**Ember.js 是什么?**

Ember is a JavaScript framework for creating ambitious web applications that eliminates boilerplate and provides a standard application architecture.

Ember 是一个旨在创建非凡web应用的JavaScript框架，它消除了样板（boilerplate）并提供了标准的应用程序架构。

**消除样板**

There are some tasks that are common to every web application. For example, taking data from the server, rendering it to the screen, then updating that information when it changes.

每个web应用中都会有些常规的任务。例如，从服务器取数据、将数据渲染到屏幕、然后当数据变更时更新信息。

Since the tools provided to do this by the browser are quite primitive, you end up writing the same code over and over. Ember.js provides tools that let you focus on your app instead of writing the same code you've written a hundred times.

由于浏览器能够提供的完成该任务的工具太过原始，致使你一遍一遍地书写相同的代码。Ember.js提供的工具可以让你专注于你的app，而不是重复地编写已经写过无数次的代码。

Because we've built dozens of applications ourselves, we've gone beyond the obvious low-level event-driven abstractions, eliminating much of the boilerplate associated with propagating changes throughout your application, and especially into the DOM itself.

因为我们自己已经创建过数十个应用，我们已经远远超越了明显低级的事件驱动（event-driven）的抽象概念，消除了大量贯穿于应用当中的、与传播变化相关联的样板，尤其是DOM本身。

To help manage changes in the view, Ember.js comes with a templating engine that will automatically update the DOM when the underlying objects change.

为了帮助你管理视图（view）中的变更，Ember.js内置了一个模板引擎，当底层的对象发生改变时它可以自动地更新DOM。

For a simple example, consider this template of a Person:

作为一个简单的例子，考虑这个Person的模板:

|  |  |
| --- | --- |
|  | {{person.name}} is {{person.age}}. |

As with any templating system, when the template is initially rendered, it will reflect the current state of the person. To avoid boilerplate, though, Ember.js will also update the DOM automatically for you if the person's name or age changes.

与其他任何的模板系统类似，模板初始渲染时会影响到person的当前状态。然而，为了避免样板，当person的name更新时Ember.js也会为你自动更新DOM。

You specify your template once, and Ember.js makes sure it's always up to date.

一次性指定模板，Ember.js就会确保它实时更新。

**提供架构**

Since web applications evolved from web pages, which were nothing more than static documents, browsers give you just enough rope to hang yourself with.

由于web应用是由web页面演变而来，其不过是静态文档而已，浏览器已赋予你充分的能力去做任何想做的事。

Ember makes it easy to divide your application into models, views, and controllers, which improves testability, makes code more modular, and helps new developers on the project quickly understand how everything fits together. The days of callback spaghetti are over.

Ember可以让你很容易地将应用划分为模型（models），视图（views）和控制器（controllers），从而提高了可测性，使代码更加模块化，并且帮助项目中的新手快速理解各部分的结合原理。意面式的回调（callback spaghetti）时代结束了。（[译者](https://twitter.com/laserji)注：[Spaghetti code](http://en.wikipedia.org/wiki/Spaghetti_code)指具有复杂控制流程的代码。）

Ember also supplies built-in support for state management, so you'll have a way to describe how your application moves through various nested states (like signed-out, signed-in, viewing-post, and viewing-comment) out of the box.

Ember同样提供了内置的对状态管理的支持，从而可以描述你的应用程序在各种不同的嵌套状态（比如登入登出，浏览文章，查看评论）间的流转。

**Ember.js 有何不同？**

Traditional web applications make the user download a new page every time they interact with the server. This means that every interaction is never faster than the latency between you and the user, and usually slower. Using AJAX to replace only parts of the page helps somewhat, but still requires a roundtrip to your server every time your UI needs to update. And if multiple parts of the page need to update all at once, most developers just resort to loading the page over again, since keeping everything in sync is tricky.

传统的web应用迫使用户在每次与服务器交互时都得下载一个新页面。这意味着每次交互的时间不可能比你到用户之间的延时更快了，并且往往更慢。使用AJAX替换页面中的一些部分，在一定程度上起到了作用，但是在每次UI需要更新时，仍然需要一次到服务器的往返。并且如果页面中的许多部分需要同时更新，绝大部分开发者又会采取重新载入页面的方式了，因为若想保持每个部分都同步还真是个技术活。

Ember.js, like some other modern JavaScript frameworks, works a little differently. Instead of the majority of your application's logic living on the server, an Ember.js application downloads everything it needs to run in the initial page load. That means that while your user is using your app, she never has to load a new page and your UI responds quickly to their interaction.

与一些其他的现代JavaScript框架类似，Ember.js的工作方式略有不同。Ember.js应用会在页面初始载入时将所需全部加在，而不是将大多数应用逻辑存放于服务器端。这意味着当用户使用你的app时，她完全不必载入新页面，并且你的UI会快速响应交互操作。

One advantage of this architecture is that your web application uses the same REST API as your native apps or third-party clients. Back-end developers can focus on building a fast, reliable, and secure API server, and don't have to be front-end experts, too.

这种架构的一个优点就是，你的web应用与本地（native）应用或者第三方客户端使用相同的REST API。后端开发者可以专注于构建一个快速，稳定，安全的API服务器，并且也不需要前端经验。

**Ember.js 概览**

These are the three features that make Ember a joy to use:

三个法宝让你快乐地使用Ember：

* 绑定（Bindings）
* 计算属性（Computed properties）
* 自动更新模板（Auto-updating templates）

**绑定**

Use bindings to keep properties between two different objects in sync. You just declare a binding once, and Ember will make sure changes get propagated in either direction.

使用绑定可以使两个不同对象的属性保持同步。你只需一次性声明绑定，然后Ember就会确保变更的双向传播。

Here's how you create a binding between two objects:

下面展示了如何在两个对象间建立绑定：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10**  11 | MyApp.president = Ember.Object.create({  name: "Barack Obama"  });  MyApp.country = Ember.Object.create({  // Ending a property with 'Binding' tells Ember to  // create a binding to the presidentName property.  presidentNameBinding: 'MyApp.president.name'  });  // Later, after Ember has resolved bindings...  MyApp.country.get('presidentName');  // "Barack Obama" |

Bindings allow you to architect your application using the MVC (Model-View-Controller) pattern, then rest easy knowing that data will always flow correctly from layer to layer.

绑定允许你以MVC（Model-View-Controller）模式架构你的应用，然后即可高枕无忧了，数据将永远正确地在层与层之间流动。

**计算属性**

Computed properties allow you to treat a function like a property:

计算属性允许你将函数视作属性：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10** | MyApp.president = Ember.Object.create({  firstName: "Barack",  lastName: "Obama",  fullName: function() {  return this.get('firstName') + ' ' + this.get('lastName');  // Call this flag to mark the function as a property  }.property()  });  MyApp.president.get('fullName');  // "Barack Obama" |

Computed properties are useful because they can work with bindings, just like any other property.

计算属性非常有用，因为它可以像其他任何属性一样与绑定一起使用。

Many computed properties have dependencies on other properties. For example, in the above example, the fullName property depends on firstName and lastName to determine its value. You can tell Ember about these dependencies like this:

许多计算出的属性依赖于其他属性。例如，在上面的例子中，属性fullName依赖于firstName和lastName以确定其值。你可以像下面这样告诉Ember这些依赖关系：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | MyApp.president = Ember.Object.create({  firstName: "Barack",  lastName: "Obama",  fullName: function() {  return this.get('firstName') + ' ' + this.get('lastName');  // Tell Ember that this computed property depends on firstName  // and lastName  }.property('firstName', 'lastName')  }); |

Make sure you list these dependencies so Ember knows when to update bindings that connect to a computed property.

要确保列出这些依赖，这样Ember才能知道何时更新与计算属性相关联的绑定。

**自动更新模板**

Ember uses Handlebars, a semantic templating library. To take data from your JavaScript application and put it into the DOM, create a <script> tag and put it into your HTML, wherever you'd like the value to appear:

Ember使用Handlebars,一个语义化的模板库，用以从你的JavaScript应用中获取数据并将其放进DOM，创建一个<script>标签，然后把它放到任何你希望展示数据的地方：

|  |  |
| --- | --- |
| 1  2  3 | <script type="text/x-handlebars">  The President of the United States is {{MyApp.president.fullName}}.  </script> |

Here's the best part: templates are bindings-aware. That means that if you ever change the value of the property that you told us to display, we'll update it for you automatically. And because you've specified dependencies, changes to *those* properties are reflected as well.

最棒的在这儿呢：模板是可感知绑定（bindings-aware）的。意思就是如果你改变了已经告知我们需要显示的属性值，我们会为你自动更新。并且由于你指定了依赖关系，对*那些*属性的更改也会体现出来。

Hopefully you can see how all three of these powerful tools work together: start with some primitive properties, then start building up more sophisticated properties and their dependencies using computed properties. Once you've described the data, you only have to say how it gets displayed once, and Ember takes care of the rest. It doesn't matter how the underlying data changes, whether from an XHR request or the user performing an action; your user interface always stays up-to-date. This eliminates entire categories of edge cases that developers struggle with every day.

希望你能看到这三个强大的工具是如何配合工作的：从一些原始的属性开始，然后使用计算属性构造一些更复杂的属性及其依赖关系。一旦你描述出数据，就只需要一次性指明数据是如何显示的，然后Ember即可帮你完成其余的任务。底层数据是如何改变的并不重要，不管是从XHR请求还是用户执行操作，你的用户界面总会保持实时更新。这消除了一大类开发人员每天为之挣扎的边缘情况（译者注：此句有点别扭，请意会）。

**入门**

Depending on your needs, there are several ways to get started creating your first Ember.js app.

根据你的需求，有几种方式可以创建你的第一个Ember.js app.

If your needs are simple or you're interested in just playing around, you can download the Ember.js Starter Kit. The Starter Kit is based on [HTML5 Boilerplate](http://html5boilerplate.com/) and does not require any build tools or other dependencies. To begin, download the Starter Kit and unzip it. You can edit the Handlebars templates directly inside the index.html file, and the Ember.js app itself lives in javascripts/app.js.

如果你的需求较简单或者只是感兴趣随便玩玩，你可以下载Ember.js入门套件（Starter Kit）。该入门套件基于[HTML5 样板](http://html5boilerplate.com/)，无需任何构建工具、没有其他依赖。首先，下载入门套件并解压缩。你可以直接在index.html文件中编辑Handlebars模板，而Ember.js app本身在javascripts/app.js中。

For larger apps, you may want to consider using Ruby on Rails. Rails helps you manage your source code and other assets, while also providing the REST API that your application will talk to.

对于大型app，你可能会考虑使用Ruby on Rails，Rails会帮你管理源代码及其他资源，同时也提供了你的应用将会与之通讯的REST API。

**EMBER对象模型**

Ember enhances the simple JavaScript object model to support bindings and observers, as well as to support a more powerful mixin-based approach to code sharing.

Ember增强了简单的JavaScript对象模型，使之能够支持绑定和观察者，同时也支持一种更加强大的、基于混合的（mixin-based）代码共享途径。

At its most basic, you create a new Ember class by using the extend method on Ember.Object.

作为最基本的形式，你可以使用Ember.Object的extend方法创建一个新的Ember类。

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person = Ember.Object.extend({  say: function(thing) {  alert(thing);  }  }); |

Once you have built a new class, you can create new instances of the class by using the createmethod. Any properties defined on the class will be available to instances.

一旦你成功创建了一个新类，就可以使用create来创建类的实例了。类中定义的任何属性在实例中都是可用的。

|  |  |
| --- | --- |
| 1  2 | var person = Person.create();  person.say("Hello") // alerts "Hello" |

When creating an instance, you can also add additional properties to the instance by passing in an object.

创建实例时，也可以通过传入对象来为实例增添额外的属性。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var tom = Person.create({  name: "Tom Dale",  helloWorld: function() {  this.say("Hi my name is " + this.get('name'));  }  });  tom.helloWorld() // alerts "Hi my name is Tom Dale" |

Because of Ember's support for bindings and observers, you will always access properties using theget method, and set properties using the set method.

由于Ember支持绑定和观察者，因此你可以随时通过get方法访问属性，也可以通过set方法设置属性。

When creating a new instance of an object, you can also override any properties or methods defined on the class. For instance, in this case, you could override the say method from the Person class.

当创建一个对象的新的实例时，也可以覆写类中定义的任何属性和方法。在本例中，作为例子，你可以覆写从Person类继承的say方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var yehuda = Person.create({  name: "Yehuda Katz",  say: function(thing) {  var name = this.get('name');  this.\_super(name + " says: " + thing);  }  }); |

You can use the \_super method on the object (super is a reserved word in JavaScript) to call the original method you overrode.

你可以使用对象的\_super方法（super是JavaScript中的保留字）来调用被你覆写的原始方法。

**类的子类**

You can also create subclasses of classes you create by using the extend method. In fact, when we created a new class above by calling extend on Ember.Object, we were **subclassing**Ember.Object.

你也可以使用extend方法为类创建子类。事实上，我们上面使用Ember.Object对象的extend方法创建新类时，即是创建了Ember.Object的**子类**。

|  |  |
| --- | --- |
| 1  2  3  4  5 | var LoudPerson = Person.extend({  say: function(thing) {  this.\_super(thing.toUpperCase());  }  }); |

When subclassing, you can use this.\_super to invoke methods you are overriding.

当创建子类时，你可以使用this.\_super来调用被你覆写的方法。

**重新打开类和实例**

You don't need to define a class all at once. You can reopen a class and define new properties using thereopen method.

无需一次性将类定义完全，你可以使用reopen方法来重新打开（reopen）一个类并为其定义新的属性。

|  |  |
| --- | --- |
| 1  2  3  4 | Person.reopen({  isPerson: true  });  Person.create().get('isPerson') // true |

When using reopen, you can also override existing methods and call this.\_super.

当使用reopen时，你也同样可以覆写已经存在的方法并调用this.\_super。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Person.reopen({  // override `say` to add an ! at the end  say: function(thing) {  this.\_super(thing + "!");  }  }); |

As you can see, reopen is used to add properties and methods to an instance. But when you need to create class method or add the properties to the class itself you can use reopenClass.

正如你所见，reopen是用来为实例添加属性和方法的。而当你需要创建类的方法或为类本身添加属性时，则可使用reopenClass。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Person.reopenClass({  createMan: function() {  return Person.create({isMan: true})  }  });  Person.createMan().get('isMan') // true |

**计算属性 （Getters）**

Often, you will want a property that is computed based on other properties. Ember's object model allows you to define computed properties easily in a normal class definition.

你可能经常需要一个基于其他属性计算而来的属性。Ember的对象模型可以使你很轻松地在常规的类定义中定义计算属性。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10**  11  12  13  14  15 | 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" |

The property method defines the function as a computed property, and defines its dependencies. Those dependencies will come into play later when we discuss bindings and observers.

property方法将函数（function）定义为一个计算属性，并且定义了它的依赖关系。这些依赖在我们待会儿讨论绑定和观察者时会提到。

When subclassing a class or creating a new instance, you can override any computed properties.

当为类创建子类或者新的实例时，你可以覆写任何计算属性。

**计算属性 （Setters）**

You can also define what Ember should do when setting a computed property. If you try to set a computed property, it will be invoked with the key and value you want to set it to.

你也可以定义当设置计算属性时Ember所执行的动作。当你尝试设置计算属性时，可以使用对应的名-值（key and value）对其进行设置。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10**  11  12  13  14  15  16  17  18  19  **20**  21  22  23 | 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 will call the computed property for both setters and getters, and you can check the number of arguments to determine whether it is being called as a getter or a setter.

对于setter和getter，Ember都会调用计算属性，你可以通过检查参数的个数来确定该计算属性究竟是被getter还是setter调用的。

**观察者**

Ember supports observing any property, including computed properties. You can set up an observer on an object by using the addObserver method.

Ember支持观察任何属性，包括计算属性。你可以使用addObserver方法在某个对象上设置观察者。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10**  11  12  13  14  15  16  17  18 | 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 |

Because the fullName computed property depends on firstName, updating firstName will fire observers on fullName as well.

由于fullName这个计算属性依赖于firstName，所以更新firstName也会触发fullName上的观察者。

Because observers are so common, Ember provides a way to define observers inline in class definitions.

由于观察者的应用非常普遍，Ember提供了一种能够在类定义的内部定义观察者的方式。

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person.reopen({  fullNameChanged: function() {  // this is an inline version of .addObserver  }.observes('fullName')  }); |

You can define inline observers by using the Ember.observer method if you are using Ember without prototype extensions:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person.reopen({  fullNameChanged: Ember.observer(function() {  // this is an inline version of .addObserver  }, 'fullName')  }); |

**以数组形式的改变**

Often, you may have a computed property that relies on all of the items in an array to determine its value. For example, you may want to count all of the todo items in a controller to determine how many of them are completed.

通常，你可能用到某个计算属性，它的值是由某个数组中的全部项来决定的。例如，你可能需要计算出某个控制器（controller）中全部的待办事项，以此来决定它们当中有多少是完成了的。

Here's what that computed property might look like:

下面是这个计算属性可能的样子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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')  }); |

Note here that the dependent key (todos.@each.isDone) contains the special key @each. This instructs Ember.js to update bindings and fire observers for this computed property when one of the following four events occurs:

注意这里的依赖键值（dependent key）（todos.@each.isDone）包含了特殊的键@each。这个是告诉Ember.js，当下面四个事件之一发生时，更新绑定并且触发这个计算属性的观察者。

* The isDone property of any of the objects in the todos array changes.
* 在todos中的任意对象的isDone属性发生改变时。
* An item is added to the todos array.
* todos数组中增添了一项。
* An item is removed from the todos array.
* todos数组中移除了一项。
* The todos property of the controller is changed to a different array.
* 控制器的todos属性被替换为不同的数组。

In the example above, the remaining count is 1:

在上面的例子中，remaining的计算结果为1：

|  |  |
| --- | --- |
| 1  2 | App.todosController.get('remaining');  // 1 |

If we change the todo's isDone property, the remaining property is updated automatically:

如果我们改变待办事项的isDone属性，则remaining属性就会自动更新。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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**

A binding creates a link between two properties such that when one changes, the other one is updated to the new value automatically. Bindings can connect properties on the same object, or across two different objects. Unlike most other frameworks that include some sort of binding implementation, bindings in Ember.js can be used with any object, not just between views and models.

绑定在两个属性之间创建了一个连接，这样一个属性改变时另一个也可以随之自动更新到最新的值。绑定也可以连接同一对象内的属性，或者跨越两个不同的对象。与其他大部分框架所包含的绑定实现不同的是，Ember.js的绑定可以用在任何对象上，而不仅仅用在视图和模型之间。

The easiest way to create a two-way binding is by creating a new property with the string Binding at the end, then specifying a path from the global scope:

最简单的创建双向绑定的方法是，创建一个以Binding字串结尾的属性，然后指定一个相对于全局作用域（global scope）的路径：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10** | 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 |

Note that bindings don't update immediately. Ember waits until all of your application code has finished running before synchronizing changes, so you can change a bound property as many times as you'd like without worrying about the overhead of syncing bindings when values are transient.

注意这个绑定不会立即更新，Ember会等待应用的所有代码运行完成之后再同步变更，因此你可以随意改变某个绑定的属性，而不用担心同步这些临时值所耗费的开销。

**单向绑定**

A one-way binding only propagates changes in one direction. Usually, one-way bindings are just a performance optimization and you can safely use the more concise two-way binding syntax (as, of course, two-way bindings are de facto one-way bindings if you only ever change one side).

单向绑定只将变更单向传播。通常，单向绑定只是为了性能优化，你可以放心地使用更加简洁的双向绑定语法（当然，如果你总是只改变一边的话，那么双向绑定事实上也是单向绑定了）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10**  11  12  13  14 | 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" |

**何时用何物？**

Sometimes new users are confused about when to use computed properties, bindings and observers. Here are some guidelines to help:

有时新用户对于何时使用计算属性、绑定和观察者很困惑。下面是一些帮助提示：

* Use *computed properties* to build a new property by synthesizing other properties. Computed properties should not contain application behavior, and should generally not cause any side-effects when called. Except in rare cases, multiple calls to the same computed property should always return the same value (unless the properties it depends on have changed, of course.)
* 使用*计算属性*创建由其他属性综合决定的新属性。计算属性不应该包含应用的行为，而且当调用时不应有任何附加影响。除非在极少数情况下，多次调用相同的计算属性应该总是返回同样的值（当然，除了属性所依赖的属性值改变的情况。）
* *Observers* should contain behavior that reacts to changes in another property. Observers are especially useful when you need to perform some behavior after a binding has finished synchronizing.
* *观察者*应该包含反映其他属性变化的行为。当你需要在绑定完成同步之后执行一些行为的时候，观察者会非常有用。
* *Bindings* are most often used to ensure objects in two different layers are always in sync. For example, you bind your views to your controller using Handlebars. You may often bind between two objects in the same layer. For example, you might have an App.selectedContactControllerthat binds to the selectedContact property of App.contactsController.
* *绑定*通常用在确保位于不同层的对象总能保持同步。例如，你使用Handlebars绑定了你的视图和控制器，你也可能经常需要绑定同一层的两个对象。例如，你可能有个App.selectedContactController需要绑定在App.contactsController的selectedContact属性上。

**创建命名空间**

Every Ember app should have an instance of Ember.Application. This object will serve as the globally-accessible namespace for all of the other classes and instances in your app. Additionally, it sets up event listeners on the page so that your views receive events when users interact with your user interface (which you'll learn about later.)

每个Ember app都应该有一个Ember.Application的实例。这个对象将会作为你的app中所有类和实例的全局可访问的命名空间（globally-accessible namespace）。此外，它也在页面上设置了事件监听器，确保当用户与你的用户界面进行交互时你的视图可以接收到事件（稍后你会了解到这些）。

Here's an example of an application:

这里是一个应用的例子：

|  |  |
| --- | --- |
|  | window.App = Ember.Application.create(); |

You can call your namespace whatever you'd like, but it must begin with a capital letter in order for bindings to find it.

你可以随意为你的命名空间命名，不过必须以大写字母开头，这样绑定才能找到它。

If you are embedding an Ember application into an existing site, you can have event listeners set up for a specific element by providing a rootElement property:

如果要将Ember应用嵌入到某个现存的网站中，可以通过提供rootElement属性来将某个特定的元素作为事件监听器。

|  |  |
| --- | --- |
| 1  2  3 | window.App = Ember.Application.create({  rootElement: '#sidebar'  }); |

**使用HANDLEBARS描述你的UI**

**Handlebars**

Ember comes bundled with [Handlebars](http://www.handlebarsjs.com/), a semantic templating language. These templates look like regular HTML, with embedded expressions.

Ember捆绑了[Handlebars](http://www.handlebarsjs.com/)，一种语义化的模板语言。这些模板看上去像是普通的HTML嵌入了表达式。

You should store your Handlebars templates inside your application's HTML file. At runtime, Ember will compile these templates so they are available for you to use in your views.

你应该把你的Handlebars模板存放在应用的HTML文件中。在运行时，Ember会编译这些模板，确保你可以在视图中使用它们。

To immediately insert a template into your document, place it inside a <script> tag within your <body>tag:

要想快速向文档中插入模板，可以将模板放入一对<script>标签，并插入<body>标签中。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <html>  <body>  <script type="text/x-handlebars">  Hello, <b>{{MyApp.name}}</b>  </script>  </body>  </html> |

To make a template available to be used later, give the <script> tag a data-template-name attribute:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <html>  <head>  <script type="text/x-handlebars" data-template-name="say-hello">  Hello, <b>{{MyApp.name}}</b>  </script>  </head>  </html> |

**Ember.View**

You can use Ember.View to render a Handlebars template and insert it into the DOM.

你可以使用Ember.View来渲染Handlebars模板并将其插入DOM。

To tell the view which template to use, set its templateName property. For example, if I had a <script>tag like this:

要想让视图知道到底该使用哪个模板，可以设置它的templateName属性。例如，如果我有个这样的<script>标签：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <html>  <head>  <script type="text/x-handlebars" data-template-name="say-hello">  Hello, <b>{{name}}</b>  </script>  </head>  </html> |

I would set the templateName property to "say-hello".

我会把templateName属性设置为"say-hello"。

|  |  |
| --- | --- |
| 1  2  3  4 | var view = Ember.View.create({  templateName: 'say-hello',  name: "Bob"  }); |

Note: For the remainder of the guide, the templateName property will be omitted from most examples. You can assume that if we show a code sample that includes an Ember.View and a Handlebars template, the view has been configured to display that template via the templateName property.

注意：作为本指南的提示，大部分例子都会省略templateName属性。你可以假定当我们展示包含Ember.View和Handlebars模板的代码例子时，视图已经通过配置templateName属性来显示指定的模板了。

You can append views to the document by calling appendTo:

你可以通过调用appendTo来像文档中追加视图：

|  |  |
| --- | --- |
|  | view.appendTo('#container'); |

As a shorthand, you can append a view to the document body by calling append:

作为简写，你可以调用append向body中追加视图：

|  |  |
| --- | --- |
|  | view.append(); |

To remove a view from the document, call remove:

调用remove可以从文档中移除视图：

|  |  |
| --- | --- |
|  | view.remove(); |

**Handlebars基础**

As you've already seen, you can print the value of a property by enclosing it in a Handlebars expression, or a series of braces, like this:

正如你所看到的，你可以使用Handlebars表达式、或者系列括号包含属性的值来将它打印出来，就像这样：

|  |  |
| --- | --- |
|  | My new car is {{color}}. |

This will look up and print the View's color property. For example, if your view looks like this:

这将会查找并打印视图的color属性。例如，如果你的视图看起来像这样：

|  |  |
| --- | --- |
| 1  2  3 | App.CarView = Ember.View.extend({  color: 'blue'  }); |

Your view would appear in the browser like this:

则你的视图在浏览器中会表现为：

|  |  |
| --- | --- |
|  | My new car is blue. |

You can also specify global paths:

你也可以指定全局路径：

|  |  |
| --- | --- |
|  | My new car is {{App.carController.color}}. |

(Ember determines whether a path is global or relative to the view by checking whether the first letter is capitalized, which is why your Ember.Application instance should start with a capital letter.)

（Ember依靠首字母是否大写来确定某个路径是全局的还是相对于当前视图的，这也是为什么你的Ember.Application实例应该以大写字母开头的缘故。）

All of the features described in this guide are **bindings aware**. That means that if the values used by your templates ever change, your HTML will be updated automatically. It's like magic.

本指南中描述的所有特性都是**可感知绑定（bindings aware）**的，这意味着模板所使用的值一旦发生变化，你的HTML就会自动更新，就像魔法一样神奇。

In order to know which part of your HTML to update when an underlying property changes, Handlebars will insert marker elements with a unique ID. If you look at your application while it's running, you might notice these extra elements:

为了得知当底层的属性改变时HTML的哪个部分需要更新，Handlebars会插入带有惟一ID的标记元素。如果你在应用运行时查看，可能会注意到这些额外的元素：

|  |  |
| --- | --- |
| 1  2  3  4 | My new car is  <script id="metamorph-0-start" type="text/x-placeholder"></script>  blue  <script id="metamorph-0-end" type="text/x-placeholder"></script>. |

Because all Handlebars expressions are wrapped in these markers, make sure each HTML tag stays inside the same block. For example, you shouldn't do this:

由于所有的Handlebars表达式都被包裹在这些标记中，所以要确保每个HTML标签都在相同的区块中。例如，你不应这么做：

|  |  |
| --- | --- |
| 1  2 | <!-- Don't do it! -->  <div {{#if isUrgent}}class="urgent"{{/if}}> |

If you want to avoid your property output getting wrapped in these markers, use the unbound helper:

如果你想避免输出的属性被包裹在这些标记中，可以使用unbound助手（helper）：

|  |  |
| --- | --- |
|  | My new car is {{unbound color}}. |

Your output will be free of markers, but be careful, because the output won't be automatically updated!

这样你的输出就不会有这些标记了，不过要小心，因为这些输出将不再自动更新！

|  |  |
| --- | --- |
|  | My new car is blue. |

**{{#if}}，{{else}}，和{{#unless}}**

Sometimes you may only want to display part of your template if a property exists. For example, let's say we have a view with a person property that contains an object with firstName and lastName fields:

有时你可能只想在属性存在时显示部分的模板。例如，假设有个含有person属性的视图包含了一个拥有firstName和lastName字段的对象：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | App.SayHelloView = Ember.View.extend({  person: Ember.Object.create({  firstName: "Joy",  lastName: "Clojure"  })  }); |

In order to display part of the template only if the person object exists, we can use the {{#if}} helper to conditionally render a block:

为了只在person对象存在时显示部分模板，我们可以使用{{#if}}助手来依条件渲染某个区块：

|  |  |
| --- | --- |
| 1  2  3 | {{#if person}}  Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!  {{/if}} |

Handlebars will not render the block if the argument passed evaluates to false, undefined, null or[] (i.e., any "falsy" value).

如果传入的参数为false、undefined、null或者[]（等等任何“假的”值）的话，Handlebars就不会渲染这个区块。

If the expression evaluates to falsy, we can also display an alternate template using {{else}}:

如果表达式执行的结果为假的（falsy），我们也可以使用{{else}}来显示一个替代的模板：

|  |  |
| --- | --- |
| 1  2  3  4  5 | {{#if person}}  Welcome back, <b>{{person.firstName}} {{person.lastName}}</b>!  {{else}}  Please log in.  {{/if}} |

To only render a block if a value is falsy, use {{#unless}}:

若想仅当某个值为假时才渲染区块，可以使用{{#unless}}：

|  |  |
| --- | --- |
| 1  2  3 | {{#unless hasPaid}}  You owe: ${{total}}  {{/unless}} |

{{#if}} and {{#unless}} are examples of block expressions. These allow you to invoke a helper with a portion of your template. Block expressions look like normal expressions except that they contain a hash (#) before the helper name, and require a closing expression.

{{#if}}和{{#unless}}是区块表达式（block expressions）的例子。这些允许你在模板的一部分上调用助手（helper）。区块表达式跟正常的表达式类似，只是他们在助手名前面包含一个哈希符（#），并且需要结束表达式。

**{{#with}}**

Sometimes you may want to invoke a section of your template with a context different than the Ember.View. For example, we can clean up the above template by using the {{#with}} helper:

有时你可能想在Ember.View之外的上下文中调用模板中的某一节。例如，我们可以使用{{#with}}助手清理上面的模板：

|  |  |
| --- | --- |
| 1  2  3 | {{#with person}}  Welcome back, <b>{{firstName}} {{lastName}}</b>!  {{/with}} |

{{#with}} changes the *context* of the block you pass to it. The context is the object on which properties are looked up. By default, the context is the Ember.View to which the template belongs.

{{#with}}改变了你所传入区块的*上下文（context）*。上下文即我们需要在其上查找属性的对象，默认的上下文为模板所属的Ember.View。

**使用{{bindAttr}}绑定元素属性**

In addition to text, you may also want your templates to dictate the attributes of your HTML elements. For example, imagine a view that contains a URL:

除了文本外，你可能也希望模板可以掌控你的HTML元素。例如，想象一个包含URL的视图：

|  |  |
| --- | --- |
| 1  2  3 | App.LogoView = Ember.View.extend({  logoUrl: 'http://www.mycorp.com/images/logo.png'  }); |

The best way to display the URL as an image in Handlebars is like this:

使用Handlebars将URL显示为图片的最好方式为：

|  |  |
| --- | --- |
| 1  2  3 | <div id="logo">  <img {{bindAttr src="logoUrl"}} alt="Logo">  </div> |

这会生成如下HTML:

|  |  |
| --- | --- |
| 1  2  3 | <div id="logo">  <img src="http://www.mycorp.com/images/logo.png" alt="Logo">  </div> |

If you use {{bindAttr}} with a Boolean value, it will add or remove the specified attribute. For example, given this Ember view:

如果你将{{bindAttr}}和布尔值一起使用，它将会添加或删除特定的属性。例如，给定如下的Ember视图：

|  |  |
| --- | --- |
| 1  2  3 | App.InputView = Ember.View.extend({  isDisabled: true  }); |

And this template:

以及如下的模板：

|  |  |
| --- | --- |
|  | <input type="checkbox" {{bindAttr disabled="isDisabled"}}> |

Handlebars will produce the following HTML element:

Handlebars会产生如下的HTML元素：

|  |  |
| --- | --- |
|  | <input type="checkbox" disabled> |

**使用{{bindAttr}}绑定类名**

The class attribute can be bound like any other attribute, but it also has some additional special behavior. The default behavior works like you'd expect:

class属性可以像其他属性一样被绑定，不过它也有一些特殊行为。默认的行为如你所期待：

|  |  |
| --- | --- |
| 1  2  3  4 | App.AlertView = Ember.View.extend({  priority: "p4",  isUrgent: true  }); |
| 1  2  3 | <div {{bindAttr class="priority"}}>  Warning!  </div> |

This template will emit the following HTML:

这个模板会生成如下的HTML：

|  |  |
| --- | --- |
| 1  2  3 | <div class="p4">  Warning!  </div> |

If the value to which you bind is a Boolean, however, the dasherized version of that property will be applied as a class:

然而，如果你需要绑定的是个布尔值，则那个属性的将下划线替换为连字符后（译者注：dasherized，意为将所有的下划线“\_”替换为连字符“-”的操作）的版本则会作为一个类：

|  |  |
| --- | --- |
| 1  2  3 | <div {{bindAttr class="isUrgent"}}>  Warning!  </div> |

This emits the following HTML:

这将产生如下的HTML：

|  |  |
| --- | --- |
| 1  2  3 | <div class="is-urgent">  Warning!  </div> |

Unlike other attributes, you can also bind multiple classes:

与其他属性不同的是，你也可以绑定多个类：

|  |  |
| --- | --- |
| 1  2  3 | <div {{bindAttr class="isUrgent priority"}}>  Warning!  </div> |

You can also specify an alternate class name to use, instead of just dasherizing.

你也可以指定一个替代的类名，而不是简单地将下划线转换为连字符。

|  |  |
| --- | --- |
| 1  2  3 | <div {{bindAttr class="isUrgent:urgent"}}>  Warning!  </div> |

In this case, if the isUrgent property is true, the urgent class will be added. If it is false, the urgentclass will be removed.

在这种情况下，如果isUrgent属性为true，则urgent类将会被添加。如果为false，则urgent类将被移除。

You can also specify a class name which shall be used when the property is false:

你也可以指定一个当属性为false时所使用的类名：

|  |  |
| --- | --- |
| 1  2  3 | <div {{bindAttr class="isEnabled:enabled:disabled"}}>  Warning!  </div> |

In this case, if the isEnabled property is true, the enabled class will be added. If the property is false, the class disabled will be added.

在这种情况下，如果isEnabled属性为true，则enabled类将被添加。如果属性为false，则disabled类将被添加。

This syntax allows the shorthand for only adding a class when a property is false, so this:

如果仅当某个属性为false时才添加类的话，那么这种语法还支持简写形式，即：

|  |  |
| --- | --- |
| 1  2  3 | <div {{bindAttr class="isEnabled::disabled"}}>  Warning!  </div> |

Will add the class disabled when isEnabled is false and add no class if isEnabled is true.

当isEnabled为false时会添加disabled类，而当isEnabled为true时则不会添加任何类。

**使用{{action}}处理事件**

Use the {{action}} helper to attach a handler in your view class to an event triggered on an element.

使用{{action}}助手可以在你的视图类中将某个元素上触发的事件绑定到某个事件处理器（handler）上。

To attach an element's click event to the edit() handler in the current view:

在当前视图中将一个元素的click事件绑定到edit()处理器上：

|  |  |
| --- | --- |
|  | <a href="#" {{action "edit" on="click"}}>Edit</a> |

Because the default event is click, this could be written more concisely as:

由于缺省事件即为click，所以可以写得更简洁一些：

|  |  |
| --- | --- |
|  | <a href="#" {{action "edit"}}>Edit</a> |

Although the view containing the {{action}} helper will be targeted by default, it is possible to target a different view:

尽管包含了{{action}}助手的视图会成为默认的事件目标，不过也可以将目标指向其他的视图：

|  |  |
| --- | --- |
|  | <a href="#" {{action "edit" target="parentView"}}>Edit</a> |

The action handler can optionally accept a jQuery event object, which will be extended to include viewand context properties. These properties can be useful when targeting a different view with your action. For instance:

这个动作处理器也可以选择性地接受一个jQuery事件对象，该对象将被扩展为包含view和context属性的对象。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | App.ListingView = Ember.View.extend({  templateName: 'listing',  edit: function(event) {  event.view.set('isEditing', true);  }  }); |

Any of the templates discussed above will produce an HTML element like this:

上面讨论的任意一种模板都会生成如下HTML：

|  |  |
| --- | --- |
|  | <a href="#" data-ember-action="3">Edit</a> |

Ember will delegate the event you specified to your target view's handler based upon the internally assigned data-ember-action id.

Ember会基于内部分配的data-ember-actionid来代理你指定在目标视图处理器上的事件

**创建视图的层次结构**

So far, we've discussed writing templates for a single view. However, as your application grows, you will often want to create a hierarchy of views to encapsulate different areas on the page. Each view is responsible for handling events and maintaining the properties needed to display it.

目前为止，我们已经讨论过为单一的视图编写模板。然而，随着应用的扩大，你经常会需要创建分层次的视图来封装页面上的不同区域。每个视图分别负责处理事件和维护需要显示的属性。

**{{view}}**

To add a child view to a parent, use the {{view}} helper, which takes a path to a view class.

可以使用{{view}}助手来给一个父视图增添子视图，它接受一个视图类的路径参数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10**  11  12 | // 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"  }); |
| 1  2 | User: {{firstName}} {{lastName}}  {{view App.InfoView}} |

|  |  |
| --- | --- |
| 1  2  3 | <b>Posts:</b> {{posts}}  <br>  <b>Hobbies:</b> {{hobbies}} |

If we were to create an instance of App.UserView and render it, we would get a DOM representation like this:

如果我们想创建一个App.UserView的实例并渲染它，我们会得到如下的DOM表示方式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | User: Albert Hofmann  <div>  <b>Posts:</b> 25  <br>  <b>Hobbies:</b> Riding bicycles  </div> |

**相对路径**

Instead of specifying an absolute path, you can also specify which view class to use relative to the parent view. For example, we could nest the above view hierarchy like this:

相对于指定绝对路径，你也可以指定一个相对于父视图的路径。例如，我们可以将上面的视图层次修改如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10** | App.UserView = Ember.View.extend({  templateName: 'user',  firstName: "Albert",  lastName: "Hofmann",  infoView: Ember.View.extend({  templateName: 'info',  posts: 25,  hobbies: "Riding bicycles"  })  }); |
| 1  2 | User: {{firstName}} {{lastName}}  {{view infoView}} |

When nesting a view class like this, make sure to use a lowercase letter, as Ember will interpret a property with a capital letter as a global property.

当以这种形式组织视图类时，要确保使用小写字母，因为Ember会把以大写字母开头的属性解释为全局属性。

**设置子视图模板**

If you'd like to specify the template your child views use inline in the main template, you can use the block form of the {{view}} helper. We might rewrite the above example like this:

如果你想在主模板中指定子视图类所使用的模板的话，可以使用{{view}}助手的区块形式。我们可以将上面的例子改写如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | App.UserView = Ember.View.extend({  templateName: 'user',  firstName: "Albert",  lastName: "Hofmann"  });  App.InfoView = Ember.View.extend({  posts: 25,  hobbies: "Riding bicycles"  }); |
| 1  2  3  4  5  6 | User: {{firstName}} {{lastName}}  {{#view App.InfoView}}  <b>Posts:</b> {{view.posts}}  <br>  <b>Hobbies:</b> {{view.hobbies}}  {{/view}} |

When you do this, it may be helpful to think of it as assigning views to portions of the page. This allows you to encapsulate event handling for just that part of the page.

当你这么做的时候，把它想象成将视图应用到页面上的一部分的会更容易理解。这可以允许你只为页面中的那一部分封装事件处理器。

**设置绑定**

So far in our examples, we have been setting static values directly on the views. But to best implement an MVC architecture, we should actually be binding the properties of our views to the controller layer.

目前在我们的例子中，我们已经在视图上直接设置了静态的值。不过作为最好的MVC架构实现，我们应该切实地将视图的属性绑定到控制器层才行。

Let's set up a controller to represent our user data:

让我们设置一个可以代表我们的用户数据的控制器吧：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | App.userController = Ember.Object.create({  content: Ember.Object.create({  firstName: "Albert",  lastName: "Hofmann",  posts: 25,  hobbies: "Riding bicycles"  })  }); |

Now let's update App.UserView to bind to App.userController:

|  |  |
| --- | --- |
| 1  2  3  4  5 | App.UserView = Ember.View.extend({  templateName: 'user',  firstNameBinding: 'App.userController.content.firstName',  lastNameBinding: 'App.userController.content.lastName'  }); |

When we only have a few bindings to configure, like with App.UserView, it is sometimes useful to be able to declare those bindings in the template. You can do that by passing additional arguments to the{{#view}} helper. If all you're doing is configuring bindings, this often allows you to bypass having to create a new subclass.

当我们只有很少的绑定需要配置时，比如使用App.UserView时，有时能够在模板中声明这些绑定是很有用的。你可以通过给{{#view}}助手传入额外的参数来实现。如果你要做的只是配置绑定的话，这些特性通常可以使你避开创建新子类的工作。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | User: {{firstName}} {{lastName}}  {{#view App.UserView postsBinding="App.userController.content.posts"  hobbiesBinding="App.userController.content.hobbies"}}  <b>Posts:</b> {{view.posts}}  <br>  <b>Hobbies:</b> {{view.hobbies}}  {{/view}} |

NOTE: You can actually pass **any** property as a parameter to {{view}}, not just bindings. However, if you are doing anything other than setting up bindings, it is generally a good idea to create a new subclass.

注意：实际上你可以将**任何**属性作为参数传给{{view}}，不仅仅是绑定。然而，如果你要做的不是设置绑定，那么最好创建一个新的子类。

**修改视图的HTML**

When you append a view, it creates a new HTML element that holds its content. If your view has any child views, they will also be displayed as child nodes of the parent's HTML element.

当追加视图时，会创建一个新的HTML元素来持有视图的内容。如果你的视图包含子视图，它们也会作为父HTML元素的子节点显示。

By default, new instances of Ember.View create a <div> element. You can override this by passing atagName parameter:

缺省时，Ember.View的新实例会创建一个<div>元素。你可以通过传入tagName参数来覆盖这个缺省值。

|  |  |
| --- | --- |
|  | {{view App.InfoView tagName="span"}} |

You can also assign an ID attribute to the view's HTML element by passing an id parameter:

你也可以通过传入id参数来为视图的HTML元素指定一个ID属性：

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

This makes it easy to style using CSS ID selectors:

这可以使你很轻松地使用CSS ID选择器来应用样式：

|  |  |
| --- | --- |
| 1  2  3  4 | /\*\* Give the view a red background. \*\*/  #info-view {  background-color: red;  } |

You can assign class names similarly:

类似地，也可以指定类名：

|  |  |
| --- | --- |
|  | {{view App.InfoView class="info urgent"}} |

You can bind class names to a property of the view by using classBinding instead of class. The same behavior as described in bindAttr applies:

你可以使用classBinding而非class来将类名绑定到视图的某个属性上。上述同样的行为也适用于bindAttr：

|  |  |
| --- | --- |
| 1  2  3  4 | App.AlertView = Ember.View.extend({  priority: "p4",  isUrgent: true  }); |
|  | {{view App.AlertView classBinding="isUrgent priority"}} |

This yields a view wrapper that will look something like this:

这将产生一个类似下面的视图容器：

|  |  |
| --- | --- |
|  | <div id="sc420" class="sc-view is-urgent p4"></div> |

**显示一个列表的项目**

If you need to enumerate over a list of objects, use Handlebar's {{#each}} helper:

如果你需要枚举一个对象列表，可以使用Handlebar的{{#each}}助手：

|  |  |
| --- | --- |
| 1  2  3  4 | App.PeopleView = Ember.View.extend({  people: [ { name: 'Yehuda' },  { name: 'Tom' } ]  }); |
| 1  2  3  4  5 | <ul>  {{#each people}}  <li>Hello, {{name}}!</li>  {{/each}}  </ul> |

This will print a list like this:

这将打印出如下的列表：

|  |  |
| --- | --- |
| 1  2  3  4 | <ul>  <li>Hello, Yehuda!</li>  <li>Hello, Tom!</li>  </ul> |

If you want to create a view for every item in a list, just set it up as follows:

如果你想为列表中的每个项单独创建一个视图，只需进行如下设置：

|  |  |
| --- | --- |
| 1  2  3  4  5 | {{#each App.peopleController}}  {{#view App.PersonView}}  {{firstName}} {{lastName}}  {{/view}}  {{/each}} |

**编写自定义助手**

Sometimes, you may use the same HTML in your application multiple times. In those case, you can register a custom helper that can be invoked from any Handlebars template.

有时你可能在应用中会多次用到同样的HTML，在那种情况下，可以注册一个自定义的助手，它可以从任意的Handlebars模板中调用。

For example, imagine you are frequently wrapping certain values in a <span> tag with a custom class. You can register a helper from your JavaScript like this:

例如，想象一下你会使用某个类频繁地将某个值包裹在<span>标签中

，就可以在你的JavaScript中注册一个助手，就像这样：

|  |  |
| --- | --- |
| 1  2  3  4 | Handlebars.registerHelper('highlight', function(property, options) {  var value = Ember.Handlebars.getPath(this, property, options);  return new Handlebars.SafeString('<span class="highlight">'+value+'</span>');  }); |

If you return HTML from a helper, and you don't want it to be escaped, make sure to return a newSafeString.

若果想从一个助手返回HTML，又不希望它被转码，则要确保返回一个新的安全字串（SafeString）。

Anywhere in your Handlebars templates, you can now invoke this helper:

在Handlebars模板的任何地方，都可以调用这个助手：

|  |  |
| --- | --- |
|  | {{highlight name}} |

and it will output the following:

|  |  |
| --- | --- |
|  | <span class="highlight">Peter</span> |

NOTE: Parameters to helper functions are passed as names, not their current values. This allows you to optionally set up observers on the values. To get the current value of the parameter, use Ember.getPath, as shown above.

注意：助手函数的参数是作为名称传递的，而不是它们当前的值。这可以允许你选择性地在值上设置观察者。要想获取参数的当前值，可以使用如上所示的Ember.getPath。

**已包含的视图**

Ember comes pre-packaged with a set of views for building a few basic controls like text inputs, check boxes, and select lists.

Ember预包装了一个视图的合集，用于构建一些基本的控件，比如文本输入框、单选框以及多选列表。

They are:

它们是：

**EMBER.CHECKBOX**

|  |  |
| --- | --- |
| 1  2  3  4 | <label>  {{view Ember.Checkbox checkedBinding="content.isDone"}}  {{content.title}}  </label> |

**EMBER.TEXTFIELD**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | App.MyText = Ember.TextField.extend({  formBlurredBinding: 'App.adminController.formBlurred',  change: function(evt) {  this.set('formBlurred', true);  }  }); |

**EMBER.SELECT**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | {{view Ember.Select viewName="select"  contentBinding="App.peopleController"  optionLabelPath="content.fullName"  optionValuePath="content.id"  prompt="Pick a person:"  selectionBinding="App.selectedPersonController.person"}} |

**EMBER.TEXTAREA**

|  |  |
| --- | --- |
| 1  2  3 | var textArea = Ember.TextArea.create({  valueBinding: 'TestObject.value'  }); |

If you would like to add one of these controls to your view, you are encouraged to extend from these controls.

如果想添加这些控件之一到你的视图中，我们鼓励你从这些控件进行扩展。

Events do not bubble from a subview to a parent view so extending these views is the only way to capture those events.

事件不会从子视图冒泡到父视图，因此扩展这些视图是捕获这些事件的惟一方法。

Example:

例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | App.MyText = Ember.TextField.extend({  formBlurredBinding: 'App.adminController.formBlurred',  change: function(evt) {  this.set('formBlurred', true);  }  }); |

You can then use this view as a sub view and capture the events. In the following example, a change to the Name input would blurr the form and cause the save button to appear.

然后即可将该视图作为子视图使用并捕获事件。在下面的例子中，Name输入的的改变会使表单失去焦点并显示保存按钮。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10**  11  12 | <script id="formDetail" data-template-name='formDetail' type="text/x-handlebars">  <form>  <fieldset>  <legend>Info:</legend>  {{view App.MyText name="Name" id="Name" valueBinding="myObj.Name"}}  <label for="Name">Name</label><br/>  {{#if formBlurred}}  <a href="#" {{action "syncData" on="click"}}>Save</a>  {{/if}}  </fieldset>  </form>  </script> |

**深入探讨视图**

Now that you're familiar with using Handlebars, let's go more in-depth on how to both handle events, and customize views to your needs.

现在你已经熟悉了Handlebars，让我们更加深入地探讨一下如何同时处理事件和根据你的需求自定义视图。

**处理事件**

Instead of having to register event listeners on elements you'd like to respond to, simply implement the name of the event you want to respond to as a method on your view.

你只需简单地将事件名字作为你的视图的方法名实现即可，而无需为需要响应的每个元素注册事件监听器。

For example, imagine we have a template like this:

例如，想象我们有如下模板：

|  |  |
| --- | --- |
| 1  2  3 | {{#view App.ClickableView}}  This is a clickable area!  {{/view}} |

Let's implement App.ClickableView such that when it is clicked, an alert is displayed:

让我们来实现App.ClickableView，使之被点击时可以出现一个警告框：

|  |  |
| --- | --- |
| 1  2  3  4  5 | App.ClickableView = Ember.View.extend({  click: function(evt) {  alert("ClickableView was clicked!");  }  }); |

Events bubble up from the target view to each parent view in succession, until the root view. These values are read-only. If you want to manually manage views in JavaScript (instead of creating them using the {{view}} helper in Handlebars), see the Ember.ContainerView documentation below.

事件会从目标视图陆续地冒泡到每一个父级视图，直到根（root）视图。这些值是只读的。如果你希望在JavaScript中手动管理视图（而不是使用Handlebars中的{{view}}助手创建），见下文的Ember.ContainerView文档。

**使用Ember.ContainerView手动管理视图**

Usually, views create their child views by using the {{view}} helper. Sometimes it is useful to manually manage a view's child views. If you create an instance of Ember.ContainerView, the childViewsarray is editable. Views that you add are rendered to the page, and views that you remove are removed from the DOM.

通常，视图使用{{view}}助手来创建子视图。不过有时手动管理一个视图的子视图也非常有用。如果你创建了一个Ember.ContainerView的实例，则childViews数组是可编辑的。你添加的视图会渲染到页面中，而移除的视图会从DOM中移除。

|  |  |
| --- | --- |
| 1  2  3  4  5 | var container = Ember.ContainerView.create();  container.append();  var coolView = App.CoolView.create(),  childViews = container.get('childViews');  childViews.pushObject(coolView); |

As a shorthand, you can specify the child views as properties and the child views as a list of keys. When the container view is created, these views will be instantiated and added to the child views array:

作为简写，你可以以属性的形式指定子视图，并将子视图作为一个键值列表。当容器视图被创建时，这些视图会被初始化并添加到子视图数组中：

|  |  |
| --- | --- |
| 1  2  3  4  5 | var container = Ember.ContainerView.create({  childViews: ['firstView', 'secondView'],  firstView: App.FirstView,  secondView: App.SecondView  }); |

**渲染管线（Render Pipeline）**

Before your views are turned into DOM elements, they first exist as a string representation. As views render, they turn each of their child views into strings and concatenate them together.

在你的视图转换为DOM元素之前，它们起初是作为字符串形式存在的。随着视图的渲染，它们把每个子视图转换为字符串并将它们连接在一起。

If you'd like to use something other than Handlebars, you can override a view's render method to generate a custom string of HTML.

如果你不想使用Handlebars，可以覆写一个视图的render方法来生成自定义的HTML字串。

|  |  |
| --- | --- |
| 1  2  3  4  5 | App.CoolView = Ember.View.create({  render: function(buffer) {  buffer.push("<b>This view is so cool!</b>");  }  }); |

This makes it easy to support template engines other than Handlebars; though do note that if you override rendering, values will not update automatically. Any updates will be your responsibility.

这使得支持Handlebars之外的模板变得非常容易，然而需要注意的是，如果覆写了渲染，这些值就不会自动更新了。任何的更新都将是你的责任了。

**自定义HTML元素**

A view is represented by a single DOM element on the page. You can change what kind of element is created by changing the tagName property.

一个视图代表页面上的一个单独的DOM元素。你可以通过修改tagName属性来改变创建的元素类型。

|  |  |
| --- | --- |
| 1  2  3 | App.MyView = Ember.View.extend({  tagName: 'span'  }); |

You can also specify which class names are applied to the view by setting its classNames property to an array of strings:

你也可以通过将视图的classNames属性设置为字符串数组来指定视图所应用的类名。

|  |  |
| --- | --- |
| 1  2  3 | App.MyView = Ember.View.extend({  classNames: ['my-view']  }); |

If you want class names to be determined by the state of properties on the view, you can use class name bindings. If you bind to a Boolean property, the class name will be added or removed depending on the value:

如果你希望类名由视图上属性的状态来决定的话，可以使用类名绑定。如果你绑定了一个布尔值的属性，则属性值将决定类的添加和删除：

|  |  |
| --- | --- |
| 1  2  3  4 | App.MyView = Ember.View.extend({  classNameBindings: ['isUrgent'],  isUrgent: true  }); |

This would render a view like this:

这将渲染出如下视图：

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

If isUrgent is changed to false, then the is-urgent class name will be removed.

如果isUrgent被改为false，则is-urgent类名将被移除。

By default, the name of the Boolean property is dasherized. You can customize the class name applied by delimiting it with a colon:

默认情况下，布尔值属性的名称已经被dasherized（译者注：上文有提到，是将下划线替换为连字符的操作）。你可以使用冒号分隔来自定义所应用的类名。

|  |  |
| --- | --- |
| 1  2  3  4 | App.MyView = Ember.View.extend({  classNameBindings: ['isUrgent:urgent'],  isUrgent: true  }); |

This would render this HTML:

这将渲染出如下HTML：

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

Besides the custom class name for the value being true, you can also specify a class name which is used when the value is false:

除了在值为true的时候自定义类名，你也可以指定一个当值为false时所使用的类名：

|  |  |
| --- | --- |
| 1  2  3  4 | App.MyView = Ember.View.extend({  classNameBindings: ['isEnabled:enabled:disabled'],  isEnabled: false  }); |

This would render this HTML:

这将渲染出如下HTML：

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

You can also specify to only add a class when the property is false by declaring classNameBindingslike this:

你也可以通过声明classNameBindings来指定当属性为false时只添加一个类：

|  |  |
| --- | --- |
| 1  2  3  4 | App.MyView = Ember.View.extend({  classNameBindings: ['isEnabled::disabled'],  isEnabled: false  }); |

This would render this HTML:

这将渲染出如下HTML：

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

If the isEnabled property is set to true, no class name is added:

如果isEnabled属性被设置为true，则不会添加任何类名：

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

If the bound value is a string, that value will be added as a class name without modification:

如果绑定的值为字符串，则这个值会不加修改地作为类名添加：

|  |  |
| --- | --- |
| 1  2  3  4 | App.MyView = Ember.View.extend({  classNameBindings: ['priority'],  priority: 'highestPriority'  }); |

This would render this HTML:

这将渲染出如下HTML：

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

**视图的属性绑定**

You can bind attributes to the DOM element that represents a view by using attributeBindings:

你可以使用attributeBindings将属性绑定到代表某个视图的DOM元素上：

|  |  |
| --- | --- |
| 1  2  3  4  5 | App.MyView = Ember.View.extend({  tagName: 'a',  attributeBindings: ['href'],  href: "http://emberjs.com"  }); |

**EMBER枚举（ENUMERABLE）API**

**什么是枚举类型？**

In Ember, an Enumerable is any object that contains a number of child objects, and which allows you to work with those children using the Enumerable interface. The most basic Enumerable is the built-in JavaScript Array.

在Ember中，一个可枚举类型（译者注：Enumerable的原意是可枚举的，但此处是作为名词存在，暂译为可枚举类型）是指任何包含一系列子对象的类型，并且允许你使用枚举接口操作这些子对象。最基本的可枚举类型是JavaScript内置的数组。

For instance, all Enumerables support the standard forEach method:

例如，所有的可枚举类型都支持标准的forEach方法：

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

In general, Enumerable methods, like forEach, take an optional second parameter, which will become the value of this in the callback function:

总体来说，像forEach这样的枚举方法，接受可选的第二个参数，该参数将作为回调函数中this的值：

|  |  |
| --- | --- |
| 1  2  3  4 | var array = [1,2,3];  array.forEach(function(item) {  console.log(item, this.indexOf(item));  }, array) |

Among other reasons, you will find this useful when using another Enumerable method as a callback toforEach:

除去其他原因，你会发现这在使用其他枚举方法作为forEach的回调函数时会非常有用：

|  |  |
| --- | --- |
| 1  2 | var array = [1,2,3];  array.forEach(array.removeObject, array); |

NOTE: This second parameter helps work around a limitation of JavaScript which sets the value of thisto window in methods used this way.

注意：这第二个参数可以帮助解决JavaScript一个局限性，即当我们这样调用方法时，this的值会被设置成window。

**Ember中的枚举类型**

In general, Ember objects that represent lists implement the Enumerable interface. Some examples:

总起来说，Ember中代表列表的对象实现了枚举接口。一些例子：

* *Array*: Ember extends the native JavaScript Array with the Enumerable interface.
* *Array*: Ember为原生的JavaScript数组扩展了枚举接口。
* *ArrayProxy*: A construct that wraps a native Array and adds additional functionality for the view layer.
* *ArrayProxy*: 一个封装了原生数组并为视图层添加了额外功能的结构体（construct）。
* *Set*: An object that can quickly answer whether it includes an object.
* *Set*: 一个可以快速确定其是否包含一个对象的对象。

**枚举接口**

**参数**

The callbacks to Enumerable methods take three arguments:

枚举方法的回调函数接受三个参数：

* *item*: the item for the current iteration.
* *item*: 当前迭代项。
* *index*: an Integer, counting up from 0.
* *index*: 一个从0开始累加的整数。
* *self*: the Enumerable itself.
* *self*: 可枚举类型本身。

**枚举**

To enumerate all the values of an enumerable object, use the forEach method:

可以使用forEach方法来枚举一个可枚举对象的所有值：

|  |  |
| --- | --- |
| 1  2  3 | enumerable.forEach(function(item, index, self) {  console.log(item);  }); |

To invoke some method on each element of an enumerable object, use the invoke method:

若想在枚举对象的每个元素上调用一些方法，可以使用invoke方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10**  11  12  13  14 | Person = Ember.Object.extend({  sayHello: function() {  console.log("Hello from " + this.get('name'));  }  });  var people = [  Person.create({ name: "Juan" }),  Person.create({ name: "Charles" }),  Person.create({ name: "Majd" })  ]  people.invoke('sayHello');  // Hello from Juan  // Hello from Charles  // Hello from Majd |

**头尾对象**

You can get the first or last object from an Enumerable by getting firstObject or lastObject.

你可以通过firstObject或者lastObject来获取一个可枚举类型的第一个或者最后一个对象。

|  |  |
| --- | --- |
| 1  2 | [1,2,3].get('firstObject') // 1  [1,2,3].get('lastObject') // 3 |

**转换为数组**

This one is simple. To convert an Enumerable into an Array, simply call its toArray method.

这个很简单。要想将一个可枚举类型转换为数组，只需调用它的toArray方法即可。

**变换**

You can transform an Enumerable into a derived Array by using the map method:

你可以使用map方法将一个可枚举类型变换为一个派生的数组：

|  |  |
| --- | --- |
| 1  2  3  4 | ['Goodbye', 'cruel', 'world'].map(function(item, index, self) {  return item + "!";  });  // returns ["Goodbye!", "cruel!", "world!"] |

**每个对象上的获取或设置**

A very common use of forEach and map is to get (or set) a property on each element. You can use thegetEach and setEach methods to accomplish these goals.

forEach和map的一个通常用法就是在每个元素上获取（或设置）一个属性。你可以使用getEach和setEach方法来完成这个目标。

|  |  |
| --- | --- |
| 1  2  3  4 | var arr = [Ember.Object.create(), Ember.Object.create()];  // we now have an Array containing two Ember.Objects  arr.setEach('name', 'unknown');  arr.getEach('name') // ['unknown', 'unknown'] |

**过滤**

Another common task to perform on an Enumerable is to take the Enumerable as input, and return an Array after filtering it based on some criteria.

在枚举类型上经常会用到的另外一个常规任务就是将枚举类型作为输入，并基于某些标准过滤后返回一个数组。

For arbitrary filtering, use the (you guessed it) filter method. The filter method expects the callback to return true if Ember should include it in the final Array, and false or undefined if Ember should not.

对于任意的过滤，使用（你已经猜到了）filter方法。如果Ember在最终的数组中应该包含该项的话，则filter方法期待回调函数返回true。

|  |  |
| --- | --- |
| 1  2  3  4  5 | var arr = [1,2,3,4,5];  arr.filter(function(item, index, self) {  if (item < 4) { return true; }  })  // returns [1,2,3] |

When working with a collection of Ember objects, you will often want to filter a set of objects based upon the value of some property. The filterProperty method provides a shortcut.

当操作一个Ember对象的集合时，你经常会需要基于某些属性的值来过滤一组对象。filterProperty方法提供了一种快捷方式。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10** | 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 just the first item |

If you want to return just the first matched value, rather than an Array containing all of the matched values, you can use find and findProperty, which work just like filter and filterProperty, but return only one item.

如果你只希望返回第一个匹配的值，而不是包含所有匹配值的数组，则可使用find和findProperty，用起来跟filter和filterProperty一样，只是仅返回一项。

**汇总信息（全部或任意的）**

If you want to find out whether every item in an Enumerable matches some condition, you can use theevery method:

如果你想知道是否一个枚举类型中的每一项都可以匹配某些条件的话，可以使用every方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  **10**  11  12 | 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 |

If you want to find out whether at least one item in an Enumerable matches some conditions, you can use the some method:

如果你想知道在一个枚举类型中是否至少有一项可以匹配某些条件的话，可以使用some方法：

|  |  |
| --- | --- |
| 1  2  3  4 | people.some(function(person, index, self) {  if(person.get('isHappy')) { return true; }  });  // returns true |

Just like the filtering methods, the every and some methods have analogous everyProperty andsomeProperty methods.

与过滤方法类似，every和some方法也有类似的everyProperty和someProperty方法。

|  |  |
| --- | --- |
| 1  2 | people.everyProperty('isHappy', true) // false  people.someProperty('isHappy', true) // true |