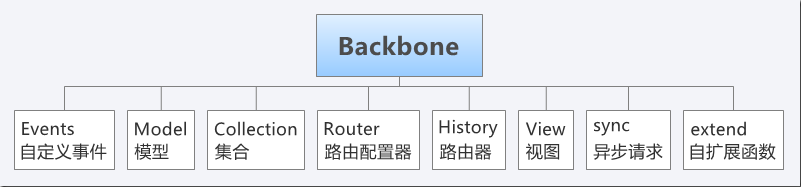
1. **简介:**
2. **http://www.cnblogs.com/aaronjs/archive/2012/08/18/2645377.html**
3. 模型(**models**)：绑定键值数据和自定义事件
4. 集合(**collections**)：附有可枚举函数的丰富API
5. 视图(**views**)：声明事件处理函数，并通过RESRful JSON接口连接到应用程序。

唯一重度依赖Underscore.js( >= 1.5.0)

基于RESTful（一个架构样式的网络系统）的约束，histroy的支持依赖于Backbone.Router ，DOM处理依赖于 Backbone.View，包括jQuery ( >= 1.11.0), 和 json2.js对旧的IE浏览器的支持.



模型可围绕数据处理类似curd的操作，Backbone中的模型类是Backbone.Model，包含了数据存储，数据验证，以及数据发生变动时触发相关动作，只要继承就能使用这些特性了

当models中值被改变时自动触发一个"change"事件、所有用于展示models数据的views都会侦听到这个事件，然后进行重新渲染。

模型构造器

模型首先就是围绕数据操作，上帝set，上帝get不能少，也是为了体现出对象封装性

与此同时数据的清理与改变也是不能少的

监听对象中属性变化

为对象添加验证规则，以及错误提示

对象的获取和保存，需要服务器端支持才能测试

**六.Backbone. Collection**

1. 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()：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。

**六.Backbone.Router（路由）**

**6.1 简介**

1.web应用程序常需要为应用的重要位置提供可链接，可收藏，可分享的 URLs。猫点（hash）片段（#page）可以被用来提供这种链接， 同时随着 History API 的到来，猫点已经可以用于处理标准 URLs （/page）。

2.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。

3.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。

**6.2 Extend**

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

创建自定义路由类。当匹配了 URL 片段便执行定义的动作，并可以通过 routes 定义路由动作键值对。要避免在路由定义时使用前导斜杠：

|  |
| --- |
| 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) {  ...  }  }); |

**6.3 routes**

router.routes

routes将带参数的 URLs 映射到路由实例的方法上(或只是直接的函数定义)，这与 View（视图)的events hash(事件键值对)非常类似。

路由可包含参数， :param，它在斜线之间匹配 URL 组件。

也支持通配符， \*splat，可以匹配多个 URL 组件。

可选部分放在括号中(/:optional)。

例:

(1)"search/:query/p:page" 能匹配#search/obama/p2 , 传入"obama" 和 "2" 到路由对应的动作中去了。

(2)"file/\*path"可匹配 #file/nested/folder/file.txt，传入动作的参数为 "nested/folder/file.txt"。

(3)"docs/:section(/:subsection)"可匹配#docs/faq 和 #docs/faq/installing，第一种情况，传入 "faq";第二种情况，传入"faq" 和 "installing"。

(4)结尾的斜杠会被当作URL的一部分， 访问时会被（正确地）当作一个独立的路由。 docs 和 docs/将触发不同的回调。如你不能避免产生这两种类型的URLs时，可定义"docs(/)"来匹配捕捉这两种情况。

(5)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 event， 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.on("route:help", function(page) {  ...  }); |

**6.4 constructor / initialize**

new Router([options])

当创建一个新路由时，可直接传入 routes 键值对象作为参数。 如定义该参数，它们将被传入 initialize 构造函数中初始化。

**6.4 route**

router.route(route, name, [callback])

为路由对象手动创建路由，route 参数可是 routing string（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。如果callback参数省略 router[name]将被用来代替。后来添加的路由可以覆盖先前声明的路由。

|  |
| --- |
| initialize: function(options) {  // Matches #page/10, passing "10"  this.route("page/:number", "page", function(number){ ... });  // Matches /117-a/b/c/open, passing "117-a/b/c" to this.open  this.route(/^(.\*?)\/open$/, "open");  },  open: function(id) { ... } |

**6.5 navigate**

router.navigate(fragment, [options])

当达到你的应用的一个点时，想保存为一个URL，可调用navigate以更新的URL。

如果您也想调用路由功能， 设置trigger选项设置为true。

无需在浏览器的历史记录创建条目来更新URL,设置 replace选项设置为true。

|  |
| --- |
| openPage: function(pageNumber) {  this.document.pages.at(pageNumber).open();  this.navigate("page/" + pageNumber);  }  # Or ...  app.navigate("help/troubleshooting", {trigger: true});  # Or ...  app.navigate("help/troubleshooting", {trigger: true, replace: true}); |

**6.6 execute**

router.execute(callback, args)

在路由内部被调用,每当路由和其相应的callback匹配时被执行。

覆盖它来执行自定义解析或包装路由,例如:在传递他们给你的路由回调之前解析查询字符串，

|  |
| --- |
| var Router = Backbone.Router.extend({  execute: function(callback, args) {  args.push(parseQueryString(args.pop()));  if (callback) callback.apply(this, args);  }  }); |

**七.Backbone.history**

**7.1 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。 不需自己去做这些事情—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,以做内部选择.不支持 pushState 的浏览器将会继续使用基于猫点的 URL 片段，如果兼容 pushState 的浏览器访问了某个 URL 猫点，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。

**7.2 Start**

Backbone.history.start([options])

当所有 Routers 创建并设置完毕，调用 Backbone.history.start() 开始监控 hashchange 事件并分配路由。后续调用Backbone.history.start()会抛出一个错误， 并且Backbone.History.started是一个布尔值，指示是否已经被调用。

如想在应用中使用 HTML5 支持的 pushState，只需要这样做：Backbone.history.start({pushState : true}) 。如果你想使用pushState的话， 对于那些本身不支持它的浏览器，需要用整页刷新代替， 您可以添加{hashChange: false}到选项。

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

因为在Internet Explorer中基于hash的历史记录依赖于<iframe>，因此需要确定DOM已准备就绪后再调用 start() 。

$(function(){

new WorkspaceRouter();

new HelpPaneRouter();

Backbone.history.start({pushState: true});

});