save()方法：**

　　save()方法用于将模型的数据保存到服务器，它可能是一条新的数据，也可能是修改服务器现有的某一条数据，这取决于模型中是否存在id（唯一标识）。

　　首先我们来看一个创建数据的例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service'
4. });
6. // 创建实例
7. **var** javabook = **new** Book();
9. // 设置初始化数据
10. javabook.set({
11. name : 'Thinking in Java',
12. author : 'Bruce Eckel',
13. price : 395.70
14. });
16. // 从将数据保存到服务器
17. javabook.save();

　　在这个例子中，我们创建了一个新的Book实例，并设置了一些数据（实际上它们可能是由用户输入的），我们通过save()方法将这些数据提交到服务器。

　　如果你抓包看一下报文头信息，能看到Request Method参数为POST，这是因为模型内部会通过isNew()方法检查是否为客户端新建，如果是客户端新建的数据，会通过POST方式发送。如果是修改服务器现有的数据，则通过PUT方式发送。

　　如果服务器接口的报文体中没有返回任何数据，你会发现保存之后的模型较之前没有发生任何变化，在你下一次调用save()方法的时候，它仍然会以POST方式通知服务器创建一条新的数据。

　　这是因为模型对象并没有获取到刚刚服务器创建成功的记录id，因此我们希望服务器接口在将数据保存成功之后，同时将新的id返回给我们，就像这样：

**Js代码  [收藏代码](javascript:void())**

1. {
2. "id" : "1001",
3. "name" : "Thinking in Java(修订版)",
4. "author" : "Bruce Eckel",
5. "price" : "395.70"
6. }

　　这一段是服务器接口返回的数据，它除了返回新记录的id，还返回了修改后的name数据（当然，你也可以只返回新记录的id，我们常常都是这样做的）。这时我们再来看现在模型中的数据，它多了一个id属性，并且name属性的值也发生了变化，也就是说模型使用服务器返回的最新数据替换了之前的数据。

　　我们将代码稍作修改：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service'
4. });
6. // 创建实例
7. **var** javabook = **new** Book();
9. // 设置初始化数据
10. javabook.set({
11. name : 'Thinking in Java',
12. author : 'Bruce Eckel',
13. price : 395.70
14. });
16. // 将数据保存到服务器
17. javabook.save(**null**, {
18. success : **function**(model) {
19. // 数据保存成功之后, 修改price属性并重新保存
20. javabook.set({
21. price : 388.00
22. });
23. javabook.save();
24. }
25. });

　　我们修改了save()方法的调用参数，像例子中那样，你可以设置一个success回调函数用来表示保存成功之后将要进行操作（你也可以设置一个error回调函数用来表示保存失败时将要进行的操作）。

　　在数据保存成功之后，我们将修改模型的price值，并从新调用save()方法保存数据。

　　我们抓包看一下请求头，发生了一些什么变化：

　　Request Method变成了PUT。

　　请求的接口地址变成了http://localhost/service/1001（这与我们刚刚讨论的URL配置有关，如果不明白可以重新阅读本节）。

　　当然，还有提交的数据也变成了我们修改后的。

　　在调用save()方法时，我们可以传递一个配置项对象，上面我们已经使用它传递了一个success回调函数。

　　在配置项中，还可以包含一个wait配置，如果我们传递了wait配置为true，那么数据会在被提交到服务器之前进行验证，当服务器没有响应新数据（或响应失败）时，模型中的数据会被还原为修改前的状态。如果没有传递wait配置，那么无论服务器是否保存成功，模型数据均会被修改为最新的状态、或服务器返回的数据。

　　我们还是用一个例子来说明：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : 'unknown',
5. author : 'unknown',
6. price : 0
7. },
8. urlRoot : '/service'
9. });
11. // 创建实例
12. **var** javabook = **new** Book();
14. // 从将数据保存到服务器
15. javabook.save({
16. name : 'Thinking in Java',
17. author : 'Bruce Eckel',
18. price : 395.70
19. }, {
20. wait : **true**
21. });

　　请运行这个例子中的代码，并且将服务器接口返回的数据设置为空（或404状态），你能看到在save()方法调用完成之后，模型中的数据被恢复成最初defaults中定义的数据，因为我们在调用save()方法时传递了wait配置。（你也可以试着将wait配置去掉，然后再运行它，你会发现虽然服务器接口并没有返回数据或保存成功，但模型对象中仍然保持着最新的数据）

　　正如我们最开始所讲得那样，save()方法用于添加一条新的数据到服务器，或修改现有的一条数据。

　　其实save()方法也可以同时实现数据修改和保存，例如：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service'
4. });
6. // 创建实例
7. **var** javabook = **new** Book();
9. // 从将数据保存到服务器
10. javabook.save({
11. name : 'Thinking in Java',
12. author : 'Bruce Eckel',
13. price : 395.70
14. });

　　在本例中，我们在调用时将数据传递给save()方法，而不是先通过set()方法设置数据。当然，你也可以像set()方法一样，只设置某一个值：

**Js代码  [收藏代码](javascript:void())**

1. javabook.save('name', 'Thinking in Java');

　　无论你通过什么方式来保存数据，它都会自动将数据同步到服务器接口（如果你没有设置url或urlRoot参数，那么所有的操作只会在本地进行）。

　　我们来讨论另一个问题：上面提到服务器接口返回的数据会被覆盖到当前模型中，在刚刚的例子里，接口返回的数据就是模型需要的数据。而实际开发中往往并没有这么顺利，我们接口返回的数据可能是这样：

**Js代码  [收藏代码](javascript:void())**

1. {
2. "resultCode" : "0",
3. "error" : "null",
4. "data" : [{
5. "isNew" : "true",
6. "bookId" : "1001",
7. "bookName" : "Thinking in Java(修订版)",
8. "bookAuthor" : "Bruce Eckel",
9. "bookPrice" : "395.70"
10. }]
11. }

　　你能看到，接口返回的数据无论从结构、还是属性名，都与模型中定义的不一样（有时甚至会返回XML或其它格式）。还好Backbone提供了一个parse()方法，用于在将服务器返回的数据覆盖到模型前，对数据进行解析。

　　parse()方法默认不会对数据进行解析，因此我们只需要重载该方法，就可以适配上面的数据格式，例如：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service',
4. // 重载parse方法解析服务器返回的数据
5. parse : **function**(resp, xhr) {
6. **var** data = resp.data[0];
7. **return** {
8. id : data.bookId,
9. name : data.bookName,
10. author : data.bookAuthor,
11. price : data.bookPrice
12. }
13. }
14. });
16. // 创建实例
17. **var** javabook = **new** Book();
19. // 从将数据保存到服务器
20. javabook.save({
21. name : 'Thinking in Java',
22. author : 'Bruce Eckel',
23. price : 395.70
24. });

　　我们重载了parse()方法，并返回了模型中能够使用的格式，这样就可以将服务器接口返回的数据与模型中的数据连接起来。虽然本例中使用了最简单的方式解析，但实际上你可能还会做一些格式化、转换和逻辑工作。

　　另外值得注意的一点是：我们常常会在数据保存成功后，对界面做一些改变。此时你可以通过许多种方式实现，例如通过save()方法中的success回调函数。

　　但我建议success回调函数中只要做一些与业务逻辑和数据无关的、单纯的界面展现即可（就像控制加载动画的显示隐藏），如果数据保存成功之后涉及到业务逻辑或数据显示，你应该通过监听模型的change事件，并在监听函数中实现它们。虽然Backbone并没有这样的要求和约束，但这样更有利于组织你的代码。

**fetch()方法：**

　　fetch()方法用于从服务器接口获取模型的默认数据，常常用于模型的数据恢复，它的参数和原理与save()方法类似，因此你可以很容易理解它。

　　先让我们看一个例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service'
4. });
6. // 创建实例
7. **var** javabook = **new** Book();
9. // 从服务器获取默认数据
10. javabook.fetch({
11. success : **function**() {
12. // 获取数据成功后, 重新读取一次
13. javabook.fetch();
14. }
15. });

　　在这个例子中，我们创建了一个空的（没有初始化数据的）Book模型实例，然后通过fetch()方法从服务器接口获取初始化数据，获取数据成功后再次调用fetch()方法重新获取一次。

　　我们将服务器接口返回的数据设置为：

**Js代码  [收藏代码](javascript:void())**

1. {
2. "id" : "1001",
3. "name" : "Thinking in Java",
4. "author" : "Bruce Eckel",
5. "price" : "395.70"
6. }

　　你需要注意观察两次请求的URL和参数：

　　第一次请求地址为http://localhost/service，Request Method参数为GET

　　第二次请求地址为http://localhost/service/1001，Request Method参数为GET

　　你会发现第二次在请求地址后加上了1001（模型id），这是因为在第一次获取数据成功后，服务器接口返回的数据会覆盖到模型中，因此模型对象具备了唯一标识（id），因此在此后的每次请求中，模型都会将id加载请求地址后面。

**destroy()方法：**

　　destroy()方法用于将数据从集合（关于集合我们将在下一章中讨论）和服务器中删除，需要注意的是，该方法并不会清除模型本身的数据。（如果需要删除模型中的数据，请手动调用unset()或clear()方法）

　　当你的模型对象从集合和服务器端删除时，只要你不再保持任何对模型对象的引用，那么它会自动从内存中移除。（通常的做法是将引用模型对象的变量或属性设置为null值）

　　当调用destroy()方法时，模型会触发destroy事件，所有监听该事件的函数将被调用。

　　我们还是通过一个例子来详细了解它：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service'
4. });
6. // 创建实例
7. **var** javabook = **new** Book({
8. id: '1001'
9. });
11. // 从服务器删除数据
12. javabook.destroy();

　　这个例子非常简单，我们创建一个模型后再调用destroy()方法将它销毁。

　　请抓包观察请求地址和Request Method：

　　我们看到请求地址为：http://localhost/service/1001，Request Method参数为DELETE。它通过Reuqest Method请求参数通知服务器接口将要进行的操作，而请求地址和save()方法及fetch()方法产生的请求地址是相同的，这正体现了我们最开始所说的REST架构。

　　在调用destroy()方法时我们同样可以传递一个配置对象，它除了success和error回调函数外，也能像save()方法一样包含一个wait配置，来看下面的例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义Book模型类
2. **var** Book = Backbone.Model.extend({
3. urlRoot : '/service'
4. });
6. // 创建实例
7. **var** javabook = **new** Book({
8. id : '1001'
9. });
11. // 监听模型的destroy事件, 在控制台输出字符串
12. javabook.on('destroy', **function**() {
13. console.log('destroy');
14. });
16. // 从服务器删除数据
17. javabook.destroy({
18. wait : **true**
19. });

　　如果你的service服务器接口能正常访问，那么你能看到在控制台输出了“destroy”字符串；如果将你的接口设置为响应失败（例如404），那么控制台就不会有输出。

　　当我们传递了wait配置后，模型会先请求服务器接口对数据进行删除，当服务器返回状态成功（状态码200）之后，本地才会进行模型的删除操作，最终触发destroy事件。

　　如果你想通过Backbone实现数据同步，而不使用RET架构，那么你可以通过重新定义Backbone.sync方法来适配现有的服务器接口。

　　在Backbone中，所有与服务器交互的逻辑都定义在Backbone.sync方法中，该方法接收method、model和options三个参数。如果你想重新定义它，可以通过method参数得到需要进行的操作（枚举值为create、read、update和delete），通过model参数得到需要同步的数据，最后根据它们来适配你自己定义的规则即可。

　　当然，你也可以将数据同步到本地数据库中，而不是服务器接口，这在开发终端应用时会非常适用。

## 7.7 小结

　　至此，Backbone模型中的核心方法和特性我们都已经讨论完了，我们总结一下本节讨论的主要内容：

* 模型封装了对象数据，并提供了一系列对数据进行操作的方法
* 我们可以在定义模型类、实例化模型对象、和调用set()方法来设置模型中的数据
* 当模型中数据发生改变时，会触发change事件和属性事件
* 我们可以定义validate方法对模型中的数据进行验证
* 通过调用save()、fetch()和destroy()方法可以让模型中的数据与服务器保持同步，但在此之前必须设置url或urlRoot属性

　　当然，模型类还包含一些实用的方法帮助我们开发，这里就不一一介绍，通过API文档你能轻易地理解它们。

8. Collection (数据模型集合)

　　如果将一个Model对象比喻成数据库中的一条记录，那么Collection就是一张数据表。它表示为一个模型集合类，用于存储和管理一系列相同类型的模型对象。

## 8.1 创建集合

　　集合用于组织和管理多个模型，但它并不是必须的，如果你的某个模型对象是唯一的（单例），那么你没必要将它放到集合中。

　　我们来看一个创建集合的例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : ''
5. }
6. });
8. // 定义集合类
9. **var** BookList = Backbone.Collection.extend({
10. model : Book
11. });
13. // 创建一系列模型对象
14. **var** book1 = **new** Book({
15. name : 'Effective Java中文版(第2版)'
16. });
17. **var** book2 = **new** Book({
18. name : 'JAVA核心技术卷II：高级特性（原书第8版）'
19. });
20. **var** book3 = **new** Book({
21. name : '精通Hibernate：Java对象持久化技术详解（第2版）'
22. });
24. // 创建集合对象
25. **var** books = **new** BookList([book1, book2, book3]);

　　在这个例子中，我们定义了模型类Book和集合类BookList，然后创建了3个模型对象，并将它们放到一个集合对象中。（你可以在控制台输出books.models属性，用来查看集合中包含的模型对象列表）

　　我们为了创建3个Book模型对象，对Book类显式实例化了3次，其实Model本身已经提供了更简单的方法来复制一个模型，例如：

**Js代码  [收藏代码](javascript:void())**

1. **var** book1 = **new** Book({
2. name : 'Effective Java中文版(第2版)'
3. });
5. **var** book2 = book1.clone();
6. book2.set('name', 'JAVA核心技术卷II：高级特性（原书第8版）');
8. **var** book3 = book1.clone();
9. book3.set('name', '精通Hibernate：Java对象持久化技术详解（第2版）');

　　在段代码中，我们使用模型的clone()方法来复制一个和当前对象相同（包括数据）的新对象，这可以简化我们创建模型的流程。

　　在实例化集合对象时，除了可以向构造函数中添加已经创建好的模型列表（就像上面的例子那样），我们还可以直接传递模型数据，集合会自动将这些数据转换为模型对象，例如：

**Js代码  [收藏代码](javascript:void())**

1. // 定义模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : ''
5. }
6. });
8. // 定义集合类
9. **var** BookList = Backbone.Collection.extend({
10. model : Book
11. });
13. **var** models = [{
14. name : 'Effective Java中文版(第2版)'
15. }, {
16. name : 'JAVA核心技术卷II：高级特性（原书第8版）'
17. }, {
18. name : '精通Hibernate：Java对象持久化技术详解（第2版）'
19. }];
21. // 创建集合对象
22. **var** books = **new** BookList(models);

　　运行这个例子，并在控制台输出books.models属性，你可以看到集合中存储的是Book模型类的实例，而并非我们在models数组中声明的原始数据。

　　这是因为我们在声明BookList集合类的时候，就已经设置了model属性，该属性指向集合中存储的模型对象的构造函数，当我们传递原始数据时，集合会自动创建model中定义的模型类，并将原始数据传递给它。

　　（如果你在定义集合类的时候没有设置model，那么集合会默认将原始数据转换为Backbone.Model类的实例）

　　我们之所以要使用extend来继承Backbone.Collection类，是因为我们希望定义一个自己的集合类，并向其中扩展更多的自定义方法。如果你的集合类仅仅是用于简单地存储和管理模型对象，且Backbone.Collection类所提供的方法已经可以满足你的要求，那么你可以直接实例化一个Backbone.Collection，同时也可以像上面一样实现原始数据和模型对象的自动转换，例如：

**Js代码  [收藏代码](javascript:void())**

1. // 定义模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : ''
5. }
6. });
8. **var** models = [{
9. name : 'Effective Java中文版(第2版)'
10. }, {
11. name : 'JAVA核心技术卷II：高级特性（原书第8版）'
12. }, {
13. name : '精通Hibernate：Java对象持久化技术详解（第2版）'
14. }];
16. // 创建集合对象
17. **var** books = **new** Backbone.Collection(models, {
18. model : Book
19. });

　　在本例中，我们没有通过extend定义自己的集合类，而是直接实例化Collection类。我们依然传入了原始数据，但同时我们在构造函数的第2个参数（配置对象）中设置了model属性，Collection通过它就能知道要将原始数据转换为哪个模型类的实例。

## 8.2 向集合中添加模型

　　集合提供了3个方法允许我们动态地向集合中动态插入模型：

* add()：向集合中的指定位置插入模型，如果没有指定位置，默认追加到集合尾部
* push()：将模型追加到集合尾部（与add方法的实现相同）
* unshift()：将模型插入到集合头部

　　这些方法很容易理解，但我们还是通过一个例子来说明：

**Js代码  [收藏代码](javascript:void())**

1. // 定义模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : '',
5. price : 0
6. }
7. });
9. // 创建集合对象
10. **var** books = **new** Backbone.Collection(**null**, {
11. model : Book
12. });
14. books.add({
15. name : '构建高性能Web站点',
16. price : 56.30
17. });
19. books.push({
20. name : '深入分析Java Web技术内幕',
21. price : 51.80
22. });
24. books.unshift({
25. name : '编写高质量代码:Web前端开发修炼之道',
26. price : 36.80
27. });
29. books.push({
30. name : '基于MVC的JavaScript Web富应用开发',
31. price : 42.50
32. }, {
33. at : 1
34. });
36. books.unshift({
37. name : 'RESTful Web Services Cookbook中文版',
38. price : 44.30
40. }, {
41. at : 2
42. });
44. // 在控制台输出集合中的模型列表
45. console.dir(books.models);

　　在例子中，我们通过3个方法向集合中添加了多个模型，最后，我们在控制台输出了集合中的模型列表。请仔细观察和分析我们调用的方法，以及最终列表中模型的排列顺序：

　　这些方法的作用和上面介绍的一样，用于将模型添加到集合中不同的位置，但当我们设置了at配置之后，它们就变得完全一样了，因为它们会忽略自身的规则，将模型插入到at所指向的位置。

　　当数据被成功添加到集合中时，集合会触发add事件，执行所有监听add事件的方法。除非我们在调用add()方法时设置了silent配置项，则会忽略事件的触发。

## 8.3 操作集合中的模型

　　在Underscore中，提供了许多对对象和数组集合进行操作的方法，这些方法已经被Backbone添加到Collection类的原型中。这意味着你可以使用Underscore中的集合方法来操作Collection集合中的数据，例如each()、map()、find()等方法。

　　但有一些方法我们还是要单独介绍它们，因为Collection对这些方法进行了重写，它们和Underscore中的同名方法不完全相同。（这也是为什么我要在上面单独介绍模型的添加方法）

**删除模型：**

　　集合类提供了3个方法用于从集合中移除模型对象，分别是：

* remove()：从集合中移除一个或多个指定的模型对象
* pop()：移除集合尾部的一个模型对象
* shift()：移除集合头部的一个模型对象

　　这些方法与添加的方法是对应的，而且当模型被移除成功后，会触发集合对象的remove事件，除非你在移除时使用了silent配置。

　　这些方法很容易理解，但还是让我们通过一个简单的例子来说明：

**Js代码  [收藏代码](javascript:void())**

1. // 定义模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : '',
5. price : 0
6. }
7. });
9. // 定义初始化数据
10. **var** data = [{
11. name : '构建高性能Web站点',
12. price : 56.30
13. }, {
14. name : '深入分析Java Web技术内幕',
15. price : 51.80
16. }, {
17. name : '编写高质量代码:Web前端开发修炼之道',
18. price : 36.80
19. }, {
20. name : '基于MVC的JavaScript Web富应用开发',
21. price : 42.50
22. }, {
23. name : 'RESTful Web Services Cookbook中文版',
24. price : 44.30
26. }]
28. // 创建集合对象
29. **var** books = **new** Backbone.Collection(data, {
30. model : Book
31. });
33. books.remove(books.models[2]);
34. books.pop();
35. books.shift();
37. // 在控制台输出集合中的模型列表
38. console.dir(books.models);

　　在本例中，我们分别调用了remove()方法移除了集合中第2个模型，调用pop()方法移除了最后一个模型，调用shift()方法移除了第一个模型。最后我们在控制台输出集合中剩下的模型列表，请查看控制台输出结果，它和你想象的结果是一致的。

**在集合中查找模型：**

Collection定义了一系列用于快速从集合中查找我们想要的模型的方法，包括：

* get()：根据模型的唯一标识（id）查找模型对象
* getByCid()：根据模型的cid查找模型对象
* at()：查找集合中指定位置的模型对象
* where()：根据数据对集合的模型进行筛选

 　　前面前面一片介绍数据模型时我们提到，每个模型对象都有一个唯一标识（id），它与数据库中记录的id保持同步。实际上，每个模型对象内部还会自动创建一个cid，它用来标识每一个模型（请注意将id和cid区分开，它们没有任何关系）。

　　集合对象提供了两个方法用于根据id和cid来查找模型对象，分别是get()方法和getByCid()方法，例如：

**Js代码  [收藏代码](javascript:void())**

1. // 定义模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : '',
5. price : 0
6. }
7. });
9. // 定义初始化数据
10. **var** data = [{
11. id : 1001,
12. name : '构建高性能Web站点',
13. price : 56.30
14. }, {
15. id : 1002,
16. name : '深入分析Java Web技术内幕',
17. price : 51.80
18. }, {
19. id : 1003,
20. name : '编写高质量代码:Web前端开发修炼之道',
21. price : 36.80
22. }, {
23. id : 1004,
24. name : '基于MVC的JavaScript Web富应用开发',
25. price : 42.50
26. }, {
27. id : 1005,
28. name : 'RESTful Web Services Cookbook中文版',
29. price : 44.30
30. }]
32. // 创建集合对象
33. **var** books = **new** Backbone.Collection(data, {
34. model : Book
35. });
37. // 根据id和cid查找模型对象
38. **var** book1 = books.get(1001);
39. **var** book2 = books.getByCid('c2');
41. // 在控制台输出模型
42. console.dir(book1);
43. console.dir(book2);

　　本例中，我们从集合中根据模型的id和cid查找出2个模型对象，但实际开发中我们不会直接在代码中写出模型的id和cid。

* id应该是从服务器接口进行同步获取到的。
* cid应该是在之前已经记录下某个模型的cid，再根据它从集合中查找的。

　　at()方法根据我们给定的索引，从集合中查找对应位置的模型，我们在上面的例子中追加以下代码：

**Js代码  [收藏代码](javascript:void())**

1. // 根据索引查找模型对象
2. **var** book3 = books.at(1);
4. // 在控制台输出模型
5. console.dir(book3);
7. 最后，我们还可以通过where()方法，实现相对复杂的查找规则，例如：
8. // 根据price从集合中查找模型
9. **var** book4 = books.where({
10. price : 51.80
11. });
13. // 在控制台输出模型
14. console.dir(book4);

　　请查看控制台输出的结果：where()方法用于给定一个或多个数据，查找并返回集合中匹配数据的模型。该方法返回一个数组，因此能够包含一个或多个结果。

　　当我们调用get()、getByCid()和at()方法没有找到到匹配对象时，会返回undefined，而where()方法在没有找到匹配对象时会返回一个空数组。你可以使用Underscore中的isEmpty()方法检查返回值是否为空，因为它能检查到空数组和空对象。

## 8.4 自动排序

　　我们常常使用数组的sort()方法对元素进行排序，Underscore也提供了sortBy()方法实现更为复杂的集合排序。但在Backbone的集合对象中，为我们提供了集合元素的实时排序，当任何模型对象被插入到集合中时，都会按照预定的排序规则放到对应的位置。

**Js代码  [收藏代码](javascript:void())**

1. // 定义模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : '',
5. price : 0
6. }
7. });
9. // 创建集合对象
10. **var** books = **new** Backbone.Collection(**null**, {
11. model : Book,
12. comparator : **function**(m1, m2) {
13. **var** price1 = m1.get('price');
14. **var** price2 = m2.get('price');
16. **if**(price1 > price2) {
17. **return** 1;
18. } **else** {
19. **return** 0;
20. }
21. }
22. });
24. books.add({
25. name : '构建高性能Web站点',
26. price : 56.30
27. });
29. books.push({
30. name : '深入分析Java Web技术内幕',
31. price : 51.80
32. });
34. books.unshift({
35. name : '编写高质量代码:Web前端开发修炼之道',
36. price : 36.80
37. });
39. books.push({
40. name : '基于MVC的JavaScript Web富应用开发',
41. price : 42.50
42. }, {
43. at : 1
44. });
46. books.unshift({
47. name : 'RESTful Web Services Cookbook中文版',
48. price : 44.30
50. }, {
51. at : 2
52. });
54. // 在控制台输出集合中的模型列表
55. console.dir(books.models);

　　这个例子和我们前面介绍添加方法时的例子相同，但集合中存储的模型顺序却不一样，因为我们在创建集合对象时设置了comparator方法。我们不需要手动调用该方法，因为它会在新模型被添加到集合中时自动被调用，并按照方法中定义的规则对集合中的数据进行重新排序。

　　comparator方法接收两个参数，表示临近的两个模型对象，你需要通过返回值表示它们的排序规则，这和JavaScript中原生的sort()方法是一样的。

　　当我们设置了comparator方法后，所有关于元素位置的方法和参数都会失效，例如push()、unshift()方法和at参数等。

　　需要注意的是：comparator方法在很多时候都是非常有用的（例如显示动态数据列表时），因为它能保证我们获取到的数据始终都是按规则排列的，但在集合中的数据量太多时，它可能会耗费很多的资源和事件来实时确保数据的排序规则。这时，你可以手动调用集合对象的sort()方法在需要的进行手动排序。

## 8.5 从服务器获取集合数据

* Collection也提供了两个与服务器进行交互的方法：
* fetch()：用于从服务器接口获取集合的初始化数据，覆盖或追加到集合列表中
* create()：在集合中创建一个新的模型，并将其同步到服务器

　　我们先来看一个fetch()方法例子：

**Js代码  [收藏代码](javascript:void())**

1. // 定义模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : '',
5. price : 0
6. }
7. });
9. // 定义集合类
10. **var** BookList = Backbone.Collection.extend({
11. model : Book,
12. url : '/service'
13. });
15. // 创建集合对象, 并从服务器同步初始化数据
16. **var** books = **new** BookList();
17. books.fetch({
18. success: **function**(collection, resp) {
19. // 同步成功后在控制台输出集合中的模型列表
20. console.dir(collection.models);
21. }
22. });

　　我们前面讨论过模型的数据同步，需要与服务器同步数据，需要设置一个url或urlPath指定服务器接口地址。同步集合数据时也不例外，本例中我们设置服务器接口地址为/service，接口返回数据为：

**Js代码  [收藏代码](javascript:void())**

1. [{
2. "id" : "1001",
3. "name" : "构建高性能Web站点",
4. "price" : "56.30"
5. }, {
6. "id" : "1002",
7. "name" : "深入分析Java Web技术内幕",
8. "price" : "51.80"
9. }, {
10. "id" : "1003",
11. "name" : "编写高质量代码:Web前端开发修炼之道",
12. "price" : "36.80"
13. }, {
14. "id" : "1004",
15. "name" : "基于MVC的JavaScript Web富应用开发",
16. "price" : "42.50"
17. }, {
18. "id" : "1005",
19. "name" : "RESTful Web Services Cookbook中文版",
20. "price" : "44.30"
21. }]

　　我们在实例化集合对象之后，调用fetch()与服务器接口进行数据同步，并在同步成功后将集合中的数据列表输出在控制台。请运行例子中的代码，你能看到控制台中输出的结果，它包含5个模型对象，正是我们服务器接口返回的这些数据。

　　在调用fetch()方法同步集合数据时，默认会以覆盖的方式进行，这意味着集合在同步之前的数据将丢失。我们可以在调用fetch()方法时传递add参数来通知集合进行添加，而不是覆盖，例如：

**Js代码  [收藏代码](javascript:void())**

1. **var** books = **new** BookList();
2. books.add({
3. id : 1000,
4. name : 'Thinking in Java',
5. price : 395.70
6. });
8. books.fetch({
9. add: **true**,
10. success : **function**(collection, resp) {
11. // 同步成功后在控制台输出集合中的模型列表
12. console.dir(collection.models);
13. }
14. });

　　我们修改了例子中的代码，在实例化集合对象后，我们向集合中添加了一条数据，然后从服务器同步了5条数据（请注意在调用fetch()方法时我们设置了add参数为true）。在控制台输出的结果中，之前的数据并没有被覆盖掉。

　　数据在成功同步到集合中之后，会触发reset事件，我们可以通过监听该事件从而进行下一步操作（比如将集合中的数据显示到页面中）。

　　集合的数据同步与模型的数据同步有许多相似之处（例如你可以重载parse()方法来对服务器返回的数据进行解析，使其能顺利被添加到集合中），这里就不再重复讨论。

　　集合提供的另一个create()方法，是根据集合的model所指向的模型类，创建一个模型对象，并把该对象添加到集合中，最后将数据同步到服务器接口。

　　我们通过例子来说明create()方法的使用：

**Js代码  [收藏代码](javascript:void())**

1. **var** books = **new** BookList();
2. // 创建一个模型
3. books.create({
4. name : 'Thinking in Java',
5. price : 395.70
6. }, {
7. success : **function**(model, resp) {
8. // 添加成功后, 在控制台输出集合中的模型列表
9. console.dir(books.models);
10. }
11. });

　　请将这段代码替换到前面的例子中，并查看运行效果。

　　（通过抓包我们能看到Request Method为POST，如果创建的模型中包含id，则Request 　　Method为PUT，这与我们之前讲模型的save()方法是相同的。）

　　集合对象默认会先将模型添加到集合中，再提交到服务器接口，无论接口返回是否成功，新建的模型对象都会被添加到集合中。我们可以通过传递wait配置，来控制只有在服务器返回成功之后（响应状态码为200），才将模型对象添加到集合中。

　　在Backbone内部，create()方法是通过add()方法将新创建的模型添加到集合中的，因此我们一般通过监听add事件，来对新模型进行下一步操作。

## 8.6 将数据批量同步到服务器

　　上一节我们讨论过，Backbone中集合提供了数据同步和创建的方法与服务器进行交互，但实际上这可能并不能满足我们的需求。例如：当我们需要对数据进行批量地添加、修改和删除操作时，就需要在Collection的基础上扩展自己的方法。

　　在下面的例子中，我扩展了对集合中的模型数据批量同步的方法：

**Js代码  [收藏代码](javascript:void())**

1. // 定义模型类
2. **var** Book = Backbone.Model.extend({
3. defaults : {
4. name : '',
5. price : 0
6. }
7. });
9. // 定义BookList类
10. **var** BookList = Backbone.Collection.extend({
11. model : Book,
12. url : '/service',
13. // 将集合中所有的模型id连接为一个字符串并返回
14. getIds : **function**() {
15. **return** \_(**this**.models).map(**function**(model) {
16. **return** model.id;
17. }).join(',');
18. },
19. // 将集合中所有模型提交到服务器接口
20. createAll : **function**(options) {
21. **return** Backbone.sync.call(**this**, 'create', **this**, options);
22. },
23. // 修改集合中的所有模型数据
24. updateAll : **function**(options) {
25. **return** Backbone.sync.call(**this**, 'update', **this**, options);
26. },
27. // 删除集合中所有的模型
28. deleteAll : **function**(options) {
29. **var** result = Backbone.sync.call(**this**, 'delete', **this**, \_.extend({
30. url : **this**.url + '/' + **this**.getIds()
31. }, options));
32. **this**.remove(**this**.models);
33. **return** result;
34. }
35. });
37. // 创建集合对象
38. **var** books = **new** BookList();
40. // 当集合触发reset事件时, 对数据进行批量同步
41. books.on('reset', **function**() {
42. books.createAll();
43. books.updateAll();
44. books.deleteAll();
45. });
47. // 从服务器接口同步默认数据
48. books.fetch();

　　来分析这个例子：

　　我们定义了BookList集合类，并扩展了createAll()、updateAll()和deleteAll()方法（稍后我们再讨论这3些方法的作用）。

　　然后我们实例化了一个BookList对象books，并监听了reset事件，reset事件会在从服务器同步数据成功之后被触发。

　　接着我们调用fetch()方法从服务器接口获取默认数据（默认数据跟前面例子中返回的数据一致），获取成功后，reset事件的监听函数将被执行。

　　我们可以在reset事件的监听函数中做许多的事情（例如写入业务逻辑），但这里为了更直观地演示，我直接调用books的自定义方法进行批量同步。

　　首先调用的是createAll()方法，它将把当前集合中的所有数据同步到服务器接口，以Request 　　Method为POST方式告诉服务器接口创建并保存这些数据。

　　在createAll()方法中，我们调用Backbone.sync()方法发送异步请求，请注意sync()方法的第2个参数，它是一个模型或集合对象，当操作为create或update时，在sync()方法内部会调用该对象的toJSON()方法，并将toJSON()方法的返回值作为Request 　　Payload请求数据发送到服务器接口。

　　你可以通过抓包并分析请求信息，来更好地理解它。

　　（toJSON()方法默认会返回模型或集合的数据对象，你可以通过重载该方法来自定义需要发送的请求数据）

　　我们假设集合中的数据已经被用户批量修改过，所以我们通过updateAll()方法将最新的数据提交到服务器接口。

　　updateAll()方法与createAll()方法几乎相同，它们的区别在于updateAll()方法在修改数据时传递给sync()方法的操作名为update而不是create，而发送给服务器的Request Method为PUT而不是POST。

　　最后我们通过deleteAll()方法，通知服务器删除集合中的模型，并从本地集合对象中删除这些数据。

　　deleteAll()方法与createAll()和updateAll()方法有些不同，因为deleteAll()方法发送给服务器的Request 　　Method为DELETE方式，这种方式下不能直接调用toJSON()方法将数据发送给接口，因此我们需要手动组装和发送数据。

　　在本例中，我们定义了getIds()方法用于将集合中的所有模型的id连接起来，在deleteAll()方法中，我们调用sync()方法发送了Request Method为DELETE的请求，并在URL中将集合中的所有模型id传递给接口进行删除。

# 9. View (视图控制器)

　　我们在前面的章节中介绍了Model和Collection，它们都是用于数据管理和交互，在本章我们讨论如何使用视图（View）将这些数据渲染到界面，以及如何管理界面事件和逻辑。

　　Backbone中的视图提供了一组处理DOM事件、和渲染模型（或集合）数据的方法（在使用视图类之前，你必须先导入jQuery或Zepto）。

　　视图类提供的方法非常简单，我们一般都会在Backbone.View的基础上进行扩展，但即使是特别复杂的视图类，它也应该仅仅是做界面事件、和渲染逻辑相关的操作，数据管理应该交由Model和Collection来完成，而业务逻辑应该由其它的类来完成。

## 9.1 定义和创建视图

　　我们通过一段例子来说明如何创建View：

**Js代码  [收藏代码](javascript:void())**

1. <div title="列表" style="color:red" id="list" **class**="listview"></div>
2. <script type="text/javascript">
3. **var** ListView = Backbone.View.extend({
4. el : '#list'
5. });
6. **var** listview = **new** ListView();
7. </script>

　　在这个例子中，我们定义了一个ListView视图类，它继承自Backbone.View，我们通过new关键字实例化一个ListView对象。

　　在定义ListView时，我们设置了el属性，它应该是一个字符串形式的DOM选择器，但视图对象在实例化时，会在内部通过这个选择器获取对应的DOM对象，并重新存放在el属性中。因此我们可以在视图的内部通过this.el来访问所关联的DOM对象。

　　每个视图对象都会关联一个DOM 对象，视图中所有操作都限定在这个DOM对象之内，这样做可以便于视图界面的控制（如渲染、隐藏和移除等），同时能提高查找视图内子元素的效率。

　　上面的例子中，id为list的标签是我们事先准备好的，在定义ListView时可以直接通过#list选择器来引用它，但实际开发时这些DOM可能是动态生成的，至少在定义视图类时它们还不存在。此时，我们可以通过另一种方式来设置视图的DOM对象：

**Js代码  [收藏代码](javascript:void())**

1. <script type="text/javascript">
2. **var** ListView = Backbone.View.extend({
3. tagName : 'div',
4. className : 'listview',
5. id : 'list',
6. attributes : {
7. title : '列表',
8. style : 'color:red'
9. },
10. render : **function**() {
11. **this**.el.innerHTML = 'Hello World!';
12. document.body.appendChild(**this**.el);
13. }
14. });
15. **var** listview = **new** ListView();
16. listview.render();
17. </script>

　　运行这个例子，页面上输出了一段红色的文字：Hello World!。如果你通过Firebug等工具查看当前的DOM结构，你能看到</body>结束标签之前多了这样一段：

**Html代码  [收藏代码](javascript:void())**

1. **<div** title="列表" style="color:red" id="list" class="listview"**>**Hello World!**</div>**

　　这段标签是视图对象在实例化时根据tagName、className、id和attributes属性自动创建的，结合我们的代码，你可以很清晰地看出：

* tagName表示新标签的名称（如果没有设置，则默认为div标签）
* className对应标签的class样式属性
* id对应标签的id属性

　　这3个是最常见的HTML属性，你可以在定义视图类时直接设置它们，如果还需要设置更多的属性，可以通过attributes属性来定义。

　　你可能注意到，我们还定义了一个render()方法，在创建ListView实例之后，我们调用该方法将新建的标签添加到页面尾部，否则它只会存储在el属性中，而不会被显示。

　　上面介绍的两种创建方式，是在两种不同的场景中使用，分别是对已经存在的DOM创建视图，和创建视图时同时创建新的DOM，因此你不应该不会同时使用到它们。

## 9.2 处理DOM事件

　　视图很重要的一个特性是帮助我们自动绑定界面事件。回想一下我们以前是如何为界面标签绑定事件的？可能就像这样：

**Js代码  [收藏代码](javascript:void())**

1. <p>
2. <input type="button" value="Create" id="create" />
3. <input type="button" value="Read" id="read" />
4. <input type="button" value="Update" id="update" />
5. <input type="button" value="Delete" id="delete" />
6. </p>
7. <script type="text/javascript">
8. **function** createData() {
9. // todo
10. }
11. **function** readData() {
12. // todo
13. }
14. **function** updateData() {
15. // todo
16. }
17. **function** deleteData() {
18. // todo
19. }
21. $('#create').on('click', createData);
22. $('#read').on('click', readData);
23. $('#update').on('click', updateData);
24. $('#delete').on('click', deleteData);
25. </script>

　　这是一个典型的通过jQuery绑定DOM事件的例子，如果你正在开发或曾经开发过一些复杂的应用，你可能尝试过通过某种方式将这些代码更好的组织起来，以便使它们看起来结构更加清晰，更易维护。

　　Backbone的视图对象为我们提供了事件的自动绑定机制，用于更好地维护DOM和事件间的关系，来看看下面的例子：

**Js代码  [收藏代码](javascript:void())**

1. <p id="view">
2. <input type="button" value="Create" id="create" />
3. <input type="button" value="Read" id="read" />
4. <input type="button" value="Update" id="update" />
5. <input type="button" value="Delete" id="delete" />
6. </p>
7. <script type="text/javascript">
8. **var** MyView = Backbone.View.extend({
9. el : '#view',
10. events : {
11. 'click #create' : 'createData',
12. 'click #read' : 'readData',
13. 'click #update' : 'updateData',
14. 'click #delete' : 'deleteData'
15. },
16. createData : **function**() {
17. // todo
18. },
19. readData : **function**() {
20. // todo
21. },
22. updateData : **function**() {
23. // todo
24. },
25. deleteData : **function**() {
26. // todo
27. }
28. });
29. **var** view = **new** MyView();
30. </script>

　　在这个例子中，我们将4个按钮放在一个id为view的标签中，并将这个标签与视图类MyView进行了关联。

　　在定义视图类时，我们声明了一个events属性，它表示视图中的用户事件列表，描述方式如下：

　　事件名称 选择器 : 事件处理函数

　　事件名称可以是DOM对象支持的任何事件，选择器可以是jQuery或Zepto支持的任意选择器字符串（包括标签选择器、类选择器、id选择器等），而事件处理函数应该是已经定义在视图类本身的方法名称。

　　视图对象会自动解析events列表中的描述，即使用jQuery或Zepto获取选择器描述的DOM对象，并将事件处理函数绑定到事件名称中。这些操作都会在视图类被实例化时自动完成，我们可以更关心视图类本身的结构，而不是刻意地去考虑如何绑定事件。

　　你可能在担心另外一个问题：如果视图的DOM结构是动态生成的，Backbone是否提供了相应的方法用于动态绑定和解除事件？

　　其实你并不需要关心这个问题，因为events中的事件是通过delegate()方法绑定到视图对象的el元素上，而并非是选择器所描述的元素。因此视图内的结构无论如何变化，events中的事件都是有效的。

　　（如果你对jQuery比较熟悉，可能了解它所提供的delegate()方法。该方法实际上将事件绑定在父层元素，然后在事件冒泡过程中，通过检查目标子元素来触发事件。）

　　视图对象通过delegate()方法绑定事件，意味着我们不需要关心视图结构变化对事件产生的影响，同时也说明events中选择器所对应的元素必须处于视图的el元素之内，否则绑定的事件是无法生效的。

　　尽管如此，有些情况下可能我们仍然需要手动绑定和解除事件，视图对象提供了delegateEvents()和undelegateEvents()方法用于动态绑定和解除events事件列表，你可以通过查看API文档来了解它们。

## 9.3 渲染视图和数据

　　我们之前提到，视图主要用于界面事件的绑定和数据渲染，然而视图对象仅仅提供了一个和渲染相关的方法render()，并且它是一个没有任何逻辑、也没有任何地方引用到的空方法，我们需要重载它来实现自己的渲染逻辑。

　　视图中可能包含许多界面逻辑，这里建议所有的视图子类都重载render()方法，并将它作为最终渲染的入口方法。在团队开发中，严格按照规范编码可以帮助别人更好地理解和维护你的代码。

# 10. Router和History (路由控制)

　　Backbone.Router担任了一部分Controller（控制器）的工作，它一般运行在单页应用中，能将特定的URL或锚点规则绑定到一个指定的方法（后文中称Action）。

　　当我们开发一个单页应用时，常常会遇到这样两个问题：

　　我们在同一个页面中通过用户的操作来隐藏、显示HTML块，为用户提供一个无刷新、完整流畅的体验，但用户可能并不知道他当前正处于同一个页面中，因此他希望通过浏览器的“前进”和“后退”按钮来返回和前进到上一步操作。当他真正这样操作时，会离开当前页面，这显然不是用户所期望的。

　　另一个问题是用户在单页应用中操作，当他读到一篇好的文章，或看到一个中意的商品时，他可能会将URL收藏起来或分享给自己的好友。但当他下一次重新打开这个链接地址，看到的却是应用的初始化状态，而并不是当初那篇文章或那个商品。

　　Backbone.Router为我们提供了解决这两个问题的方法，我们先来看一个例子：

**Js代码  [收藏代码](javascript:void())**

1. **var** AppRouter = Backbone.Router.extend({
2. routes : {
3. '' : 'main',
4. 'topic' : 'renderList',
5. 'topic/:id' : 'renderDetail',
6. '\*error' : 'renderError'
7. },
8. main : **function**() {
9. console.log('应用入口方法');
10. },
11. renderList : **function**() {
12. console.log('渲染列表方法');
13. },
14. renderDetail : **function**(id) {
15. console.log('渲染详情方法, id为: ' + id);
16. },
17. renderError : **function**(error) {
18. console.log('URL错误, 错误信息: ' + error);
19. }
20. });
22. **var** router = **new** AppRouter();
23. Backbone.history.start();

　　将例子中的代码复制到你的页面中。假设你的页面地址为http://localhost/index.html，请依次访问下面的地址，并注意控制台的输出结果：

* http://localhost/index.html // 输出：应用入口方法
* http://localhost/index.html#topic // 输出：渲染列表方法
* http://localhost/index.html#topic/1023 // 输出：渲染详情方法, id为:1023
* http://localhost/index.html#about // 输出：URL错误, 错误信息: about

　　然后再使用浏览器的“前进”、“返回”等按钮进行切换，你会看到当你的URL切换时，控制台输出了对应的结果，说明它已经调用了相应的方法。而在进行这些操时，页面并没有刷新。这个例子很好地解决了我们在一开始所说的两个问题。

## 10.1 路由规则设定

　　你可能经常听说“路由器”这个词，但它常常是指一种网络设备，这种设备是网络连接、数据传输的导航和枢纽。而Backbone中的“路由器”功能与它类似，从上面的例子中你就能看出，它可以将不同的URL锚点导航到对应的Action方法。

　　（许多服务端Web框架中也提供了这样的机制，但Backbone.Router更侧重前端单页应用的导航。）

　　Backbone的路由导航是由Backbone.Router和Backbone.History两个类共同完成的：

Router类用于定义和解析路由规则，并将URL映射到Action。

History类用于监听URL的变化，和触发Action方法。

　　我们一般不会直接实例化一个History，因为我们在第一次创建Router实例时，会自动创建一个History的单例对象，你可以通过Backbone.history来访问这个对象。

　　要使用路由功能，首先我们需要定义一个Router类来声明需要监听的URL规则和Action，在刚才的例子中，我们在定义时通过routes属性来定义需要监听的URL列表，其中Key表示URL规则，Value表示当URL处于该规则时所执行的Action方法。

## 10.2 Hash规则

　　URL规则表示当前URL中的Hash（锚点）片段，我们除了能在规则中指定一般的字符串外，还需要注意两种特别的动态规则：

　　规则中以/（斜线）为分隔的一段字符串，在Router类内部会被转换为表达式([^\/]+)，表示以/（斜线）开头的多个字符，如果在这一段规则中设置了:（冒号），则表示URL中这一段字符串将被作为参数传递给Action。

　　例如我们设置了规则topic/:id，当锚点为#topic/1023时，1023将被作为参数id传递给Action，规则中的参数名（:id）一般会和Action方法的形参名称相同，虽然Router并没有这样的限制，但使用相同的参数名更容易让人理解。

　　规则中的\*（星号）会在Router内部被转换为表达式(.\*?)，表示零个或多个任意字符，与:（冒号）规则相比，\*（星号）没有/（斜线）分隔的限制，就像我们在上面的例子中定义的\*error规则一样。

　　Router中的\*（星号）规则在被转换为正则表达式后使用非贪婪模式，因此你可以使用例如这样的组合规则：\*type/:id，它能匹配#hot/1023，同时会将hot和1023作为参数传递给Action方法。

　　上面介绍了规则的定义方式，这些规则都会对应一个Action方法名称，该方法必须处于Router对象中。

　　在定义好Router类之后，我们需要实例化一个Router对象，并调用Backbone.history对象的start()方法，该方法会启动对URL的监听。在History对象内部，默认会通过onhashchange事件监听URL中Hash（锚点）的变化，对于不支持onhashchange事件的浏览器（例如IE6），History会通过setInterval心跳的方式监听。

## 10.3 pushState规则

　　Backbone.History还支持pushState方式的URL，pushState是HTML5提供的一种新特性，它能操作当前浏览器的URL（而不是仅仅改变锚点），同时不会导致页面刷新，从而使单页应用使用起来更像一套完整的流程。

　　要使用pushState特性，你需要先了解HTML5为该特性提供的一些方法和事件（这些方法都被定义在window.history对象中）：

* pushState()：该方法可以将指定的URL添加一个新的history实体到浏览器历史里
* replaceState()：该方法可以将当前的history实体替换为指定的URL

　　调用pushState()和replaceState()方法，仅仅是替换当前页面的URL，而并不会真正转到这个URL地址（当使用后退或前进按钮时，也不会跳转到该URL），我们可以通过onpopstate事件来监听这两个方法引起的URL变化。

　　值得注意的是，使用pushState特性时，需要在服务器端对URL规则进行配置，否则用户在下一次访问该地址时，可能无法正确导航到当前页面。另外低于以下版本的浏览器是不支持该特性的：Chrome 5，Firefox 4.0，IE 10，Opera 11.5，Safari 5.0

　　关于pushState规则这里不再深入讨论，如果需要了解，请参考我所注释的Backbone源码。

## 10.4 路由相关方法

**route()方法**

　　在设定好路由规则之后，如果需要动态调整，可以调用Router.route()方法来动态添加路由规则及Action方法，例如：

**Js代码  [收藏代码](javascript:void())**

1. router.route('topic/:pageno/:pagesize', 'page', **function**(pageno, pagesize){
2. // todo
3. });

　　我们调用route()方法时，给定的规则不仅仅可以是字符串，也可以是一个正则表达式：

**Js代码  [收藏代码](javascript:void())**

1. router.route(/^topic/(.\*?)/(.\*?)$/, 'page', **function**(pageno, pagesize){
2. // todo
3. });

**navigate()方法**

　　在前面的例子中，URL规则都是由我们手动输入触发的，在实际应用中，有时可能需要手动进行跳转、导航，这时可以调用Router.navigate()方法进行控制，例如：

**Js代码  [收藏代码](javascript:void())**

1. router.navigate('topic/1000', {
2. trigger: **true**
3. });

　　这段代码将URL更改为http://localhost/index.html#topic/1000，并触发了renderDetail方法。需要注意的是，我们在第二个参数传入了trigger配置，该配置用于表示更改URL的同时是否触发相应的Action方法。

**stop()方法**

　　还记得我们是通过Backbone.history.start()方法来启动路由监听的，你也可以随时调用Backbone.history.stop()方法来停止监听，例如：

**Js代码  [收藏代码](javascript:void())**

1. router.route('topic/:pageno/:pagesize', 'page', **function**(pageno, pagesize) {
2. Backbone.history.stop();
3. });

　　运行这段代码，并访问URL：http://localhost/index.html#topic/5/20，你会发现这个Action被执行之后，监听已经不再生效了。