Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

0、入门

核心概念

上手Ember.js，必须先理解几个核心概念。

我们希望开发者能够建立雄心勃勃的、与原生应用抗衡的大型web应用，要做到这一点，必须拥有先进的工具和正确的概念，以帮助开发者进行沟通协作。

我们耗费了大量时间去参考原生应用框架的思想，比如Cocoa，但是那些概念不仅没能形成帮助，反而还成为了一种障碍，而且，有些概念确实也并不适合web开发的特征。于是我们转去向其他一些流行的开源项目寻找灵感，比如Ruby on Rails以及Backbone.js。

因此，Ember.js是一个综合体，它结合了原生应用的强大，且兼具现代网络的轻量与敏捷。

概念

模板 TEMPLATES

Ember.js使用Handlebars模板引擎定义应用程序的用户界面，模板里可以包含HTML以及以下内容：

- **表达式（Expressions）：**比如{{firstname}}，用来提取控制器或者模型中的信息，将之转换成HTML，并且保持自动更新。
- **插口（Outlets）：**给其他模板预留的占位符，当用户操作应用程序的时候，路由器可以控制不同的模板接入到插口的位置。通过{{outlet}}标记可以在模板里创建插口。
- **视图（Views）：**负责处理用户事件，通过{{view}}标记在模板里创建。

视图 VIEWS

嵌入在模板里的视图负责将原始事件（比如点击、手势、滑动）转换成应用里定义的事件，并将事件发送给控制器。

比如，视图可以将click事件转换成更有意义的deleteItem事件，并发送给控制器。如果控制器没有实现deleteItem事件，则事件会发送给当前的路由器。

控制器 CONTROLLERS

控制器储存应用程序状态，模板用来连接控制器，并将控制器的当前状态转换成HTML。

控制器通常用来为模板描述模型，在这种情况下，控制器将模型的属性传给模板，并以模板期望的方式修改或增加模型。

模型 MODELS

模型用来处理持久化的数据，比如在应用程序中由用户赋值的数据。模型对象一般由服务器加载，当客户端做出修改的时候，再存回服务器。

通常来说，数据可以转换为JSON格式由服务器传递给Ember.js的模型，实现关联、属性计算，以及其他一些功能。

路由器 ROUTER

路由器负责管理应用程序的状态。

当用户启动应用程序，它会检查URL，确保载入正确的模板进行展示，并为模板解析相应的模型对象。

当应用程序在运行到不同状态时，路由器会自动更新URL，用户可以保存URL，以便返回之前的一些状态，或者通过URL与其他用户分享当前页面。

上面这些就是开发Ember.js应用程序需要了解的核心概念。我们设计的系统支持复杂的扩展，如果你能遵循这些基本内容，在增加新功能的时候就不必回溯整个系统。

我们认为，让多个开发者通过框架的模式，对一个问题在解决方式上达成一致是很重要的。现在已经解释了一些基本对象所扮演的角色，接下来就要深入Ember.js，了解各个部分的细节和工作方式。

回到页顶
xbingoz.com 首页

Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

1、应用程序 APPLICATION

创建一个应用程序

创建Ember.js应用程序的第一步是创建一个Ember.Application类的实例化对象：

```
window.App = Ember.Application.create();
```

这里将实例化的对象命名为App，开发者可以根据应用程序的用途，选择意义相符的名字。

创建一个应用程序的实例对象非常重要，原因如下：

- 定义应用程序的命名空间，所有类都定义成该对象的属性（比如App.PostsView、App.PostsController）。这样做可以避免污染全局作用域。
- 在document上增加事件监听，并负责将事件发送给视图。
- 自动渲染模板，包括根模板，以及其他放入根模板的模板，都将被渲染。
- 基于当前URL创建路由器并开始路由。

应用程序模板

应用程序模板是应用程序启动时被渲染的默认模板，模板内应该包含header、footer以及其他一些修饰内容。另外，还应该包含至少一个

{{outlet}}占位符，路由器会根据应用程序的状态，在这里插入相应的模板。

例：

```
<header>
  <h1>Igor's Blog</h1>
</header>

<div>
  {{outlet}}
</div>

<footer>
  &copy;2013 Igor's Publishing, Inc.
</footer>
```

header和footer会保持不动，而div里的内容会被改变，比如用户将URL路径定位到/posts或/posts/15，div里的内容就会做出相应变化。

如果在HTML里保存模板，可以创建一个匿名的script标签，它将会被自动调用到屏幕上。

```
<script type="text/x-handlebars">
  <div>
    {{outlet}}
  </div>
</script>
```

如果使用工具加载模板，一定要给模板命名为application。

更多内容参考模板和路由的章节。

回到页顶
xbingoz.com 首页

Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

2、模板 TEMPLATES

HANDLEBARS基础

Ember.js使用Handlebars模板类库增强应用的用户界面。Handlebars模板和普通的HTML很像，但是Handlebars可以嵌入表达式更改显示内容。

我们采用Handlebars并且给它增加了许多强大的扩展，有助于让Handlebars模板更像整洁的HTML代码一样利于思考，从而帮助定义用户界面。并且，一旦让Ember.js将一个给定的模板渲染到屏幕上，可以不必为了保持内容更新而增加额外的代码。

定义模板

如果不使用编译工具，可以在HTML里增加 `<script>` 标签，从而给应用程序创建一个主模板，例如：

```
<html>
  <body>
    <script type="text/x-handlebars">
      Hello, <strong>{{firstName}} {{lastName}}</strong>!
    </script>
  </body>
</html>
```

此模板会自动被编译，并在应用程序载入的时候，由路由器呈现到页面上。

也可以给模板定义名称，以便稍后使用。比如，要定义一个可以重复使用的控件，应用到界面里的不同位置，可以给script标签增加一个 `data-template-name` 属性，该属性可以让Ember.js保存此模板，不立即进行渲染：

```
<html>
  <head>
    <script type="text/x-handlebars" data-template-name="say-hello">
      <div class="my-cool-control">{{name}}</div>
    </script>
  </head>
</html>
```

也可以使用编译工具管理应用程序的资源，编译工具可以更好地对模板进行预处理，生成Ember.js可用的模板。

HANDLEBARS表达式

每个模板都有一个相应的控制器，模板可以在控制器的属性里找到需要现实的内容。

将控制器的属性名包裹在大括号里，可以将属性值显示在页面上，就像这样：

```
Hello, <strong>{{firstName}} {{lastName}}</strong>!
```

这里的标记会从控制器里寻找firstName和lastName两个属性，并将它们插入到HTML里定义的模板上，同时也会插入DOM。

默认情况下，应用程序的顶层模板会绑定到一个特定的控制器 `ApplicationController` ：

```
App.ApplicationController = Ember.Controller.extend({
  firstName: "Trek",
  lastName: "Glowacki"
});
```

上面的模板和控制器结合起来，就会渲染出以下HTML内容：

```
Hello, <strong>Trek Glowacki</strong>!
```

Handlebars的表达式绑定了监听，这意味着模板里用到的值一旦发生变化，HTML内容就会自动更新。

随着应用程序的规模不断扩大，模板的数量也会增加，相应的控制器也会增加，每个模板都会绑定到不同的控制器上。

条件语句

如果只想在属性存在的情况下，才将模板里的相应部分显示出来，可以使用{{#if}}标记，定义有条件的渲染模块：

```
{{#if person}}
  Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
{{/if}}
```

如果传入参数的计算值是false、undefined、null或者[]（任何等同false的值），该语句块就不会被Handlebars渲染。

另外，表达式值为假时，可以用{{else}}标记指向备用的渲染模板：

```
{{#if person}}
  Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
{{else}}
  Please log in.
{{/if}}
```

如果只有值为假时才渲染，可以使用{{#unless}}：

```
{{#unless hasPaid}}
  You owe: ${{total}}
{{/unless}}
```

{{#if}}和{{#unless}}是块级表达式的例子，它们可以调用模板里的一部分内容。块级表达式和一般编程里的表达式类似，不过在调用标记前要加hash(#)，在结束位置要标明关闭（/）。

显示一个项目列表

假如要枚举一个对象列表，可以使用Handlebar的 `{{#each}}` 标记：

```
<ul>
  {{#each people}}
    <li>Hello, {{name}}!</li>
  {{/each}}
</ul>
```

模板里的{{#each}}代码块，会将上下文中设置的item的元素循环输出。

上面的例子会输出这样的内容：

```
<ul>
  <li>Hello, Yehuda!</li>
  <li>Hello, Tom!</li>
  <li>Hello, Trek!</li>
</ul>
```

Handlebars里的一切都绑定了监听器，{{#each}}也一样。如果item数组里增加或者删除item，DOM就会自动更新，而且不需要为此多写任何代码。

还有一种不必改变{{#each}}所在模板作用域的使用方法，当遇到需要循环外部作用域的属性时，这种方法会非常有效。

```
{{name}}'s Friends

<ul>
  {{#each friend in friends}}
    <li>{{name}}'s friend {{friend.name}}</li>
  {{/each}}
<ul>
```

这段代码会打印以下内容：

```
Trek's Friends

<ul>
  <li>Trek's friend Yehuda</li>
  <li>Trek's friend Tom!</li>
</ul>
```

{{#each}}也可以搭配一个{{else}}使用，当循环的内容为空时，就会输出{{else}}的代码块：

```
{{#each people}}
  Hello, {{name}}!
{{else}}
  Sorry, nobody is here.
{{/each}}
```

改变作用域

有时候调用模板里的不同段落，需要切换到不同的上下文，比如重复路径：

```
Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!
```

这时候可以使用{{#with}}标记，使代码更简洁：

```
{{#with person}}
  Welcome back, <b>{{firstName}} {{lastName}}</b>!
{{/with}}
```

通过{{#with}}可以设置代码段的上下文，默认上下文是控制器，而通过{{#with}}可以一次性切换代码段里所有表达式的上下文关系。

绑定元素属性

除了普通的文本，可能还要将模板中的HTML元素属性绑定到控制器。

例如控制器的一个属性是某个图片的URL：

```
<div id="logo">
  <img {{bindAttr src="logoUrl"}} alt="Logo">
</div>
```

生成以下代码：

```
<div id="logo">
  <img src="http://www.example.com/images/logo.png" alt="Logo">
</div>
```

如果使用布尔值，可以增加或删除一些特殊属性，例如：

```
<input type="checkbox" {{bindAttr disabled="isAdministrator"}}>
```

如果isAdministrator是true：

```
<input type="checkbox" disabled>
```

如果isAdministrator是false：

```
<input type="checkbox">
```

绑定元素的class类名

HTML元素的class属性可以像其他属性一样被绑定：

```
<div {{bindAttr class="priority"}}>
  Warning!
</div>
```

如果控制器的priority属性值是p4，就会生成以下内容：

```
<div class="p4">
  Warning!
</div>
```

绑定到布尔值

如果绑定的是一个布尔值，Ember.js会将属性名用横线转换成类名：

```
<div {{bindAttr class="isUrgent"}}>
  Warning!
</div>
```

如果isUrgent是true，将渲染出以下HTML：

```
<div class="is-urgent">
  Warning!
</div>
```

如果isUrgent是false，将不会加入类名：

```
<div>
  Warning!
</div>
```

如果想明确地定义一个类名，而不是用横线修改属性名，可以使用以下语法：

```
<div {{bindAttr class="isUrgent:urgent"}}>
  Warning!
</div>
```

如果isUrgent为true，urgent将成为类名：

```
<div class="urgent">
  Warning!
</div>
```

也可以指定一个false时添加的类名：

```
<div {{bindAttr class="isEnabled:enabled:disabled"}}>
  Warning!
</div>
```

在上面的代码中，如果isEnabled为true，类名将是enabled，如果isEnabled为false，类名将是disabled。

这种语法还有两外一种用法，当条件为假时，增加一个类名，当条件为真时，则移除类名：

```
<div {{bindAttr class="isEnabled::disabled"}}>
  Warning!
</div>
```

当isEnabled为false时，会增加一个disabled类名，当isEnabled为true时，这里不会出现类名。

静态类

如果在元素里，需要同时运用静态类名和绑定类名，则需要将静态类名放在绑定类名的列表里，并且在前面加一个冒号：

```
<div {{bindAttr class=":high-priority isUrgent"}}>
  Warning!
</div>
```

类名high-priority会直接添加到元素里：

```
<div class="high-priority is-urgent">
  Warning!
</div>
```

绑定类名和静态类名不能混合使用，下面的例子不会被正常渲染：

```
<div class="high-priority" {{bindAttr class="isUrgent"}}>
  Warning!
</div>
```

绑定多个类名

和其他属性不同的是，类名可以绑定多个：

```
<div {{bindAttr class="isUrgent priority"}}>
  Warning!
</div>
```

这段代码会按照上面说过的规则渲染：

```
<div class="is-urgent p4">
  Warning!
</div>
```

链接 `{{linkTo}}`标记

运用{{linkTo}}可以给路由创建链接：

```
App.Router.map(function() {
  this.resource("posts", function(){
    this.route("post", { path: "/:post_id" });
  });
});

<!-- posts.handlebars -->

<ul>
{{#each post in posts}}
  <li>{{#linkTo posts.post post}}{{post.title}}{{/linkTo}}</li>
{{/each}}
</ul>
```

如果posts模板的对象是三个posts的列表，HTML会被渲染成以下内容：

```
<ul>
  <li><a href="/posts/1">Infinity Madness</a></li>
  <li><a href="/posts/2">Hexadecimal Weirdness</a></li>
  <li><a href="/posts/3">Slashes!</a></li>
</ul>
```

{{linkTo}}的要素：

- 一个路由名称，上面的例子就可以使用index、posts或者post作为名称。

- 如果路由里有一个动态字段，该字段代表一个对象，默认情况下，Ember.js会用对象的id属性替换该字段的值。
- 可以选择设置title，该属性会绑定到链接的title属性上。

多个上下文

如果路由是嵌套的，可以为每一个动态字段设置一个模型。

```
App.Router.map(function() {
  this.resource("posts", function(){
    this.resource("post", { path: "/:post_id" }, function(){
      this.route("comments");
      this.route("comment", { path: "/comments/:comment_id" });
    });
  });
});
```

postIndex模板：

```
<div class="post">
  {{body}}
</div>

<p>{{#linkTo post.comment primaryComment}}Main Comment{{/linkTo}}</p>
```

由于只提供一个模型，该链接将继承当前post的动态字段：post_id。primaryComment将生成一个comment路由处理的新模型。

或者也可以将post和comment都放在标记里：

```
<p>
  {{#linkTo post.comment nextPost primaryComment}}
    Main Comment for the Next Post
  {{/linkTo}}
</p>
```

这种情况下，模型将由制定的:post_id和:comment_id填充，指定的nextPost将生成为post处理程序生成一个新对象，primaryComment也会为comment处理程序生成一个新对象。

当跳转到一个新的URL时，以下情况会让路由器执行处理程序：

- 处理程序更新
- 处理程序的模型发生变化

活动 `{{action}}`标记

一些情况下需要用简单的用户事件（比如点击）触发更高级的事件，此类事件可以操作控制器上的某些属性，通过绑定修改当前模板。

例如，有一个用于显示博客文章的模板，该模板还支持追加信息的扩展。

```
<!-- post.handlebars -->

<div class='intro'>
  {{intro}}
</div>

{{#if isExpanded}}
  <div class='body'>{{body}}</div>
  <button {{action "contract"}}>Contract</button>
{{else}}
  <button {{action "expand"}}>Show More...</button>
{{/if}}
```

在此例中，文章的控制器是Ember.ObjectController，其内容是App.Post的一个实例。

```
App.PostController = Ember.ObjectController.extend({
  // initial value
  isExpanded: false,

  expand: function() {
    this.set('isExpanded', true);
  },

  contract: function() {
    this.set('isExpanded', false);
  }
});
```

{{action}}标记默认触发当前控制器的方法，另外也可以给方法传递参数路径。在下面的例子中，点击事件将触发 `controller.select( context.post )` :

```
<p><button {{action "select" post}}>✓</button> {{post.title}}</p>
```

目标绑定

如果当前控制器没有找到相应的活动，事件将会冒泡到当前路由器，然后是父级路由器，最后是应用程序的路由。

在路由事件属性上定义活动：

```
App.PostsIndex = Ember.Route.extend({
  events: {
    myCoolAction: function() {
      // do your business.
    }
  }
});
```

此例是创建一个按钮，并根据用户在应用程序中所处的位置，实现不同的功能。例如，在侧边栏上有一个按钮，当用户处于/post的路由时，它可以完成一件事，当用户处于/about路由时，可以实现另一件事。

使用自定义标记

有时候可能需要在应用中多次使用相同的HTML内容，在这种情况下，可以注册一个可以在任意模板中使用的自定义标记。

假设经常要用标签包裹某些值，且标签都使用相同的类名，那么可以通过js注册一个标记：

```
Ember.Handlebars.registerBoundHelper('highlight', function(value, options) {
  escaped = Handlebars.Utils.escapeExpression(value);
  return new Handlebars.SafeString('<span class="highlight">' + escaped + '</span>');
});
```

如果要用标记返回一段HTML，且不希望内容被转义，那么要确保使用的是安全的字符，且要确保避开用户数据。

在Handlebars的任意位置，都可以调用标记：

```
{{highlight name}}
```

它将输出以下内容：

```
<span class="highlight">Peter</span>
```

如果当前的上下文环境中，name属性被修改，Ember.js将自动处理标记，并在DOM里更新数据。

归属

假设要创建一个 App.Person 的全名标记，且当实例发生变化时，或者firstName或lastName属性发生变化时，需要自动更新输出。

```
Ember.Handlebars.registerBoundHelper('fullName', function(person) {
  return person.get('firstName') + ' ' + person.get('lastName');
}, 'firstName', 'lastName');
```

该标记用法如下：

```
{{fullName person}}
```

此时，当上下文环境中的person对象发生变化，或者任何依赖的键发生变化时，输出都会自动更新。

传给fullName标记的路径，以及它依赖的键，都应该是属性的完整路径。（比如person.address.country）

回到页顶
xbingoz.com 首页

Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

3、路由 ROUTING

当用户操作应用的时候，应用会在若干个交互状态间运转。Ember.js提供了非常有效的工具，帮助开发者管理应用的状态。

为了更好地理解状态管理的重要性，可以设想一下，如果要设计一个管理博客的WEB应用，有些状态必须随时掌握：用户是否已经登录？用户是否是管理员？用户正在浏览那篇文章？是否打开了设置窗口？是否正在编辑当前文章？

在Ember.js中，任何一种状态都由URL表示，它们都被路由处理器封装在URL里，所以回答上面的问题会非常简单而且准确。

应用程序在任何时候都可能会有一个或多个活跃的路由处理器，它们会由于以下原因发生改变：

- 用户交互中产生一个修改URL的事件
- 用户手动改变URL（比如使用后退按钮），或者是页面第一次加载。

一旦URL发生改变，新的活跃路由会做出以下响应：

- 有条件的重定向到一个新的URL。
- 将控制器更新到一个特定的模型。
- 更新模板，或将新模板放到插口的位置。

记录路由变化

随着应用程序越来越复杂，开发者非常需要随时了解路由器的运行情况。只需对Ember.Application进行非常简单的修改，就可以让Ember打印将转换事件写入日志

```
App = Ember.Application.create({
  LOG_TRANSITIONS: true
});
```

定义路由器

当应用程序启动时，开发者定义的路由器会根据URL做出各种响应，比如显示模板、加载数据，以及其他一些应用中的设定状态。

```
App.Router.map(function() {
  this.route("about", { path: "/about" });
  this.route("favorites", { path: "/favs" });
});
```

当用户访问 `/`，Ember.js将渲染 index 模板。访问 `å/about` 渲染 about 模板， 而 `å/favs` 渲染 favorites 模板。

注意，如果路径和路由器的名称相同，可以不写路径，如下例所示：

```
App.Router.map(function() {
  this.route("about");
  this.route("favorites", { path: "/favs" });
});
```

在模板中可以使用{{linkTo}}指向路由器的名称，实现在路由之间的切换（`/` 的路由名是 index）

```
{{#linkTo "index"}}<img class="logo">{{/linkTo}}

<nav>
  {{#linkTo "about"}}About{{/linkTo}}
  {{#linkTo "favorites"}}Favorites{{/linkTo}}
</nav>
```

{{linkTo}}标记还会给链接增加active类名，指出当前所处的路由。

开发者可以通过创建Ember.Route的子类自定义一个路由行为，例如，通过创建App.IndexRoute定义用户访问 `/` 时会发生什么。

```
App.IndexRoute = Ember.Route.extend({
  setupController: function(controller) {
    // Set the IndexController's `title`
    controller.set('title', "My App");
  }
});
```

IndexController 是 index 模板的起始上下文，通过模板可以设置title：

```
<!-- get the title from the IndexController -->
<h1>{{title}}</h1>
```

如果不明确定义App.IndexController，Ember.js会自动生成一个。

Ember.js会通过路由器的名字，判断出相应的路由和控制器的名字：

URL	Route Name	Controller	Route	Template
/	index	IndexController	IndexRoute	index
/about	about	AboutController	AboutRoute	about
/favs	favorites	FavoritesController	FavoritesRoute	favorites

资源

通过resource可以定义一组路由的工作方式：

```
App.Router.map(function() {
  this.resource('posts', { path: '/posts' }, function() {
    this.route('new');
  });
});
```

和this.route一样，resource定义的路由如果和路径同名，也可以省略路径，如下所示：

```
App.Router.map(function() {
  this.resource('posts', function() {
    this.route('new');
  });
});
```

该路由器可以创建三个路由：

URL	Route Name	Controller	Route	Template
/	index	IndexController	IndexRoute	index
N/A	posts[1]	PostsController	PostsRoute	posts
/posts	posts.index	PostsController ↳PostsIndexController	PostsRoute ↳PostsIndexRoute	posts ↳posts/index
/posts/new	posts.new	PostsController ↳PostsNewController	PostsRoute ↳PostsNewRoute	posts ↳posts/new

注1：跳转到`posts`或者创建一个指向`posts`的链接，相当于跳转或链接到`posts.index`。

注意：如果通过this.resource创建一组资源，但是不提供任何方法,那么隐含的resource.index路由将不会被创建，/resource只能应用ResourceRoute, ResourceController, 以及 resource 模板.

如果路由器嵌套在资源当中，需要将资源名加它们的名字，构成路由的名字。如果要跳转到一个路由（使用transitionTo或者{{#linkTo}}），需要确保使用完整路由名（posts.new而不是new）。

访问 `/` 会直接渲染index模板。

访问 `/posts` 有点不同，它会首先渲染posts模板，会将posts/index模板渲染到posts模板的插口处。

最后，访问 `/posts/new` 会先渲染posts模板，然后将posts/new模板渲染到posts模板的插口处。

注意：this.resource是对URL来说是名词，而this.route对URL来说是修饰名称的动词或形容词。

动态字段

资源路由的作用之一是将URL转换到一个模型。

例如，有一个资源this.resource('/blog_posts')，那么路由可能就是这样的：

```
App.BlogPostsRoute = Ember.Route.extend({
  model: function() {
    return App.BlogPost.all();
  }
});
```

blog_posts模板会从当前上下文中获取全部posts的列表。

因为/blog_posts代表一个固定的模型，不需其他附加信息，就可以知道如何使用。但是，如果要让一个路由指向唯一的文章，可能无法将所有的路由和文章的映射关系手动编码。

输入动态字段。

动态字段是URL中的一部分，以冒号开头，后面是标示符。

```
App.Router.map(function() {
  this.resource('posts');
  this.resource('post', { path: '/posts/:post_id' });
});

App.PostRoute = Ember.Route.extend({
  model: function(params) {
    return App.Post.find(params.post_id);
  }
});
```

由于这种模式很常用，所以上面的模型钩子是默认行为。

例如，如果动态字段是post_id，Ember.js会很智能地通过URL确定ID，返回 App.Post 模型，具体来说，除非模型被覆盖，否则路由会自动返回App.Post.find(params.post_id)。这正是Ember Data希望达到的效果，所以如果针对Ember Data应用Ember router，动态字段将如预期一般开箱即用。

嵌套资源

路由器不能嵌套，但是资源可以嵌套。

```
App.Router.map(function() {
  this.resource('post', { path: '/post/:post_id' }, function() {
    this.route('edit');
    this.resource('comments', function() {
      this.route('new');
    });
  });
});
```

此路由器会创建五个路由：

URL	Route Name	Controller	Route	Template
/	index	App.IndexController	App.IndexRoute	index
N/A	post	App.PostController	App.PostRoute	post
/post/:post_id[1]	post.index	App.PostIndexController	App.PostIndexRoute	post/index
/post/:post_id/edit	post.edit	App.PostEditController	App.PostEditRoute	post/edit
N/A	comments	App.CommentsController	App.CommentsRoute	comments
/post/:post_id/comments	comments.index	App.CommentsIndexController	App.CommentsIndexRoute	comments/index
/post/:post_id/comments/new	comments.new	App.CommentsNewController	App.CommentsNewRoute	comments/new

注1:post_id 就是文章的id，比如文章的id是1，路由就是/post/1

comments模板将在posts模板的插口位置渲染，所有comments下的模板(comments/index 和 comments/new)，将在comments模板的插口位置渲染。

指定一个路由的模型

在路由器中，一个URL会被关联到一个或多个路由处理程序，路由处理程序负责将URL转换到模型对象，告诉控制器呈现该模型，然后渲染绑定到控制器的模板。

专属模型

如果路由没有动态字段，可以通过硬编码的方式，明确路由的模型钩子，将模型和URL关联起来。

```
App.Router.map(function() {
  this.resource('posts');
});

App.PostsRoute = Ember.Route.extend({
  model: function() {
    return App.Post.find();
  }
});
```

默认情况下，模型钩子的值将被分配到posts控制器的model属性上，通过设置setupControllers钩子可以改变这种行为。posts控制器是posts模板的上下文。

动态模型

如果路由有一个动态字段，就需要传入参数以明确使用的模型：

```
App.Router.map(function() {
  this.resource('post', { path: '/posts/:post_id' });
});

App.PostRoute = Ember.Route.extend({
  model: function(params) {
    return App.Post.find(params.post_id);
  }
});
```

由于这种模式很常用，所以上面的模型钩子是默认行为。

例如，如果动态字段是post_id，Ember.js会很智能地通过URL确定ID，返回 App.Post 模型，具体来说，除非模型被覆盖，否则路由会自动返回App.Post.find(params.post_id)。这正式Ember Data希望达到的效果，所以如果针对Ember Data应用Ember router，动态字段将如预期一般开箱即用。

设置控制器

改变URL可能也要更换显示的模板。模板可以有若干个数据源，但是一次只能用一个。

在Ember.js中，模板可以检索控制器中的信息进行呈现。

Ember.ObjectController和Ember.ArrayController是两个内置控制器，通过它们，可以很简单地将当前模型的属性传给模板，而不需要附加任何用于显示的代码。

设置路由处理程序的setupController钩子，可以为控制器指明当前模型。

```
App.Route.map(function() {
  this.resource('post', { path: '/posts/:post_id' });
});

App.PostRoute = Ember.Route.extend({
  setupController: function(controller, model) {
    controller.set('content', model);
  }
});
```

setupController接收路由处理程序匹配的控制器作为第一个参数，上面的例子里，PostRoute的setupController将接收一个App.PostController的实例作为参数。

第二个参数是路由处理程序的模型，详细信息参考定义路由模型的章节。

默认的setupController钩子会将控制器匹配的模型传给路由处理器的模型。

如果要在路由处理器匹配的模型之外，另外设置一个控制器，可以使用controllerFor方法：

```
App.PostRoute = Ember.Route.extend({
  setupController: function(controller, model) {
    this.controllerFor('topPost').set('content', model);
  }
});
```

渲染模板

路由处理程序的一个重要工作就是渲染适当的模板。

路由处理程序默认将模板渲染到最近的父级模板上。

```
App.Router.map(function() {
  this.resource('posts');
});

App.PostsRoute = Ember.Route.extend();
```

通过renderTemplate钩子，可以渲染另外一个模板，而不是路由处理程序匹配的模板：

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render('favoritePost');
  }
});
```

如果不用路由处理程序的控制器，而使用另外一个控制器，可以通过controller项进行设置：

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render({ controller: 'favoritePost' });
  }
});
```

如果要将模板渲染到另外一个插口：

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render({ outlet: 'posts' });
  }
});
```

上述所有设置项都可以合并到一起使用：

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    var controller = this.controllerFor('favoritePost');

    // Render the `favoritePost` template into
    // the outlet `posts`, and display the `favoritePost`
    // controller.
    this.render('favoritePost', {
      outlet: 'posts',
      controller: controller
    });
  }
});
```

URL重定向

如果要从一个路由定向到另一个路由，可以设置implement钩子。

```
App.Router.map(function() {
  this.resource('posts');
});

App.IndexRoute = Ember.Route.extend({
  redirect: function() {
    this.transitionTo('posts');
  }
});
```

基于应用的各种状态，可以有条件地进行转换。

```
App.Router.map(function() {
  this.resource('topCharts', function() {
    this.route('choose', { path: '/' });
    this.route('albums');
    this.route('songs');
    this.route('artists');
    this.route('playlists');
  });
});

App.TopChartsChooseRoute = Ember.Route.extend({
  redirect: function() {
    var lastFilter = this.controllerFor('application').get('lastFilter');
    this.transitionTo('topCharts.' + lastFilter || 'songs');
  }
});

// Superclass to be used by all of the filter routes below
App.FilterRoute = Ember.Route.extend({
  enter: function() {
    var controller = this.controllerFor('application');
    controller.set('lastFilter', this.templateName);
  }
});

App.TopChartsSongsRoute = App.FilterRoute.extend();
App.TopChartsAlbumsRoute = App.FilterRoute.extend();
App.TopChartsArtistsRoute = App.FilterRoute.extend();
App.TopChartsPlaylistsRoute = App.FilterRoute.extend();
```

此例中，访问 / 目录时会立即跳转到用户上一次访问的URL。如果这是第一次访问，则跳转到 /songs 。

路由也可以选择只在某些情况下跳转。如果没有重定向到一个新的路由，其他钩子（model, setupController, renderTemplates）会照常执行。

指定LOCATION API

路由器默认使用浏览器的hash加载应用的初始状态，并在运行期间同步。这种功能目前是依赖浏览器的hashchange事件。

下面的例子里，输入/#/posts/new会转到posts.new路由。

```
App.Router.map(function() {
  this.resource('posts', function() {
    this.route('new');
  });
});
```

如果希望可以后退到/posts/new，可以使用浏览器的history API：

```
App.Router.reopen({
  location: 'history'
});
```

最后，如果不想让应用和浏览器的URL发生交互，可以完全禁用location API，这种做法在测试时很有用。如果要用路由器管理应用状态，

且不像被URL妨碍（比如应用嵌入另外页面），也可以这样做。

回到页顶
xbingoz.com 首页

Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

4、控制器 CONTROLLERS

控制器可以完成以下工作：

- 为模板代理一个模型
- 存放不需要服务器保存的应用属性

控制器很小，在其他框架里，应用的状态会分散在很多控制器里，与此不同的是，Ember.js将这些状态封装在路由器里，这意味着控制器可以更轻量，且更专注一件事。

代理模型

模板应该连接到控制器，而不是连接到模型。这样易于从模型的属性中分离出特定的展示属性。

然而，如果模板能检索模型的属性，通常用起来会比较方便。Ember.js里的Ember.ObjectController和Ember.ArrayController可以轻易做到这一点。

保存应用属性

应用程序经常有一些不属于模型的信息需要存储，这时不一定非要保存到服务器，也可以保存到控制器里。

例如，假设应用程序里需要有一个搜索框，你可以使用一个有query属性的SearchController绑定搜索框。

用户搜索时，可以在搜索框里输入内容，然后敲击回车。

当路由器向服务器发送请求时，是从SearchController寻找搜索内容，而不是回到搜索框去找。

代理唯一模型

使用Ember.ObjectController可以代理唯一模型，在路由的setupController方法中，设置content属性，可以为Ember.ObjectController指明需要代理的模型。

如果模板在ObjectController中查找一个属性，ObjectController会先从自己的属性中寻找，如果定义了该名字的属性，就将当前值返回过去。

但是，如果控制器没有定义该属性，就会在模型中检索该属性，并返回属性值。

例如，假设要写一个乐手，通过SongController代理当前演奏的歌。

```
App.SongController = Ember.ObjectController.extend({
  soundVolume: 1
});
```

在路由器中，设置content，给控制器指明当前演奏的歌。

```
App.SongRoute = Ember.Route.extend({
  setupController: function(controller, song) {
    controller.set('content', song);
  }
});
```

在模板里，显示正在演奏的歌名和音量。

```
<p>
  <strong>Song</strong>: {{name}} by {{artist}}
</p>
<p>
  <strong>Current Volume</strong>: {{soundVolume}}
</p>
```

name和artist是需要持久化的信息，保存在模型当中，所以控制器要在模型中找到它们，然后传给模板。

soundVolume由当前代码定义，存在控制器中，控制器不需要查询模型，就可以直接把值返回出去。

这种体系的好处是，通过控制器访问模型的属性，可以更容易地开始浏览网页。但是，如果需要将模型的属性直接翻译给模板，有一点很明确，这里不存在向模型添加特定视图的问题。

例如，要显示歌曲的长短：

```
<p>
  <strong>Song</strong>: {{name}} by {{artist}}
</p>
<p>
  <strong>Duration</strong>: {{duration}}
</p>
```

duration是一个表示歌曲妙数的整数，直接输出会像下面这样：

```
<p>
  <strong>Song</strong>: 4 Minute Warning by Radiohead
</p>
<p>
  <strong>Duration</strong>: 257
</p>
```

然而用户是人而不是机器，所以更希望将时间格式化一下。通过控制器很容易就可以实现，控制器可以定义属性的计算，从而将模型的值转换成模板更适用的形式：

```
App.SongController = Ember.ObjectController.extend({
  duration: function() {
    var duration = this.get('content.duration'),
         minutes = Math.floor(duration / 60),
         seconds = duration % 60;

    return [minutes, seconds].join(':');
  }.property('content.duration')
});
```

现在，模板的输出变得更友好了：

```
<p>
  <strong>Song</strong>: 4 Minute Warning by Radiohead
</p>
<p>
  <strong>Duration</strong>: 4:17
</p>
```

代理多个模型

使用Ember.ArrayController可以代理多个模型组成的数组，通过setupController设置content属性，可以为ArrayController指明需要代理的模型。

ArrayController可以像一般的数组一样处理，例如，假设需要输出当前的歌曲列表，可以先在路由器里给SongsController设置歌曲列表。

```
App.SongsRoute = Ember.Route.extend({
  setupController: function(controller, playlist) {
    controller.set('content', playlist.get('songs'));
  }
});
```

然后在songs模板里使用{{#each}}标记，循环输出每一首歌曲：

```
<h1>Playlist</h1>

<ul>
  {{#each controller}}
    <li>{{name}} by {{artist}}</li>
  {{/each}}
</ul>
```

ArrayController还可以收集模型的信息，例如，要显示所有长度超过30秒的歌曲，可以给控制器增加一个名为longSongCount的属性计算方法：

```
App.SongsController = Ember.ArrayController.extend({
  longSongCount: function() {
    var longSongs = this.filter(function(song) {
      return song.get('duration') > 30;
    });
    return longSongs.get('length');
  }.property('@each.duration')
});
```

现在就可以在模板里运用该属性：

```
<ul>
  {{#each controller}}
    <li>{{name}} by {{artist}}</li>
  {{/each}}
</ul>

{{longSongCount}} songs over 30 seconds.
```

回到页顶
xbingoz.com 首页

Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

5、模型 MODELS

在大多数Ember.js应用中，模型是由Ember Data处理的。Ember Data是一个为Ember.js而建立的类库，可以用来检索服务器的数据记录，更新浏览器的变化，并将用户更改的信息存回服务器。

Ember Data提供了很多功能，就像服务器端的ActiveRecord一样，但是它主要针对浏览器环境的JavaScript设计。

无需任何设置，只要遵循一定的规则，Ember Data就可以通过RESTful JSON API和服务器进行通信，加载或保存数据。

目前有很多WEB服务API，但是很多是疯狂的、不一致的、难以控制的。Ember Data的目的是实现一般工作的固化管理。

目前Ember Data是Ember.js分离出的库，正在扩充API以支持更多功能。本节涉及的API都是稳定的。开发者可以到Github上获取Ember Data的完整版本。

待译

回到页顶
xbingoz.com 首页

Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

0. 入门
1. 应用程序 APPLICATION
2. 模板 TEMPLATES
3. 路由 ROUTING
4. 控制器 CONTROLLERS
5. 模型 MODELS（待译）
6. 视图 VIEWS
7. 对象模型 THE OBJECT MODEL
8. 枚举 ENUMERABLES
9. Ember.js配置 CONFIGURING EMBER.JS
10. 理解Ember.js UNDERSTANDING EMBER.JS（待译）

6、视图 VIEWS

视图简介

Ember.js使用的Handlebars模板系统非常强大，应用程序的大多数界面都可以通过Handlebars定义。如果开发者是从其他类库转过来，肯定会惊讶于Ember.js需要创建的视图是如此简洁。

Ember.js的视图通常只在以下情况时才创建：

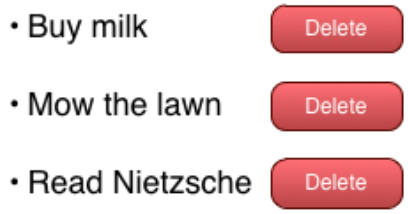
- 需要处理复杂的用户事件
- 构建可重用的组件

这两个条件往往会同时出现。

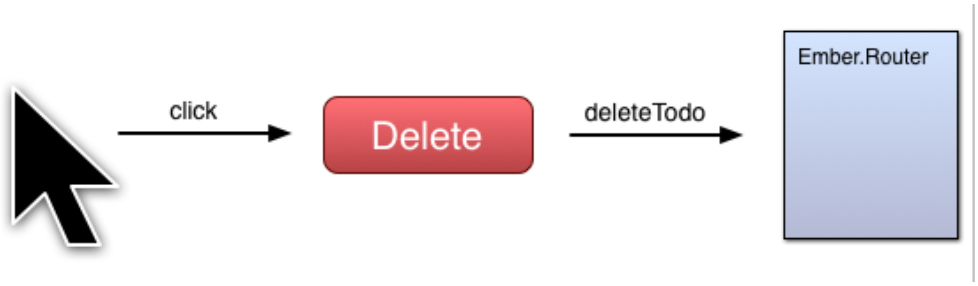
事件处理

视图在Ember.js应用中的任务，是将一个浏览器中的事件，转换成应用中定义的事件。

例如，假设有一个todo项目的列表，每个todo项目后面都有一个删除按钮。



视图负责将原始事件（click）转换成语义事件（删除todo项目）。语意事件将被发送给应用的路由器，路由器会根据当前的状态做出响应。



定义视图

通过Ember.View可以渲染一个Handlebars模板，并将模板放入DOM。

templateName属性代表视图需要渲染的模板，例如，如果有一个下面这样的script标签：

```
<html>
  <head>
    <script type="text/x-handlebars" data-template-name="say-hello">
      Hello, <b>{{view.name}}</b>
    </script>
  </head>
</html>
```

将templateName属性值设为"say-hello"。

```
var view = Ember.View.create({
  templateName: 'say-hello',
  name: "Bob"
});
```

注意：在其余的指南示例中，templateName属性一般会被省略。如果一个例子中包含Ember.View和一个handlebars模板，可以视为它们已经由templateName属性关联起来。

通过appendTo可以将视图放入页面。

```
view.appendTo('#container');
```

用简写可以直接追加到body中。

```
view.append();
```

用remove移除一个视图：

```
view.remove();
```

处理事件

在元素上注册事件监听做出响应的方式被取代，只需要在视图上注册一个方法，该方法与需要对应的事件使用相同的名字即可。

例如，假设有一个这样的模板：

```
{{#view App.ClickableView}}
This is a clickable area!
{{/view}}
```

设置App.ClickableView，点击时有弹窗出现：

```
App.ClickableView = Ember.View.extend({
  click: function(evt) {
    alert("ClickableView was clicked!");
  }
});
```

事件会从目标视图开始，一层层向上冒泡，直到根视图。这种冒泡是不可修改的。如果想在JS中手动管理视图，可以在handlebars里创建{{view}}标记，具体可以参看Ember.ContainerView的文档。

在模板中插入视图

到目前为止，我们讨论的都是为一个视图创建模板。然而随着应用的规模扩大，就需要给视图设计层次结构，封装在页面的不同区域。每个视图负责一部分事件，并维护一部分需要显示的属性。

{{VIEW}}

创建一个{{view}}标记，并指明视图类的路径，就可以创建一个子视图。

```
// Define parent view
App.UserView = Ember.View.extend({
  templateName: 'user',

  firstName: "Albert",
  lastName: "Hofmann"
});

// Define child view
App.InfoView = Ember.View.extend({
  templateName: 'info',

  posts: 25,
  hobbies: "Riding bicycles"
});

User: {{view.firstName}} {{view.lastName}}
{{view App.InfoView}}

<b>Posts:</b> {{view.posts}}
<br>
<b>Hobbies:</b> {{view.hobbies}}
```

如果实例化App.UserView并且渲染，就可以得到下面的内容：

```
User: Albert Hofmann
<div>
  <b>Posts:</b> 25
  <br>
  <b>Hobbies:</b> Riding bicycles
</div>
```

相对路径

定义子视图不一定要用绝对路径，也可以通过一个相对路径：相对于父视图的路径：

```
App.UserView = Ember.View.extend({
  templateName: 'user',

  firstName: "Albert",
  lastName: "Hofmann",

  infoView: Ember.View.extend({
    templateName: 'info',

    posts: 25,
    hobbies: "Riding bicycles"
  })
});

User: {{view.firstName}} {{view.lastName}}
{{view view.infoView}}
```

使用嵌套的视图类时，一定要使用小写字符。在Ember.js里，大写字母的属性会被解释为全局属性。

设置子视图模板

如果要在主模板里设置一个子视图的模板，可以在{{view}}标记的代码段里实现。所以上面的例子可以改成这样：

```
App.UserView = Ember.View.extend({
  templateName: 'user',

  firstName: "Albert",
  lastName: "Hofmann"
});

App.InfoView = Ember.View.extend({
  posts: 25,
  hobbies: "Riding bicycles"
});

User: {{view.firstName}} {{view.lastName}}
{{#view App.InfoView}}
  <b>Posts:</b> {{view.posts}}
  <br>
  <b>Hobbies:</b> {{view.hobbies}}
{{/view}}
```

这样做的好处是，有助于整理页面各部分视图的设置思路，开发者可以针对页面的一个部分，封装相应的事件处理。

给视图增加一个layout

视图可以有一个辅助模板，用于包裹主模板。和模板一样，layout也是handlebars模板，被放在view标签里。

设置layoutName属性，给模板指明layout模板。

设置{{yield}}标记，告诉layout模板插入主模板的位置。视图通过模板渲染的HTML会被放在{{yield}}标记的位置。

首先，定义一个layout模板：

```
<script type="text/x-handlebars" data-template-name="my_layout">
  <div class="content-wrapper">
    {{yield}}
  </div>
</script>
```

然后设置主模板：

```
<script type="text/x-handlebars" data-template-name="my_content">
  Hello, <b>{{view.name}}</b>!
</script>
```

最后定义视图，定义layout模板，设置模板的包裹结构。

```
AViewWithLayout = Ember.View.extend({
  name: 'Teddy',
  layoutName: 'my_layout',
  templateName: 'my_content'
});
```

最终渲染的HTML结果：

```
<div class="content-wrapper">
  Hello, Teddy!
</div>
```

layout模板的实践应用

如果一个视图拥有通用的包裹内容和通用的行为，且它的主模板经常需要更换，那么layout模板会非常有用。一种可能的情况是弹出视图。

首先定义弹出视图的layout模板：

```
<script type="text/x-handlebars" data-template-name="popup">
  <div class="popup">
    <button class="popup-dismiss">x</button>
    <div class="popup-content">
    {{yield}}
    </div>
  </div>
</script>
```

定义弹出视图：

```
App.PopupView = Ember.View.extend({
  layoutName: 'popup'
});
```

现在就可以让不同的模板重用弹出视图：

```
{{#view App.PopupView}}
  <form>
    <label for="name">Name:</label>
    <input id="name" type="text" />
  </form>
{{/view}}

{{#view App.PopupView}}
  <p> Thank you for signing up! </p>
{{/view}}
```

自定义视图元素

视图由一个DOM元素呈现在页面上，通过修改tagName属性可以更换呈现视图的元素。

```
App.MyView = Ember.View.extend({
  tagName: 'span'
});
```

通过classNames属性，可以给视图的元素设置样式。classNames属性的值是数组。

```
App.MyView = Ember.View.extend({
  classNames: ['my-view']
});
```

如果要用视图属性确定元素的类名，可以使用类名绑定，如果绑定的是一个布尔属性，可以实现类名的增加或删除。

```
App.MyView = Ember.View.extend({
  classNameBindings: ['isUrgent'],
  isUrgent: true
});
```

渲染结果大概是：

```
<div class="ember-view is-urgent">
```

如果isUrgent的值改为false，is-urgent类名将被删除。

默认情况下，布尔属性通过横线转换成类名，如果要明确定义一个类名，可以用冒号分隔：

```
App.MyView = Ember.View.extend({
  classNameBindings: ['isUrgent:urgent'],
  isUrgent: true
});
```

渲染出的HTML：

```
<div class="ember-view urgent">
```

除了用true来自定义类名，用false也同样可以：

```
App.MyView = Ember.View.extend({
  classNameBindings: ['isEnabled:enabled:disabled'],
  isEnabled: false
});
```

渲染结果：

```
<div class="ember-view disabled">
```

也可以只在false时定义类名：

```
App.MyView = Ember.View.extend({
  classNameBindings: ['isEnabled::disabled'],
  isEnabled: false
});
```

渲染结果：

```
<div class="ember-view disabled">
```

如果isEnabled的值改成true，渲染结果会变成：

```
<div class="ember-view">
```

如果绑定的属性值是字符串，该字符串会直接成为类名，而不会再经过修改：

```
App.MyView = Ember.View.extend({
  classNameBindings: ['priority'],
  priority: 'highestPriority'
});
```

渲染结果：

```
<div class="ember-view highestPriority">
```

给视图绑定属性

通过attributeBindings，可以给展示视图的DOM元素绑定属性。

```
App.MyView = Ember.View.extend({
  tagName: 'a',
  attributeBindings: ['href'],
  href: "http://emberjs.com"
});
```

通过HANDLEBARS自定义视图的元素

追加视图会创建一个HTML元素装载视图的内容。如果一个视图拥有子视图，则子视图都会以其子节点的形式呈现。

默认情况下，Ember.View的实例会创建div元素。通过tagName参数可以修改这种设置。

```
{{view App.InfoView tagName="span"}}
```

通过id属性，可以给视图的HTML元素设置id。

```
{{view App.InfoView id="info-view"}}
```

这样就可以通过CSS的选择器设置样式：

```
/** Give the view a red background. **/
#info-view {
  background-color: red;
}
```

也可以设置类名：

```
{{view App.InfoView class="info urgent"}}
```

通过classBinding可以将视图属性绑定成类名，属性名也可以使用同样的处理方法：

```
App.AlertView = Ember.View.extend({
  priority: "p4",
  isUrgent: true
});

{{view App.AlertView classBinding="isUrgent priority"}}
```

渲染结果大概是：

```
<div id="ember420" class="ember-view is-urgent p4"></div>
```

内建的视图元素

Ember.js预先设置了一些视图元素，用于一些基本的控制。例如文本输入、多选框、下来选项：

EMBER.CHECKBOX

```
<label>
  {{view Ember.Checkbox checkedBinding="content.isDone"}}
  {{content.title}}
</label>
```

EMBER.TEXTFIELD

```
App.MyText = Ember.TextField.extend({
  formBlurredBinding: 'App.adminController.formBlurred',
  change: function(evt) {
    this.set('formBlurred', true);
  }
});
```

EMBER.SELECT

```
{{view Ember.Select viewName="select"
                    contentBinding="App.peopleController"
                    optionLabelPath="content.fullName"
                    optionValuePath="content.id"
                    prompt="Pick a person:"
                    selectionBinding="App.selectedPersonController.person"}}
```

EMBER.TEXTAREA

```
var textArea = Ember.TextArea.create({
  valueBinding: 'TestObject.value'
});
```

使用EMBER.CONTAINERVIEW管理视图

在视图中创建子视图通常是用{{view}}标记完成。有时要用另一个手动管理子视图的方法：创建Ember.ContainerView实例，该实例会创建一个可编辑的childViews数组，在该数组中增删子视图，都会自动在DOM中完成相应操作。

```
var container = Ember.ContainerView.create();
container.append();

var coolView = App.CoolView.create(),
    childViews = container.get('childViews');

childViews.pushObject(coolView);
```

还有一种简写方式，将子视图设置成Ember.ContainerView实例的属性：

```
var container = Ember.ContainerView.create({
  childViews: ['firstView', 'secondView'],

  firstView: App.FirstView,
  secondView: App.SecondView
});
```

回到页顶
xbingoz.com 首页

Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

7、对象模型 THE OBJECT MODEL

类和实例

Ember.Object的extend()方法可以创建一个新的类：

```
App.Person = Ember.Object.extend({
  say: function(thing) {
    alert(thing);
  }
});
```

上面新建的App.Person类有一个say()方法。

通过extend()方法可以给现有的类创建子类，比如，可以给内建的Ember.View类创建子类：

```
App.PersonView = Ember.View.extend({
  tagName: 'li',
  classNameBindings: ['isAdministrator']
});
```

定义子类的时候可以覆盖父类的方法，但是通过_super()方法可以重新调用被覆盖的父类方法：

```
App.Person = Ember.Object.extend({
  say: function(thing) {
    var name = this.get('name');

    alert(name + " says: " + thing);
  }
});

App.Soldier = App.Person.extend({
  say: function(thing) {
    this._super(thing + ", sir!");
  }
});

var yehuda = App.Soldier.create({
  name: "Yehuda Katz"
});

yehuda.say("Yes");
// alerts "Yehuda Katz says: Yes, sir!"
```

创建实例

定义一个类后，可以用create()方法创建类的实例。类里定义的方法、属性、属性计算，都可以在实例里使用。

```
var person = Person.create();
person.say("Hello") // alerts "Hello"
```

通过给create()方法传递hash设置，可以给实例的属性设置初始值。

```
Person = Ember.Object.extend({
  helloWorld: function() {
    alert("Hi, my name is " + this.get('name'));
  }
});

var tom = Person.create({
  name: "Tom Dale"
});

tom.helloWorld() // alerts "Hi my name is Tom Dale"
```

由于性能的原因，在使用create()方法创建实例的时候，不能再定义任何方法和属性计算，在create()里只能定义属性。如果要增加方法或者属性计算，可以定义一个子类，然后再实例化。

通常情况下，指向类的属性或变量应该首字母大写，而实例不用这样做。例如，Person是一个类，而person是一个实例。在Ember.js的应用程序中，最好坚持这种命名方式。

创建一个新实例的时候，init()方法会被自动调用。所以在这里实例进行设置会非常理想。

```
Person = Ember.Object.extend({
  init: function() {
    var name = this.get('name');
    alert(name + ", reporting for duty!");
  }
});

Person.create({
  name: "Stefan Penner"
});

// alerts "Stefan Penner, reporting for duty!"
```

如果给一个内建的类创建子类，比如Ember.View或Ember.ArrayController，并且给子类设置了新的init()方法，那么一定要在init里调用this._super()方法。否则系统无法进行一些设置，应用就会出现一些怪异的情况。

访问一个对象的属性，请使用get和set方法：

```
var person = App.Person.create();

var name = person.get('name');
person.set('name', "Tobias Fünke");
```

务必使用这些访问方法，否则属性计算不会执行，观察者不会触发，模板无法更新。

观察者

Ember.js对所有属性都支持观察方法，包括属性计算。通过addObserver方法可以给一个对象设置观察者。

```
Person = Ember.Object.extend({
  // these will be supplied by `create`
  firstName: null,
  lastName: null,

  fullName: function() {
    var firstName = this.get('firstName');
    var lastName = this.get('lastName');

    return firstName + ' ' + lastName;
  }.property('firstName', 'lastName')
});

var person = Person.create({
  firstName: "Yehuda",
  lastName: "Katz"
});

person.addObserver('fullName', function() {
  // deal with the change
});

person.set('firstName', "Brohuda"); // observer will fire
```

由于fullName是基于firstName计算出来的，所以更新firstName，会触发fullName的观察者。

由于观察者很常见，所以Ember.js提供一个定义类时设置内联观察者的方法：

```
Person.reopen({
  fullNameChanged: function() {
    // this is an inline version of .addObserver
  }.observes('fullName')
});
```

如果没有对Ember的原型做扩展，可以用Ember.observer方法定义内联的观察者：

```
Person.reopen({
  fullNameChanged: Ember.observer(function() {
    // this is an inline version of .addObserver
  }, 'fullName')
});
```

属性计算

对象中的某些属性经常是由其他属性计算出来的。Ember.js的对象模型允许在定义类的时候定义属性计算的方法。

```
Person = Ember.Object.extend({
  // these will be supplied by `create`
  firstName: null,
  lastName: null,

  fullName: function() {
    var firstName = this.get('firstName');
    var lastName = this.get('lastName');

   return firstName + ' ' + lastName;
  }.property('firstName', 'lastName')
});

var tom = Person.create({
  firstName: "Tom",
  lastName: "Dale"
});

tom.get('fullName') // "Tom Dale"
```

属性计算是通过在属性上定义一个函数方法实现的，同时也要定义好依赖关系。当使用绑定和观察者时，这些依赖会发挥作用。

当创建一个类的子类时，可以覆盖属性计算。

访问属性计算

属性计算可以被访问，响应方法需要开发者定义。当访问属性计算时，会根据传入的键和值进行调用。

```
Person = Ember.Object.extend({
  // these will be supplied by `create`
  firstName: null,
  lastName: null,

  fullName: function(key, value) {
    // getter
    if (arguments.length === 1) {
      var firstName = this.get('firstName');
      var lastName = this.get('lastName');

      return firstName + ' ' + lastName;

    // setter
    } else {
      var name = value.split(" ");

      this.set('firstName', name[0]);
      this.set('lastName', name[1]);

      return value;
    }
  }.property('firstName', 'lastName')
});

var person = Person.create();
person.set('fullName', "Peter Wagenet");
person.get('firstName') // Peter
person.get('lastName') // Wagenet
```

Ember.js可以通过setter和getter调用属性计算，使用它们时需要检查参数数量，以明确是否可以使用。

属性计算和数据聚合

属性计算经常会依赖一个数组，通过数组的所有元素才能计算出值。例如一个todo列表，需要将控制器里的全部todo项目都拿出来，才能确定有多少已经完成。

此类属性计算可以这样处理：

```
App.todosController = Ember.Object.create({
  todos: [
    Ember.Object.create({ isDone: false })
  ],

  remaining: function() {
    var todos = this.get('todos');
    return todos.filterProperty('isDone', false).get('length');
  }.property('todos.@each.isDone')
});
```

注意这里的依赖(todos.@each.isDone)中包含一个特殊的@each，当以下四种情况出现时，它会指示Ember.js更新绑定或者触发观察者。

- todos中任何对象的isDone属性发生变化
- todos增加新元素
- todos删除元素
- 控制器中的todos属性指向其他数组

上面的例子里，remaining计算的结果是1：

```
App.todosController.get('remaining');
// 1
```

如果改变todos里元素的isDone属性，remaining属性会自动更新：

```
var todos = App.todosController.get('todos');
var todo = todos.objectAt(0);
todo.set('isDone', true);

App.todosController.get('remaining');
// 0

todo = Ember.Object.create({ isDone: false });
todos.pushObject(todo);

App.todosController.get('remaining');
// 1
```

绑定

绑定是在两个属性之间建立联系，当其中一个被改变时，另一个会自动更新到同样的值。绑定的属性可以在一个对象上，也可以在两个不同的对象上。与大多数支持绑定的框架不同，Ember.js可以对任何对象进行绑定，而不限于视图和模型之间。

创建绑定的最简单方法，是定义一个新的属性，该属性名以Binding结尾，然后给这个属性指明一个全局作用于中的路径：

```
App.wife = Ember.Object.create({
  householdIncome: 80000
});

App.husband = Ember.Object.create({
  householdIncomeBinding: 'App.wife.householdIncome'
});

App.husband.get('householdIncome'); // 80000

// Someone gets raise.
App.husband.set('householdIncome', 90000);
App.wife.get('householdIncome'); // 90000
```

注意，绑定不是立即更新。直到应用的所有代码跑完，Ember.js才会进行同步修改。所以，如果一个属性的值是临时的，那么它可以被修改很多次，而且不必担心同步修改的开销。

单向绑定

单向绑定仅在一个方向上传播变化，相比于双向绑定，单向绑定的性能通常更好，语法也更简洁。（如果只改变一侧的值，其实双向绑定也是一种单向绑定。）

```
App.user = Ember.Object.create({
  fullName: "Kara Gates"
});

App.userView = Ember.View.create({
  userNameBinding: Ember.Binding.oneWay('App.user.fullName')
});

// Changing the name of the user object changes
// the value on the view.
App.user.set('fullName', "Krang Gates");
// App.userView.userName will become "Krang Gates"

// ...but changes to the view don't make it back to
// the object.
App.userView.set('userName', "Truckasaurus Gates");
App.user.get('fullName'); // "Krang Gates"
```

重启类和实例

定义类的时候，不必一次性完成全部定义。通过reopen方法可以重新开启类，并追加新的属性。

```
Person.reopen({
  isPerson: true
});

Person.create().get('isPerson') // true
```

使用reopen时，可以覆盖已存在的属性，也可以调用this.super方法。

```
Person.reopen({
  // override `say` to add an ! at the end
  say: function(thing) {
    this._super(thing + "!");
  }
});
```

reopen用来给实例增加属性和方法，如果需要给类本身增加方法，需要使用reopenClass：

```
Person.reopenClass({
  createMan: function() {
    return Person.create({isMan: true})
  }
});

Person.createMan().get('isMan') // true
```

绑定、观察者、属性计算的应用场景

初学者经常会被绑定、观察者、属性计算的使用场景所困扰，下面是一些帮助理解的指引：

- 属性计算的作用是将一些属性合并成一个新属性，它不应该包含应用程序的行为，也不能在调用时产生其他作用。除了极少数情况，多次调用属性计算的结果应该保持一致。（除非依赖的属性发生变化）。
- 观察者包含的内容是，当一个属性发生变化时，应该做出的响应行为。当绑定完成同步后需要触发一些行为时，观察者会非常有用。
- 绑定通常用来让两个不同层面的对象保持同步。比如，通过Handlebars将视图和控制器绑定在一起。

回到页顶
xbingoz.com 首页

Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

8、枚举 ENUMERABLES

在Ember.js里，包含子对象的任何对象都是可枚举的，通过Ember.Enumerable API可以操作这些子对象。在大多数应用中，原生的JS数组拥有最常见的可枚举性。Ember.js对其进行了扩展形成枚举接口。

通过处理枚举的标准接口，Ember.js可以彻底改变数据存储的方式，且不需要改变应用的其他部分。

例如，一个从开发处获取数据用于显示的列表，当切换到服务器获取数据时，视图、模板、控制器的代码完全不需要修改。

枚举API最大程度地符合ECMA标准，Ember.js通过浏览器原生提供的数组实现枚举，从而保持与其他类库的兼容。

例如，所有的枚举接口都支持forEach方法：

```
[1,2,3].forEach(function(item) {
  console.log(item);
});

//=> 1
//=> 2
//=> 3
```

通常，forEach之类的枚举方法都接受第二个参数，该参数指向回调函数中的this：

```
var array = [1,2,3];

array.forEach(function(item) {
  console.log(item, this.indexOf(item));
}, array)

//=> 1 0
//=> 2 1
//=> 3 2
```

Ember.js中的枚举

列表的对象通常实现了枚举接口，比如：

- **数组** - Ember通过枚举接口扩展了js的原生数组。
- **Ember.ArrayController** - 包装基本数组的控制器，为视图层增加了附加功能。
- **Ember.Set** - 可以检查一个数据里是否包含对象。

API概览

这里只讨论最常见的枚举功能，更多详细内容可以参考API文档。

枚举迭代

使用forEach枚举对象里的所有值：

```
var food = ["Poi", "Ono", "Adobo Chicken"];

food.forEach(function(item, index) {
  console.log('Menu Item %@: %@'.fmt(index+1, item));
});

// Menu Item 1: Poi
// Menu Item 2: Ono
// Menu Item 3: Adobo Chicken
```

复制数组

使用toArray方法，支持Ember.Enumerable的对象都可以变成原生数组。

```
var states = Ember.Set.create();

states.add("Hawaii");
states.add("California")

states.toArray()
//=> ["Hawaii", "California"]
```

需要注意的是，许多枚举对象，例如上面例子里的Ember.Set，所得到的数组的顺序是不能确定的。

首尾对象

所有的枚举都可以使用firstObject和lastObject属性：

```
var animals = ["rooster", "pig"];

animals.get('lastObject');
//=> "pig"

animals.pushObject("peacock");

animals.get('lastObject');
//=> "peacock"
```

MAP

通过map()方法可以转换枚举中的所有元素，转换是通过回调函数对每个元素进行调用，返回的结果形成一个新的数组。

```
var words = ["goodbye", "cruel", "world"];

var emphaticWords = words.map(function(item) {
  return item + "!";
});
// ["goodbye!", "cruel!", "world!"]
```

如果枚举是由对象组成，有一个mapProperty()方法，可以将所有对象中的某个属性提取出来，形成一个新的数组。

```
var hawaii = Ember.Object.create({
  capital: "Honolulu"
});

var california = Ember.Object.create({
  capital: "Sacramento"
});

var states = [hawaii, california];

states.mapProperty('capital');
//=> ["Honolulu", "Sacramento"]
```

FILTERING

另一种常见的任务是将枚举作为输入，通过某些条件对项目进行过滤并返回数组。

filter方法可以处理任何过滤，项目传给该方法的回调函数，如果返回true则保留该项，如果为false则滤去该项。

```
var arr = [1,2,3,4,5];

arr.filter(function(item, index, self) {
  if (item < 4) { return true; }
})

// returns [1,2,3]
```

如果是处理一个Ember对象的集合，经常需要基于对象的某些属性值进行过滤。filterProperty是处理这种情况的捷径：

```
Todo = Ember.Object.extend({
  title: null,
  isDone: false
});

todos = [
  Todo.create({ title: 'Write code', isDone: true }),
  Todo.create({ title: 'Go to sleep' })
];

todos.filterProperty('isDone', true);

// returns an Array containing only items with `isDone == true`
```

如果只想返回第一个匹配的项目，而不是包含全部匹配项目的数组，可以使用find和findProperty方法，用法类似filter和filterProperty，但是只返回一个项目。

信息汇总（全部或者局部）

如果要检查枚举的所有元素是否都满足某些条件，可以使用every方法：

```
Person = Ember.Object.extend({
  name: null,
  isHappy: false
});

var people = [
  Person.create({ name: 'Yehuda', isHappy: true }),
  Person.create({ name: 'Majd', isHappy: false })
];

people.every(function(person, index, self) {
  if(person.get('isHappy')) { return true; }
});

// returns false
```

如果要判断枚举中是否至少有一个元素符合条件，可以用some方法：

```
people.some(function(person, index, self) {
  if(person.get('isHappy')) { return true; }
});

// returns true
```

和过滤方法相似，every和some也有对应的everProperty和someProperty方法：

```
people.everyProperty('isHappy', true) // false
people.someProperty('isHappy', true)  // true
```

回到页顶
xbingoz.com 首页

Ember.js官方指南 [译]

Ember.js确实非常强大，但是上手的难度也较高。Ember.js的官方指南做的还是挺详细的，对很多细节进行了解读，对于刚开始了解Ember.js的开发者会比较有帮助。

以下是官方指南的译文。 草稿，不断修正中。

目录

0. 入门
1. 应用程序 APPLICATION
2. 模板 TEMPLATES
3. 路由 ROUTING
4. 控制器 CONTROLLERS
5. 模型 MODELS（待译）
6. 视图 VIEWS
7. 对象模型 THE OBJECT MODEL
8. 枚举 ENUMERABLES
9. Ember.js配置 CONFIGURING EMBER.JS
10. 理解Ember.js UNDERSTANDING EMBER.JS（待译）

9、Ember.js配置

禁用原型扩展

默认情况下，Ember.js会通过以下方式对原生的js对象进行扩展：

- **Array** 扩展支持Ember.Enumerable, Ember.MutableEnumerable, Ember.MutableArray和Ember.Array接口。如果浏览器不支持ECMA5的数组方法会增加上去，并添加属性和方法，使数组拥有观察者。
- **String** 增加camelize()和fmt()之类的方法。
- **Function** 增加对property()方法的支持，扩展出处理属性计算的方法。增加对observes()方法的支持，扩展出处理观察者方法。

这些是对原生原型对象进行的扩展，我们已经对所有的原型变换进行了细致的权衡，并建议开发者使用它们。它们可以显著减少代码输入的数量。

然而我们也理解，Ember.js应用可能会嵌入到一个开发者不能完全控制的环境中。最常见的情况是创建一个嵌入其他页面的第三方js控件，或将将应用一块块地拆分到更现代的Ember.js框架中。

在上述情况中，开发者不能或不想修改原生原型，可以通过Ember.js禁用前面提供的扩展。

将EXTEND_PROTOTYPES标记为false就可以达到目的：

```
window.Ember = {};
Ember.EXTEND_PROTOTYPES = false;
```

注意，上面这段代码必须放在Ember.js文件加载之前。如果在设置禁用前已经加载了Ember.js，那么原生的原型已经被修改。

没有扩展的处理方法

为了保证应用程序运行正常，在禁用原生扩展的时候，需要在原生对象创建之前，手动扩展或创建对象。

数组

数组不支持基于观察者的功能，如果禁用了原生扩展，同时又将原生数组和{{#each}}之类的东西放在一起使用，Ember.js将无法监测到数组的变化，当数组变化时模板也就不会更新。

另外，如果将一个原生数组放进Ember.ArrayController的content里，由于原生数组不再支持Ember.Array接口，这里会抛出一个异常。

使用Ember.A方法，可以将一个原生数组强制转换成一个支持各种接口的数组：

```
var islands = ['Oahu', 'Kauai'];
islands.contains('Oahu');
//=> TypeError: Object Oahu,Kauai has no method 'contains'

// Convert `islands` to an array that implements the
// Ember enumerable and array interfaces
Ember.A(islands);

islands.contains('Oahu');
//=> true
```

字符串

字符串不再支持Ember.String API文档里定义的各种便捷方法，但是可以通过Ember.String调用相应的方法，并将需要使用该方法的字符串，作为第一个参数传进去。

```
"my_cool_class".camelize();
//=> TypeError: Object my_cool_class has no method 'camelize'

Ember.String.camelize("my_cool_class");
//=> "myCoolClass"
```

函数

针对属性计算，可以用Ember.computed()包装函数：

```
// This won't work:
fullName: function() {
  return this.get('firstName') + ' ' + this.get('lastName');
}.property('firstName', 'lastName')

// Instead, do this:
fullName: Ember.computed(function() {
  return this.get('firstName') + ' ' + this.get('lastName');
}).property('firstName', 'lastName')
```

观察者则通过Ember.observer()实现：

```
// This won't work:
fullNameDidChange: function() {
  console.log("Full name changed");
}.observes('fullName')

// Instead, do this:
fullNameDidChange: Ember.observer(function() {
  console.log("Full name changed");
}, 'fullName')
```

回到页顶