目錄

knockoutjs极简开发指南

谁适合看这本书？

希望使用knockoutjs来开发web应用的前端程序员

你需要具备哪些知识？

了解基础的css、js、html知识

为什么到了**2017**年我们还要使用**knockoutjs**？

其实knockoutjs也不赖，这家伙现在还在更新呢。对于一些场景单项数据流和状态管理有点太复杂了，都是双向也是很好操作的嘛！

使用须知：

1.本文是极简开发指南旨在方便大家快速上手，阅读高级教程之外的章节即可入门

2.本书基于knockoutjs 3.4.x版本进行，也会涉及到一些knockoutjs 3.5.0 beta的内容

3.深入了解请访问官方文档http://knockoutjs.com/documentation/introduction.html

最后把这本书献给懒得看英文文档的开发者们，还是中文最贴切

联系作者：宋江凌

邮箱： songhlc@yonyou.com

MVVM

Model View ViewModel

Model表示数据模型（通常对应于前端ajax请求后台返回的数据）

View表示视图层（通常就是html页面）

ViewModel（基于Model但和View做了双向数据绑定，同时也可能包含一些操作model改变的方法）

来个例子：

```
// view
<body>
    <input data-bind="value:inputValue"/>
</body>
```

```
<script>
   var model = {
      inputValue: '2'
   }
   var viewModel = {
      inputValue: ko.observable(model.inputValue)
   }
</script>
```

如上，当我在页面的input输入框输入值之后，viewModel.inputValue 值就自动改变了，

同样如果我给viewModel.inputValue赋值：viewModel.inputValue("test")，那么输入框里的值就自动变成test了。

所以这就是所谓的双向数据绑定。

Model和viewModel的区别

model通常可以理解为是要提交到后端的数据模型，而viewModel除了数据模型本身可能还有一些和view相关的状态，

比如：给viewModel里包含一个状态isVisible，用于根据某些规则判断输入框是否要显示出来。

```
<body>
   <div data-bind="visible:isVisible">
    <input data-bind="value:inputValue"/>
   <div>
</body>
<script>
   var model = {
      inputValue: '2'
   }
   var viewModel = {
      inputValue: ko.observable(model.inputValue),
      isVisible: ko.observable(false) // 属于view中需要使用到的vm但
model不需要
   }
</script>
```

> 注解：
>
> 1.此处data-bind可以理解为设置双向绑定，value表示绑定input的value属性，visible表示根据后面的值动态切换当前element的隐藏/显示（相当于动态设置display:none）
>
> 2.ko.observable表示定义一个支持双向数据绑定的对象

监听对象Observables

1.如何声明一个监听对象（下称ko对象）

通过ko创建一个viewmodel，例如:

```
<script>
var viewModel = {
    id: ko.observable(123),
    name: ko.observable('john')
}
</script>
```

创建一个简单的view，例如：

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

激活**Knockout**

html标签本身并不能识别data-bind属性，所以需要在js中进行绑定，将viewmodel和view关联起来

```
ko.applyBindings(viewModel);
// ko.applyBindings支持两个参数，// 可以把viewmodel绑定到指定id上
// ko.applyBindings(viewModel, document.getElementById('id')).
```

最终的渲染效果

```
My name is <span data-bind="text: name">john</span>
```

2.ko对象读写

- 读取ko对象的值，viewModel.name() 会返回"john"，viewModel.id() 会返回

123

- 设置ko对象的值，访问ko对象并传入一个新的值做为参数，例如viewModel.name('jerry')会把name的值设置成jerry（由于data-bind中text指令的存在，所以页面会渲染成：My name is jerry）

3.订阅到ko对象的改变

如果你需要在ko对象值改变之后，做一些自己的业务操作，比如：当输入的手机号改变之后，自动触发手机号合法性验证；那么你可以使用：

```
viewModel.phone.subscribe(function(newValue) {
    // 检测newValue的值是否是一个合法的电话号码
});
```

高级应用

subscribe添加beforeChange的订阅，可以监控到phone改变以前的值

```
viewModel.phone.subscribe(function(oldValue) {
    // 获取改变以前的值
}, null, "beforeChange");
```

4.数组对象的监听

```
viewModel = {
 myObservableArray = ko.observableArray();
}
viewModel.myObservableArray([1,2,3])
```

observable => observableArray

关于数组对象的使用，和javascript里的array一样，同样可以使用以下数组常用语法：

```
viewModel.myObservableArray.indexOf(1)  // 0
viewModel.myObservableArray.push(4)  // array的值会变成[1,2,3,4]
```

以下事件会触发数组值的改变并重新渲染相关页面

1. push
2. pop
3. shift
4. unshift
5. reverse
6. sort
7. splice

> 注意以下操作不会通知到observableArray的改变（实际值改变，但相关联的view不会自动改变）

```
viewModel = {
 myObservableArray = ko.observableArray([{a:1,b:2}]);
}
viewModel.myObservableArray()[0].a = 2 // 改变数组对象中object里参数的值
```

应当这么处理：

```
var item = viewModel.myObservableArray(0)
item.a = 2

viewModel.myObservableArray.splice(0, 1, item) // 删除某一项，然后再同样位置再插入该项修改后的结果
```

计算属性

假设我们页面上有一个区域用于显示币种+金额，我们可以使用计算属性来实现这种需求

> 当currency或amout改变之后，amountDisplay也会自动计算并改变值

```
<html>
    <div data-bind="text:amountDisplay"></div>
</html>
<script>
    // 这里我们通过initVm方法来初始化viewmodel，能更好的使用this上下文
    function initVm () {
        this.currency = ko.observable('$')
        this.amout = ko.observable('2000.00')
        this.amountDisplay = ko.computed(function () {
            // 计算属性需要return最终的计算值
            return this.currency() + this.amout()
        })
    }
    var viewModel = new initVm()
    ko.applyBindings(viewModel)
</script>
```

> 当然，我们也可以这么写
>
> ```
> 
> ```
>
> 不过如果计算规则复杂之后，在html里写这些以后就不容易维护了
>
> ## 原则一：尽量少在**html**里写复杂的业务逻辑，使用计算属性来代替

关于pureComputed,请参见 高级教程 章节

关于指令

1.什么是指令

```
<div data-bind="text:title"></div>
```

html中跟在data-bind后面的text就是指令。

2.ko预置的指令都有哪些

2.1用于控制文本和显示的

- **visible/hidden(3.5.0-beta版本新增，和visible相对应)**

```
// 例子
<div data-bind="visible: shouldShowMessage">
    只有 "shouldShowMessage" 为true时才会显示在页面上（否则 displ
ay:none）
</div>

<script type="text/javascript">
    var viewModel = {
        shouldShowMessage: ko.observable(true) // Message in
itially visible
    };
    viewModel.shouldShowMessage(false); // ... now it's hidd
en
    viewModel.shouldShowMessage(true); // ... now it's visib
le again
</script>
```

- **text**

```
// 显示值
<span data-bind="text: title"></span>

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

- **html**

```
// 用于展示html片段
<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 rep
ort <a href='report.html'>here</a>.</em>"); // HTML content
appears
</script>
```

- **css**

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

<script type="text/javascript">
    var viewModel = {
        currentProfit: ko.observable(150000) // Positive val
ue, so initially we don't apply the "profitWarning" class
    };
    viewModel.currentProfit(-50); // Causes the "profitWarni
ng" class to be applied
</script>
```

- **style**

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

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

- **attr**

```
// 动态绑定属性
<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>
```

- **class(3.5.0+ 新增)**

```
// 动态绑定属性
<a data-bind="class: classes">
    Report
</a>

<script type="text/javascript">
    var viewModel = {
        classes: ko.observable("classA classB classC")
    };
</
```

2.2控制流程

- **foreach**

表单中数据循环, $index 可以用于获取当前需要的索引（从0开始）

```
<table>
    <thead>
        <tr><th>index</th><th>First name</th><th>Last name</th><
/tr>
    </thead>
    <tbody data-bind="foreach: people">
        <tr>
            <td data-bind="text: $index"></td>
            <td data-bind="text: firstName"></td>
            <td data-bind="text: lastName"></td>
        </tr>
    </tbody>
</table>

<script type="text/javascript">
    var vm = {
        people: ko.observableArray([
            { firstName: 'Bert', lastName: 'Bertington' },
            { firstName: 'Charles', lastName: 'Charlesforth' },
            { firstName: 'Denise', lastName: 'Dentiste' }
        ])
    }
    ko.applyBindings(vm);
</script>
```

```
// 灵活使用as别名，在多重循环中灵活使用
<ul data-bind="foreach: { data: categories, as: 'category' }">
    <li>
        <ul data-bind="foreach: { data: items, as: 'item' }">
            <li>
                <span data-bind="text: category.name"></span>:
                <span data-bind="text: item"></span>
            </li>
        </ul>
    </li>
</ul>

<script>
    var viewModel = {
        categories: ko.observableArray([
            { name: 'Fruit', items: [ 'Apple', 'Orange', 'Banana
' ] },
            { name: 'Vegetables', items: [ 'Celery', 'Corn', 'Sp
inach' ] }
        ])
    };
    ko.applyBindings(viewModel);
</script>
```

- **if/ifnot**（和**if**相对应）

和visible的区别，visible只是切换div的css中的display样式，if为false时，内部的dom结构是不会渲染到页面上的

```
<label><input type="checkbox" data-bind="checked: displayMessage
" /> Display message</label>

<div data-bind="if: displayMessage">Here is a message. Astonishi
ng.</div>
```

组件

> 组件是目前所有MVVM框架之中最基础、最重要的部分，学好如何正确的使用组件对于快速开发一个工程有非常大的好处

组件产生的目的为了复用

概述：组件和自定义标签

组件十分强大,它可以让你十分轻松的组织并重用你的程序代码（尤其是UI层的代码）。

- 可用于展现独立的控件或小部件，或者程序的整个部分
- 包含独立的view层，通常也拥有对应的viewmodel
- 支持接受多种自定义参数和回调事件
- 可以很方便的进行集成和组件嵌套
- 极易封装并提供跨工程使用场景
- 非常灵活的自定义和加载

这种模式有助于大型应用程序，它通过清晰的组织和封装来简化开发，并有助于按需加载应用程序代码和模板来提高运行时的性能。

自定义标签的开发方式能够让你快速的写出一个组件，相比于通过一个** **标签再加入一堆的data-bind指令，你可以使用一些更具语义化的命名方式（比如：**<product-list>** 或 **<img-list>**）

例子

```
ko.components.register('like-widget', {
    viewModel: function(params) {
        // Data: value is either null, 'like', or 'dislike'
        this.chosenValue = params.value;

        // Behaviors
        this.like = function() { this.chosenValue('like'); }.bin
d(this);
        this.dislike = function() { this.chosenValue('dislike');
 }.bind(this);
    },
    template:
        '<div class="like-or-dislike" data-bind="visible: !chose
nValue()">\
            <button data-bind="click: like">Like it</button>\
            <button data-bind="click: dislike">Dislike it</butto
n>\
        </div>\
        <div class="result" data-bind="visible: chosenValue">\
            You <strong data-bind="text: chosenValue"></strong>
it\
        </div>'
});
```

例子效果和原文参见http://knockoutjs.com/documentation/component-overview.html

组件注册

如同上一个章节，组件注册需要使用 `ko.components.register` 进行声明

viewmodel/template 组合声明

```
ko.components.register('some-component-name', {
    viewModel: <see below>,
    template: <see below>
});
```

- 上面例子中 `some-component-name` 表示组件名，通常我们建议使用小写字母，并且单词之间使用 `-` 进行分割
- viewModel 是可选择的（对于一些纯静态展示的组件，可以不使用 viewModel）
- template是必须选项

三种声明viewModel的方式

1.通过构造函数

```
function SomeComponentViewModel(params) {
    // 'params' is an object whose key/value pairs are the param
eters
    // passed from the component binding or custom element.
    this.someProperty = params.something;
}

SomeComponentViewModel.prototype.doSomething = function() { ...
};

ko.components.register('my-component', {
    viewModel: SomeComponentViewModel,
    template: ...
});
```

组件会在加载的时候自动调用构造函数，并对template中的字段进行绑定，这是通常大部分组件使用的方式

注：可以理解为ko会主动调用 `new SomeComponentViewModel(params)`

2.可共享的实例

如果你需要在不同组件中共享一个viewModel对象，那么可以这么使用：

```
var sharedViewModelInstance = { ... };

ko.components.register('my-component', {
    viewModel: { instance: sharedViewModelInstance },
    template: ...
});
```

想象一下，假设你有一个商品列表，列表中有一个tab页签用于切换商品列表的展现形式（纵向排列/九宫格展示），同时可能还存在一些数据勾选的交互。如果不共享viewModel对象，那么你就要考虑到两个组件之间数据的同步。而如果使用了可共享实例，那么你们操作的就是同一个viewModel了。

注：适用于一些出现在同一个页面的不同组件，数据来源一致，但展现形式不一致的场景

3.通过 `createViewModel` 工厂函数

如果你想要对于当前组件绑定的dom元素做一些配置或者操作，或者对于params有一些动态的处理，那么可以这么使用：

```
ko.components.register('my-component', {
    viewModel: {
        createViewModel: function(params, componentInfo) {
            // - 'params' is an object whose key/value pairs are
 the parameters
            //   passed from the component binding or custom ele
ment
            // - 'componentInfo.element' is the element the comp
onent is being
            //   injected into. When createViewModel is called,
the template has
            //   already been injected into this element, but is
n't yet bound.
            // - 'componentInfo.templateNodes' is an array conta
ining any DOM
            //   nodes that have been supplied to the component.
 See below.

            // Return the desired view model instance, e.g.:
            return new MyViewModel(params);
        }
    },
    template: ...
});
```

- componentInfo.element 指向了当前组件所在的dom元素，如ycloud中将componentInfo.element赋值给了this.$el
- componentInfo.templateNodes 是组件标签内部的自定义模板，通常如果需要自定义内部模板的时候可以使用它

四种声明**template**的方式

1.通过一个已存在的元素id

```
<template id='my-component-template'>
    <h1 data-bind='text: title'></h1>
    <button data-bind='click: doSomething'>Click me right now</b
utton>
</template>
```

```
ko.components.register('my-component', {
    template: { element: 'my-component-template' },
    viewModel: ...
});
```

2.已存在的dom元素

```
var elemInstance = document.getElementById('my-component-templat
e');

ko.components.register('my-component', {
    template: { element: elemInstance },
    viewModel: ...
});
```

3.html片段

```
ko.components.register('my-component', {
    template: '<h1 data-bind="text: title"></h1>\
               <button data-bind="click: doSomething">Clickety</
button>',
    viewModel: ...
});
```

4.dom节点数组

```
var myNodes = [
    document.getElementById('first-node'),
    document.getElementById('second-node'),
    document.getElementById('third-node')
];

ko.components.register('my-component', {
    template: myNodes,
    viewModel: ...
});
```

使用组件

通常我们有两种方式来调用我们的组件，假设组件定义如下：

```
ko.components.register('message-editor', {
    viewModel: function(params) {
        this.text = ko.observable(params && params.initialText |
| '');
    },
    template: 'Message: <input data-bind="value: text" /> '
            + '(length: <span data-bind="text: text().length"></
span>)'
});
```

请注意组件需要传入参数: `initialText`

1.自定义标签

```
<message-editor
    params="initialText:ko.observable('testme')">
</message-editor>
```

在html中使用自定义标签就可以了，然后通过params传入参数。

2.component指令

```
<div data-bind='component: {
    name: "message-editor",
    params: { initialText: "Hello, world!" }
}'></div>
```

组件生命周期

1.组件读取构造函数和模板

2.组件克隆模板并注入element对象之中

3.如果包含viewModel,将viewModel实例化

4.viewModel被绑定到view之中

5.组件被激活

6.如果component指令绑定的组件名改变，将会销毁之前的viewModel

如果你还没有开始正式项目的开发，请略过这个章节

这个章节暂时不会更新，因为是极简开发指南，所以所谓的高级教程就是ko的官方文档了。

http://knockoutjs.com/documentation