

Knockout 应用开发指南

Knockout应用开发指南 第一章：入门

1 Knockout 简介 (Introduction)

Knockout 是一个轻量级的 UI 类库，通过应用 MVVM 模式使 JavaScript 前端 UI 简单化。

Knockout 有如下4大重要概念：

- **声明式绑定 (Declarative Bindings)**: 使用简明易读的语法很容易地将模型 (model)数据关联到 DOM 元素上。
- **UI 界面自动刷新 (Automatic UI Refresh)**: 当您的模型状态(model state)改变时，您的 UI 界面将自动更新。
- **依赖跟踪 (Dependency Tracking)**: 为转变和联合数据，在你的模型数据之间隐式建立关系。
- **模板 (Templating)**: 为您的模型数据快速编写复杂的可嵌套的 UI。

简称: KO

官方网站: <http://knockoutjs.com>

2 入门介绍 (Getting started)

2.1 KO 工作原理及带来的好处

Knockout 是一个以数据模型 (data model) 为基础的能够帮助你创建富文本，响应显示和编辑用户界面的 JavaScript 类库。任何时候如果你的 UI 需要自动更新 (比如：更新依赖于用户的行为或者外部数据源的改变)，KO 能够很简单的帮你实现并且很容易维护。

重要特性:

- **优雅的依赖追踪**- 不管任何时候你的数据模型更新，都会自动更新相应的内容。
- **声明式绑定**- 浅显易懂的方式将你的用户界面指定部分关联到你的数据模型上。
- **灵活全面的模板**- 使用嵌套模板可以构建复杂的动态界面。
- **轻易可扩展**- 几行代码就可以实现自定义行为作为新的声明式绑定。

额外的好处:

- **纯 JavaScript 类库** – 兼容任何服务器端和客户端技术
- **可添加到 Web 程序最上部** – 不需要大的架构改变
- **简洁的** – Gzip 之前大约25kb

- **兼容任何主流浏览器** (IE 6+、Firefox 2+、Chrome、Safari、其它)
- **Comprehensive suite of specifications** (采用行为驱动开发) - 意味着在新的浏览器和平台上可以很容易通过验证。

开发人员如果用过 Silverlight 或者 WPF 可能会知道 KO 是 MVVM 模式的一个例子。如果熟悉 Ruby on Rails 或其它 MVC 技术可能会发现它是一个带有声明式语法的 MVC 实时 form。换句话说,你可以把 KO 当成通过编辑 JSON 数据来制作 UI 用户界面的一种方式... 不管它为你做什么

OK, 如何使用它?

简单来说: 声明你的数据作为一个 JavaScript 模型对象 (model object), 然后将 DOM 元素或者模板 (templates) 绑定到它上面。

让我们来看一个例子

想想在一个页面上, 航空旅客可以为他们的旅行升级高级食物套餐, 当他们选择一个套餐的时候, 页面立即显示套餐的描述和价格。首先, 声明可用的套餐:

```
var availableMeals = [  
  { mealName: 'Standard', description: 'Dry crusts of bread', extraCost: 0 },  
  { mealName: 'Premium', description: 'Fresh bread with cheese', extraCost:  
    9.95 },  
  { mealName: 'Deluxe', description: 'Caviar and vintage Dr Pepper', extraCost:  
    18.50 }  
];
```

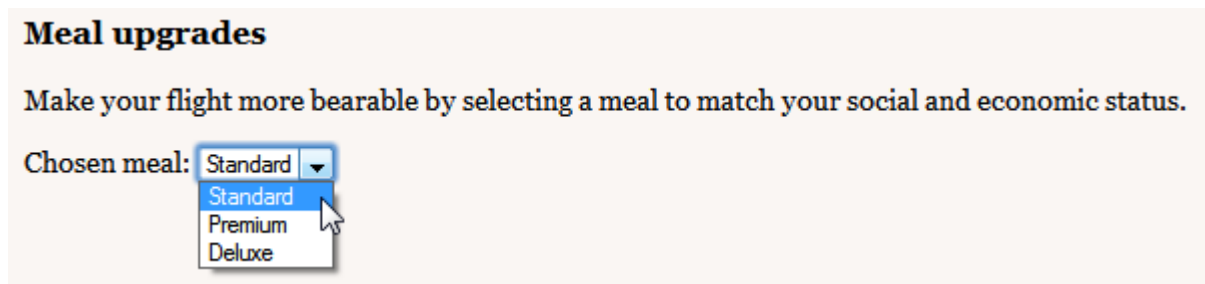
如果想让这3个选项显示到页面上, 我们可以绑定一个下拉菜单 (例如: <select>元素) 到这个数据上。例如:

```
<h3>Meal upgrades</h3>  
<p>Make your flight more bearable by selecting a meal to match your social and  
economic status.</p>  
Chosen meal: <select data-bind="options: availableMeals,  
                      optionsText: 'mealName'"></select>
```

启用 Knockout 并使你的绑定生效, 在 availableMeals 变量声明之后添加如下代码:

```
var viewModel = {  
  /* we'll populate this in a moment */  
};  
  
ko.applyBindings(viewModel); // Makes Knockout get to work  
// 注意: ko. applyBindings 需要在上述 HTML 之后应用才有效
```

你可以在这个系列里将学习更多的 view model 和 MVVM。现在你的页面将 render 成如下的样子：



响应选择

下一步，声明一个简单的 data model 来描述旅客已经选择的套餐，添加一个属性到当前的 view model 上：

```
var viewModel = {  
    chosenMeal: ko.observable(availableMeals[0])  
};
```

ko.observable 是什么？它是 KO 里的一个基础概念。UI 可以监控（observe）它的值并且回应它的变化。这里我们设置 chosenMeal 是 UI 可以监控已经选择的套餐，并初始化它，使用 availableMeal 里的第一个值作为它的默认值（例如：Standard）。

让我们将 chosenMeal 关联到下拉菜单上，仅仅是更新<select>的 data-bind 属性，告诉它让<select>元素的值读取/写入 chosenMeal 这个模型属性：

```
Chosen meal: <select data-bind="options: availableMeals,  
                        optionsText: 'mealName',  
                        value: chosenMeal"></select>
```

理论上说，我们现在可以读/写 chosenMeal 属性了，但是我们不知道它的作用。让我们来显示已选择套餐的描述和价格：

```
<p>  
    You've chosen:  
    <b data-bind="text: chosenMeal().description"></b>  
    (price: <span data-bind="text: chosenMeal().extraCost"></span>)  
</p>
```

于是，套餐信息和价格，将根据用户选择不同的套餐项而自动更新：

Meal upgrades

Make your flight more bearable by selecting a meal to match your social and economic status.

Chosen meal:

You've chosen: **Caviar and vintage Dr Pepper** (price: 18.5)

更多关于 observables 和 dependency tracking 的使用

最后一件事：如果能将价格格式化成带有货币符号的就好了，声明一个 JavaScript 函数就可以实现了...

```
function formatPrice(price) {  
    return price == 0 ? "Free" : "$" + price.toFixed(2);  
}
```

... 然后更新绑定信息使用这个函数 ...

```
(price: <span data-bind='text: formatPrice(chosenMeal().extraCost)'></span>)
```

... 界面显示结果将变得好看多了：

Meal upgrades

Make your flight more bearable by selecting a meal to match your social and economic status.

Chosen meal:

You've chosen: **Fresh bread with cheese** (price: \$9.95)

Price 的格式化展示了，你可以在你的绑定里写任何 JavaScript 代码，KO 仍然能探测到你的绑定依赖代码。这就展示了当你的 model 改变时，KO 如何只进行局部更新而不用重新 render 整个页面 – 仅仅是有依赖值改变的那部分。

链式的 observables 也是支持的（例如：总价依赖于价格和数量）。当链改变的时候，依赖的下游部分将会重新执行，同时所有相关的 UI 将自动更新。不需要在各个 observables 之间声明关联关系，KO 框架会在运行时自动执行的。

你可以从 observables 和 observable arrays 获取更多信息。上面的例子非常简单，没有覆盖很多 KO 的功能。你可以获取更多的内嵌的绑定和模板绑定。

KO 和 jQuery (或 Prototype 等)是竞争关系还是能一起使用?

所有人都喜欢 jQuery! 它是一个在页面里操作元素和事件的框架, 非常出色并且易使用, 在 DOM 操作上肯定使用 jQuery, KO 解决不同的问题。

如果页面要求复杂, 仅仅使用 jQuery 需要花费更多的代码。例如: 一个表格里显示一个列表, 然后统计列表的数量, Add 按钮在数据行 TR 小于5调的时候启用, 否则就禁用。jQuery 没有基本的数据模型的概念, 所以需要获取数据的数量 (从 table/div 或者专门定义的 CSS class), 如果需要在某些 SPAN 里显示数据的数量, 当添加新数据的时候, 你还要记得更新这个 SPAN 的 text。当然, 你还要判断当总数>=5条的时候禁用 Add 按钮。然后, 如果还要实现 Delete 功能的时候, 你不得不指出哪一个 DOM 元素被点击以后需要改变。

Knockout 的实现有何不同?

使用 KO 非常简单。将你的数据描绘成一个 JavaScript 数组对象 myItems, 然后使用模板 (template) 转化这个数组到表格里 (或者一组 DIV)。不管什么时候数组改变, UI 界面也会响应改变 (不用指出如何插入新行<tr>或在哪里插入), 剩余的工作就是同步了。例如: 你可以声明绑定如下一个 SPAN 显示数据数量 (可以放在页面的任何地方, 不一定非要在 template 里):

```
There are <span data-bind="text: myItems().count"></span> items
```

就是这些! 你不需要写代码去更新它, 它的更新依赖于数组 myItems 的改变。同样, Add 按钮的启用和禁用依赖于数组 myItems 的长度, 如下:

```
<button data-bind="enable: myItems().count < 5">Add</button>
```

之后, 如果你要实现 Delete 功能, 不必指出如何操作 UI 元素, 只需要修改数据模型就可以了。

总结: KO 没有和 jQuery 或类似的 DOM 操作 API 对抗竞争。KO 提供了一个关联数据模型和用户界面的高级功能。KO 本身不依赖 jQuery, 但是你可以一起同时使用 jQuery, 生动平缓的 UI 改变需要真正使用 jQuery。

2.2 下载安装

Knockout 的核心类库是纯 JavaScript 代码, 不依赖任何第三方的类库。所以按照如下步骤即可添加 KO 到你的项目里:

- 1 下载 Knockout 类库的最新版本, 在正式开发和产品使用中, 请使用默认的压缩版本 (knockout-x.x.js)。

下载地址: <https://github.com/SteveSanderson/knockout/downloads>

Debug 调试目的，可使用非压缩版本(*knockout-x.x.debug.js*)。和压缩版本同样的功能，但是具有全变量名和注释的可读性源代码，并且没有隐藏内部的 *API*。

- 2 在你的 HTML 页面里使用<script>标签引用 Knockout 类库文件。

这就是你需要的一切...

开启模板绑定

...除非你想使用模板绑定功能（您很有可能使用它，因为非常有用），那你需要再引用两个 JavaScript 文件。KO1.3版的默认模板引擎是依赖 jQuery 的 jquery.tmpl.js（最新版2.0版已经不依赖 jquery tmp 了）。所以你需要下载下面的2个文件并在**引用 KO 之前**引用：

- jQuery 1.4.2 或更高版本
- jquery-tmpl.js — 此版本 可以很容易使用，或者你访问[官方网站](#) 查找最新版本。

正确的引用顺序：

```
<script type='text/javascript' src='jquery-1.4.2.min.js'></script>
<script type='text/javascript' src='jquery-tmpl.js'></script>
<script type='text/javascript' src='knockout-1.2.1.js'></script>
```

（当然，您要根据你的文件路径更新上面的文件路径和文件名。）

下一步，开始学习 监控属性。

Knockout应用开发指南 第二章：监控属性(Observables)

关于 Knockout 的3个重要概念（Observables,DependentObservables,ObservableArray），本人无法准确表达它的准确含义，所以暂定翻译为（监控属性、依赖监控属性和监控数组），如果有好的建议请指正，多谢。

1 创建带有监控属性的 view model

Observables

Knockout 是在下面三个核心功能建立起来的：

- 1 监控属性（Observables）和依赖跟踪（Dependency tracking）

- 2 声明式绑定 (Declarative bindings)
- 3 模板 (Templating)

这一节，你讲学到3个功能中的第一个。在这之前，我们来解释一下 MVVM 模式和 view model 的概念。

MVVM and View Models

Model-View-View Model (MVVM) 是一种创建用户界面的设计模式。描述的是如何将复杂的 UI 用户界面分成3个部分：

- **model:** 你程序里存储的数据。这个数据包括对象和业务操作（例如：银子账户可以完成转账功能），并且独立于任何 UI。使用 KO 的时候，通常说是向服务器调用 Ajax 读写这个存储的模型数据。
- **view model:** 在 UI 上，纯 code 描述的数据以及操作。例如，如果你实现列表编辑，你的 view model 应该是一个包含列表项 items 的对象和暴露的 add/remove 列表项 (item) 的操作方法。

注意这不是 UI 本身：它不包含任何按钮的概念或者显示风格。它也不是持续数据模型 – 包含用户正在使用的未保存数据。使用 KO 的时候，你的 view models 是不包含任何 HTML 知识的纯 JavaScript 对象。保持 view model 抽象可以保持简单，以便你能管理更复杂的行为。

- **view:** 一个可见的，交互式的，表示 view model 状态的 UI。从 view model 显示数据，发送命令到 view model（例如：当用户 click 按钮的时候），任何 view model 状态改变的时候更新。

使用 KO 的时候，你的 view 就是你带有绑定信息的 HTML 文档，这些声明式的绑定管理到你的 view model 上。或者你可以使用模板从你的 view model 获取数据生成 HTML。

创建一个 view model，只需要声明任意的 JavaScript object。例如：

```
var myViewModel = {  
  personName: 'Bob',  
  personAge: 123  
};
```

你可以为 view model 创建一个声明式绑定的简单 view。例如：下面的代码显示 personName 值：

```
The name is <span data-bind="text: personName"></span>
```

Activating Knockout

data-bind 属性尽快好用但它不是 HTML 的原生属性（它严格遵从 HTML5 语法，虽然 HTML4 验证器提示有不可识别的属性但依然可用）。由于浏览器不识别它是什么意思，所以你需要激活 Knockout 来让他起作用。

激活 Knockout，需要添加如下的 `<script>` 代码块：

```
ko.applyBindings(myViewModel);
```

你可以将这个代码块放在 HTML 底部，或者放在 jQuery 的 `$` 函数或者 `ready` 函数里，然后放在页面上面，最终生成结果就是如下的 HTML 代码：

```
The name is <span>Bob</span>
```

你可能奇怪 `ko.applyBindings` 使用的是什么样的参数，

- 第一个参数是你想用于声明式绑定
- 第二个参数（可选），可以声明成使用 `data-bind` 的 HTML 元素或者容器。例如，`ko.applyBindings(myViewModel, document.getElementById('someElementId'))`。它的现在是只有作为 `someElementId` 的元素和子元素才能激活 KO 功能。好处是你可以同一个页面声明多个 view model，用来区分区域。

Observables

现在已经知道如何创建一个简单的 view model 并且通过 binding 显示它的属性了。但是 KO 一个重要的功能是当你的 view model 改变的时候能自动更新你的界面。当你的 view model 部分改变的时候 KO 是如何知道的呢？答案是：你需要将你的 model 属性声明成 observable 的，因为它是非常特殊的 JavaScript objects，能够通知订阅者它的改变以及自动探测到相关的依赖。

例如：将上述例子的 view model 改成如下代码：

```
var myViewModel = {
  personName: ko.observable('Bob'),
  personAge: ko.observable(123)
};
```

你根本不需要修改 view – 所有的 `data-bind` 语法依然工作，不同的是他能监控到变化，当值改变时，view 会自动更新。

监控属性（observables）的读和写

不是所有的浏览器都支持 JavaScript 的 `getters and setters` (比如 IE)，所以为了兼容性，使用 `ko.observable` 监控的对象都是真实的 function 函数。

- 读取监控属性（observable）的值，只需要直接调用监控属性（observable）（不需要参数），例如 `myViewModel.personName()` 将返回 'Bob'，`myViewModel.personAge()` 将返回 123。
- 写一个新值到监控属性（observable）上，调用这个 observable 属性并当新值作为

参数。例如：调用 `myViewModel.personName('Mary')` 将更新 `name` 值为'Mary'。

- 给一个 `model` 对象的多个属性写入新值，你可以使用链式语法。例如：
`myViewModel.personName('Mary').personAge(50)` 将会将 `name` 更新为 'Mary' 并且将 `age` 更新为 50。

监控属性（`observables`）的特征就是监控（`observed`），例如其它代码可以说我需要得到对象变化的通知，所以 KO 内部有很多内置的绑定语法。所以如果你的代码写成 `data-bind="text: personName"`，`text` 绑定注册到自身，一旦 `personName` 的值改变，它就能得到通知。

当然调用 `myViewModel.personName('Mary')` 改变 `name` 的值，`text` 绑定将自动更新这个新值到相应的 DOM 元素上。这就是如何将 `view model` 的改变传播到 `view` 上的。

监控属性（Observables）的显式订阅

通常情况下，你不用手工订阅，所以新手可以忽略此小节。高级用户，如果你要注册自己的订阅到监控属性（`observables`），你可以调用它的 `subscribe` 函数。例如：

```
myViewModel.personName.subscribe(function (newValue) {  
    alert("The person's new name is " + newValue);  
});
```

这个 `subscribe` 函数在内部很多地方都用到的。你也可以终止自己的订阅：首先得到你的订阅，然后调用这个对象的 `dispose` 函数，例如：

```
var subscription = myViewModel.personName.subscribe(function (newValue) { /* do  
stuff */ });  
// ...then later...  
subscription.dispose(); // I no longer want notifications
```

大多数情况下，你不需要做这些，因为内置的绑定和模板系统已经帮你做好很多事情了，可以直接使用它们。

2 使用依赖监控属性(Dependent Observables)

如果你已经有了监控属性 `firstName` 和 `lastName`，你想显示全称怎么办？这就需要用到依赖监控属性了 – 这些函数是一个或多个监控属性，如果他们的依赖对象改变，他们会自动跟着改变。

例如，下面的 `view model`,

```
var viewModel = {  
    firstName: ko.observable('Bob'),  
    lastName: ko.observable('Smith')  
};
```

... 你可以添加一个依赖监控属性来返回姓名全称:

```
viewModel.fullName = ko.dependentObservable(function () {  
    return this.firstName() + " " + this.lastName();  
}, viewModel);
```

并且绑定到 UI 的元素上, 例如:

```
The name is <span data-bind="text: fullName"></span>
```

... 不管 `firstName` 还是 `lastName` 改变, 全称 `fullName` 都会自动更新 (不管谁改变, 执行函数都会调用一次, 不管改变成什么, 他的值都会更新到 UI 或者其他依赖监控属性上)。

管理‘this’

新手可忽略此小节, 你只需要安装上面例子中的代码模式写就行了, 无需知道/关注这个 `this`。

你可能疑惑 `ko.dependentObservable` 的第二个参数是做什么用的 (上面的例子中我传的是 `viewModel`), 它是声明执行依赖监控属性的 `this` 用的。没有它, 你不能引用到 `this.firstName()` 和 `this.lastName()`。老练的 `JavaScript` 开发人员不觉得 `this` 怎么样, 但是如果你不熟悉 `JavaScript`, 那就对它就会很陌生。(C# 和 Java 需要不需要为 `set` 一个值为设置 `this`, 但是 `JavaScript` 需要, 因为默认情况下他们的函数自身不是任何对象的一部分)。

不幸的是, `JavaScript` 对象没有任何办法能引用他们自身, 所以你需要通过 `myViewModelObject.myDependentObservable = ...` 的形式添加依赖监控属性到 `view model` 对象上。你不能直接在 `view model` 里声明他们, 换句话说, 你不能写成下面这样:

```
var viewModel = {  
    myDependentObservable: ko.dependentObservable(function () {  
        ...  
    }, /* can't refer to viewModel from here, so this doesn't work */)  
}
```

... 相反你必须写成如下这样:

```
var viewModel = {  
    // Add other properties here as you wish  
};  
  
viewModel.myDependentObservable = ko.dependentObservable(function () {  
    ...  
}, viewModel); // This is OK
```

只要你知道期望什么, 它确实不是个问题。J

依赖链

理所当然，如果你想你可以创建一个依赖监控属性的链。例如：

- 监控属性 **items** 表述一组列表项
- 监控属性 **selectedIndexes** 保存着被用户选上的列表项的索引
- 依赖监控属性 **selectedItems** 返回的是 **selectedIndexes** 对应的列表项数组
- 另一个依赖监控属性返回的 **true** 或 **false** 依赖于 **selectedItems** 的各个列表项是否包含一些属性（例如，是否新的或者还未保存的）。一些 UI element（像按钮的启用/禁用）的状态取决于这个值）。

然后，**items** 或者 **selectedIndexes** 的改变将会影响到所有依赖监控属性的链，所有绑定这些属性的 UI 元素都会自动更新。多么整齐与优雅！

可写的依赖监控属性

新手可忽略此小节，可写依赖监控属性真的是太 **advanced** 了，而且大部分情况下都用不到。

正如所学到的，依赖监控属性是通过计算其它的监控属性而得到的。感觉是依赖监控属性正常情况下应该是只读的。那么，有可能让依赖监控属性支持可写么？你只需要声明自己的 **callback** 函数然后利用写入的值再处理一下相应的逻辑即可。

你可以像使用普通的监控属性一样使用依赖监控属性 – 数据双向绑定到 **DOM** 元素上，并且通过自定义的逻辑拦截所有的读和写操作。这是非常牛逼的特性并且可以在大范围内使用。

例1：分解用户的输入

返回到经典的“**first name + last name = full name**”例子上，你可以让事情调回来看：让依赖监控属性 **fullName** 可写，让用户直接输入姓名全称，然后输入的值将被解析并映射写入到基本的监控属性 **firstName** 和 **lastName** 上：

```
var viewModel = {
    firstName: ko.observable("Planet"),
    lastName: ko.observable("Earth")
};

viewModel.fullName = ko.dependentObservable({

    read: function () {
        return this.firstName() + " " + this.lastName();
    },

    write: function (value) {
        var lastSpacePos = value.lastIndexOf(" ");
```

```

        if (lastSpacePos > 0) { // Ignore values with no space character
            this.firstName(value.substring(0, lastSpacePos)); // Update
"firstName"
            this.lastName(value.substring(lastSpacePos + 1)); // Update
"lastName"
        }
    },
    owner: viewModel
});

```

这个例子里，写操作的 **callback** 接受写入的值，把值分离出来，分别写入到“**firstName**”和“**lastName**”上。你可以像普通情况一样将这个 **view model** 绑定到 **DOM** 元素上，如下：

```

<p>First name: <span data-bind="text: firstName"></span></p>
<p>Last name: <span data-bind="text: lastName"></span></p>
<h2>Hello, <input data-bind="value: fullName"/>!</h2>

```

这是一个 **Hello World** 例子的反例子，姓和名都不可编辑，相反姓和名组成的姓名全称却是可编辑的。

上面的 **view model** 演示的是通过一个简单的参数来初始化依赖监控属性。你可以给下面的属性传入任何 **JavaScript** 对象：

- **read** — 必选，一个用来执行取得依赖监控属性当前值的函数。
- **write** — 可选，如果声明将使你的依赖监控属性可写，别的代码如果这个可写功能写入新值，通过自定义逻辑将值再写入各个基础的监控属性上。
- **owner** — 可选，如果声明，它就是 **KO** 调用 **read** 或 **write** 的 **callback** 时用到的 **this**。查看“管理 **this**”获取更新信息。

例2：Value 转换器

有时候你可能需要显示一些不同格式的数据，从基础的数据转化成显示格式。比如，你存储价格为 **float** 类型，但是允许用户编辑的字段需要支持货币单位和小数点。你可以用可写的依赖监控属性来实现，然后解析传入的数据到基本 **float** 类型里：

```

viewModel.formattedPrice = ko.dependentObservable({

    read: function () {
        return "$" + this.price().toFixed(2);
    },

    write: function (value) {
        // Strip out unwanted characters, parse as float, then write the raw data
back to the underlying "price" observable
        value = parseFloat(value.replace(/[^\.\d]/g, ""));
        this.price(isNaN(value) ? 0 : value); // Write to underlying storage
    },

```

```
owner: viewModel
});
```

然后我们绑定 `formattedPrice` 到 `text box` 上:

```
<p>Enter bid price: <input data-bind="value: formattedPrice"/></p>
```

所以, 不管用户什么时候输入新价格, 输入什么格式, `text box` 里会自动更新为带有2位小数点和货币符号的数值。这样用户可以看到你的程序有多聪明, 来告诉用户只能输入2位小数, 否则的话自动删除多余的位数, 当然也不能输入负数, 因为 `write` 的 `callback` 函数会自动删除负号。

例3: 过滤并验证用户输入

例1展示的是写操作过滤的功能, 如果你写的值不符合条件的话将不会被写入, 忽略所有不包括空格的值。

再多走一步, 你可以声明一个监控属性 `isValid` 来表示最后一次写入是否合法, 然后根据真假值显示相应的提示信息。稍后详细介绍, 先参考如下代码:

```
var viewModel = {
    acceptedNumericValue: ko.observable(123),
    lastInputWasValid: ko.observable(true)
};

viewModel.attemptedValue = ko.dependentObservable({
    read: viewModel.acceptedNumericValue,
    write: function (value) {
        if (isNaN(value))
            this.lastInputWasValid(false);
        else {
            this.lastInputWasValid(true);
            this.acceptedNumericValue(value); // Write to underlying storage
        }
    },
    owner: viewModel
});
```

... 按照如下格式声明绑定元素:

```
<p>Enter a numeric value: <input data-bind="value: attemptedValue"/></p>
<div data-bind="visible: !lastInputWasValid()">That's not a number!</div>
```

现在, `acceptedNumericValue` 将只接受数字, 其它任何输入的值都会触发显示验证信息, 而会更新 `acceptedNumericValue`。

备注: 上面的例子显得杀伤力太强了, 更简单的方式是在 `<input>` 上使用 `jQuery Validation`

和 `number class`。Knockout 可以和 jQuery Validation 一起很好的使用，参考例子：[grid editor](#)。当然，上面的例子依然展示了一个如何使用自定义逻辑进行过滤和验证数据，如果验证很复杂而 jQuery Validation 很难使用的话，你就可以用它。

依赖跟踪如何工作的

新手没必要知道太清楚，但是高级开发人员可以需要知道为什么依赖监控属性能够自动跟踪并且自动更新 UI...

事实上，非常简单，甚至说可爱。跟踪的逻辑是这样的：

- 4 当你声明一个依赖监控属性的时候，KO 会立即调用执行函数并且获取初始化值。
- 5 当你的执行函数运行的时候，KO 会把所有需要依赖的依赖属性（或者监控依赖属性）都记录到一个 Log 列表里。
- 6 执行函数结束以后，KO 会向所有 Log 里需要依赖到的对象进行订阅。订阅的 callback 函数是重新运行你的执行函数。然后回头重新执行上面的第一步操作（并且注销不再使用的订阅）。
- 7 最后 KO 会通知上游所有订阅它的订阅者，告诉它们我已经设置了新值。

所有说，KO 不仅仅是在第一次执行函数执行时候探测你的依赖项，每次它都会探测。举例来说，你的依赖属性可以是动态的：依赖属性 A 代表你是否依赖于依赖属性 B 或者 C，这时候只有当 A 或者你当前的选择 B 或者 C 改变的时候执行函数才重新执行。你不需要再声明其它的依赖：运行时会自动探测到的。

另外一个技巧是：一个模板输出的绑定是依赖监控属性的简单实现，如果模板读取一个监控属性的值，那模板绑定就会自动变成依赖监控属性依赖于那个监控属性，监控属性一旦改变，模板绑定的依赖监控属性就会自动执行。嵌套的模板也是自动的：如果模板 X render 模板 Y，并且 Y 需要显示监控属性 Z 的值，当 Z 改变的时候，由于只有 Y 依赖它，所以只有 Y 这部分进行了重新绘制（render）。

3 使用 observable 数组

如果你要探测和响应一个对象的变化，你应该用 `observables`。如果你需要探测和响应一个集合对象的变化，你应该用 `observableArray`。在很多场景下，它都非常有用，比如你要在 UI 上需要显示/编辑的一个列表数据集合，然后对集合进行添加和删除。

```
var myObservableArray = ko.observableArray(); // Initially an empty array
myObservableArray.push('Some value');         // Adds the value and notifies
observers
```

关键点：监控数组跟踪的是数组里的对象，而不是这些对象自身的状态。

简单说，将一对象放在 `observableArray` 里不会使这个对象本身的属性变化可监控的。当然你自己也可以声明这个对象的属性为 `observable` 的，但它就成了一个依赖监控对象了。一

个 `observableArray` 仅仅监控他拥有的对象，并在这些对象添加或者删除的时候发出通知。

预加载一个监控数组 `observableArray`

如果你想让你的监控数组在开始的时候就有一些初始值，那么在声明的时候，你可以在构造器里加入这些初始对象。例如：

```
// This observable array initially contains three objects
var anotherObservableArray = ko.observableArray([
  { name: "Bungle", type: "Bear" },
  { name: "George", type: "Hippo" },
  { name: "Zippy", type: "Unknown" }
]);
```

从 `observableArray` 里读取信息

一个 `observableArray` 其实就是一个 `observable` 的监控对象，只不过他的值是一个数组（`observableArray` 还加了很多其他特性，稍后介绍）。所以你可以像获取普通的 `observable` 的值一样，只需要调用无参函数就可以获取自身的值了。例如，你可以像下面这样获取它的值：

```
alert('The length of the array is ' + myObservableArray().length);
alert('The first element is ' + myObservableArray()[0]);
```

理论上你可以使用任何原生的 **JavaScript** 数组函数来操作这些数组，但是 **KO** 提供了更好的功能等价函数，他们非常有用是因为：

- 8 兼容所有浏览器。（例如 `indexOf` 不能在 **IE8**和早期版本上使用，但 **KO** 自己的 `indexOf` 可以在所有浏览器上使用）
- 9 在数组操作函数方面（例如 `push` 和 `splice`），**KO** 自己的方式可以自动触发依赖跟踪，并且通知所有的订阅者它的变化，然后让 **UI** 界面也相应的自动更新。
- 10 语法更方便，调用 **KO** 的 `push` 方法，只需要这样写：`myObservableArray.push(...)`。比如原生数组的 `myObservableArray().push(...)`好用多了。

下面讲解的均是 `observableArray` 的读取和写入的相关函数。

indexOf

`indexOf` 函数返回的是第一个等于你参数数组项的索引。例如：

`myObservableArray.indexOf('Blah')`将返回以0为第一个索引的第一个等于 `Blah` 的数组项的索引。如果没有找到相等的，将返回-1。

slice

`slice` 函数是 `observableArray` 相对于 JavaScript 原生函数 `slice` 的等价函数（返回给定的从开始索引到结束索引之间所有的对象集合）。调用 `myObservableArray.slice(...)` 等价于调用 JavaScript 原生函数（例如：`myObservableArray().slice(...)`）。

操作 `observableArray`

`observableArray` 展现的是数组对象相似的函数并通知订阅者的功能。

pop, push, shift, unshift, reverse, sort, splice

所有这些函数都是和 JavaScript 数组原生函数等价的，唯一不同的数组改变可以通知订阅者：

`myObservableArray.push('Some new value')` 在数组末尾添加一个新项

`myObservableArray.pop()` 删除数组最后一个项并返回该项

`myObservableArray.unshift('Some new value')` 在数组头部添加一个项

`myObservableArray.shift()` 删除数组头部第一项并返回该项

`myObservableArray.reverse()` 翻转整个数组的顺序

`myObservableArray.sort()` 给数组排序

默认情况下，是按照字符排序（如果是字符）或者数字排序（如果是数字）。

你可以排序传入一个排序函数进行排序，该排序函数需要接受2个参数（代表该数组里需要比较的项），如果第一个项小于第二个项，返回-1，大于则返回1，等于返回0。例如：用 `lastname` 给 `person` 排序，你可以这样写：`myObservableArray.sort(function (left, right) {return left.lastName == right.lastName? 0: (left.lastName < right.lastName? -1: 1)})`

`myObservableArray.splice()` 删除指定开始索引和指定数目的数组对象元素。例如 `myObservableArray.splice(1, 3)` 从索引1开始删除3个元素（第2,3,4个元素）然后将这些元素作为一个数组对象返回。

更多 `observableArray` 函数的信息，请参考等价的 [JavaScript 数组标准函数](#)。

remove 和 removeAll

`observableArray` 添加了一些 JavaScript 数组默认没有但非常有用的函数：

`myObservableArray.remove(someItem)` 删除所有等于 `someItem` 的元素并将被删除元素作为一个数组返回

`myObservableArray.remove(function(item) { return item.age < 18 })` 删除所有 `age` 属性小于18的元素并将被删除元素作为一个数组返回

`myObservableArray.removeAll(['Chad', 132, undefined])` 删除所有等于 'Chad', 123, or undefined 的元素并将被删除元素作为一个数组返回

destroy 和 destroyAll (注: 通常只和 Ruby on Rails 开发者有关)

destroy 和 destroyAll 函数是为 Ruby on Rails 开发者方便使用而开发的:

`myObservableArray.destroy(someItem)` 找出所有等于 someItem 的元素并给他们添加一个属性 `_destroy`, 并赋值为 true

`myObservableArray.destroy(function(someItem) { return someItem.age < 18 })` 找出所有 age 属性小于 18 的元素并给他们添加一个属性 `_destroy`, 并赋值为 true

`myObservableArray.destroyAll(['Chad', 132, undefined])` 找出所有等于 'Chad', 123, or undefined 的元素并给他们添加一个属性 `_destroy`, 并赋值为 true

那么, `_destroy` 是做什么用的? 正如我提到的, 这只是为 Rails 开发者准备的。在 Rails 开发过程中, 如果你传入一个 JSON 对象, Rails 框架会自动转换成 ActiveRecord 对象并且保存到数据库。Rails 框架知道哪些对象以及在数据库中存在, 哪些需要添加或更新, 标记 `_destroy` 为 true 就是告诉框架删除这条记录。

注意的是: 在 KO render 一个 foreach 模板的时候, 会自动隐藏带有 `_destroy` 属性并且值为 true 的元素。所以如果你的“delete”按钮调用 `destroy(someItem)` 方法的话, UI 界面上的相对应的元素将自动隐藏, 然后等你提交这个 JSON 对象到 Rails 上的时候, 这个元素项将从数据库删除 (同时其它的元素项将正常的插入或者更新)。

Knockout应用开发指南 第三章: 绑定语法 (1)

第三章所有代码都需要启用 KO 的 `ko.applyBindings(viewModel);` 功能, 才能使代码生效, 为了节约篇幅, 所有例子均省略了此行代码。

1 visible 绑定

目的

visible 绑定到 DOM 元素上, 使得该元素的 hidden 或 visible 状态取决于绑定的值。

例子

```
<div data-bind="visible: shouldShowMessage">
    You will see this message only when "shouldShowMessage" holds a true value.
</div>

<script type="text/javascript">
    var viewModel = {
        shouldShowMessage: ko.observable(true) // Message initially visible
    };
    viewModel.shouldShowMessage(false); // ... now it's hidden
    viewModel.shouldShowMessage(true); // ... now it's visible again
</script>
```

参数

主参数

当参数设置为一个**假值**时（例如：布尔值 `false`，数字值 `0`，或者 `null`，或者 `undefined`），该绑定将设置该元素的 `style.display` 值为 `none`，让元素隐藏。它的优先级高于你在 CSS 里定义的任何 `display` 样式。

当参数设置为一个**真值**时（例如：布尔值 `true`，或者非空 `non-null` 的对象或者数组），该绑定会删除该元素的 `style.display` 值，让元素可见。然后你在 CSS 里自定义的 `display` 样式将会自动生效。

如果参数是监控属性 `observable` 的，那元素的 `visible` 状态将根据参数值的变化而变化，如果不是，那元素的 `visible` 状态将只设置一次并且以后不在更新。

其它参数

无

注：使用函数或者表达式来控制元素的可见性

你也可以使用 JavaScript 函数或者表达式作为参数。这样的话，函数或者表达式的结果将决定是否显示/隐藏这个元素。例如：

```
<div data-bind="visible: myValues().length > 0">
    You will see this message only when 'myValues' has at least one member.
</div>

<script type="text/javascript">
    var viewModel = {
```

```
    myValues: ko.observableArray([]) // Initially empty, so message hidden
  };
  viewModel.myValues.push("some value"); // Now visible
</script>
```

依赖性

除 KO 核心类库外，无依赖。

2 text 绑定

目的

text 绑定到 DOM 元素上，使得该元素显示的文本值为你绑定的参数。该绑定在显示或者上非常有用，但是你可以用在任何元素上。

例子

```
Today's message is: <span data-bind="text: myMessage"></span>

<script type="text/javascript">
  var viewModel = {
    myMessage: ko.observable() // Initially blank
  };
  viewModel.myMessage("Hello, world!"); // Text appears
</script>
```

参数

主参数

KO 将参数值会设置在元素的 innerText（IE）或 textContent（Firefox 和其它相似浏览器）属性上。原来的文本将会被覆盖。

如果参数是监控属性 observable 的，那元素的 text 文本将根据参数值的变化而更新，如果不是，那元素的 text 文本将只设置一次并且以后不在更新。

如果你传的是不是数字或者字符串（例如一个对象或者数组），那显示的文本将是 yourParameter.toString() 的等价内容。

其它参数

无

注1：使用函数或者表达式来决定 text 值

如果你想让你的 text 更可控，那选择是创建一个依赖监控属性（dependent observable），然后在它的执行函数里编码，决定应该显示什么样的 text 文本。

例如：

```
The item is <span data-bind="text: priceRating"></span> today.
```

```
<script type="text/javascript">
    var viewModel = {
        price: ko.observable(24.95)
    };

    viewModel.priceRating = ko.dependentObservable(function () {
        return this.price() > 50 ? "expensive" : "affordable";
    }, viewModel);
</script>
```

现在，text 文本将在“expensive”和“affordable”之间替换，取决于价格怎么改变。

然而，如果有类似需求的话其实没有必要再声明一个依赖监控属性（dependent observable），你只需要按照如下代码写 JavaScript 表达式就可以了：

```
The item is <span data-bind="text: price() > 50 ? 'expensive' :
'affordable'"></span> today.
```

结果是一样的，但我们不需要再声明依赖监控属性（dependent observable）。

注2：关于 HTML encoding

因为该绑定是设置元素的 innerText 或 textContent（而不是 innerHTML），所以它是安全的，没有 HTML 或者脚本注入的风险。例如：如果你编写如下代码：

```
viewModel.myMessage("<i>Hello, world!</i>");
```

... 它不会显示斜体字，而是原样输出标签。如果你需要显示 HTML 内容，请参考 html 绑定。

注3: 关于 IE 6 的白空格 `whitespace`

IE6有个奇怪的问题，如果 `span` 里有空格的话，它将自动变成一个空的 `span`。如果你想编写如下的代码的话，那 Knockout 将不起任何作用：

```
Welcome, <span data-bind="text: userName"></span> to our web site.
```

... IE6 将不会显示 `span` 中间的那个空格，你可以通过下面这样的代码避免这个问题：

```
Welcome, <span data-bind="text: userName">&nbsp;</span> to our web site.
```

IE6以后版本和其它浏览器都没有这个问题

依赖性

除 KO 核心类库外，无依赖。

3 html 绑定

目的

html 绑定到 DOM 元素上，使得该元素显示的 HTML 值为你绑定的参数。如果在你的 view model 里声明 HTML 标记并且 render 的话，那非常有用。

例子

```
<div data-bind="html: details"></div>

<script type="text/javascript">
    var viewModel = {
        details: ko.observable() // Initially blank
    };

    viewModel.details("<em>For further details, view the report <a
href='report.html'>here</a>.</em>");
    // HTML content appears
</script>
```

参数

主参数

KO 设置该参数值到元素的 `innerHTML` 属性上，元素之前的内容将被覆盖。

如果参数是监控属性 `observable` 的，那元素的内容将根据参数值的变化而更新，如果不是，那元素的内容将只设置一次并且以后不在更新。

如果你传的是不是数字或者字符串（例如一个对象或者数组），那显示的文本将是 `yourParameter.toString()` 的等价内容。

其它参数

无

注：关于 HTML encoding

因为该绑定设置元素的 `innerHTML`，你应该注意不要使用不安全的 HTML 代码，因为有可能引起脚本注入攻击。如果你不确信是否安全（比如显示用户输入的内容），那你应该使用 `text` 绑定，因为这个绑定只是设置元素的 `text` 值 `innerText` 和 `textContent`。

依赖性

除 KO 核心类库外，无依赖。

4 CSS 绑定

目的

css 绑定是添加或删除一个或多个 CSS class 到 DOM 元素上。非常有用，比如当数字变成负数时高亮显示。（注：如果你不想应用 CSS class 而是想引用 `style` 属性的话，请参考 **style 绑定**。）

例子

```
<div data-bind="css: { profitWarning: currentProfit() < 0 }">
  Profit Information
</div>
```

```
<script type="text/javascript">
    var viewModel = {
        currentProfit: ko.observable(150000)
        // Positive value, so initially we don't apply the "profitWarning" class
    };

    viewModel.currentProfit(-50);
    // Causes the "profitWarning" class to be applied
</script>
```

效果就是当 `currentProfit` 小于0的时候，添加 `profitWarning` CSS class 到元素上，如果大于0则删除这个 CSS class。

参数

主参数

该参数是一个 JavaScript 对象，属性是你的 CSS class 名称，值是比较用的 `true` 或 `false`，用来决定是否应该使用这个 CSS class。

你可以一次设置多个 CSS class。例如，如果你的 view model 有一个叫 `isSevere` 的属性，

```
<div data-bind="css: { profitWarning: currentProfit() < 0, majorHighlight: isSevere }">
```

非布尔值会被解析成布尔值。例如，`0`和 `null` 被解析成 `false`，`21`和非 `null` 对象被解析成 `true`。

如果参数是监控属性 `observable` 的，那随着值的变化将会自动添加或者删除该元素上的 CSS class。如果不是，那 CSS class 将会只添加或者删除一次并且以后不在更新。

你可以使用任何 JavaScript 表达式或函数作为参数。KO 将用它的执行结果来决定是否应用或删除 CSS class。

其它参数

无

注：应用的 CSS class 的名字不是合法的 JavaScript 变量命名

如果你想使用 `my-class` class，你不能写成这样：

```
<div data-bind="css: { my-class: someValue }">...</div>
```

... 因为 `my-class` 不是一个合法的命名。解决方案是：在 `my-class` 两边加引号作为一个字符串使用。这是一个合法的 `JavaScript` 对象 文字（从 `JSON` 技术规格说明来说，你任何时候都应该这样使用，虽然不是必须的）。例如，

```
<div data-bind="css: { 'my-class': someValue }">...</div>
```

依赖性

除 `KO` 核心类库外，无依赖。

5 style 绑定

目的

`style` 绑定是添加或删除一个或多个 `DOM` 元素上的 `style` 值。比如当数字变成负数时高亮显示，或者根据数字显示对应宽度的 `Bar`。（注：如果你不是应用 `style` 值而是应用 `CSS class` 的话，请参考 **CSS 绑定**。）

例子

```
<div data-bind="style: { color: currentProfit() < 0 ? 'red' : 'black' }">
    Profit Information
</div>

<script type="text/javascript">
    var viewModel = {
        currentProfit: ko.observable(150000) // Positive value, so initially black
    };
    viewModel.currentProfit(-50); // Causes the DIV's contents to go red
</script>
```

当 `currentProfit` 小于0的时候 `div` 的 `style.color` 是红色，大于的话是黑色。

参数

主参数

该参数是一个 `JavaScript` 对象，属性是你的 `style` 的名称，值是该 `style` 需要应用的值。

你可以一次设置多个 style 值。例如，如果你的 view model 有一个叫 isSevere 的属性，

```
<div data-bind="style: { color: currentProfit() < 0 ? 'red' : 'black', fontWeight: isSevere() ? 'bold' : '' }">...</div>
```

如果参数是监控属性 observable 的，那随着值的变化将会自动添加或者删除该元素上的 style 值。如果不是，那 style 值将会只应用一次并且以后不在更新。

你可以使用任何 JavaScript 表达式或函数作为参数。KO 将用它的执行结果来决定是否应用或删除 style 值。

其它参数

无

注：应用的 style 的名字不是合法的 JavaScript 变量命名

如果你需要应用 font-weight 或者 text-decoration，你不能直接使用，而是要使用 style 对应的 JavaScript 名称。

错误： { font-weight: someValue };

正确： { fontWeight: someValue }

错误： { text-decoration: someValue };

正确： { textDecoration: someValue }

参考：style 名称和对应的 JavaScript 名称列表。

依赖性

除 KO 核心类库外，无依赖。

6 attr 绑定

目的

attr 绑定提供了一种方式可以设置 DOM 元素的任何属性值。你可以设置 img 的 src 属性，连接的 href 属性。使用绑定，当模型属性改变的时候，它会自动更新。

例子

```
<a data-bind="attr: { href: url, title: details }">
  Report
```

```
</a>

<script type="text/javascript">
  var viewModel = {
    url: ko.observable("year-end.html"),
    details: ko.observable("Report including final year-end statistics")
  };
</script>
```

呈现结果是该连接的 href 属性被设置为 year-end.html， title 属性被设置为 Report including final year-end statistics。

参数

主参数

该参数是一个 JavaScript 对象，属性是你的 attribute 名称，值是该 attribute 需要应用的值。

如果参数是监控属性 observable 的，那随着值的变化将会自动添加或者删除该元素上的 attribute 值。如果不是，那 attribute 值将会只应用一次并且以后不在更新。

其它参数

无

注：应用的属性名字不是合法的 JavaScript 变量命名

如果你要用的属性名称是 data-something 的话，你不能这样写：

```
<div data-bind="attr: { data-something: someValue }">...</div>
```

... 因为 data-something 不是一个合法的命名。解决方案是：在 data-something 两边加引号作为一个字符串使用。这是一个合法的 JavaScript 对象 文字（从 JSON 技术规格说明来说，你任何时候都应该这样使用，虽然不是必须的）。例如，

```
<div data-bind="attr: { 'data-something': someValue }">...</div>
```

Knockout应用开发指南 第三章：绑定语法（2）

7 click 绑定

目的

click 绑定在 DOM 元素上添加事件句柄以便元素被点击的时候执行定义的 JavaScript 函数。大部分是用在 button, input 和连接 a 上，但是可以在任意元素上使用。

例子

```
<div>
  You've clicked <span data-bind="text: numberOfClicks"></span> times
  <button data-bind="click: incrementClickCounter">Click me</button>
</div>

<script type="text/javascript">
  var viewModel = {
    numberOfClicks: ko.observable(0),
    incrementClickCounter: function () {
      var previousCount =this.numberOfClicks();
      this.numberOfClicks(previousCount +1);
    }
  };
</script>
```

每次点击按钮的时候，都会调用 incrementClickCounter() 函数，然后更新自动更新点击次数。

参数

主参数

Click 点击事件时所执行的函数。

你可以声明任何 JavaScript 函数 – 不一定非要是 view model 里的函数。你可以声明任意对象上的任何函数，例如： **someObject.someFunction**。

View model 上的函数在用的时候有一点点特殊，就是不需要引用对象的，直接引用函数本身就行了，比如直接写 `incrementClickCounter` 就可以了，而无需写成：
`viewModel.incrementClickCounter`（尽管是合法的）。

其它参数

无

注1：传参数给你的 `click` 句柄

最简单的办法是传一个 `function` 包装的匿名函数：

```
<button data-bind="click: function() { viewModel.myFunction('param1',  
'param2') }">  
    Click me  
</button>
```

这样，KO 就会调用这个匿名函数，里面会执行 `viewModel.myFunction()`，并且传进了 `'param1'` 和 `'param2'` 参数。

注2：访问事件源对象

有些情况，你可能需要使用事件源对象，Knockout 会将这个对象传递到你函数的第一个参数：

```
<button data-bind="click: myFunction">  
    Click me  
</button>  
  
<script type="text/javascript">  
    var viewModel = {  
        myFunction: function (event) {  
            if (event.shiftKey) {  
                //do something different when user has shift key down  
            } else {  
                //do normal action  
            }  
        }  
    };
```

```
</script>
```

如果你需要的话，可以使用匿名函数的第一个参数传进去，然后在里面调用：

```
<button data-bind="click: function(event) { viewModel.myFunction(event,
'param1', 'param2') }">
    Click me
</button>
```

这样，KO 就会将事件源对象传递给你的函数并且使用了。

注3：允许执行默认事件

默认情况下，Knockout 会阻止冒泡，防止默认的事件继续执行。例如，如果你点击一个 **a** 连接，在执行完自定义事件时它不会连接到 **href** 地址。这特别有用是因为你的自定义事件主要就是操作你的 **view model**，而不是连接到另外一个页面。

当然，如果你想让默认的事件继续执行，你可以在你 **click** 的自定义函数里返回 **true**。

注4：控制 **this** 句柄

初学者可以忽略这小节，因为大部分都用不着，高级用户可以参考如下内容：

KO 在调用你定义的函数时，会将 **view model** 传给 **this** 对象（也就是 **ko.applyBindings** 使用的 **view model**）。主要是方便你在调用你在 **view model** 里定义的方法的时候可以很容易再调用 **view model** 里定义的其它属性。例如：**this.someOtherViewModelProperty**。

如果你想引用其它对象，我们有两种方式：

- 你可以和注1里那样使用匿名函数，因为它支持任意 **JavaScript** 对象。
- 你也可以直接引用任何函数对象。你可以使用 **bind** 使 **callback** 函数设置 **this** 为任何你选择的对象。例如：

```
<button data-bind="click: someObject.someFunction.bind(someObject)">
    Click me
</button>
```

如果你是 C# 或 Java 开发人员，你可以疑惑为什么我们还要用 `bind` 函数到一个对象上，特别是像调用 `someObject.someFunction`。原因是在 JavaScript 里，函数自己不是类的一部分，他们在单独存在的对象，有可能多个对象都引用同样的 `someFunction` 函数，所以当这个函数被调用的时候它不知道谁调用的（设置 `this` 给谁）。在你 `bind` 之前运行时是不会知道的。KO 默认情况下设置 `this` 对象是 `view model`，但你可以用 `bind` 语法重定义它。

在注1里使用匿名函数的时候没有具体的要求，因为 JavaScript 代码 `someObject.someFunction()` 就意味着调用 `someFunction`，然后设置 `this` 到 `someObject` 对象上。

注5：防止事件冒泡

默认情况下，Knockout 允许 `click` 事件继续在更高一层的事件句柄上冒泡执行。例如，如果你的元素和父元素都绑定了 `click` 事件，那当你点击该元素的时候两个事件都会触发的。如果需要，你可以通过额外的绑定 `clickBubble` 来禁止冒泡。例如：

```
<div data-bind="click: myDivHandler">
  <button data-bind="click: myButtonHandler, clickBubble: false">
    Click me
  </button>
</div>
```

默认情况下，`myButtonHandler` 会先执行，然后会冒泡执行 `myDivHandler`。但一旦你设置了 `clickBubble` 为 `false` 的时候，冒泡事件会被禁止。

依赖性

除 KO 核心类库外，无依赖。

8 event 绑定

目的

event 绑定在 DOM 元素上添加指定的事件句柄以便元素被触发的时候执行定义的 JavaScript 函数。大部分情况下是用于 `keypress`，`mouseover` 和 `mouseout` 上。

例子

```
<div>
  <div data-bind="event: { mouseover: enableDetails, mouseout:
disableDetails }">
    Mouse over me
  </div>
  <div data-bind="visible: detailsEnabled">
    Details
  </div>
</div>

<script type="text/javascript">
  var viewModel = {
    detailsEnabled: ko.observable(false),
    enableDetails: function () {
      this.detailsEnabled(true);
    },
    disableDetails: function () {
      this.detailsEnabled(false);
    }
  };
</script>
```

每次鼠标在第一个元素上移入移出的时候都会调用 view model 上的方法来 toggle detailsEnabled 的值，而第二个元素会根据 detailsEnabled 的值自动显示或者隐藏。

参数

主参数

你需要传入的是一个 JavaScript 对象，他的属性名是事件名称，值是你所需要执行的函数。

你可以声明任何 JavaScript 函数 – 不一定非要是 view model 里的函数。你可以声明任意对象上的任何函数，例如： `event: { mouseover: someObject.someFunction }`。

View model 上的函数在用的时候有一点点特殊，就是不需要引用对象的，直接引用函数本身就行了，比如直接写 `event: { mouseover: enableDetails }` 就可以了，而无需写成：`event: { mouseover: viewModel.enableDetails }`（尽管是合法的）。

其它参数

无

注1：传参数给你的 click 句柄

最简单的办法是传一个 function 包装的匿名函数：

```
<button data-bind="event: { mouseover: function()
{ viewModel.myFunction('param1', 'param2') } }">
    Click me
</button>
```

这样，KO 就会调用这个匿名函数，里面会执行 `viewModel.myFunction()`，并且传进了 'param1' 和 'param2' 参数。

注2：访问事件源对象

有些情况，你可能需要使用事件源对象，Knockout 会将这个对象传递到你函数的第一个参数：

```
<div data-bind="event: { mouseover: myFunction }">
    Mouse over me
</div>

<script type="text/javascript">
    var viewModel = {
        myFunction: function (event) {
            if (event.shiftKey) {
                //do something different when user has shift key down
            } else {
                //do normal action
            }
        }
    };
</script>
```

如果你需要的话，可以使用匿名函数的第一个参数传进去，然后在里面调用：

```
<div data-bind="event: { mouseover: function(event) { viewModel.myFunction(event,
'param1', 'param2') } }">
    Mouse over me
</div>
```


这样，KO 就会将事件源对象传递给你的函数并且使用了。

注3：允许执行默认事件

默认情况下，Knockout 会阻止冒泡，防止默认的事件继续执行。例如，如果在一个 `input` 标签上绑定一个 `keypress` 事件，当你输入内容的时候，浏览器只会调用你的函数而不是天价你输入的值。另外一个例子 `click` 绑定，当你点击一个 `a` 连接，在执行完自定义事件时它不会连接到 `href` 地址。因为你的自定义事件主要就是操作你的 `view model`，而不是连接到另外一个页面。

当然，如果你想让默认的事件继续执行，你可以在你 `event` 的自定义函数里返回 `true`。

注4：控制 `this` 句柄

初学者可以忽略这小节，因为大部分都用不着，高级用户可以参考如下内容：

KO 在调用你定义的 `event` 绑定函数时，会将 `view model` 传给 `this` 对象（也就是 `ko.applyBindings` 使用的 `view model`）。主要是方便你在调用你在 `view model` 里定义的方法的时候可以很容易再调用 `view model` 里定义的其它属性。例如：

`this.someOtherViewModelProperty`。

如果你想引用其它对象，我们有两种方式：

- 你可以和注1里那样使用匿名函数，因为它支持任意 JavaScript 对象。
- 你也可以直接引用任何函数对象。你可以使用 `bind` 使 `callback` 函数设置 `this` 为任何你选择的对象。例如：

```
<div data-bind="event: { mouseover:
someObject.someFunction.bind(someObject) }">
    Mouse over me
</div>
```

如果你是 C# 或 Java 开发人员，你可以疑惑为什么我们还要用 `bind` 函数到一个对象想，特别是像调用 `someObject.someFunction`。原因是在 JavaScript 里，函数自己不是类的一部分，他们在单独存在的对象，有可能多个对象都引用同样的 `someFunction` 函数，所以当这个函数被调用的时候它不知道谁调用的（设置 `this` 给谁）。在你 `bind` 之前运行时是不会知道的。KO 默认情况下设置 `this` 对象是 `view model`，但你可以用 `bind` 语法重定义它。

在注1里使用匿名函数的时候没有具体的要求，因为 JavaScript 代码 `someObject.someFunction()` 就意味着调用 `someFunction`，然后设置 `this` 到 `someObject` 对象上。

注5：防止事件冒泡

默认情况下，Knockout 允许 event 事件继续在更高层的事件句柄上冒泡执行。例如，如果你的元素和父元素都绑定了 `mouseover` 事件，那么如果你的鼠标在该元素移动的时候两个事件都会触发的。如果需要，你可以通过额外的绑定 `youreventBubble` 来禁止冒泡。例如：

```
<div data-bind="event: { mouseover: myDivHandler }">
  <button data-bind="event: { mouseover: myButtonHandler }, mouseoverBubble:
false">
    Click me
  </button>
</div>
```

默认情况下，`myButtonHandler` 会先执行，然后会冒泡执行 `myDivHandler`。但一旦你设置了 `mouseoverBubble` 为 `false` 的时候，冒泡事件会被禁止。

依赖性

除 KO 核心类库外，无依赖。

9 submit 绑定

目的

`submit` 绑定在 `form` 表单上添加指定的事件句柄以便该 `form` 被提交的时候执行定义的 JavaScript 函数。只能用在表单 `form` 元素上。

当你使用 `submit` 绑定的时候，Knockout 会阻止 `form` 表单默认的 `submit` 动作。换句话说，浏览器会执行你定义的绑定函数而不会提交这个 `form` 表单到服务器上。可以很好地解释这个，使用 `submit` 绑定就是为了处理 `view model` 的自定义函数的，而不是再使用普通的 HTML `form` 表单。如果你要继续执行默认的 HTML `form` 表单操作，你可以在你的 `submit` 句柄里返回 `true`。

例子

```
<form data-bind="submit: doSomething">
  ... form contents go here ...
  <button type="submit">Submit</button>
</div>

<script type="text/javascript">
  var viewModel = {
    doSomething: function (formElement) {
      // ... now do something
    }
  };
</script>
```

这个例子里，KO 将把整个 form 表单元素作为参数传递到你的 submit 绑定函数里。你可以忽略不管，但是有些例子里是否有用，参考：[ko.postJson](#) 工具。

为什么不在 submit 按钮上使用 click 绑定？

在 form 上，你可以使用 click 绑定代替 submit 绑定。不过 submit 可以 handle 其它的 submit 行为，比如在输入框里输入回车的时候可以提交表单。

参数

主参数

你绑定到 submit 事件上的函数

你可以声明任何 JavaScript 函数 – 不一定非要是 view model 里的函数。你可以声明任意对象上的任何函数，例如：[submit: someObject.someFunction](#)。

View model 上的函数在用的时候有一点点特殊，就是不需要引用对象的，直接引用函数本身就行了，比如直接写 [submit: doSomething](#) 就可以了，而无需写成：[submit: viewModel.doSomething](#)（尽管是合法的）。

其它参数

无

备注:

关于如果传递更多的参数给 submit 绑定函数，或者当调用非 view model 里的函数的时如何控制 this，请参考 click 绑定。所有 click 绑定相关的 notes 也都适用于 submit 绑定。

依赖性

除 KO 核心类库外，无依赖。

10 enable 绑定

目的

enable 绑定使 DOM 元素只有在参数值为 true 的时候才 enabled。在 form 表单元素 input, select, 和 textarea 上非常有用。

例子

```
<p>
  <input type='checkbox' data-bind="checked: hasCellphone"/>
  I have a cellphone
</p>

<p>
  Your cellphone number:
  <input type='text' data-bind="value: cellphoneNumber, enable:
hasCellphone"/>
</p>

<script type="text/javascript">
  var viewModel = {
    hasCellphone: ko.observable(false),
    cellphoneNumber: ""
  };
</script>
```

这个例子里，“Your cellphone number”后的 text box 初始情况下是禁用的，只有当用户点击标签 “I have a cellphone”的时候才可用。

参数

主参数

声明 DOM 元素是否可用 `enabled`。

非布尔值会被解析成布尔值。例如 `0` 和 `null` 被解析成 `false`，`21` 和非 `null` 对象被解析给 `true`。

如果你的参数是 `observable` 的，那绑定会随着 `observable` 值的改变而自动更新 `enabled/disabled` 状态。如果不是，则只会设置一次并且以后不再更新。

其它参数

无

注：任意使用 JavaScript 表达式

不紧紧限制于变量 – 你可以使用任何 JavaScript 表达式来控制元素是否可用。例如，

```
<button data-bind="enabled: parseAreaCode(viewModel.cellphoneNumber()) !=  
'555'">  
    Do something  
</button>
```

依赖性

除 KO 核心类库外，无依赖。

11 disable 绑定

目的

`disable` 绑定使 DOM 元素只有在参数值为 `true` 的时候才 `disabled`。在 form 表单元素 `input`，`select`，和 `textarea` 上非常有用。

`disable` 绑定和 `enable` 绑定正好相反，详情请参考 `enable` 绑定。

Knockout应用开发指南 第三章：绑定语法（3）

12 value 绑定

目的

value 绑定是关联 DOM 元素的值到 view model 的属性上。主要是用在表单控件

当用户编辑表单控件的时候，view model 对应的属性值会自动更新。同样，当你更新 view model 属性的时候，相对应的元素值在页面上也会自动更新。

注：如果你在 checkbox 或者 radio button 上使用 checked 绑定来读取或者写入元素的 checked 状态，而不是 value 值的绑定。

例子

```
<p>Login name: <input data-bind="value: userName"/></p>
<p>Password: <input type="password" data-bind="value: userPassword"/></p>

<script type="text/javascript">
    var viewModel = {
        userName: ko.observable(""), // Initially blank
        userPassword: ko.observable("abc"), // Prepopulate
    };
</script>
```

参数

主参数

KO 设置此参数为元素的 value 值。之前的值将被覆盖。

如果参数是监控属性 observable 的，那元素的 value 值将根据参数值的变化而更新，如果不是，那元素的 value 值将只设置一次并且以后不在更新。

如果你提供的参数不是一个数字或者字符串（而是对象或者数组）的话，那显示的 value

值就是 `yourParameter.toString()` 的内容（通常没用，所以最好都设置为数字或者字符串）。

不管什么时候，只要你更新了元素的值，那 KO 都会将 **view model** 对应的属性值自动更新。默认情况下当用户离开焦点（例如 `onchange` 事件）的时候，KO 才更新这个值，但是你可以通过第2个参数 `valueUpdate` 来特别指定改变值的时机。

其它参数

valueUpdate

如果你使用 `valueUpdate` 参数，那就是意味着 KO 将使用自定义的事件而不是默认的离开焦点事件。下面是一些最常用的选项：

“**change**”（默认值） - 当失去焦点的时候更新 **view model** 的值，或者是 `<select>` 元素被选择的时候。

“**keyup**” - 当用户敲完一个字符以后立即更新 **view model**。

“**keypress**” - 当用户正在敲一个字符但没有释放键盘的时候就立即更新 **view model**。不像 `keyup`，这个更新和 `keydown` 是一样的。

“**afterkeydown**” - 当用户开始输入字符的时候就更新 **view model**。主要是捕获浏览器的 `keydown` 事件或异步 `handle` 事件。

上述这些选项，如果你想让你的 **view model** 进行实时更新，使用“**afterkeydown**”是最好的选择。

例子：

```
<p>Your value: <input data-bind="value: someValue, valueUpdate:
'afterkeydown'"/></p>
<p>You have typed: <span data-bind="text: someValue"></span></p> <!-- updates
in real-time -->

<script type="text/javascript">
    var viewModel = {
        someValue: ko.observable("edit me")
    };
</script>
```

注1: 绑定下拉菜单 drop-down list (例如 SELECT)

Knockout 对下拉菜单 drop-down list 绑定有一个特殊的支持, 那就是在读取和写入绑定的时候, 这个值可以是任意 JavaScript 对象, 而不必非得是字符串。在你让你用户选择一组 model 对象的时候非常有用。具体例子, 参考 options 绑定。

类似, 如果你想创建一个 multi-select list, 参考 selectedOptions 绑定。

注2: 更新 observable 和 non-observable 属性值

如果你用 value 绑定将你的表单元素和你的 observable 属性关联起来, KO 设置的2-way 的双向绑定, 任何一方改变都会更新另外一方的值。

但是, 如果你的元素绑定的是一个 non-observable 属性 (例如是一个原始的字符串或者 JavaScript 表达式), KO 会这样执行:

- 如果你绑定的 non-observable 属性是简单对象, 例如一个常见的属性值, KO 会设置这个值为 form 表单元素的初始值, 如果你改变 form 表单元素的值, KO 会将值重新写回到 view mode 的这个属性。但当这个属性自己改变的时候, 元素却不会再变化了 (因为不是 observable 的), 所以它仅仅是1-way 绑定。
- 如果你绑定的 non-observable 属性是复杂对象, 例如复杂的 JavaScript 表达式或者子属性, KO 也会设置这个值为 form 表单元素的初始值, 但是改变 form 表单元素的值的时候, KO 不会再写会 view model 属性, 这种情况叫 one-time-only value setter, 不是真正的绑定。

例子:

```
<p>First value: <input data-bind="value: firstValue"/></p>           <!-- two-way
binding -->
<p>Second value: <input data-bind="value: secondValue"/></p>       <!-- one-way
binding -->
<p>Third value: <input data-bind="value: secondValue.length"/></p> <!-- no
binding -->

<script type="text/javascript">
    var viewModel = {
        firstValue: ko.observable("hello"), // Observable
        secondValue: "hello, again" // Not observable
    };
    ko.applyBindings(viewModel);
```



```
</script>
```

依赖性

除 KO 核心类库外，无依赖。

13 checked 绑定

目的

checked 绑定是关联到 checkable 的 form 表单控件到 view model 上 - 例如 checkbox (`<input type='checkbox'>`) 或者 radio button (`<input type='radio'>`)。当用户 check 关联的 form 表单控件的时候，view model 对应的值也会自动更新，相反，如果 view model 的值改变了，那控件元素的 check/uncheck 状态也会跟着改变。

注：对 text box, drop-down list 和所有 non-checkable 的 form 表单控件，用 [value 绑定](#) 来读取和写入是该元素的值，而不是 checked 绑定。

例子

```
<p>Send me spam: <input type="checkbox" data-bind="checked: wantsSpam" /></p>

<script type="text/javascript">
    var viewModel = {
        wantsSpam: ko.observable(true) // Initially checked
    };

    // ... then later ...
    viewModel.wantsSpam(false); // The checkbox becomes unchecked
</script>
```

Checkbox 关联到数组

```
<p>Send me spam: <input type="checkbox" data-bind="checked: wantsSpam" /></p>
<div data-bind="visible: wantsSpam">
    Preferred flavors of spam:
    <div><input type="checkbox" value="cherry" data-bind="checked:
spamFlavors" /> Cherry</div>
```

```

    <div><input type="checkbox" value="almond" data-bind="checked:
spamFlavors"/> Almond</div>

    <div><input type="checkbox" value="msg" data-bind="checked: spamFlavors"/>
Monosodium Glutamate</div>
</div>

<script type="text/javascript">

    var viewModel = {
        wantsSpam: ko.observable(true),
        spamFlavors: ko.observableArray(["cherry", "almond"]) // Initially checks
the Cherry and Almond checkboxes
    };

    // ... then later ...
    viewModel.spamFlavors.push("msg"); // Now additionally checks the Monosodium
Glutamate checkbox
</script>

```

添加 radio button

```

<p>Send me spam: <input type="checkbox" data-bind="checked: wantsSpam"/></p>

<div data-bind="visible: wantsSpam">
    Preferred flavor of spam:
    <div><input type="radio" name="flavorGroup" value="cherry"
data-bind="checked: spamFlavor"/> Cherry</div>
    <div><input type="radio" name="flavorGroup" value="almond"
data-bind="checked: spamFlavor"/> Almond</div>
    <div><input type="radio" name="flavorGroup" value="msg" data-bind="checked:
spamFlavor"/> Monosodium Glutamate</div>
</div>

<script type="text/javascript">

    var viewModel = {
        wantsSpam: ko.observable(true),
        spamFlavor: ko.observable("almond") // Initially selects only the Almond
radio button
    };

```

```
// ... then later ...  
viewModel.spamFlavor("msg"); // Now only Monosodium Glutamate is checked  
</script>
```

参数

主参数

KO 会设置元素的 **checked** 状态匹配到你的参数上，之前的值将被覆盖。对参数的解析取决于你元素的类型：

对于 **checkbox**，当参数为 **true** 的时候，KO 会设置元素的状态为 **checked**，反正设置为 **unchecked**。如果你传的参数不是布尔值，那 KO 将会解析成布尔值。也就是说非 **0** 值和非 **null** 对象，非空字符串将被解析成 **true**，其它值都被解析成 **false**。

当用户 **check** 或者 **uncheck** 这个 **checkbox** 的时候，KO 会将 **view model** 的属性值相应地设置为 **true** 或者 **false**。

一个特殊情况是参数是一个数组，如果元素的值存在于数组，KO 就会将元素设置为 **checked**，如果数组里不存在，就设置为 **unchecked**。如果用户对 **checkbox** 进行 **check** 或 **uncheck**，KO 就会将元素的值添加数组或者从数组里删除。

对于 **radio buttons**，KO 只有当参数值等于 **radio button value** 属性值的时候才设置元素为 **checked** 状态。所以参数应是字符串。在上面的例子里只有当 **view model** 的 **spamFlavor** 属性等于“**almond**”的时候，该 **radio button** 才会设置为 **checked**。

当用户将一个 **radio button** 选择上的时候 **is selected**，KO 会将该元素的 **value** 属性值更新到 **view model** 属性里。上面的例子，当点击 **value=“cherry”** 的选项上，**viewModel.spamFlavor** 的值将被设置为“**cherry**”。

当然，最有用的是设置一组 **radio button** 元素对应到一个单个的 **view model** 属性。确保一次只能选择一个 **radio button** 需要将他们的 **name** 属性名都设置成一样的值（例如上个例子的 **flavorGroup** 值）。这样的话，一次就只能选择一个了。

如果参数是监控属性 **observable** 的，那元素的 **checked** 状态将根据参数值的变化而更新，如果不是，那元素的 **value** 值将只设置一次并且以后不在更新。

其它参数

无

依赖性

除 KO 核心类库外，无依赖。

14 options 绑定

目的

options 绑定控制什么样的 options 在 drop-down 列表里(例如: <select>)或者 multi-select 列表里 (例如: <select size='6'>) 显示。此绑定不能用于<select>之外的元素。关联的数据应是数组 (或者是 observable 数组), <select>会遍历显示数组里的所有的项。

注: 对于 multi-select 列表, 设置或者获取选择的多项需要使用 [selectedOptions](#) 绑定。对于 single-select 列表, 你也可以使用 [value](#) 绑定读取或者设置元素的 selected 项。

例1: Drop-down list

```
<p>Destination country: <select data-bind="options:
availableCountries"></select></p>

<script type="text/javascript">
    var viewModel = {
        availableCountries: ko.observableArray(['France', 'Germany', 'Spain'])
    // These are the initial options
    };

    // ... then later ...
    viewModel.availableCountries.push('China'); // Adds another option
</script>
```

例2: Multi-select list

```
<p>Choose some countries you'd like to visit: <select data-bind="options:
```

```

availableCountries" size="5" multiple="true"></select></p>

<script type="text/javascript">
    var viewModel = {
        availableCountries: ko.observableArray(['France', 'Germany', 'Spain'])
    };
</script>

```

例3: Drop-down list 展示的任意 JavaScript 对象，不仅仅是字符串

```

<p>
    Your country:
    <select data-bind="options: availableCountries,
        optionsText: 'countryName', value: selectedCountry,
optionsCaption: 'Choose...'"></select>
</p>

<div data-bind="visible: selectedCountry"> <!-- Appears when you select something
-->
    You have chosen a country with population
    <span data-bind="text: selectedCountry() ?
selectedCountry().countryPopulation : 'unknown'"></span>.
</div>

<script type="text/javascript">
    // Constructor for an object with two properties
var country =function (name, population) {
    this.countryName = name;
    this.countryPopulation = population;
};

    var viewModel = {
        availableCountries: ko.observableArray([
            new country("UK", 65000000),
            new country("USA", 320000000),
            new country("Sweden", 29000000)
        ]),
        selectedCountry: ko.observable() // Nothing selected by default
    };
</script>

```

例4: Drop-down list 展示的任意 JavaScript 对象，显示 text 是 function 的返回值

```
<!-- Same as example 3, except the <select> box expressed as follows: -->

<select data-bind="options: availableCountries,
    optionsText: function(item) {
        return item.countryName + ' (pop: ' + item.countryPopulation
+ ' ) '
    },
    value: selectedCountry,
    optionsCaption: 'Choose...'"></select>
```

注意例3和例4在 optionsText 值定义上的不同。

参数

主参数

该参数是一个数组（或者 observable 数组）。对每个 item，KO 都会将它作为一个 <option> 添加到<select>里，之前的 options 都将被删除。

如果参数是一个 string 数组，那你不需要再声明任何其它参数。<select>元素会将每个 string 显示为一个 option。不过，如果你让用户选择的是一个 JavaScript 对象数组（不仅仅是 string），那就需要设置 optionsText 和 optionsValue 这两个参数了。

如果参数是监控属性 observable 的，那元素的 options 项将根据参数值的变化而更新，如果不是，那元素的 value 值将只设置一次并且以后不在更新。

其它参数

optionsCaption

有时候，默认情况下不想选择任何 option 项。但是 single-select drop-down 列表由于每次都要默认选择以项目，怎么避免这个问题呢？常用的方案是加一个“请选择的”或者“Select an item”的提示语，或者其它类似的，然后让这个项作为默认选项。

我们使用 optionsCaption 参数就能很容易实现，它的值是字符串型，作为默认项

显示。例如：

```
<select data-bind='options: myOptions, optionsCaption: "Select an item...",
value: myChosenValue'></select>
```

KO 会在所有选项上加上这一个项，并且设置 value 值为 undefined。所以，如果 myChosenValue 被设置为 undefined（默认是 observable 的），那么上述的第一个项就会被选中。

optionsText

上面的例3展示的绑定 JavaScript 对象到 option 上 – 不仅仅是字符串。这时候你需要设置这个对象的那个属性作为 drop-down 列表或 multi-select 列表的 text 来显示。例如，例3中使用的是设置额外的参数 optionsText 将对象的属性名 countryName 作为显示的文本。

如果不想仅仅显示对象的属性值作为每个 item 项的 text 值，那你可以设置 optionsText 为 JavaScript 函数，然后再函数里通过自己的逻辑返回相应的值（该函数参数为 item 项本身）。例4展示的就是返回 item 的2个属性值合并的结果。

optionsValue

和 optionsText 类似，你也可以通过额外参数 optionsValue 来声明对象的那个属性值作为该<option>的 value 值。

经典场景：如在更新 options 的时候想保留原来的已经选择的项。例如，当你重复多次调用 Ajax 获取 car 列表的时候，你要确保已经选择的某个 car 一直都是被选择上，那你就需要设置 optionsValue 为“carId”或者其它的 unique 标示符，否则的话 KO 找不知道之前选择的 car 是新 options 里的哪一项。

selectedOptions

对于 multi-select 列表，你可以用 selectedOptions 读取和设置多个选择项。技术上看它是一个单独的绑定，有自己的文档，请参考：[selectedOptions 绑定](#)。

注：已经被选择的项会再 options 改变的时候保留

当使用 `options` 绑定 `<select>` 元素的时候，如果 `options` 改变，KO 将尽可能保留之前已经被选择的项不变（除非是你事先手工删除一个或多个已经选择的项）。这是因为 `options` 绑定尝试依赖 `value` 值的绑定（`single-select` 列表）和 `selectedOptions` 绑定（`multi-select` 列表）。

依赖性

除 KO 核心类库外，无依赖。

15 selectedOptions 绑定

目的

`selectedOptions` 绑定用于控制 `multi-select` 列表已经被选择的元素，用在使用 `options` 绑定的 `<select>` 元素上。

当用户在 `multi-select` 列表选择或反选一个项的时候，会将 `view model` 的数组进行相应的添加或者删除。同样，如果 `view model` 上的这个数组是 `observable` 数组的话，你添加或者删除任何 `item`（通过 `push` 或者 `splice`）的时候，相应的 UI 界面里的 `option` 项也会被选择上或者反选。这种方式是 **2-way** 绑定。

注：控制 `single-select` 下拉菜单选择项，你可以使用 `value` 绑定。

例子

```
<p>
  Choose some countries you'd like to visit:
  <select data-bind="options: availableCountries, selectedOptions:
chosenCountries" size="5" multiple="true"></select>
</p>

<script type="text/javascript">
  var viewModel = {
    availableCountries: ko.observableArray(['France', 'Germany', 'Spain']),
    chosenCountries: ko.observableArray(['Germany']) // Initially, only
Germany is selected
  };

  // ... then later ...
```



```
viewModel.chosenCountries.push('France'); // Now France is selected too
</script>
```

参数

主参数

该参数是数组（或 observable 数组）。KO 设置元素的已选项为和数组里 `match` 的项，之前的已选项将被覆盖。

如果参数是依赖监控属性 observable 数组，那元素的已选项 `selected options` 项将根据参数值的变化（通过 `push`，`pop`，或其它 [observable 数组方法](#)）而更新，如果不是，那元素的已选项 `selected options` 将只设置一次并且以后不在更新。

不管该参数是不是 observable 数组，用户在 `multi-select` 列表里选择或者反选的时候，KO 都会探测到，并且更新数组里的对象以达到同步的结果。这样你就可以获取 `options` 已选项。

其它参数

无

注：支持让用户选择任意 JavaScript 对象

在上面的例子里，用户可以选择数组里的字符串值，但是选择不限于字符串，如果你愿意你可以声明包含任意 JavaScript 对象的数组，查看 [options 绑定](#) 如何显示 JavaScript 对象到列表里。

这种场景，你可以用 `selectedOptions` 来读取或设置这些对象本身，而不是页面上显示的 `option` 表示形式，这样做在大部分情况下都非常清晰。`view model` 就可以探测到你从数组对象里选择的项了，而不必关注每个项和页面上展示的 `option` 项是如何 `map` 的。

依赖性

除 KO 核心类库外，无依赖。

16 uniqueName 绑定

目的

`uniqueName` 绑定确保所绑定的元素有一个非空的 `name` 属性。如果该元素没有 `name` 属性，那绑定会给它设置一个 `unique` 的字符串值作为 `name` 属性。你不会经常用到它，只有在某些特殊的场景下才用到，例如：

- 在使用 KO 的时候，一些技术可能依赖于某些元素的 `name` 属性，尽快他们没有什么意义。例如，`jQuery Validation` 验证当前只验证有 `name` 属性的元素。为配合 `Knockout UI` 使用，有些时候需要使用 `uniqueName` 绑定避免让 `jQuery Validation` 验证出错。
- IE 6 下，如果 `radio button` 没有 `name` 属性是不允许被 `checked` 了。大部分时候都没问题，因为大部分时候 `radio button` 元素都会有 `name` 属性的作为一组互相的 `group`。不过，如果你没声明，KO 内部会在这些元素上使用 `uniqueName` 那么以确保他们可以被 `checked`。

例子

```
<input data-bind="value: someModelProperty, uniqueName: true"/>
```

参数

主参数

就像上面的例子一样，传入 `true`（或者可以转成 `true` 的值）以启用 `uniqueName` 绑定。

其它参数

无

依赖性

除 KO 核心类库外，无依赖。

Knockout应用开发指南 第四章：模板绑定

模板绑定 The template binding

目的

template 绑定通过模板将数据 render 到页面。模板绑定对于构建嵌套结构的页面非常方便。默认情况， Knockout 用的是流行的 jquery.tmpl 模板引擎。使用它的话，需要在安装页面下载和引用 jquery.tmpl 和 jQuery 框架。或者你也可以集成其它的模板引擎（虽然需要了解 Knockout 内部知识才行）。

例子

```
<div data-bind='template: "personTemplate"'> </div>
<script id='personTemplate' type='text/html'>
    ${ name } is ${ age } years old
    <button data-bind='click: makeOlder'>Make older</button>
</script>

<script type='text/javascript'>
    var viewModel = {
        name: ko.observable('Bert'),
        age: ko.observable(78),
        makeOlder: function () {
            this.age(this.age() +1);
        }
    };
    ko.applyBindings(viewModel);
</script>
```

当引用的 observable(dependent observable)数据改变的时候, Knockout 会自动重新render 模板。在这个例子里，每次点击 button 的时候都会重新 render 模板。

语法

你可以使用任何你模板引擎支持的语法。jquery.tmpl 执行如下语法：

- `${ someValue }` — [参考文档](#)
- `{{html someValue}}` — [参考文档](#)

- `{{if someCondition}}` — [参考文档](#)
- `{{else someCondition}}` — [参考文档](#)
- `{{each someArray}}` — [参考文档](#)

和 **observable** 数组一起使用`{{each}}`

当然使用`{{each someArray}}`的时候，如果你的值是 **observableArray**，你必须使用 JavaScript 类型的基础数组类型`{{each myObservableArray()}}`，而不是`{{each myObservableArray}}`。

参数

主参数

语法快速记忆：如果你声明的仅仅是字符串（上个例子），KO 会使用模板的 ID 来 render。应用在模板上的数据是你的整个 view model 对象（例如 `ko.applyBindings` 绑定的对象）。

更多控件，你可以传带有如下属性的 JavaScript 对象：

name（必选项） — 需要 render 的模板 ID – 参考 注5 如何使用 function 函数声明 ID。

data（可选项） — 需要 render 到模板的数据。如果你忽略整个参数，KO 将查找 foreach 参数，或者是应用整个 view model 对象。

foreach（可选项） — 指定 KO 按照“foreach”模式 render 模板 – 参考 注3。

afterAdd 或 **beforeRemove**（可选项） — 在 foreach 模式下使用 callback 函数。

templateOptions（可选项） — 在 render 模板的时候，传递额外数据以便使用。参考 注6。

传递多个参数的例子：

```
<div data-bind='template: { name: "personTemplate", data: someObject }'> </div>
```

注1: Render 嵌套模板

因为在模板里使用的是 `data-bind` 属性来声明的，所以嵌套模板你可以再次使用 `data-bind='template: ...'`，在上层模板的元素里。

这比模板引起的原生语法好用多了（例如 `jquery.tmpl` 里的 `{{tmpl}}`）。Knockout 语法的好处在于可以在每层模板的跟着相关的依赖值，所以如果依赖改变了，KO 将只会重新 render 依赖所在的那个模板。这将极大地改善了性能。

注2: `${ val }`和``有何不同?

当你在模板内部使用 `data-bind` 属性的时候，KO 是单独为这个绑定单独跟踪依赖项的。当 `model` 改变的时候，KO 只会更新绑定的元素以及子元素而不需要重新 render 整个模板。所以如果你声明这样的代码是 ``，当 `someObservableValue` 改变的时候，KO 将只是简单地更新 `` 元素的 `text` 值而不需要重新 render 整个模板。

不过，如果模板内部使用的 `observable` 值（例如 `${someObservableValue}`），如果这个 `observable` 值改变了，那 KO 将重新 render 整个模板。

这就是说，很多情况下 `` 性能要比 `${ someObservableValue }` 要好，因为值改变的话不会影响临近元素的状态。不过 `${ someObservableValue }` 语法比较简洁，如果你的模板比较小的话，还是更合适的，不会带来大的性能问题。

注3: 使用 `foreach`

如果需要为集合里的每一个 item render 一次模板，有2种方式：

你可以使用模板引擎里的原生“each”语法，对 `jquery.tmpl` 来说就是用 `{{each}}` 语法迭代数组。

另外一种方式就是用 Knockout 的 `foreach` 模式来 render。

例子:

```
<div data-bind='template: { name: "personTemplate",  
    foreach: someObservableArrayOfPeople }'> </div>
```

foreach 模板模式的好处是：

- 当往你的 **collection** 集合里添加新 **item** 项的时候，KO 只会对这个新 **item** 进行 **render** 模板，并且将结果附加到现有的 **DOM** 上。
- 当从 **collection** 集合里删除 **item** 的时候，KO 将不会重新 **render** 任何模板，而只是简单地删除相关的元素。
- KO 允许通过自定义的方式声明 **afterAdd** 和 **beforeRemove** 的 **callback** 函数添加/删除 **DOM** 元素。然后这个 **callback** 会在删除元素的时候进行一些动画或者其它操作。

与原生的 **each** 不同之处是：在改变之后，模板引擎强制重新 **render** 模板里所有的内容，因为它根本就不关注 KO 里所谓的依赖跟踪内容。

关于使用 **foreach** 模式的例子，参考 **grid editor** 和 **animated transitions**。

注4：使用 **afterRender** 选项

有时候，你需要在模板生成的 **DOM** 元素上深度定义逻辑。例如，你可能想再模板输出的时候进行截获，然后在 **render** 的元素上允许 **jQuery UI** 命令（比如 **date picker**, **slider**，或其它）。

你可以使用 **afterRender** 选项，简单声明一个 **function** 函数（匿名函数或者 **view model** 里的函数），在 **render** 或者重新 **render** 模板之后 **Knockout** 会重新调用它。如果你使用的是 **foreach**，那在每个 **item** 添加到 **observable** 数组之后，**Knockout** 会立即调用 **afterRender** 的 **callback** 函数。例如，

```
<div data-bind='template: { name: "personTemplate",
                        data: myData,
                        afterRender: myPostProcessingLogic }'> </div>
```

... 在 **view model** 里声明一个类似的函数（例如，对象包含 **myData**）：

```
viewModel.myPostProcessingLogic = function (elements) {
    // "elements" is an array of DOM nodes just rendered by the template
    // You can add custom post-processing logic here
}
```

注5：动态决定使用哪个模板

有时候，你可能需要根据数据的状态来决定使用哪个模板的 ID。可以通过 `function` 的返回 ID 应用到 `name` 选择上。如果你用的是 `foreach` 模板模式，Knockout 会对每个 `item` 执行 `function`（将 `item` 作为参数）从而将返回值作为 ID，否则，该 `function` 接受的参数是整个 `data option` 或者是整个 `view model`。

例子：

```
<ul data-bind='template: { name: displayMode,
                        foreach: employees }'> </ul>
<script type='text/javascript'>
var viewModel = {
    employees: ko.observableArray([
        { name: "Kari", active: ko.observable(true) },
        { name: "Brynn", active: ko.observable(false) },
        { name: "Nora", active: ko.observable(false) }
    ]),
    displayMode: function (employee) {
        return employee.active() ? "active" : "inactive";
        // Initially "Kari" uses the "active" template, while the others use
        "inactive"
    }
};

// ... then later ...
viewModel.employees()[1].active(true);
// Now "Brynn" is also rendered using the "active" template.
</script>
```

如果你的 `function` 引用的是 `observable` 值，那当这些值改变的时候，绑定的值会随着改变的。这将导致相应的模板重新 `render`。

注6：使用 `templateOptions` 传递额外的参数

如果你在绑定模板的时候需要传入额外的数据的话，你可以使用 `templateOptions` 对象来传递这些值。这可以帮助你通过一些 不属于 `view model` 过滤条件或者字符来重用模板。另外一个好处是用在范围控制，你可以引用通过你的模板访问数据的数据。

例子，

```
<ul data-bind='template: { name: "personTemplate",
```

```

        foreach: employees,
        templateOptions: { label: "Employee:",
                           selectedPerson: selectedEmployee } }'>
</ul>

<script id='personTemplate' type='text/html'>
    <div data-bind="css: { selected: $data === $item.selectedPerson() }">
        ${ $item.label } <input data-bind="value: name" />
    </div>
</script>

```

在整个例子里，`personTemplate` 有可能都使用 `employee` 和自定义对象。通过 `templateOptions` 我们可以传递一个字符 `label` 和当前已选择项作为 `selectedPerson` 来控制 `style`。在 `jquery.tmpl` 模板里，这些值可以通过访问 `$item` 对象的属性得到。

注7：模板是被预编译和缓存的

为了最大性能，Knockout 内嵌模板引擎 `jquery.tmpl` 会利用自身的功能对你的模板进行预编译成可执行的 JavaScript 代码，然后从编译流程里缓存输出。这将使模板更快更加具有可执行性，尤其是使用 `foreach` 循环来 `render` 相同模板的时候。

一般情况你不会注意到这个，所以经常会忘记。不过，当你在某种原因下通过编程重写模板 `<script>` 元素的时候并且该模板之前已经用过一次的话，你的改变不会带来任何 `render` 的变化，因为在第一次使用的时候已经预编译了并且缓存起来了。（如果这些会带来问题，我们将考虑在 KO 新版本里提供一个禁用或重设模板缓存的功能，不过好像没有好的原因去动态改变模板 `<script>` 元素的内容）。

注8：使用不同的模板引擎

如果你想使用不同的 JavaScript 模板引擎（或者是因为某些原因你不想在 jQuery 上使用依赖）。我们可以去为 KO 来写一个不同的模板引擎，例如，在 KO 源代码里的 `jqueryTmplTemplateEngine.js`，尽管他是为了支持多个版本的 `jquery.tmpl` 而编译。支持一个单独的模板引擎版本相对简单多了。

依赖性

`template` 绑定只能在引用合适的模板引擎情况下才能工作。例如提到的 `jquery.tmpl` 引擎。

Knockout应用开发指南 第五章：创建自定义绑定

创建自定义绑定

你可以创建自己的自定义绑定 – 没有必要非要使用内嵌的绑定（像 `click`, `value` 等）。你可以封装复杂的逻辑或行为，自定义很容易使用和重用的绑定。例如，你可以在 `form` 表单里自定义像 `grid`, `tabset` 等这样的绑定。

重要：以下文档只应用在 **Knockout 1.1.1**和**更高版本**，**Knockout 1.1.0**和以前的版本在**注册 API** 上是不同的。

注册你的绑定

添加子属性到 `ko.bindingHandlers` 来注册你的绑定：

```
ko.bindingHandlers.yourBindingName = {
  init: function(element, valueAccessor, allBindingsAccessor, viewModel) {
    // This will be called when the binding is first applied to an element
    // Set up any initial state, event handlers, etc. here
  },

  update: function(element, valueAccessor, allBindingsAccessor, viewModel) {
    // This will be called once when the binding is first applied to an element,
    // and again whenever the associated observable changes value.
    // Update the DOM element based on the supplied values here.
  }
};
```

... 然后就可以在任何 DOM 元素上使用了：

```
<div data-bind="yourBindingName: someValue"> </div>
```

注：你实际上没必要把 `init` 和 `update` 这两个 `callbacks` 都定义，你可以只定义其中的任意一个。

update 回调

当管理的 `observable` 改变的时候，KO 会调用你的 `update callback` 函数，然后传递以下参数：

- **element** — 使用这个绑定的 DOM 元素
- **valueAccessor** — JavaScript 函数, 通过 `valueAccessor()` 可以得到应用到这个绑定的 model 上的当前属性值。
- **allBindingsAccessor** — JavaScript 函数, 通过 `allBindingsAccessor()` 得到这个元素上所有 model 的属性值。
- **viewModel** — 传递给 `ko.applyBindings` 使用的 view model 参数, 如果是模板内部的话, 那这个参数就是传递给该模板的数据。

例如, 你可能想通过 `visible` 绑定来控制一个元素的可见性, 但是你想让该元素在隐藏或者显示的时候加入动画效果。那你可以自定义自己的绑定来调用 jQuery 的 `slideUp/slideDown` 函数:

```
ko.bindingHandlers.slideVisible = {
  update: function(element, valueAccessor, allBindingsAccessor) {
    // First get the latest data that we're bound to
    var value = valueAccessor(), allBindings = allBindingsAccessor();

    // Next, whether or not the supplied model property is observable, get its
    current value
    var valueUnwrapped = ko.utils.unwrapObservable(value);

    // Grab some more data from another binding property
    var duration = allBindings.slideDuration || 400;

    // 400ms is default duration unless otherwise specified

    // Now manipulate the DOM element

    if (valueUnwrapped == true)
      $(element).slideDown(duration); // Make the element visible
    else
      $(element).slideUp(duration);   // Make the element invisible
  }
};
```

然后你可以像这样使用你的绑定:

```
<div data-bind="slideVisible: giftWrap, slideDuration:600">You have selected the
```

```

option</div>
<label><input type="checkbox" data-bind="checked: giftWrap"/> Gift wrap</label>

<script type="text/javascript">
    var viewModel = {
        giftWrap: ko.observable(true)
    };
    ko.applyBindings(viewModel);
</script>

```

当然，看来可能代码很多，但是一旦你创建了自定义绑定，你就可以在很多地方重用它。

init 回调

Knockout 在 DOM 元素使用自定义绑定的时候会调用你的 `init` 函数。`init` 有两个重要的用途：

- 为 DOM 元素设置初始值
- 注册事件句柄，例如当用户点击或者编辑 DOM 元素的时候，你可以改变相关的 observable 值的状态。

KO 会传递和 `update` 回调函数一样的参数。

继续上面的例子，你可以像让 `slideVisible` 在页面第一次显示的时候设置该元素的状态（但是不使用任何动画效果），而只是让动画在以后改变的时候再执行。你可以这样做：

```

ko.bindingHandlers.slideVisible = {
    init: function(element, valueAccessor) {
        var value = ko.utils.unwrapObservable(valueAccessor());
        // Get the current value of the current property we're bound to
        $(element).toggle(value);
        // jQuery will hide/show the element depending on whether "value" or true
        or false
    },

    update: function(element, valueAccessor, allBindingsAccessor) {
        // Leave as before
    }
};

```

这就是说 `giftWrap` 的初始值声明的是 `false`（例如 `giftWrap: ko.observable(false)`），然后让初始值会让关联的 `DIV` 隐藏，之后用户点击 `checkbox` 的时候会让元素显示出来。

DOM 事件之后更新 observable 值

你已经值得了如何使用 `update` 回调,当 `observable` 值改变的时候,你可以更新相关的 `DOM` 元素。但是其它形式的事件怎么做呢? 比如当用户对某个 `DOM` 元素有某些 `action` 操作的时候,你想更新相关的 `observable` 值。

你可以使用 `init` 回调来注册一个事件句柄,这样可以改变相关的 `observable` 值,例如,

```
ko.bindingHandlers.hasFocus = {

  init: function (element, valueAccessor) {

    $(element).focus(function () {
      var value = valueAccessor();
      value(true);
    });

    $(element).blur(function () {
      var value = valueAccessor();
      value(false);
    });
  },

  update: function (element, valueAccessor) {
    var value = valueAccessor();
    if (ko.utils.unwrapObservable(value))
      element.focus();
    else
      element.blur();
  }
};
```

现在你可以通过 `hasFocus` 绑定来读取或者写入这个 `observable` 值了:

```
<p>Name: <input data-bind="hasFocus: editingName"/></p>
<!-- Showing that we can both read and write the focus state -->
<div data-bind="visible: editingName">You're editing the name</div>
<button data-bind="enable: !editingName(), click:function()
{ editingName(true) }">Edit name</button>
<script type="text/javascript">
  var viewModel = {
    editingName: ko.observable()
  };
  ko.applyBindings(viewModel);
</script>
```

Knockout应用开发指南 第六章：加载或保存JSON 数据

加载或保存 JSON 数据

Knockout 可以实现很复杂的客户端交互，但是几乎所有的 web 应用程序都要和服务端交换数据（至少为了本地存储需要序列化数据），交换数据最方便的就是使用 **JSON 格式** – 大多数的 Ajax 应用程序也是使用这种格式。

加载或保存数据

Knockout 不限制你用任何技术加载和保存数据。你可以使用任何技术和服务器来交互。用的最多的是使用 jQuery 的 Ajax 帮助，例如：getJSON，post 和 ajax。你可以通过这些方法从服务器端获取数据：

```
$.getJSON("/some/url", function (data) {  
    // Now use this data to update your view models,  
    // and Knockout will update your UI automatically  
})
```

... 或者向服务器端发送数据：

```
var data = /* Your data in JSON format - see below */;  
$.post("/some/url", data, function(returnedData) {  
    // This callback is executed if the post was successful  
})
```

或者，如果你不想用 jQuery，你可以用任何其它的方式来读取或保存 JSON 数据。所以，Knockout 需要你做的仅仅是：

对于保存，让你的 view model 数据转换成简单的 JSON 格式，以方便使用上面的技术来保存数据。

对于加载，更新你接收到的数据到你的 view model 上。

转化 View Model 数据到 JSON 格式

由于 view model 都是 JavaScript 对象，所以你需要使用标准的 JSON 序列化工具让转化 view model 为 JSON 格式。例如，可以使用 JSON.serialize()（新版本浏览器才支持的原生方法），或者使用 **json2.js** 类库。不过你的 view model 可能包括 observables，依赖对象

dependent observables 和 observable 数组，有可能不能很好的序列化，你需要自己额外的处理一下数据。

为了使 view model 数据序列化方便（包括序列化 observables 等格式），Knockout 提供了2个帮助函数：

- ko.toJS — 克隆你的 view model 对象，并且替换所有的 observable 对象为当前的值，这样你可以得到一个干净的和 Knockout 无关的数据 copy。
- ko.toJSON — 将 view model 对象转化成 JSON 字符串。原理就是：先调在 view model 上调用 ko.toJS，然后调用浏览器原生的 JSON 序列化器得到结果。注：一些老浏览器版本不支持原生的 JSON 序列化器（例如：IE7和以前的版本），你需要引用 [json2.js](#) 类库。

声明一个 view model:

```
var viewModel = {
    firstName: ko.observable("Bert"),
    lastName: ko.observable("Smith"),
    pets: ko.observableArray(["Cat", "Dog", "Fish"]),
    type: "Customer"
};

viewModel.hasALotOfPets = ko.dependentObservable(function () {
    return this.pets().length > 2
}, viewModel)
```

该 view model 包含 observable 类型的值，依赖类型的值 dependent observable 以及依赖数组 observable array，和普通对象。你可以像如下代码一样使用 ko.toJSON 将此转化成服务器端使用的 JSON 字符串：

```
var jsonData = ko.toJSON(viewModel);

// Result: jsonData is now a string equal to the following value
//
'{"firstName":"Bert","lastName":"Smith","pets":["Cat","Dog","Fish"],"type":"Customer","hasALotOfPets":true}'
```

或者，序列化之前，你想得到 JavaScript 简单对象的话，直接使用像这样一样使用 ko.toJS:

```
var plainJs = ko.toJS(viewModel);
```

```
// Result: plainJS is now a plain JavaScript object in which nothing is observable.
It's just data.
// The object is equivalent to the following:
// {
//   firstName: "Bert",
//   lastName: "Smith",
//   pets: ["Cat", "Dog", "Fish"],
//   type: "Customer",
//   hasALotOfPets: true
// }
```

使用 JSON 更新 View Model 数据

如果你从服务器端获取数据并且更新到 view model 上，最简单的方式是自己实现。例如，

```
// Load and parse the JSON
var someJSON = /* Omitted: fetch it from the server however you want */;
var parsed = JSON.parse(someJSON);

// Update view model properties
viewModel.firstName(parsed.firstName);
viewModel.pets(parsed.pets);
```

很多情况下，最直接的方法就是最简单而且最灵活的方式。当然，如果你更新了 view model 的属性，Knockout 会自动帮你更新相关的 UI 元素的。

不过，很多开发人员还是喜欢使用一种好用而不是每次都写代码的方式来转化数据到 view model 上，尤其是 view model 有很多属性或者嵌套的数据结构的时候，这很有用，因为可以节约很多代码量。knockout.mapping 插件可以帮你做到这一点。

Knockout应用开发指南 第七章：Mapping 插件

Mapping 插件

Knockout 设计成允许你使用任何 JavaScript 对象作为 view model。必须 view model 的一些属性是 observable 的，你可以使用 KO 绑定他们到你的 UI 元素上，当这些 observable 值改变的时候，这些 UI 元素就会自动更新。

绝大多数程序都需要从服务器端获取数据，但是由于服务器不知道 observable 的概念是什么，它只支持简单的 JavaScript 对象（通常是序列化以后的 JSON），mapping 插件可以让你很方便地将简单 JavaScript 对象 map 到带有 observable 值的 view model。你也可以自己写 JavaScript 代码将从服务器获取的数据来构建 view model，mapping 插件只是一种很好的替代而已。

下载

[Version 2.0](#) （最小版本8.6kb）

例子：手工 mapping

显示当前服务器时间和你网站上的当前用户数。你应该使用如下的 view model 来代表你的这些信息：

```
var viewModel = {  
    serverTime: ko.observable(),  
    numUsers: ko.observable()  
}
```

然后绑定 view model 到 HTML 元素上，如下：

```
The time on the server is: <span data-bind='text: serverTime'></span>  
and <span data-bind='text: numUsers'></span> user(s) are connected.
```

由于 view model 属性是 observable 的，在他们变化的时候，KO 会自动更新绑定的 HTML 元素。

接下来，从服务器获取最新的数据。或许每隔5秒你要调用一次 Ajax 请求（例如，使用 jQuery 的 \$.getJSON 或 \$.ajax 函授）：

```
var data = getDataUsingAjax();           // Gets the data from the server
```


然后，服务器返回和下面相似的 JSON 数据：

```
{
  serverTime: '2010-01-07',
  numUsers: 3
}
```

最后，用这些数据更新你的 view model（不使用 mapping 插件），代码如下：

```
// Every time data is received from the server:
viewModel.serverTime(data.serverTime);
viewModel.numUsers(data.numUsers);
```

为了使数据显示在页面上，所有的属性都要像这样写代码。如果你的数据结构很复杂的话（例如，包含子对象或者数组），那就维护起来就相当痛苦。mapping 插件就是来让你让你的 JavaScript 简单对象（或 JSON 结构）转换成 observable 的 view model 的。

例子：使用 ko.mapping

通过 mapping 插件创建 view model，直接使用 ko.mapping.fromJS 函数来创建：

```
var viewModel = ko.mapping.fromJS(data);
```

它会自动将 data 里所有的属性创建成 observable 类型的属性。你可以通过 ko.mapping.fromJS 函数定期从服务器获取数据，然后更新你的 view model：

```
// Every time data is received from the server:
ko.mapping.fromJS(data, viewModel);
```

如何 mapping?

对象的所有属性都被转换成 observable 类型值，如果获取的对象的值改变了，就会更新这个 observable 类型的值。

数组也被转换成了 observable 数组，如果服务器更新改变了数组的个数，mapping 插件也会添加或者删除相应的 item 项，也会尽量保持和原生 JavaScript 数组相同的 order 顺序。

Unmapping

如果你想让 map 过的对象转换成原来的 JavaScript 对象，使用如下方式：

```
var unmapped = ko.mapping.toJS(viewModel);
```

会创建一个 **unmapped** 对象, 只包含你之前 **map** 过的对象属性, 换句话说, 你在 **view model** 上手工添加的属性或者函数都会被忽略的, 唯一例外的是 **_destroy** 属性是可以 **unmapped** 回来的, 因为你从 **ko.observableArray** 里 **destroy** 一个 **item** 项的时候会生成这个属性。请参考“高级用户”小节如何配置使用。

与 JSON 字符串一起使用

如果你的 Ajax 调用返回的是 JSON 字符串（而不是反序列化后的 JavaScript 对象），你可以使用 **ko.mapping.fromJSON** 函数来创建或者更新你的 **view model**。用 **ko.mapping.toJSON** 实现 **unmap**。

使用 **.from/toJSON** 函数处理 JSON 字符串和使用 **.from/toJS** 函数处理 JS 对象是等价的。

高级用法

有时候, 在使用 **ko.mapping.fromJS** 的时候, 可能有必要去使用 **mapping** 的高级用法来定义 **mapping** 的详细过程, 以后定义了, 以后再调用的时候就不必再定义了。这里有一些情形, 你可能需要使用这些 **option**。

用法1: 使用 **keys** 来使对象 **unique** 化

你有一个 JavaScript 对象, 如下:

```
var data = {
  name: 'Scot',
  children: [
    { id: 1, name: 'Alicw' }
  ]
}
```

使用 **map** 插件, 你可以将它 **map** 到 **view model** 上 (没有任何问题):

```
var viewModel = ko.mapping.fromJS(data);
```

现在, 数据被更新成如下这样:

```
var data = {
  name: 'Scott',
  children: [
```

```
    { id: 1, name: 'Alice' }  
  ]  
}
```

这里发生了两件事：**name** 从 **Scot** 变成了 **Scott**，**children[0].name** 从 **Alicw** 变成了 **Alice**。你可以用如下代码更新 **view model**：

```
ko.mapping.fromJS(data, viewModel);
```

于是，**name** 像我们期望的一样更新了，但是在 **children** 数组里，子项 **Alicw** 被删除而新项 **Alice** 被添加到数组里。这不是我们所期望的，我们期望的是只是把 **name** 从 **Alicw** 更新成 **Alice**，不是替换整个 **item** 项。发生的原因是，默认情况下 **mapping plugin** 插件只是简单地比较数组里的两个对象是否相等。因为 JavaScript 里 **{ id: 1, name: 'Alicw' }** 和 **{ id: 1, name: 'Alice' }** 是不相等的，所以它认为喜欢将新项替换掉老项。

解决这个问题，你需要声明一个 **key** 让 **mapping** 插件使用，用来判断一个对象是新对象还是旧对象。代码如下：

```
var mapping = {  
  'children': {  
    key: function (data) {  
      return ko.utils.unwrapObservable(data.id);  
    }  
  }  
}  
  
var viewModel = ko.mapping.fromJS(data, mapping);
```

这样，每次 **map** 的时候，**mapping** 插件都会检查数组项的 **id** 属性来判断这个数组项是需要合并的还是全新 **replace** 的。

用法2：用 **create** 自定义对象的构造器

如果你想自己控制 **mapping**，你也可以使用 **create** 回调。使用回调可以让你自己控制 **mapping**。

举例，你有一个像这样的 JavaScript 对象：

```
var data = {  
  name: 'Graham',  
  children: [  
    { id: 1, name: 'Lisa' }  
  ]  
}
```

如果你想自己 `map children` 数组，你可以这样声明：

```
var mapping = {
  'children': {
    create: function (options) {
      return new myChildModel(options.data);
    }
  }
}

var viewModel = ko.mapping.fromJS(data, mapping);
```

支持 `create` 回调的 `options` 参数是一个 JavaScript 对象，包含如下：

- **data:** JavaScript 对象，包含 `child` 用到的数据
- **parent:** `child` 对象所属的父对象或者数组

当然，在内部的 `create` 回调里，你也可以再次调用 `ko.mapping.fromJS`。一个例子就是：如果你想让初始的 JavaScript 对象带有额外的依赖属性 `dependent observables`：

```
var myChildModel = function (data) {
  ko.mapping.fromJS(data, {}, this);

  this.nameLength = ko.dependentObservable(function () {
    return this.name().length;
  }, this);
}
```

用法3：用 `update` 自定义对象的 `updating`

你也可以使用 `update` 回调来自定义一个对象如何更新。它接受一个需要替代的对象以及和 `create` 回调一样的 `options` 参数，你应该 `return` 更新后的值。

`update` 回调使用的 `options` 参数是一个 JavaScript 对象，包含如下内容：

- **data:** JavaScript 对象，包含 `child` 用到的数据
- **parent:** `child` 对象所属的父对象或者数组
- **observable:** 如果属性是 `observable` 的，这将会写入到实际的 `observable` 里

例子，在数据显示之前，在新数据后面附加额外的字符串：

```
var data = {
  name: 'Graham',
```

```

}

var mapping = {
  'name': {
    update: function(options) {
      return options.data + 'foo!';
    }
  }
}

var viewModel = ko.mapping.fromJS(data, mapping);
alert(viewModel.name());

```

alert 的结果是：Grahamfoo!。

用法4：使用 ignore 忽略不需要 map 的属性

如果在 map 的时候，你想忽略一些属性，你可以使用 ignore 累声明需要忽略的属性名称集合：

```

var mapping = {
  'ignore': ["propertyToIgnore", "alsoIgnoreThis"]
}

var viewModel = ko.mapping.fromJS(data, mapping);

```

你声明的忽略数组被编译到默认的 ignore 数组里。你可以像下面代码一样来维护它：

```

var oldOptions = ko.mapping.defaultOptions().ignore;
ko.mapping.defaultOptions().ignore = ["alwaysIgnoreThis"];

```

用法5：使用 include 声明需要 map 的属性

默认情况下，当 map 你的 view model 回到 JS 对象是时候，只 map 原始 view model 里拥有的属性（除了例外的 _destroy 属性），不过，你可以使用 include 参数来定制：

```

var mapping = {
  'include': ["propertyToInclude", "alsoIncludeThis"]
}

var viewModel = ko.mapping.fromJS(data, mapping);

```

你声明的 include 数组被编译到默认的 include 数组里，默认只有 _destroy。你可以像这样

来维护：

```
var oldOptions = ko.mapping.defaultOptions().include;
ko.mapping.defaultOptions().include = ["alwaysIncludeThis"];
```

用法6：使用 **copy** 来复制属性

默认情况下，`map` 的时候是把所有的值都转换成 `observable` 的，如果你只是想 `copy` 属性值而不是替换成 `observable` 的，你可以将属性名称添加到 `copy` 数组：

```
var mapping = {
  'copy': ["propertyToCopy"]
}

var viewModel = ko.mapping.fromJS(data, mapping);
```

你声明的 `copy` 数组被编译到默认的 `copy` 数组里，默认值是空。你可以像这样来维护：

```
var oldOptions = ko.mapping.defaultOptions().copy;
ko.mapping.defaultOptions().copy = ["alwaysCopyThis"];
```

用法7：Specifying the update target

在上面的例子，如果你想再一个 `class` 内 `map`，你可以使用第三个参数作为操作的目标，例如：

```
ko.mapping.fromJS(data, {}, someObject); // overwrites properties on someObject
```

所以，如果你想 `map` 一个 `JavaScript` 对象到 `this` 上，你可以这样声明：

```
ko.mapping.fromJS(data, {}, this);
```

从多数据源 **map**

你可以通过多次使用 `ko.mapping.fromJS` 来将多个 `JS` 对象的数据源 `map` 到一起，例如：

```
var viewModel = ko.mapping.fromJS(alice, aliceMappingOptions);
ko.mapping.fromJS(bob, bobMappingOptions, viewModel);
```

你声明的 `mapping` 选项 `option` 在每次调用的时候都会合并。

Map 以后的 observable 数组

map 插件 map 以后生产的 observable 数组，带有几个额外的函数来处理带有 keys 的 mapping:

- mappedRemove
- mappedRemoveAll
- mappedDestroy
- mappedDestroyAll
- mappedIndexOf

它们是和 ko.observableArray 里的函数等价的，不同是他们通过 key 来处理对象。例如:

```
var obj = [
  { id: 1 },
  { id: 2 }
]

var result = ko.mapping.fromJS(obj, {
  key: function (item) {
    return ko.utils.unwrapObservable(item.id);
  }
});

result.mappedRemove({ id: 2 });
```

map 过的 observable 数组，除了上面的函数还支持一个 mappedCreate 函数:

```
var newItem = result.mappedCreate({ id: 3 });
```

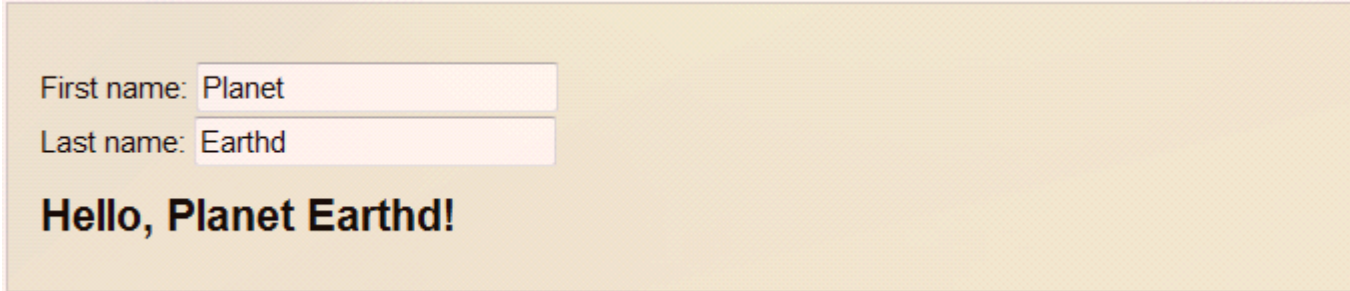
首先会检查 key (id=3) 在数组里是否存在 (如果存在则抛出异常)，然后，如果有 create 和 update 回调的话会调用他们，最后创建一个新对象，并将新对象添加到数组然后返回该新对象。

Knockout应用开发指南 第八章：简单应用举例（1）

本章展示的4个例子主要是利用了 Knockout 的基本语法特性，让大家感受到使用 Kncokout 的快感。

1 Hello world

这个例子里，2个输入框都被绑定到 data model 上的 observable 变量上。“full name”显示的是一个 dependent observable，它的值是前面2个输入框的值合并一起的结果。



First name: Planet

Last name: Earthd

Hello, Planet Earthd!

无论哪个输入框更新，都会看到“full name”显示结果都会自动更新。查看 HTML 源代码可以看到我们不需要声明 onchange 事件。Knockout 知道什么时候该更新 UI。

代码: View

```
<p>First name: <input data-bind="value: firstName"/></p>
<p>Last name: <input data-bind="value: lastName"/></p>
<h2>Hello, <span data-bind="text: fullName"> </span>!</h2>
```

代码: View model

```
// 这里是声明的 view model

var viewModel = {
  firstName: ko.observable("Planet"),
  lastName: ko.observable("Earth")
};

viewModel.fullName = ko.dependentObservable(function () {
  // Knockout tracks dependencies automatically.
  //It knows that fullName depends on firstName and lastName,
  //because these get called when evaluating fullName.
  return viewModel.firstName() + " " + viewModel.lastName();
});
```



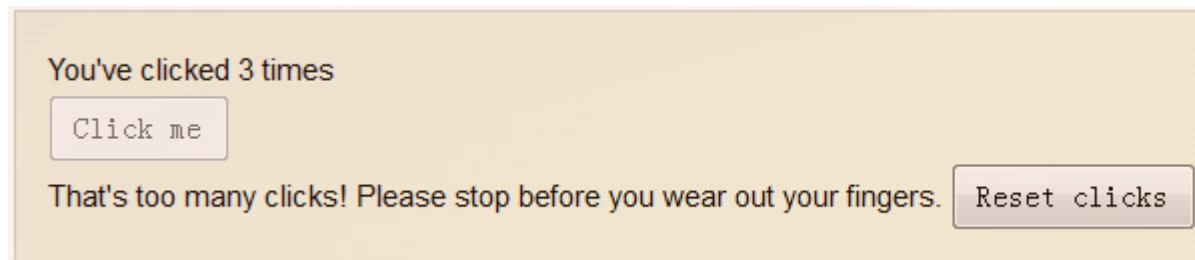
```
});  
  
ko.applyBindings(viewModel); // This makes Knockout get to work
```

2 Click counter

这个例子展示创建一个 view model 并且绑定各种绑定到 HTML 元素标记上,以便展示和修改 view model 的状态。

Knockout 根据依赖项。在内部, `hasClickedTooManyTimes` 在 `numberOfClicks` 上有个订阅,以便当 `numberOfClicks` 改变的时候,强制 `hasClickedTooManyTimes` 重新执行。相似的, UI 界面上多个地方引用 `hasClickedTooManyTimes`, 所以当 `hasClickedTooManyTimes` 改变的时候,也讲导致 UI 界面更新。

不需要手工声明或者订阅这些 `subscription` 订阅,他们由 KO 框架自己创建和销毁。参考如下代码实现:



代码: View

```
<div>You've clicked <span data-bind="text: numberOfClicks">&nbsp;</span>  
times</div>  
  
<button data-bind="click: registerClick,  
enable: !hasClickedTooManyTimes()">Click me</button>  
  
<div data-bind="visible: hasClickedTooManyTimes">  
    That's too many clicks! Please stop before you wear out your fingers.  
    <button data-bind="click: function() { numberOfClicks(0) }">Reset  
clicks</button>  
</div>
```

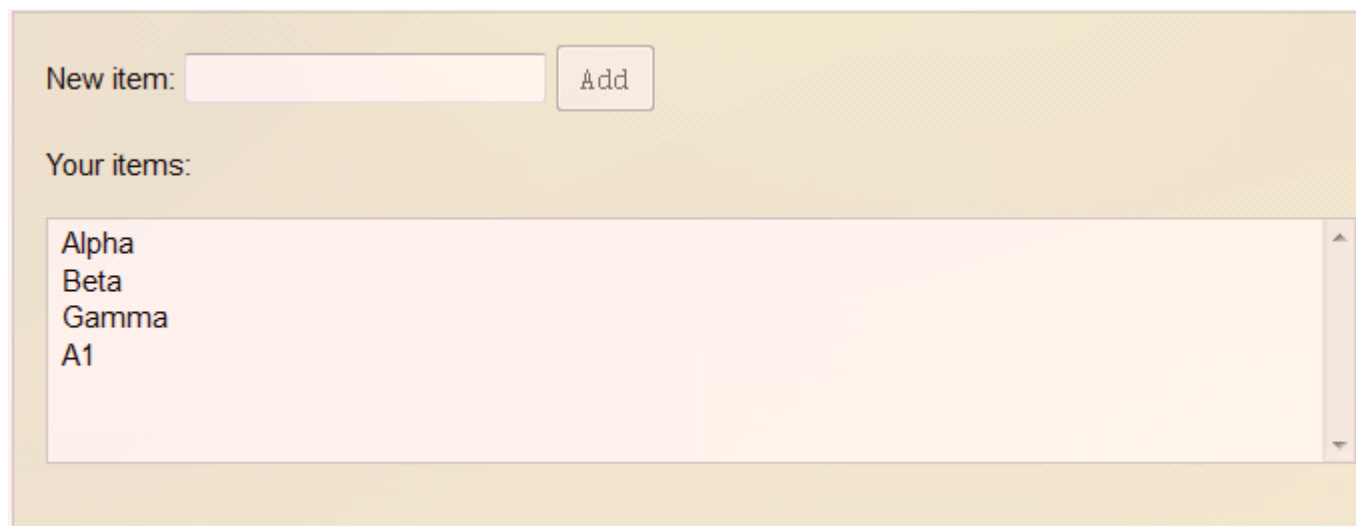
代码: View model

```
var clickCounterViewModel = function () {  
    this.numberOfClicks = ko.observable(0);  
  
    this.registerClick = function () {  
        this.numberOfClicks(this.numberOfClicks() + 1);  
    }  
  
    this.hasClickedTooManyTimes = ko.dependentObservable(function () {  
        return this.numberOfClicks() >= 3;  
    }, this);  
};  
  
ko.applyBindings(new clickCounterViewModel());
```

3 Simple list

这个例子展示的是绑定到数组上。

注意到，只有当你在输入框里输入一些值的时候，Add 按钮才可用。参考下面的 HTML 代码是如何使用 `enable` 绑定。



New item:

Your items:

- Alpha
- Beta
- Gamma
- A1

代码: View

```
<form data-bind="submit: addItem">
```

```

New item:
<input data-bind='value: itemToAdd, valueUpdate: "afterkeydown"' />
<button type="submit" data-bind="enable: itemToAdd().length >
0">Add</button>
<p>Your items:</p>
<select multiple="multiple" width="50" data-bind="options: items"> </select>
</form>

```

代码: View model

```

var viewModel = {};
viewModel.items = ko.observableArray(["Alpha", "Beta", "Gamma"]);
viewModel.itemToAdd = ko.observable("");
viewModel.addItem = function () {
    if (viewModel.itemToAdd() != "") {
        viewModel.items.push(viewModel.itemToAdd());
        // Adds the item. Writing to the "items" observableArray causes any
        associated UI to update.

        viewModel.itemToAdd("");
        // Clears the text box, because it's bound to the "itemToAdd" observable
    }
}
ko.applyBindings(viewModel);

```

4 Better list

这个例子是在上个例子的基础上添加 **remove item** 功能（**multi-selection**）和排序功能。“**remove**”和“**sort**”按钮在不能用的时候会变成 **disabled** 状态（例如，没有足够的 **item** 来排序）。

参考 **HTML** 代码是如何实现这些功能的，另外这个例子也展示了如何使用匿名函数来实现排序。

Add item:

Add

Your values:

Fries

Eggs Benedict

Ham

Cheese

Remove

Sort

代码: View

```
<form data-bind="submit:addItem">
  Add item: <input type="text" data-bind='value:itemToAdd, valueUpdate:
"afterkeydown"' />
  <button type="submit" data-bind="enable: itemToAdd().length >
0">Add</button>
</form>

<p>Your values:</p>
<select multiple="multiple" height="5" data-bind="options:allItems,
selectedOptions:selectedItems"> </select>

<div>
  <button data-bind="click: removeSelected, enable: selectedItems().length >
0">Remove</button>
  <button data-bind="click: function() { allItems.sort() }, enable:
allItems().length > 1">Sort</button>
</div>
```

代码: View model

```
// In this example, betterListModel is a class, and the view model is an instance
of it.

// See simpleList.html for an example of how to construct a view model without
defining a class for it. Either technique works fine.
```

```
var betterListModel = function () {
    this.itemToAdd = new ko.observable("");
    this.allItems = new ko.observableArray(["Fries", "Eggs Benedict", "Ham",
"Cheese"]);

    // Initial items

    this.selectedItems = new ko.observableArray(["Ham"]);

    // Initial selection

    this.addItem = function () {
        if ((this.itemToAdd() != "") && (this.allItems.indexOf(this.itemToAdd())
< 0))
            // Prevent blanks and duplicates
            this.allItems.push(this.itemToAdd());
            this.itemToAdd(""); // Clear the text box
    }

    this.removeSelected = function () {
        this.allItems.removeAll(this.selectedItems());
        this.selectedItems([]); // Clear selection
    }
};

ko.applyBindings(new betterListModel());
```

Knockout应用开发指南 第八章：简单应用举例（2）

5 Control types

这个例子，对 **view model** 没有什么特殊的展示，只是展示如何绑定到各种元素上（例如，**select**，**radio button** 等）。

HTML controls		What's in the model?	
Text value (updates on change):	<input type="text" value="Hello"/>	Text value:	Hello
Text value (updates on keystroke):	<input type="text" value="Hello"/>	Password:	mypass
Text value (multi-line):	<input type="text" value="Hello"/>	Bool value:	True
		Selected option:	Gamma
Password:	<input type="password" value="•••••"/>	Multi-selected options:	Alpha
Checkbox:	<input checked="" type="checkbox"/>	Radio button selection:	Beta
Drop-down list:	<input type="text" value="Gamma"/>		
Multi-select drop-down list:	<div>Alpha Beta Gamma</div>		
Radio buttons:	<input type="radio"/> Alpha <input checked="" type="radio"/> Beta <input type="radio"/> Gamma		

代码: View

[View Code](#)

代码: View model

```
var viewModel = {
    stringValue: ko.observable("Hello"),
    passwordValue: ko.observable("mypass"),
    booleanValue: ko.observable(true),
    optionValues: ["Alpha", "Beta", "Gamma"],
    selectedOptionValue: ko.observable("Gamma"),
    multipleSelectedOptionValues: ko.observable(["Alpha"]),
    radioSelectedOptionValue: ko.observable("Beta")
};

ko.applyBindings(viewModel);
```

6 Templating

这个例子展示的 **render** 模板，以及在模板内部如何使用 **data binding** 属性的。

Template 很容易嵌套，当任何依赖数据改变的时候，**Knockout** 会自动重新 **render** 模板。参考演示（启用‘**Show render times**’），**Knockout** 知道只需要重新 **render** 改变的那些数据所绑定的最近的模板。

People

- Annabelle has 3 children: [Add child](#) (person rendered at 16)
 - Arnie (child rendered at 16)
 - Anders (child rendered at 16)
 - Apple (child rendered at 16)
- Bertie has 4 children: [Add child](#) (person rendered at 16)
 - Boutros-Boutros (child rendered at 16)
 - Brianna (child rendered at 16)
 - Barbie (child rendered at 16)
 - Bee-bop (child rendered at 16)
- Charles has 2 children: [Add child](#) (person rendered at 16)
 - Cayenne (child rendered at 16)
 - Cleopatra (child rendered at 16)

☒ Show render times

代码: View

```
<div data-bind='template: "peopleTemplate"'>
</div>
<label>
  <input type="checkbox" data-bind="checked: showRenderTimes"/>
  Show render times</label>
<script type="text/html" id="peopleTemplate">
<h2>People</h2>
<ul>
  {{each people}}
  <li>
    <div>
      ${ name } has <span data-bind="text: children().length">&nbsp;</span>
      children:
      <a href="#" data-bind="click: addChild ">Add child</a>
      <span class="renderTime" data-bind="visible: showRenderTimes">
        (person rendered at <span data-bind="text: new
Date().getSeconds()"></span>
      </span>
    </div>
  </li>
</ul>
<div data-bind='template: { name: "childrenTemplate", data: children }'></div>
```

```

    </li>
  {{/each}}
</ul>
</script>
<script type="text/html" id="childrenTemplate">
  <ul>
    {{each $data}}
      <li>
        ${ this }
        <span class="renderTime" data-bind="visible:
viewModel.showRenderTimes">
          (child rendered at <span data-bind="text: new
Date().getSeconds()"></span>)
        </span>
      </li>
    {{/each}}
  </ul>
</script>

```

代码: View model

```

// Define a "person" class that tracks its own name and children, and has a method
to add a new child

var person = function (name, children) {
  this.name = name;
  this.children = ko.observableArray(children);

  this.addChild = function () {
    this.children.push("New child");
  } .bind(this);
}

// The view model is an abstract description of the state of the UI, but without
any knowledge of the UI technology (HTML)

var viewModel = {
  people: [
    new person("Annabelle", ["Arnie", "Anders", "Apple"]),
    new person("Bertie", ["Boutros-Boutros", "Brianna", "Barbie", "Bee-bop"]),
    new person("Charles", ["Cayenne", "Cleopatra"])
  ],
  showRenderTimes: ko.observable(false)
};

```

```
ko.applyBindings(viewModel);
```

7 Paged grid

`data-bind="..."` 绑定（像 `text`, `visible`, 和 `click` 不是固定死的） - 你可以很容易自定义自己的绑定。如果你的自定义绑定仅仅是添加事件或者更新 **DOM** 元素的属性，那几行就可以实现。不过，你依然可以自定义可以重用的绑定（或插件），就行本例的 `simpleGrid` 绑定。

如果一个插件需要自己标准的 **view model**（例如本例的 `ko.simpleGrid.viewModel`），它提供既提供了该如何配置插件实例（分页大小，列声明）工作，也提供了 **view model** 上的属性是否是 **observable** 的（例如 `currentpage` 索引）。也可以扩展代码让这些属性很容易地改变，并且让 **UI** 自动更新。例如，“Jump to first page”按钮的工作原理。

查看 **HTML** 源代码可以看到非常容易使用这个 `simple grid` 插件。`simpleGrid` 源码地址是：
<http://knockoutjs.com/examples/resources/knockout.simpleGrid.js>

Item Name	Sales Count	Price
Well-Travelled Kitten	352	\$75.95
Speedy Coyote	89	\$190.00
Furious Lizard	152	\$25.00
Indifferent Monkey	1	\$99.95

Page: 1 2

代码: View

```
<div data-bind="simpleGrid: gridViewModel"> </div>

<button data-bind='click: function() { items.push({ name: "New item", sales: 0,
price: 100 }) }'>
  Add item
```

```

</button>

<button data-bind="click: sortByName">
    Sort by name
</button>

<button data-bind="click: function() { gridViewModel.currentPageIndex(0) }">
    Jump to first page
</button>

```

代码: View model

```

var myModel = {
    items: ko.observableArray([
        { name: "Well-Travelled Kitten", sales: 352, price: 75.95 },
        { name: "Speedy Coyote", sales: 89, price: 190.00 },
        { name: "Furious Lizard", sales: 152, price: 25.00 },
        { name: "Indifferent Monkey", sales: 1, price: 99.95 },
        { name: "Brooding Dragon", sales: 0, price: 6350 },
        { name: "Ingenious Tadpole", sales: 39450, price: 0.35 },
        { name: "Optimistic Snail", sales: 420, price: 1.50 }
    ]),

    sortByName: function () {
        this.items.sort(function (a, b) {
            return a.name < b.name ? -1 : 1;
        });
    }
};

myModel.gridViewModel = new ko.simpleGrid.viewModel({
    data: myModel.items,
    columns: [
        { headerText: "Item Name", rowText: "name" },
        { headerText: "Sales Count", rowText: "sales" },
        { headerText: "Price", rowText: function (item) { return "$" +
item.price.toFixed(2) } }
    ],
    pageSize: 4
});

ko.applyBindings(myModel);

```

8 Animated transitions

该例子展示了2种方式实现动画过渡效果：

当使用 `template/foreach` 绑定的时候，你可以使用 `afterAdd` 和 `beforeRemove` 回调函数，他们可以让你写代码真实操作添加和删除元素，这样你就可以使用像 `jQuery` 的 `slideUp/slideDown()` 这样的动画效果。在 `planet types` 之间切换或添加新的 `planet` 可以看到效果。

通过 `observable` 类型的值，我们不难定义自己的 `Knockout` 绑定，查看 `HTML` 源代码可以看到一个自定义绑定 `fadeVisible`，不管什么时候它改变了，`jQuery` 就会在相关的元素上执行 `fadeIn/fadeOut` 动画效果。点击“advanced options” checkbox 可以看到效果。



Planets

☒ Display advanced options

Show: ☒ All ☐ Rocky planets ☐ Gas giants

Mercury

Venus

Earth

Mars

Jupiter

Saturn

Uranus

Neptune

Pluto

Add rocky planet

Add gas giant

代码: View

```
<h2>Planets</h2>
<p>
  <label>
    <input type="checkbox" data-bind="checked: displayAdvancedOptions"/>
    Display advanced options
  </label>
</p>
<p data-bind="fadeVisible: displayAdvancedOptions">
  Show:
  <label><input type="radio" value="all" data-bind="checked:
typeToShow"/>All</label>
  <label><input type="radio" value="rock" data-bind="checked:
typeToShow"/>Rocky planets</label>
  <label><input type="radio" value="gasgiant" data-bind="checked:
typeToShow"/>Gas giants</label>
```

```

</p>
<div data-bind='template: { name: "planetsTemplate",
    foreach: planetsToShow,
    beforeRemove: function(elem)
{ $(elem).slideUp(function() { $(elem).remove(); }) },
    afterAdd: function(elem)
{ $(elem).hide().slideDown() } }'>
</div>
<script type="text/html" id="planetsTemplate">
    <div class="planet ${ type }">${ name }</div>
</script>
<p data-bind="fadeVisible: displayAdvancedOptions">
    <button data-bind='click: function() { addPlanet("rock") }'>
        Add rocky planet</button>
    <button data-bind='click: function() { addPlanet("gasgiant") }'>
        Add gas giant</button>
</p>

```

代码: View model

```

var viewModel = {
    planets: ko.observableArray([
        { name: "Mercury", type: "rock" },
        { name: "Venus", type: "rock" },
        { name: "Earth", type: "rock" },
        { name: "Mars", type: "rock" },
        { name: "Jupiter", type: "gasgiant" },
        { name: "Saturn", type: "gasgiant" },
        { name: "Uranus", type: "gasgiant" },
        { name: "Neptune", type: "gasgiant" },
        { name: "Pluto", type: "rock" }
    ]),

    typeToShow: ko.observable("all"),
    displayAdvancedOptions: ko.observable(false),

    addPlanet: function (type) { this.planets.push({ name: "New planet", type:
type }); }
};

viewModel.planetsToShow = ko.dependentObservable(function () {
    // Represents a filtered list of planets
    // i.e., only those matching the "typeToShow" condition

    var desiredType = this.typeToShow();

```

```

    if (desiredType == "all")
        return this.planets();

    return ko.utils.arrayFilter(this.planets(), function (planet) {
        return planet.type == desiredType;
    });
} .bind(viewModel));

// Here's a custom Knockout binding that makes elements shown/hidden via jQuery's
fadeIn()/fadeOut() methods
// Could be stored in a separate utility library

ko.bindingHandlers.fadeVisible = {
    init: function (element, valueAccessor) {
        // Initially set the element to be instantly visible/hidden depending on
the value
        var value = valueAccessor();

        $(element).toggle(ko.utils.unwrapObservable(value));
        // Use "unwrapObservable" so we can handle values that may or may not be
observable
    },

    update: function (element, valueAccessor) {
        // Whenever the value subsequently changes, slowly fade the element in or
out
        var value = valueAccessor();
        ko.utils.unwrapObservable(value) ? $(element).fadeIn() :
$(element).fadeOut();
    }
};

ko.applyBindings(viewModel);

```


Knockout应用开发指南 第九章：高级应用举例

1 Contacts editor

这个例子和微软为演示 jQuery Data Linking Proposal 例子提供的例子一样的提供的，我们可以看看 Knockout 实现是难了还是容易了。

代码量的多少不重要（尽快 Knockout 的实现很简洁），重要的看起来是否容易理解且可读。查看 HTML 源代码，看看如何实现的 view model 以及绑定的。

Contacts

First name	Last name	Phone numbers		
Danny	LaRusso	Mobile	(555) 121-2121	Delete
		Home	(555) 123-4567	Delete
Add number				
Sensei	Miyagi	Mobile	(555) 444-2222	Delete
		Home	(555) 999-1212	Delete
Add number				
				Delete
Add number				

Add a contact

Save to JSON

```
[
  {
    "firstName": "Danny",
    "lastName": "LaRusso",
    "phones": [
      {
```

代码: View

View Code 



```
<h2>Contacts</h2>
<div id="contactsList" data-bind='template: "contactsListTemplate"'>
</div>
<script type="text/html" id="contactsListTemplate">
  <table class='contactsEditor'>
    <tr>
      <th>First name</th>
      <th>Last name</th>
      <th>Phone numbers</th>
    </tr>

    {{each(i, contact) contacts()}}
      <tr>
        <td>
          <input data-bind="value: firstName"/>
          <div><a href="#" data-bind="click: function()
{ viewModel.removeContact(contact) }">Delete</a></div>
        </td>
        <td><input data-bind="value: lastName"/></td>
        <td>
          <table>
            {{each(i, phone) phones}}
              <tr>
                <td><input data-bind="value: type"/></td>
                <td><input data-bind="value: number"/></td>
                <td><a href="#" data-bind="click: function()
{ viewModel.removePhone(contact, phone) }">Delete</a></td>
              </tr>
            {{/each}}
          </table>
          <a href="#" data-bind="click: function() { viewModel.addPhone(contact) }">Add
number</a>
        </td>
      </tr>
    {{/each}}
  </table>
</script>
<p>
  <button data-bind="click: addContact">
    Add a contact</button>
  <button data-bind="click: save, enable: contacts().length > 0">
    Save to JSON</button>
</p>
```

```
</p>
<textarea data-bind="value: lastSavedJson" rows="5" cols="60"
disabled="disabled"> </textarea>
```



代码: View model

 View Code 

```
var viewModel = {
  contacts: new ko.observableArray([
    { firstName: "Danny", lastName: "LaRusso", phones: [
      { type: "Mobile", number: "(555) 121-2121" },
      { type: "Home", number: "(555) 123-4567" }]
    },
    { firstName: "Sensei", lastName: "Miyagi", phones: [
      { type: "Mobile", number: "(555) 444-2222" },
      { type: "Home", number: "(555) 999-1212" }]
    }
  ]),

  addContact: function () {
    viewModel.contacts.push({ firstName: "", lastName: "", phones: [] });
  },

  removeContact: function (contact) {
    viewModel.contacts.remove(contact);
  },

  addPhone: function (contact) {
    contact.phones.push({ type: "", number: "" });
    viewModel.contacts.valueHasMutated();
  },

  removePhone: function (contact, phone) {
    ko.utils.arrayRemoveItem(contact.phones, phone);
    viewModel.contacts.valueHasMutated();
  },

  save: function () {
```

```

        viewModel.lastSavedJson(JSON.stringify(viewModel.contacts(), null, 2));
    },

    lastSavedJson: new ko.observable("")
};

ko.applyBindings(viewModel);

```

2 Editable grid

该例是使用“foreach”绑定为数组里的每一项来 render 到 template 上。好处（相对于模板内部使用 for 循环）是当你添加或者删除 item 项的时候，Knockout 不需要重新 render – 只需要 render 新的 item 项。就是说 UI 上其它控件的状态（比如验证状态）不会丢失。

如何一步一步构建这个例子并集成 ASP.NET MVC，请[参阅此贴](#)。

You have asked for 4 gift(s)

Gift name	Price	
<input type="text" value="Tall Hat"/>	<input type="text" value="39.95"/>	Delete
<input type="text" value="Long Cloak"/>	<input type="text" value="120.00"/>	Delete
<input type="text" value=""/>	<input type="text" value="s"/>	Delete
This field is required.		Please enter a valid number.
<input type="text" value=""/>	<input type="text" value=""/>	Delete

代码: View

View Code

```

<form action="/someServerSideHandler">
<p>

```

```

    You have asked for <span data-bind="text: gifts().length">&nbsp;</span>
gift(s)</p>
<table data-bind="visible: gifts().length > 0">
  <thead>
    <tr>
      <th>Gift name</th>
      <th>Price</th>
      <th></th>
    </tr>
  </thead>
  <tbody data-bind='template: { name: "giftRowTemplate", foreach: gifts }'>
  </tbody>
</table>
<button data-bind="click: addGift">
  Add Gift</button>
<button data-bind="enable: gifts().length > 0" type="submit">
  Submit</button>
</form>
<script type="text/html" id="giftRowTemplate">
  <tr>
    <td><input class="required" data-bind="value: name, uniqueName:
true"/></td>
    <td><input class="required number" data-bind="value: price, uniqueName:
true"/></td>
    <td><a href="#" data-bind="click: function()
{ viewModel.removeGift($data) }">Delete</a></td>
  </tr>
</script>

```



代码: View model

 View Code

```

var viewModel = {
  gifts: ko.observableArray([
    { name: "Tall Hat", price: "39.95" },
    { name: "Long Cloak", price: "120.00" }
  ]),

  addGift: function () {

```

```

        this.gifts.push({ name: "", price: "" });
    },

    removeGift: function (gift) {
        this.gifts.remove(gift);
    },

    save: function (form) {
        alert("Could now transmit to server: " +
ko.utils.stringifyJson(this.gifts));
        // To transmit to server, write this: ko.utils.postJson($("#form")[0],
this.gifts);
    }
};

ko.applyBindings(viewModel);

$("#form").validate({ submitHandler: function () { viewModel.save() } });

```

3 Shopping cart screen

这个例子展示的是依赖监控属性（**dependent observable**）怎么样链在一起。每个 **cart** 对象都有一个 **dependentObservable** 对象去计算自己的 **subtotal**，这些又被一个进一步的 **dependentObservable** 对象依赖计算总的价格。当改变数据的时候，整个链上的依赖监控属性都会改变，所有相关的 UI 元素也会被更新。



这个例子也展示了如何创建联动的下拉菜单。

Category	Product	Price	Quantity	Subtotal	
Classic Cars	1993 Mazda R	\$83.51	1	\$83.51	Remove
Planes	Select...				Remove

Total value: \$83.51

Add product Submit order

代码: View

 View Code 

```
<div id="cartEditor">
  <table width="100%">
    <thead>
      <tr>
        <th width="25%">Category</th>
        <th width="25%">Product</th>
        <th width="15%" class='price'>Price</th>
        <th width="10%" class='quantity'>Quantity</th>
        <th width="15%" class='price'>Subtotal</th>
        <th width="10%"></th>
      </tr>
    </thead>
    <tbody data-bind='template: {name: "cartRowTemplate", foreach: lines}'>
    </tbody>
  </table>
  <p class="grandTotal">
    Total value: <span data-bind="text:
formatCurrency(grandTotal())"></span>
  </p>
  <button data-bind="click: addLine">
    Add product</button>
  <button data-bind="click: save">
    Submit order</button>
</div>
<script type="text/html" id="cartRowTemplate">
```

```

<tr>
  <td><select data-bind='options: sampleProductCategories, optionsText:
"name", optionsCaption: "Select...", value: category'></select></td>
  <td><select data-bind='visible: category, options: category() ?
category().products : null, optionsText: "name", optionsCaption: "Select...",
value: product'></select></td>
  <td class='price'><span data-bind='text: product() ?
formatCurrency(product().price) : ""'></span></td>
  <td class='quantity'><input data-bind='visible: product, value: quantity,
valueUpdate: "afterkeydown"'></td>
  <td class='price'><span data-bind='visible: product, text:
formatCurrency(subtotal())'></span></td>
  <td><a href="#" data-bind='click: function()
{ cartViewModel.removeLine($data) }'>Remove</a></td>
</tr>
</script>

```



代码: View model

View Code 

```

function formatCurrency(value) { return "$" + value.toFixed(2); }

var cartLine = function () {
  this.category = ko.observable();
  this.product = ko.observable();
  this.quantity = ko.observable(1);
  this.subtotal = ko.dependentObservable(function () {
    return this.product() ? this.product().price * parseInt("0" +
this.quantity(), 10) : 0;
  }).bind(this);

  // Whenever the category changes, reset the product selection
  this.category.subscribe(function ()
{ this.product(undefined); }).bind(this));
};

var cart = function () {
  // Stores an array of lines, and from these, can work out the grandTotal
  this.lines = ko.observableArray([new cartLine()]); // Put one line in by

```



```

default
    this.grandTotal = ko.dependentObservable(function () {
        var total = 0;
        for (var i = 0; i < this.lines().length; i++)
            total += this.lines()[i].subtotal();
        return total;
    }).bind(this);

    // Operations
    this.addLine = function () { this.lines.push(new cartLine()) };
    this.removeLine = function (line) { this.lines.remove(line) };

    this.save = function () {
        var dataToSave = $.map(this.lines(), function (line) {
            return line.product() ? { productName: line.product().name, quantity:
line.quantity() } : undefined
        });

        alert("Could now send this to server: " + JSON.stringify(dataToSave));
    };
};

var cartViewModel = new cart();

ko.applyBindings(cartViewModel, document.getElementById("cartEditor"));

```

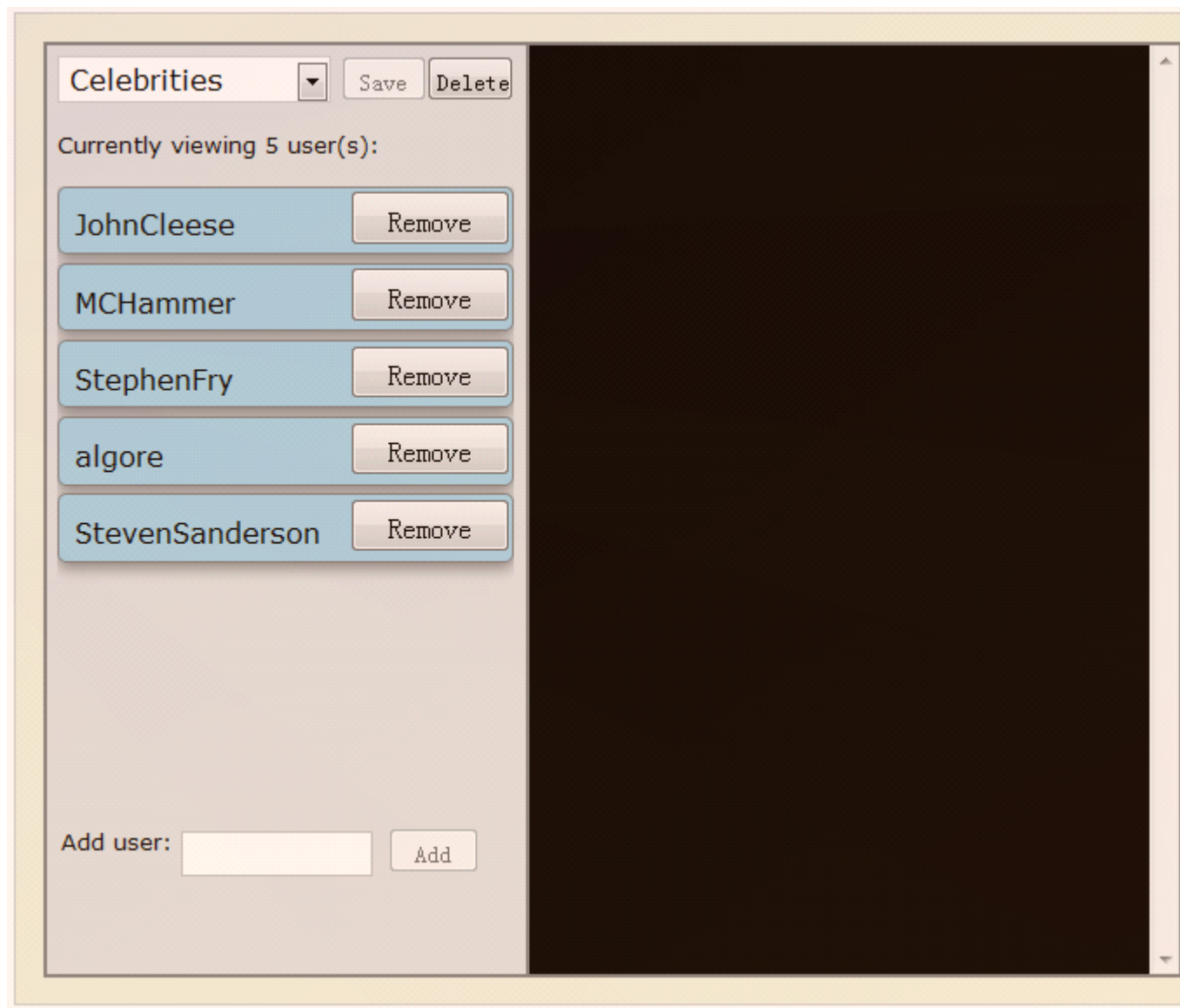
4 Twitter client

这是一个复杂的例子，展示了几几乎所有 **Knockout** 特性来构建一个富客户端。


用户数据存在一个 **JavaScript** 模型里，通过模板来展示。就是说我们可以通过清理用户列表里的数据来达到隐藏用户信息的目的，而不需要手动去隐藏 **DOM** 元素。

按钮将根据他们是否可操作来自动变成 **enabled** 或 **disabled** 状态。例如，有一个叫 **hasUnsavedChanges** 的依赖监控属性(**dependentObservable**)控制这“Save”按钮的 **enabled** 状态。

可以非常方便地从外部 **JSON** 服务获取数据，并集成到 **view model** 里，然后显示在页面上。



代码: View

View Code 

```
<div class="loadingIndicator">
  Loading...</div>
<div class="configuration">
  <div class="listChooser">
    <button data-bind='click: deleteList, enable: editingList.name'>
      Delete</button>
    <button data-bind='click: saveChanges, enable: hasUnsavedChanges'>
      Save</button>
    <select data-bind='options: savedLists, optionsValue: "name", value:
```

```

editingList.name'>
    </select>
</div>
<p>
    Currently viewing <span data-bind="text:
editingList.userNames().length">&nbsp;</span>
    user(s):</p>
    <div class="currentUsers" data-bind='template: { name: "usersTemplate", data:
editingList }'>
        </div>
        <form data-bind="submit: addUser">
            <label>
                Add user:</label>
                <input data-bind='value: userNameToAdd, valueUpdate: "keyup", css:
{ invalid: !userNameToAddIsValid() }' />
                <button type="submit" data-bind='enable: userNameToAddIsValid() &&
userNameToAdd() != ""'>
                    Add</button>
            </form>
        </div>
        <div class="tweets" data-bind='template: { name: "tweetsTemplate", data:
currentTweets }'>
        </div>
        <script type="text/html" id="tweetsTemplate">
            <table width="100%">
                {{each $data}}
                    <tr>
                        <td></td>
                        <td>
                            <a class="twitterUser"
href="http://twitter.com/$ { from_user }">${ from_user }</a>
                            ${ text }
                            <div class="tweetInfo">${ created_at }</div>
                        </td>
                    </tr>
                {{/each}}
            </table>
        </script>
        <script type="text/html" id="usersTemplate">
            <ul>
                {{each(i, userName) userNames()}}
                    <li><button data-bind="click: function()
{ userNames.remove(userName) }">Remove</button> <div>${ userName }</div></li>
                {{/each}}
            </ul>
        </script>
    </div>
</p>
</div>

```

```
</ul>
</script>
```



代码: View model

View Code

```
// The view model holds all the state we're working with. It also has methods
// that can edit it, and it uses
// dependentObservables to compute more state in terms of the underlying data
// --
// The view (i.e., the HTML UI) binds to this using data-bind attributes, so it
// always stays up-to-date with
// the view model, even though the view model does not know or care about any
// view that binds to it

var viewModel = {
  savedLists: ko.observableArray([
    { name: "Celebrities", userNames: ['JohnCleese', 'MCHammer', 'StephenFry',
    'algore', 'StevenSanderson'] },
    { name: "Microsoft people", userNames: ['BillGates', 'shanselman',
    'haacked', 'ScottGu'] },
    { name: "Tech pundits", userNames: ['Scobleizer', 'LeoLaporte',
    'techcrunch', 'BoingBoing', 'timoreilly', 'codinghorror'] }
  ]),

  editingList: {
    name: ko.observable("Tech pundits"),
    userNames: ko.observableArray()
  },

  userNameToAdd: ko.observable(""),
  currentTweets: ko.observableArray([])
};

viewModel.findSavedList = function (name) {
  var lists = this.savedLists();

  for (var i = 0; i < lists.length; i++)
    if (lists[i].name === name)
```

```

        return lists[i];
    };

// Methods
viewModel.addUser = function () {
    if (this.userNameToAdd() && this.userNameToAddIsValid()) {
        this.editingList.userNames.push(this.userNameToAdd());
        this.userNameToAdd("");
    }
}

viewModel.saveChanges = function () {
    var saveAs = prompt("Save as", this.editingList.name());

    if (saveAs) {
        var dataToSave = this.editingList.userNames().slice(0);
        var existingSavedList = this.findSavedList(saveAs);
        if (existingSavedList)
            existingSavedList.userNames = dataToSave; // Overwrite existing list
        else
            this.savedLists.push({ name: saveAs, userNames: dataToSave }); // Add
new list

        this.editingList.name(saveAs);
    }
}

viewModel.deleteList = function () {
    var nameToDelete = this.editingList.name();
    var savedListsExceptOneToDelete = $.grep(this.savedLists(), function (list)
{ return list.name != nameToDelete });
    this.editingList.name(savedListsExceptOneToDelete.length == 0 ? null :
savedListsExceptOneToDelete[0].name);
    this.savedLists(savedListsExceptOneToDelete);
};

ko.dependentObservable(function () {
    // Observe viewModel.editingList.name(), so when it changes (i.e., user
selects a different list) we know to copy the saved list into the editing list
    var savedList = viewModel.findSavedList(viewModel.editingList.name());

    if (savedList) {
        var userNamesCopy = savedList.userNames.slice(0);
        viewModel.editingList.userNames(userNamesCopy);
    }
}

```

```

    } else
        viewModel.editingList.userNames([]);
});

viewModel.hasUnsavedChanges = ko.dependentObservable(function () {
    if (!this.editingList.name())
        return this.editingList.userNames().length > 0;

    var savedData = this.findSavedList(this.editingList.name()).userNames;
    var editingData = this.editingList.userNames();
    return savedData.join("|") != editingData.join("|");
}, viewModel);

viewModel.userNameToAddIsValid = ko.dependentObservable(function () {
    return (this.userNameToAdd() == "") ||
    (this.userNameToAdd().match(/^\\s*[a-zA-Z0-9_]{1,15}\\s*$/) != null);
}, viewModel);

// The active user tweets are (asynchronously) computed from editingList.userNames
ko.dependentObservable(function () {
    twitterApi.getTweetsForUsers(this.editingList.userNames(), function (data)
    { viewModel.currentTweets(data) })
}, viewModel);

ko.applyBindings(viewModel);

// Using jQuery for Ajax loading indicator - nothing to do with Knockout
$(".loadingIndicator").ajaxStart(function () { $(this).fadeIn(); })
    .ajaxComplete(function () { $(this).fadeOut(); });

```

Knockout应用开发指南 第十章：更多信息（完结篇）

1 浏览器支持

Knockout 在如下浏览器通过测试：

- Mozilla Firefox 2.0+（最新测试版本：3.6.8）
- Google Chrome（通过 Windows and Mac 下的 version 5测试；其它低版本也该可以工作）
- Microsoft Internet Explorer 6, 7, 8
- Apple Safari（Windows 下的 Safari 5测试，Mac OS X 下的 Safari 3.1.2测试，以及 iPhone 下的 Safari for iOS 4测试；高低版本应该都可以工作）
- Opera 10 for Windows

Knockout 应该在以上这个浏览器的各版本上工作，但是由于太多版本，没有逐一测试。最新测试结果显示，Knockout 在如下浏览器也是可以工作的（尽管没有对每个版本逐一测试）：

Opera Mini

Google Android OS browser (OS version 2.2)

测试 Knockout 能否在一个新平台或浏览器下工作，只需要下载[源代码](#)，然后在该浏览器里运行里面的/spec/runner.html 文件即可测试。这个文件可以验证超过100个行为特性，如果有问题则会生成报表。上述浏览器的测试结果都应该是100%通过。

2 寻求帮助

有任何问题，你可以在 Google group 进去寻求帮助。

地址：<http://groups.google.com/group/knockoutjs>

3 更多教程和例子

这里有更多使用 Knockout 和其它相关技术的页面和例子：

[Knock Me Out](#) — Ryan Niemeyer 的博客, 包括 KnockoutJS 和相关技术的很多好创意、想法和讨论

[Editing a variable-length list, Knockout-style](#) — Steve Sanderson 展示的在 ASP.NET

MVC 下使用 Knockout 的好处

[Knockout+WebSockets](#) — Carl Hörberg 使用 Knockout, Sinatra, SQLite 和 WebSockets 实现的实时讨论版

[Knockout – quick asp.net mvc sample](#) — Steve Gentile 提供的另外一篇博客， 关于 ASP.NET MVC 下如何使用 Knockout

[Log4Play: Log4j Live Streaming with Play Framework, Knockout.js and WebSockets](#)
— Felipe Oliveira 关于使用 KO 和 WebSockets 创建实时 server log 功能

[Wiki – 攻略](#) — 网友贡献的攻略和例子

[Wiki – 插件](#) — 网友贡献的各种插件列表