



BACKBONE.JS

简单术语翻译对照：

散列表 (hash) 模型 (model) 视图 (view) 集合 (collection) 回调函数 (callback) 绑定 (bind)

[Backbone](#) 为复杂Javascript应用程序提供**模型**(models)、**集合**(collections)、**视图**(views)的结构。其中模型用于绑定键值数据和自定义事件；集合附有可枚举函数的丰富API；视图可以声明事件处理函数，并通过RESTful JSON接口连接到应用程序。

Backbone项目 [托管在Github](#)

Backbone是 [DocumentCloud](#) 的一个开源组件。

下载和依赖

[一回](#) 翻译的为 0.5.3 版本，下载请前往 [Backbone官网](#) 。

Backbone.js 唯一重度依赖 [Underscore.js](#)。对于 RESTful , history 的支持依赖于 [Backbone.Router](#) , DOM 处理依赖于 [Backbone.View](#) , [json2.js](#), 和 [jQuery](#) (> 1.4.2) 或 [Zepto](#) 之一。

简介

当我们开发含有大量Javascript的web应用程序时，首先你需要做的事情之一便是停止向DOM对象附加数据。通过复杂多变的jQuery选择符和回调函数创建Javascript应用程序，包括在HTML UI，Javascript逻辑和数据之间保持同步，都不复杂。但对富客户端应用来说，良好的架构通常是有很多益处的。

Backbone将数据呈现为 [模型](#)，你可以创建模型、对模型进行验证和销毁，甚至将它保存到服务器。当UI的变化引起模型属性改变时，模型会触发 *"change"* 事件；所有显示模型数据的 [视图](#) 会接收到该事件的通知，继而视图重新渲染。你无需查找DOM来搜索指定 *id* 的元素去手动更新HTML。 — 当模型改变了，视图便会自动变化。

Backbone.Events

Events 是一个可以被mix到任意对象的模块，它拥有让对象绑定和触发自定义事件的能力。事件在被绑定之前是不需要事先声明的，还可以携带参数。我们通过一个例子来看：

```
var object = {};  
  
_.extend(object, Backbone.Events);  
  
object.bind("alert", function(msg) {  
    alert("Triggered " + msg);  
});  
  
object.trigger("alert", "www.csser.com");
```

bind `object.bind(event, callback, [context])`

绑定 **callback** 函数到 **object** 对象。当**事件**触发时执行回调函数 **callback**。如果一个页面中有大量不同的事件，按照惯例使用冒号指定命名空间：“poll:start”，或“change:selection”

当 **callback** 执行时提供第三个可选参数，可以为 **this** 指定上下文：`model.bind('change', this.render, this)`

绑定到特殊事件“all”的回调函数会在任意事件发生时被触发，其第一个参数为事件的名称。例如，将一个对象的所有事件代理到另一对象：

```
proxy.bind("all", function(eventName) {  
    object.trigger(eventName);  
});
```

unbind `object.unbind([event], [callback])`

从 `object` 对象移除先前绑定的 `callback` 函数。如果不指定第二个参数，所有 `event` 事件绑定的回调函数都被移除。如果第一个参数也不指定，对象所绑定的所有回调函数都将被移除。

```
object.unbind("change", onChange); // 只移除onChange回调函数
```

```
object.unbind("change");           // 移除所有 "change" 回调函数
```

```
object.unbind();                   // 移除对象的所有回调函数
```

trigger `object.trigger(event, [*args])`

触发 `event` 事件的回调函数。后续传入 `trigger` 的参数会被依次传入事件回调函数。

模型 是所有 Javascript 应用程序的核心，包括交互数据及相关的大量逻辑：转换、验证、计算属性和访问控制。你可以用特定的方法扩展 `Backbone.Model`，**模型** 也提供了一组基本的管理变化的功能。

下面的示例演示了如何定义一个模型，包括自定义方法、设置属性、以及触发该属性变化的事件。

```
var Sidebar = Backbone.Model.extend({
  promptColor: function() {
    var cssColor = prompt("请输入一个CSS颜色值：");
    this.set({color: cssColor});
  }
});

window.sidebar = new Sidebar;

sidebar.bind('change:color', function(model, color) {
  $('#sidebar').css({background: color});
});

sidebar.set({color: 'white'});

sidebar.promptColor();
```

extend `Backbone.Model.extend(properties, [classProperties])`

要创建自己的 **模型** 类，你可以扩展 `Backbone.Model` 并提供实例 **属性**，以及可选的可以直接注册到构造函数的 **类属性** (classProperties)。

extend 可以正确的设置原型链，因此通过 **extend** 创建的子类 (subclasses) 也可以被深度扩展。

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

  initialize: function() { ... },

  author: function() { ... },

  coordinates: function() { ... },

  allowedToEdit: function(account) {
    return true;
  }

});

var PrivateNote = Note.extend({

  allowedToEdit: function(account) {
    return account.owns(this);
  }

});
```

constructor / initialize `new Model([attributes])`

当创建模型实例时，可以传入 **属性** 初始值，这些值会被 **set** 到模型。如果定义了 **initialize** 函数，该函数会在模型创建后执行。

```
new Site({
  title: "CSSer, 关注web前后端技术",
  author: "一回"
});
```

get `model.get(attribute)`

从模型获取当前属性值，比如： `csser.get("title")`

set `model.set(attributes, [options])`

向模型设置一个或多个散列属性。如果任何一个属性改变了模型的状态，在不传入 `{silent: true}` 选项参数的情况下，会触发“change”事件。可以绑定事件到某个属性，例如： `change:title`，及 `change:content`。

```
csser.set({title: "CSSer", content: "http://www.csser.com"});
```

如果模型拥有 **validate** 方法，那么属性验证会在 **set** 之前执行，如果验证失败，模型不会发生变化，这时 **set** 会返回 `false`。也可以在选项传入 `error` 回调函数，此时验证失败时会执行它而不触发“error”事件。

escape `model.escape(attribute)`

与 **get** 类似，但返回模型属性值的 HTML 转义后的版本。如果将数据从模型插入 HTML，使用 **escape** 取数据可以避免 **XSS** 攻击。

```
var hacker = new Backbone.Model({
  name: "<script>alert('xss')</script>"
});
```

```
alert(hacker.escape('name'));
```

has `model.has(attribute)`

属性值为非 null 或非 undefined 时返回 true

```
if (note.has("title")) {  
  ...  
}
```

unset `model.unset(attribute, [options])`

从内部属性散列表中删除指定属性。如果未设置 `silent` 选项，会触发 “change” 事件。

clear `model.clear([options])`

从模型中删除所有属性。如果未设置 `silent` 选项，会触发 “change” 事件。

id `model.id`

模型的特殊属性，**id** 可以是任意字符串（整型 id 或 UUID）。在属性中设置的 **id** 会被直接拷贝到模型属性上。我们可以从集合（collections）中通过 id 获取模型，另外 id 通常用于生成模型的 URLs。

cid `model.cid`

模型的特殊属性，**cid** 或客户 id 是当所有模型创建时自动产生的唯一标识符。客户 ids 在模型尚未保存到服务器之前便存在，此时模型可能仍不具有最终的 **id**，客户 ids 的形式为：c1, c2, c3 ...

attributes `model.attributes`

attributes 属性是包含模型状态的内部散列表。建议采用 `set` 更新属性而不要直接修改。如要获取模型属性的副本，用 `toJSON` 取而代之。

defaults`model.defaults` or `model.defaults()`

defaults 散列（或函数）用于为模型指定默认属性。创建模型实例时，任何未指定的属性会被设置为其默认值。

```
var Meal = Backbone.Model.extend({
  defaults: {
    "appetizer": "caesar salad",
    "entree":    "ravioli",
    "dessert":   "cheesecake"
  }
});

alert("Dessert will be " + (new Meal).get('dessert'));
```

需要提醒的是，在 Javascript 中，对象是按引用传值的，因此包含对象作为默认值，它会被所有实例共享。

toJSON`model.toJSON()`

返回模型 `attributes` 副本的 JSON 字符串化形式。它可用于模型的持久化、序列化，或者传递到视图前的扩充。该方法的名称有点混乱，因为它事实上并不返回 JSON 字符串，但 [JavaScript API for JSON.stringify](#) 可以实现。

```
var artist = new Backbone.Model({
  firstName: "立华",
  lastName: "咸"
});

artist.set({birthday: "December 13, 1979"});

alert(JSON.stringify(artist));
```

fetch`model.fetch([options])`

从服务器重置模型状态。这对模型尚未填充数据，或者服务器端已有最新状态的情况很有用处。如果服务器端状态与当前属性不同，则触发“change”事件。选项的散列表参数接受 `success` 和 `error` 回调函数，回调函数中可以传入（`model, response`）作为参数。


```
// 每隔 10 秒从服务器拉取数据以保持频道模型是最新的
setInterval(function() {
    channel.fetch();
}, 10000);
```

save `model.save([attributes], [options])`

通过委托 `Backbone.sync` 保存模型到数据库（或可替代的持久层）。**attributes** 散列表（在 `set`）应当包含想要改变的属性，不涉及的键不会被修改。如果模型含有 `validate` 方法，并且验证失败，模型不会保存。如果模型 `isNew`，保存将采用“create” (HTTP POST) 方法，如果模型已经在服务器存在，保存将采用“update” (HTTP PUT) 方法。

在下面的示例，注意我们是如何在模型初次保存时接收到“create”请求，第二次接收到“update”请求的。

```
Backbone.sync = function(method, model) {
    alert(method + ": " + JSON.stringify(model));
    model.id = 1;
};
```

```
var book = new Backbone.Model({
    title: "The Rough Riders",
    author: "Theodore Roosevelt"
});
```

```
book.save();
```

```
book.save({author: "Teddy"});
```

save 支持在选项散列表中传入 `success` 和 `error` 回调函数，回调函数支持传入 (`model`, `response`) 作为参数。如果模型拥有 `validate` 方法并且验证失败，`error` 回调函数会执行。如果服务端验证失败，返回非 200 的 HTTP 响应码，将产生文本或 JSON 的错误内容。

```
book.save({author: "F.D.R."}, {error: function(){ ... }});
```

destroy `model.destroy([options])`

通过委托 HTTP DELETE 请求到 `Backbone.sync` 销毁服务器上的模型. 接受 `success` 和 `error` 回调函数作为选项散列表参数。将在模型上触发“destroy”事件，该事件可以通过任意包含它的集合向上冒泡。

```
book.destroy({success: function(model, response) {  
    ...  
}});
```

validate `model.validate(attributes)`

该方法是未定义的，如果有在Javascript执行的需要，建议用自定义的验证逻辑重载它。**validate** 会在 `set` 和 `save` 之前调用，并传入待更新的属性。如果模型和属性通过验证，不返回任何值；如果属性不合法，返回一个可选择的错误。该错误可以是简单的用于显示的字符串错误信息，或者是一个可以描述错误详细的 `error` 对象。如果 **validate** 返回错误，`set` 和 `save` 将不会执行。失败的验证会触发一个“error”事件。

```
var Chapter = Backbone.Model.extend({  
    validate: function(attrs) {  
        if (attrs.end < attrs.start) {  
            return "can't end before it starts";  
        }  
    }  
});
```

```
var one = new Chapter({  
    title : "Chapter One: The Beginning"  
});
```

```
one.bind("error", function(model, error) {  
    alert(model.get("title") + " " + error);  
});
```

```
one.set({  
    start: 15,  
    end:   10  
});
```

“error”事件对模型和集合级别提供粗粒度的错误信息很有帮助，但如果想设计更好的处理错误的特定视图，可以直接传入 error 回调函数重载事件。

```
account.set({access: "unlimited"}, {
  error: function(model, error) {
    alert(error);
  }
});
```

url `model.url()`

返回模型资源在服务器上位置的相对 URL 。如果模型放在其它地方，可通过合理的逻辑重载该方法。生成 URLs 的形式为：“/[collection.url]/[id]”，如果模型不是集合的一部分，则 URLs 形式为：“/[urlRoot]/id”。

由于是委托到 `Collection#url` 来生成 URL，所以首先需要确认它是否定义过，或者所有模型共享一个通用根 URL 时，是否存在 `urlRoot` 属性。例如，一个 id 为 101 的模型，存储在 url 为 “/documents/7/notes” 的 `Backbone.Collection` 中，那么该模型的 URL 为：“/documents/7/notes/101”

urlRoot `model.urlRoot`

如果使用的集合外部的模型，通过指定 urlRoot 来设置生成基于模型 id 的 URLs 的默认 url 函数。“/[urlRoot]/id”

```
var Book = Backbone.Model.extend({urlRoot : '/books'});

var solaris = new Book({id: "1083-lem-solaris"});

alert(solaris.url());
```

parse `model.parse(response)`

parse 会在通过 `fetch` 从服务器返回模型数据，以及 `save` 时执行。传入本函数的为原始 `response` 对象，并且应当返回可以 `set` 到模型的属性散列表。默认实现是自动进行的，仅简单传入 JSON 响应。如果需要使用已存在的 API，或者更好的命名空间响应，可以重载它。

如果使用的 Rails 后端，需要注意 Rails's 默认的 `to_json` 实现已经包含了命名空间之下的模型属性。对于无缝的后端集成环境禁用这种行为：

```
ActiveRecord::Base.include_root_in_json = false
```

clone `model.clone()`

返回与模型属性一致的新的实例。

isNew `model.isNew()`

模型是否已经保存到服务器。如果模型尚无 `id`，则被视为新的。

change `model.change()`

手动触发“change”事件。如果已经在 `set` 函数传入选项参数 `{silent: true}`，当所有操作结束时，可以手动调用 `model.change()`。

hasChanged `model.hasChanged([attribute])`

标识模型从上次“change”事件发生后是否改变过。如果传入 `attribute`，当指定属性改变后返回 `true`。

注意，本方法以及接下来 `change` 相关的方法，仅对“change”事件发生有效。

```
book.bind("change", function() {  
  if (book.hasChanged("title")) {  
    ...  
  }  
});
```

changedAttributes `model.changedAttributes([attributes])`

仅获取模型属性已改变的散列表。 或者也可以传入外来的 **attributes** 散列，返回该散列与模型不同的属性。 一般用于指出视图的哪个部分已被更新，或者确定哪些需要与服务器进行同步。

previous `model.previous(attribute)`

在“change”事件发生的过程中，本方法可被用于获取已改变属性的旧值。

```
var bill = new Backbone.Model({
  name: "二回"
});
```

```
bill.bind("change:name", function(model, name) {
  alert("名字已从 " + bill.previous("name") + " 改为 " + name);
});
```

```
bill.set({name : "一回"});
```

previousAttributes `model.previousAttributes()`

返回模型的上一个属性散列的副本。一般用于获取模型的不同版本之间的区别，或者当发生错误时回滚模型状态。

Backbone.Collection

集合是模型的有序组合，我们可以在集合上绑定“change”事件，从而当集合中的模型发生变化时获得通知，集合也可以监听“add”和“remove”事件，从服务器更新，并能使用 [Underscore.js 提供的方法](#)

集合中的模型触发的任何事件都可以在集合身上直接触发，所以我们可以监听集合中模型的变化：`Documents.bind("change:selected", ...)`

extend `Backbone.Collection.extend(properties, [classProperties])`

通过扩展 **Backbone.Collection** 创建一个 **Collection** 类。实例属性参数 **properties** 以及 类属性参数 **classProperties** 会被直接注册到集合的构造函数。

model `collection.model`

指定集合的模型类。可以传入原始属性对象（和数组）来 `add`，`create`，以及 `reset`，传入的属性会被自动转换为适合的模型类型。

```
var Library = Backbone.Collection.extend({
  model: Book
});
```

constructor / initialize `new Collection([models], [options])`

当创建集合时，你可以选择传入初始的 **模型** 数组。集合的 `comparator` 函数也可以作为选项传入。如果定义了 `initialize` 函数，会在集合创建时被调用。

```
var tabs = new TabSet([tab1, tab2, tab3]);
```

models `collection.models`

访问集合中模型的原始值。通常我们使用 `get`，`at`，或 **Underscore方法** 访问模型对象，但偶尔也需要直接访问。

toJSON `collection.toJSON()`

返回集合中包含的每个模型对象的数组。可用于集合的序列化和持久化。本方法名称容易引起混淆，因为它与 `JavaScript's JSON API` 命名相同。

```
var collection = new Backbone.Collection([
  {name: "Tim", age: 5},
  {name: "Ida", age: 26},
  {name: "Rob", age: 55}
]);
```

```
alert(JSON.stringify(collection));
```

Underscore 方法 (26)

Backbone 代理了 **Underscore.js** 从而为 **Backbone.Collection** 提供了26个迭代函数。这里没有列出这些函数的使用方法，你可以[点击链接](#)前往查看：

- [forEach \(each\)](#)
- [map](#)
- [reduce \(foldl, inject\)](#)
- [reduceRight \(foldr\)](#)
- [find \(detect\)](#)
- [filter \(select\)](#)
- [reject](#)
- [every \(all\)](#)
- [some \(any\)](#)
- [include](#)
- [invoke](#)
- [max](#)
- [min](#)
- [sortBy](#)
- [groupBy](#)
- [sortedIndex](#)
- [toArray](#)
- [size](#)
- [first](#)
- [rest](#)
- [last](#)
- [without](#)
- [indexOf](#)
- [lastIndexOf](#)
- [isEmpty](#)
- [chain](#)

```
Books.each(function(book) {  
    book.publish();  
});  
  
var titles = Books.map(function(book) {  
    return book.get("title");  
});  
  
var publishedBooks = Books.filter(function(book) {  
    return book.get("published") === true;  
});  
  
var alphabetical = Books.sortBy(function(book) {  
    return book.author.get("name").toLowerCase();  
});
```

add `collection.add(models, [options])`

向集合中增加模型（或模型数组）。默认会触发“add”事件，可以传入 {silent : true} 关闭。如果定义了 [模型](#) 属性，也可以传入原始的属性对象让其看起来像一个模型实例。传入 {at: index} 可以将模型插入集合中特定的位置。

```
var ships = new Backbone.Collection;  
  
ships.bind("add", function(ship) {  
    alert("Ahoy " + ship.get("name") + "!");  
});  
  
ships.add([  
    {name: "Flying Dutchman"},  
    {name: "Black Pearl"}  
]);
```


remove `collection.remove(models, [options])`

从集合中删除模型（或模型数组）。会触发“remove”事件，同样可以使用 `silent` 关闭。

get `collection.get(id)`

返回集合中 `id` 为 `id` 的模型。

```
var book = Library.get(110);
```

getCid `collection.getCid(cid)`

通过指定客户id返回集合中的模型。客户id是指模型创建时自动生成的 `.cid` 属性。在模型尚未保存到服务器时其还没有id值，所以通过cid获取模型很有用处。

at `collection.at(index)`

返回集合中指定索引的模型对象。不论你是否对模型进行了重新排序，`at` 始终返回其在集合中插入时的索引值。

length `collection.length`

与数组类似，集合拥有 `length` 属性，返回该集合拥有的模型数量。

comparator `collection.comparator`

默认情况下，集合没有声明 **comparator** 函数。如果定义了该函数，集合中的模型会按照指定的算法进行排序。换言之，模型被增加的同时会插入适合的位置。Comparator接收模型作为参数，返回数值或字符串作为相对其它模型的排序依据。

注意即使下面例子中的章节是后加入集合中的，但它们都会遵循正确的排序：

```
var Chapter = Backbone.Model;
var chapters = new Backbone.Collection;

chapters.comparator = function(chapter) {
  return chapter.get("page");
};

chapters.add(new Chapter({page: 9, title: "The End"}));
chapters.add(new Chapter({page: 5, title: "The www.csser.com"}));
chapters.add(new Chapter({page: 1, title: "The Beginning"}));

alert(chapters.pluck('title'));
```

说明: *comparator* 函数与 Javascript 的 "sort" 并不相同, 后者必须返回 0, 1, 或 -1, 前者则更像 *sortBy* — 一个更友好的API。

sort `collection.sort([options])`

强制对集合进行重排序。一般情况下不需要调用本函数, 因为 `comparator` 函数会实时排序。如果不指定 `{silent: true}`, 调用 **sort** 会触发集合的 "reset" 事件。

pluck `collection.pluck(attribute)`

从集合中的每个模型拉取 `attribute`。等价于调用 `map`, 并从迭代器中返回单个属性。

```
var stooges = new Backbone.Collection([
  new Backbone.Model({name: "Curly"}),
  new Backbone.Model({name: "Larry"}),
  new Backbone.Model({name: "Moe"})
]);

var names = stooges.pluck("name");

alert(JSON.stringify(names));
```

url `collection.url` or `collection.url()`

设置 **url** 属性（或函数）以指定集合对应的服务器位置。集合内的模型使用 **url** 构造自身的 URLs。

```
var Notes = Backbone.Collection.extend({
  url: '/notes'
});
```

// 或者，更复杂一些的方式：

```
var Notes = Backbone.Collection.extend({
  url: function() {
    return this.document.url() + '/notes';
  }
});
```

parse `collection.parse(response)`

每一次调用 **fetch** 从服务器拉取集合的模型数据时，**parse**都会被调用。本函数接收原始 **response** 对象，返回可以 **add** 到集合的模型属性数组。默认实现是无需操作的，只需简单传入服务端返回的JSON对象。如果需要处理遗留API，或者在返回数据定义自己的命名空间，可以重写本函数。

```
var Tweets = Backbone.Collection.extend({
  // Twitter 搜索 API 在 "result" 键下返回 tweets
  parse: function(response) {
    return response.results;
  }
});
```

fetch`collection.fetch([options])`

从服务器拉取集合的默认模型，成功接收数据后会重置（reset）集合。**options** 支持 `success` 和 `error` 回调函数，回调函数接收（`collection`, `response`）作为参数。可以委托 `Backbone.sync` 在随后处理个性化需求。处理 **fetch** 请求的服务器应当返回模型的 JSON 数组。

```
Backbone.sync = function(method, model) {  
    alert(method + ": " + model.url);  
};
```

```
var Accounts = new Backbone.Collection;  
Accounts.url = '/accounts';
```

```
Accounts.fetch();
```

如果希望向当前集合追加模型数据而不是替换，传入 `{add: true}` 作为 **fetch** 的参数。

fetch 的参数可以支持直接传入 `jQuery.ajax` 作为参数，所以拉取指定页码的集合数据可以这样写：`Documents.fetch({data: {page: 3}})`

不建议在页面加载完毕时利用 **fetch** 拉取并填充集合数据 — 所有页面初始数据应当在 `bootstrapped` 时已经就绪。**fetch** 适用于惰性加载不需立刻展现的模型数据。

reset`collection.reset(models, [options])`

每次一个的向集合做增删操作已经很好了，但有时会有很多的模型变化以至于需要对集合做大批量的更新操作。利用 **reset** 可将集合替换为新的模型（或键值对象），结束后触发“reset”事件。传入 `{silent: true}` 忽略“reset”事件的触发。不传入任何参数将清空整个集合。

这里有一个在页面加载完毕后 **reset** 初始启动集合的例子：

```
<script>
  Accounts.reset(<%= @csser.to_json %>);
</script>
```

create `collection.create(attributes, [options])`

在集合中创建一个模型。 等价于用键值对象实例一个模型，然后将模型保存到服务器，保存成功后将模型增加到集合中。 如果验证失败会阻止模型创建，返回 `false`，否则返回该模型。 为了能正常运行，需要在集合中设置 `model` 属性。**create** 方法接收键值对象或者已经存在尚未保存的模型对象作为参数。

```
var Library = Backbone.Collection.extend({
  model: Book
});

var NYPL = new Library;

var othello = NYPL.create({
  title: "Backbone.js API 中文手册",
  author: "一回( www.csser.com )"
});
```

Backbone.Router

web应用程序通常需要为应用的重要位置提供可链接，可收藏，可分享的 URLs。 直到最近， 锚点（hash）片段（#page）可以被用来提供这种链接， 同时随着 History API 的到来，锚点已经可以用于处理标准 URLs（/page）。 **Backbone.Router** 为客户端路由提供了许多方法，并能连接到指定的动作（actions）和事件（events）。 对于不支持 History API 的旧浏览器，路由提供了优雅的回调函数并可以透明的进行 URL 片段的转换。

页面加载期间，当应用已经创建了所有的路由，需要调用 `Backbone.history.start()`，或 `Backbone.history.start({pushState : true})` 来确保驱动初始化 URL 的路由。

extend `Backbone.Router.extend(properties, [classProperties])`

创建一个自定义的路由类。可以通过 `routes` 定义路由动作键值对，当匹配了 URL 片段便执行定义的动作。

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

  routes: {
    "help":           "help",    // #help
    "search/:query":  "search",  // #search/kiwis
    "search/:query/p:page": "search" // #search/kiwis/p7
  },

  help: function() {
    ...
  },

  search: function(query, page) {
    ...
  }

});
```

routes `router.routes`

`routes` 将带参数的 URLs 映射到路由实例的方法上，这与 [视图](#) 的 [事件键值对](#) 非常类似。路由可以包含参数，`:param`，它在斜线之间匹配 URL 组件。路由也支持通配符，`*splat`，可以匹配多个 URL 组件。

举个例子，路由 `"search/:query/p:page"` 能匹配 `#search/obama/p2`，这里传入了 `"obama"` 和 `"2"` 到路由对应的动作中去了。`"file/*path"` 路由可以匹配 `#file/nested/folder/file.txt`，这时传入动作的参数为 `"nested/folder/file.txt"`。

当访问者点击浏览器后退按钮，或者输入 URL，如果匹配一个路由，此时会触发一个基于动作名称的 [事件](#)，其它对象可以监听这个路由并接收到通知。下面的示例中，用户访问 #help/uploading 将从路由中触发 route:help 事件。

```
routes: {
  "help/:page":      "help",
  "download/*path":  "download",
  "folder/:name":    "openFolder",
  "folder/:name-:mode": "openFolder"
}

router.bind("route:help", function(page) {
  ...
});
```

constructor / initialize `new Router([options])`

实例化一个路由对象，你可以直接传入 [routes](#) 键值对象作为参数。如果定义该参数，它们将被传入 initialize 构造函数中初始化。

route `router.route(route, name, callback)`

为路由对象手动创建路由，route 参数可以是 [路由字符串](#) 或 正则表达式。每个捕捉到的被传入的路由或正则表达式，都将作为参数传入回调函数（callback）。一旦路由匹配，name 参数会触发 “route:name” 事件。

```
initialize: function(options) {

  // 匹配 #page/10, 传入回调函数 "10"
  this.route("page/:number", "page", function(number){ ... });

  // 匹配 /csser.com/b/c/open, 传入回调函数 "csser.com/b/c"
  this.route(/^(.*)/open$/, "open", function(id){ ... });

}
```

navigate `router.navigate(fragment, [triggerRoute])`

手动到达应用程序中的某个位置。传入 **triggerRoute** 以执行路由动作函数。

```
openPage: function(pageNumber) {  
  this.document.pages.at(pageNumber).open();  
  this.navigate("page/" + pageNumber);  
}
```

或者 ...

```
app.navigate("help/troubleshooting", true);
```

Backbone.history

History 作为全局路由服务用于处理 hashchange 事件或 pushState，匹配适合的路由，并触发回调函数。我们不需要自己去做这些事情 — 如果使用带有键值对的 **路由**，Backbone.history 会被自动创建。

Backbone 会自动判断浏览器对 **pushState** 的支持，以做内部的选择。不支持 pushState 的浏览器将会继续使用基于锚点的 URL 片段，如果兼容 pushState 的浏览器访问了某个 URL 锚点，将会被透明的转换为真实的 URL。注意使用真实的 URLs 需要 web 服务器支持直接渲染那些页面，因此后端程序也需要做修改。例如，如果有这样一个路由 /document/100，如果浏览器直接访问它，web 服务器必须能够处理该页面。趋于对搜索引擎爬虫的兼容，让服务器完全为该页面生成静态 HTML 是非常好的做法 ... 但是如果要做的是一个 web 应用，只需要利用 Javascript 和 Backbone 视图将服务器返回的 REST 数据渲染就很好了。

start `Backbone.history.start([options])`

当所有的 **路由** 创建并设置完毕，调用 Backbone.history.start() 开始监控 hashchange 事件并分配路由。

需要指出的是，如果想在应用中使用 HTML5 支持的 pushState，只需要这样做：Backbone.history.start({pushState : true})。

如果应用不是基于域名的根路径 /，需要告诉 History 基于什么路径：Backbone.history.start({pushState: true, root: "/public/search/"})

当执行后，如果某个路由成功匹配当前 URL，Backbone.history.start() 返回 true。如果没有定义的路由匹配当前 URL，返回 false。

如果服务器已经渲染了整个页面，但又不希望开始 History 时触发初始路由，传入 `silent : true` 即可。

```
$(function(){
  new WorkspaceRouter();
  new HelpPaneRouter();
  Backbone.history.start({pushState: true});
});
```

Backbone.sync

Backbone.sync 是 Backbone 每次向服务器读取或保存模型时都要调用执行的函数。默认情况下，它使用 (jQuery/Zepto).ajax 方法发送 RESTful json 请求。如果想采用不同的持久化方案，比如 WebSockets, XML, 或 Local Storage，我们可以重载该函数。

Backbone.sync 的语法为 `sync(method, model, [options])`。

- **method** – CRUD 方法 (“create”, “read”, “update”, 或 “delete”)
- **model** – 要被保存的模型（或要被读取的集合）
- **options** – 成功和失败的回调函数，以及所有 jQuery 请求支持的选项

默认情况下，当 **Backbone.sync** 发送请求以保存模型时，其属性会被序列化为 JSON，并以 application/json 的内容类型发送。当接收到来自服务器的 JSON 响应后，对经过服务器改变的模型进行拆解，然后在客户端更新。当 “read” 请求从服务器端响应一个集合 ([Collection#fetch](#)) 时，便拆解模型属性对象的数组。

默认 **sync** 映射 REST 风格的 CRUD 类似下面这样：

- **create** → POST /collection
- **read** → GET /collection[/id]
- **update** → PUT /collection/id
- **delete** → DELETE /collection/id

emulateHTTP `Backbone.emulateHTTP = true`

老的浏览器不支持 Backbone 默认的 REST/HTTP，此时可以开启 `Backbone.emulateHTTP`。设置该选项将通过 POST 方法伪造 PUT 和 DELETE 请求，此时该请求会向服务器传入名为 `_method` 的参数。设置该选项同时也会向服务器发送 `X-HTTP-Method-Override` 头。

```
Backbone.emulateHTTP = true;
```

```
model.save(); // POST 到 "/collection/id", 附带 "_method=PUT" + header.
```

emulateJSON `Backbone.emulateJSON = true`

同样老的浏览器也不支持发送 `application/json` 编码的请求，设置 `Backbone.emulateJSON = true`；后 JSON 模型会被序列化为 `model` 参数，请求会按照 `application/x-www-form-urlencoded` 的内容类型发送，就像提交表单一样。

Backbone.View

Backbone 视图的使用相当方便 — 它不会影响任何的 HTML 或 CSS 代码，并且可以与任意 Javascript 模板引擎兼容。基本的做法就是，将界面组织到逻辑视图，之后是模型，当模型数据发生改变，视图立刻自动更新，这一切都不需要重绘页面。我们再也不必钻进 JSON 对象中，查找 DOM 元素，手动更新 HTML 了，通过绑定视图的 `render` 函数到模型的 “change” 事件 — 模型数据会即时的显示在 UI 中。

extend `Backbone.View.extend(properties, [classProperties])`

创建自定义的视图类。通常我们需要重载 `render` 函数，声明 [事件](#)，以及通过 `tagName`，`className`，或 `id` 为视图指定根元素。

```
var DocumentRow = Backbone.View.extend({

  tagName: "li",

  className: "document-row",

  events: {
    "click .icon": "open",
    "click .button.edit": "openEditDialog",
    "click .button.delete": "destroy"
  },

  render: function() {
    ...
  }

});
```

constructor / initialize `new View([options])`

每次实例化一个视图时，传入的选项参数会被注册到 `this.options` 中以备后用。这里有多个特殊的选项，如果传入，则直接注册到视图中去：`model`, `collection`, `el`, `id`, `className`, 以及 `tagName`. 如果视图定义了 **initialize** 函数，当视图实例化时该函数便立刻执行。如果希望创建一个指向 DOM 中已存在的元素的视图，传入该元素作为选项：`new View({el: existingElement})`

```
var doc = Documents.first();

new DocumentRow({
  model: doc,
  id: "document-row-" + doc.id
});
```

el view.el

所有的视图都拥有一个 DOM 元素（**el** 属性），即使该元素仍未插入页面中去。视图可以在任何时候渲染，然后一次性插入 DOM 中去，这样能尽量减少 reflows 和 repaints 从而获得高性能的 UI 渲染。this.el 可以从视图的 tagName, className, 以及 id 属性创建，如果都未指定，**el** 会是一个空 div。

如果希望将 **el** 赋给页面 DOM 中已经存在的元素，直接设置其值为真实的 DOM 元素或 CSS 选择符字符串。

```
var ItemView = Backbone.View.extend({
  tagName: 'li'
});
```

```
var BodyView = Backbone.View.extend({
  el: 'body'
});
```

```
var item = new ItemView();
var body = new BodyView();
```

```
alert(item.el + ' ' + body.el);
```

\$ (jQuery 或 Zepto) view.\$(selector)

如果页面中引入了 jQuery 或 Zepto，每个视图都将拥有 \$ 函数，可以在视图元素查询作用域内运行。如果使用该作用域内的 jQuery 函数，就不需要从列表中指定的元素获取模型的 ids 这种查询了，我们可以更多的依赖 HTML class 属性。它等价于运行：\$(selector, this.el)。

```
ui.Chapter = Backbone.View.extend({
  serialize : function() {
    return {
      title: this.$(".title").text(),
      start: this.$(".start-page").text(),
      end:   this.$(".end-page").text()
    };
  }
});
```

render `view.render()`

render 默认实现是没有操作的。重载本函数可以实现从模型数据渲染视图模板，并可用新的 HTML 更新 `this.el`。推荐的做法是在 **render** 函数的末尾 `return this` 以开启链式调用。

```
var Bookmark = Backbone.View.extend({
  render: function() {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  }
});
```

Backbone 并不知道开发者使用何种模板引擎。**render** 函数中可以采用拼字符串，或者利用 `document.createElement` 创建 DOM 树等等。但还是建议选择一个好的 Javascript 模板引擎。[Mustache.js](#), [Haml-js](#), 以及 [Eco](#) 都是很好的选择。因为 [Underscore.js](#) 已经引入页面了，所以为了防止 XSS 攻击带给数据的安全威胁，`_.template` 可以使用并是一个很好的选择。

无论基于什么考虑，都永远不要在 Javascript 中拼接 HTML 字符串。

remove `view.remove()`

从 DOM 中移除视图。它等价与下面的语句：`$(view.el).remove()`；

make `view.make(tagName, [attributes], [content])`

借助给定的元素类型（**tagName**），以及可选的 `attributes` 和 HTML 内容创建 DOM 元素。通常用于内部创建初始的 `view.el`。

```
var view = new Backbone.View;

var el = view.make("b", {className: "bold"}, "Bold! ");

$("#make-demo").append(el);
```

delegateEvents `delegateEvents([events])`

采用 jQuery 的 `delegate` 函数来为视图内的 DOM 事件提供回调函数声明。如果未传入 **events** 对象，使用 `this.events` 作为事件源。事件对象的书写格式为 `{"event selector" : "callback"}`。省略 `selector` 则事件被绑定到视图的根元素 (`this.el`)。默认情况下，`delegateEvents` 会在视图的构造函数内被调用，因此如果有 `events` 对象，所有的 DOM 事件已经被连接，并且我们永远不需要去手动调用本函数。

`events` 属性也可以被定义成返回 **events** 对象的函数，这样让我们定义事件，以及实现事件的继承变得更加方便。

视图 [渲染](#) 期间使用 **delegateEvents** 相比用 jQuery 向子元素绑定事件有更多优点。所有注册的函数在传递给 jQuery 之前已被绑定到视图上，因此当回调函数执行时，`this` 仍将指向视图对象。当 **delegateEvents** 再次运行，此时或许需要一个不同的 `events` 对象，所以所有回调函数将被移除，然后重新委托 — 这对模型不同行为也不同的视图挺有用处。

搜索结果页面显示文档的视图看起来类似这样：

```
var DocumentView = Backbone.View.extend({

  events: {
    "dblclick"           : "open",
    "click .icon.doc"     : "select",
    "contextmenu .icon.doc" : "showMenu",
    "click .show_notes"   : "toggleNotes",
    "click .title .lock"  : "editAccessLevel",
    "mouseover .title .date" : "showTooltip"
  },
```

```
render: function() {
$(this.el).html(this.template(this.model.toJSON()));
return this;
},

open: function() {
window.open(this.model.get("viewer_url"));
},

select: function() {
this.model.set({selected: true});
},

...

});
```

Utility Functions

noConflict `var backbone = Backbone.noConflict();`

返回 Backbone 对象的原始值。通常用于在第三方网站上引入了多个 Backbone 文件，避免冲突。

```
var localBackbone = Backbone.noConflict();
var model = localBackbone.Model.extend(...);
```