<http://www.cnblogs.com/landeanfen/p/5400654.html>

[**JS组件系列——BootstrapTable+KnockoutJS实现增删改查解决方案（一）**](http://www.cnblogs.com/landeanfen/p/5400654.html)

**阅读目录**

* [一、Knockout.js简介](http://www.cnblogs.com/landeanfen/p/5400654.html#_label0)
  + [1、Knockout.js和MVVM](http://www.cnblogs.com/landeanfen/p/5400654.html#_label0_0)
  + [2、最简单的实例](http://www.cnblogs.com/landeanfen/p/5400654.html#_label0_1)
  + [3、监控属性](http://www.cnblogs.com/landeanfen/p/5400654.html#_label0_2)
  + [4、ko里面常见的data-bind属性](http://www.cnblogs.com/landeanfen/p/5400654.html#_label0_3)
  + [5、Json对象和监控属性的转化及关系](http://www.cnblogs.com/landeanfen/p/5400654.html#_label0_4)
  + [6、创建自己的data-bind属性](http://www.cnblogs.com/landeanfen/p/5400654.html#_label0_5)
* [二、第一个增删改查实例](http://www.cnblogs.com/landeanfen/p/5400654.html#_label1)

**正文**

前言：出于某种原因，需要学习下Knockout.js，这个组件很早前听说过，但一直没尝试使用，这两天学习了下，觉得它真心不错，双向绑定的机制简直太爽了。今天打算结合bootstrapTable和Knockout去实现一个简单的增删改查，来体验一把神奇的MVVM。关于WebApi的剩余部分，博主一定抽时间补上。

[回到顶部](http://www.cnblogs.com/landeanfen/p/5400654.html#_labelTop)

**一、Knockout.js简介**

[回到顶部](http://www.cnblogs.com/landeanfen/p/5400654.html#_labelTop)

**1、Knockout.js和MVVM**

如今，各种前端框架应接不暇，令人眼花缭乱，有时不得不感叹作为程序猿也真是苦逼，总有学不完的技术，何时是尽头，除非你转化！苦海无边，回头是不是岸，由你决定！

Knockout.js是一个基于MVVM模式的轻量级的前端框架，有多轻？根据官网上面显示的最新版本v3.4.0，仅22kb。能够友好地处理数据模型和界面DOM的绑定，最重要的是，它的绑定是双向的，也就是说数据模型变化了，界面DOM上的数据也会跟着发生变化，反过来，界面DOM上的数据变化了，数据模型也会相应这个变化。这样能够大大减少我们的前端代码量，并且使得我们界面易于维护，再也不用写一大堆事件监控数据模型和界面DOM的变化了。下面博主会根据一个使用实例来说明这两点。

Knockout.js官网：[http://knockoutjs.com](http://knockoutjs.com/)

Knockout.js开源地址：<https://github.com/knockout/knockout>

MVVM模式：这是一种创建用户界面的设计模式，MVVM把它拆分成三块就是Model、View、ViewModel，Model就是数据模型，View就是我们的视图，ViewModel就是一个视图模型，用来绑定数据模型和视图上面的dom元素。如果你使用过WPF和Silverlight，理解这个应该不是啥问题；没有使用过也什么关系，看完此文，你会有一个大致的认识。

[回到顶部](http://www.cnblogs.com/landeanfen/p/5400654.html#_labelTop)

**2、最简单的实例**

 一般来说，如果你从零开始使用Knockout.js，你至少需要做以下四部

**2.1、去官网下载knockout.js文件，然后引用到view页面里面。**

<script src="~/scripts/knockout/knockout-3.4.0.min.js"></script>

注意：knockout.js并不需要jquery的支持，如果你的项目需要用到jquery的相关操作，则引用jquery；否则只引用以上文件即可。

**2.2、定义ViewModel**

viewmodel是什么？其实，在js里面，它看上去就像一个json对象。我们定义一个viewmodel：

var myViewModel = {

Name: "Lilei",

profession: "软件工程师",

};

**2.3、view视图里面定义绑定data-bind的标签**

<div>

姓名：<label data-bind="text:Name"></label><br />

职业：<input type="text" data-bind="textinput:Profession" />

</div>

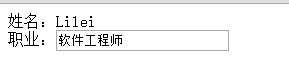
注意：对应input标签的文本，需要使用textinput，而普通标签的文本使用text即可。

**2.4、激活绑定**

做了以上三步，还需要激活knockout的绑定

ko.applyBindings(myViewModel);

做到这四部，基本实现了一个最简单的viewmodel的数据绑定。得到效果：



**如果你够细心，会发现ko.applyBindings()方法有两个参数，第一个就是我们需要绑定的viewmodel，第二个又是什么呢？由 ko.applyBindings(myViewModel); 可知，第二个参数是一个可选参数，它表示viewmodel绑定的标签的作用域，比如，我们将以上代码改一下：**

<div>

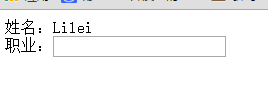
姓名：<label id="lb\_name" data-bind="text:Name"></label><br />

职业：<input type="text" data-bind="textinput:Profession" />

</div>

ko.applyBindings(myViewModel,document.getElementById("lb\_name"));

得到结果：



由此可知：第二个参数限定了myViewModel的作用范围，也就是说，只有在id="lb\_name"的标签上面绑定才会生效，如果第二个参数是div等容器标签，它表示该绑定的范围为该div下面的所有子标签。

[回到顶部](http://www.cnblogs.com/landeanfen/p/5400654.html#_labelTop)

**3、监控属性**

截止到上面的四步，我们看不到任何效果，看到的无非就是将一个json对象的的数据绑定到了html标签上面，这样做有什么意义呢？不是把简单的问题复杂化吗？别急，马上见证奇迹！上文说了，knockout最重要的意义在于双向绑定，那么如何实现我们的双向绑定呢？答案就是监控属性。

在knockout里面，核心的有三个监控属性：**Observables,DependentObservables,ObservableArray**，Observe的意思翻译过来是观察、观测的意思，如果说成观察属性或者观测属性感觉不太恰当，我们暂且叫监控属性。

**3.1、Observables：监控属性**

我们将上面的例子改成这样：

[复制代码](javascript:void(0);)

<head>

<meta name="viewport" content="width=device-width" />

<title>Index3</title>

<script src="~/scripts/jquery-1.9.1.min.js"></script>

<script src="~/scripts/knockout/knockout-3.4.0.min.js"></script>

</head>

<body>

<div>

姓名：<label data-bind="text:Name"></label><br />

职业：<input type="text" data-bind="textinput:Profession" />

</div>

<div>

<input type="text" id="txt\_testobservable" />

</div>

<script type="text/javascript">

//1.定义ViewModel

var myViewModel = {

Name: ko.observable("Lilei"),

Profession: "软件工程师",

};

//2.激活绑定

ko.applyBindings(myViewModel);

$(function () {

//注册文本框的textchange事件

$("#txt\_testobservable").on("input", function () {

myViewModel.Name($(this).val());

});

});

</script>

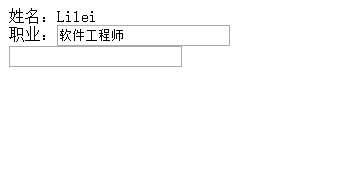
</body>

[复制代码](javascript:void(0);)

 ko.observable("Lilei") 这一句的意义是将viewmodel的Name属性添加成为监控属性，一定Name属性变成监控属性，神奇的事情就发生了，我们来看看当我们写myViewModel.的时候：



Name由原来的属性变成方法了，也就是说一旦添加了ko.observable()，那么对应的属性就会变成方法，那么对于Name的取值和赋值都需要使用myViewModel.Name()来处理。我们先来看看效果：



**代码释疑：很显然  myViewModel.Name($(this).val()); 这一句将当前文本框的值赋给了Name属性，由于界面绑定了Name属性，所以label里面的值也随之发生了变化。或者你会说，这个使用textchange事件也可以做到的，只要将当前文本框的值赋给label标签，也可以达到这个效果，这个不算什么。确实，你的写法也可以达到目的，但是我们的监控属性的意义在于，任何地方改变了Name的值，界面都会随之变化，而不用每个地方去给label标签赋值，JS里面只需要把关注点方法myViewModel.Name()上面即可。是不是很厉害~~**

**3.2、DependentObservables：监控依赖属性**

如果看了上面的监控属性还没过瘾？下面再来看看监控依赖属性的使用。

我们将代码再改下看看：

[复制代码](javascript:void(0);)

<head>

<meta name="viewport" content="width=device-width" />

<title>Index3</title>

<script src="~/scripts/jquery-1.9.1.min.js"></script>

<script src="~/scripts/knockout/knockout-3.4.0.min.js"></script>

</head>

<body>

<div>

姓名：<input type="text" data-bind="textinput:Name" /><br />

职业：<input type="text" data-bind="textinput:Profession" /><br />

描述：<label data-bind="text:Des"></label>

</div>

<script type="text/javascript">

//1.定义ViewModel

var myViewModel = {

Name: ko.observable("Lilei"),

Profession: ko.observable("软件工程师"),

};

myViewModel.Des = ko.dependentObservable(function () {

return "本人姓名——" + myViewModel.Name() + "，职业——" + myViewModel.Profession();

});

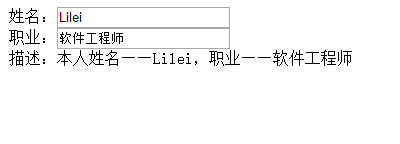
//2.激活绑定

ko.applyBindings(myViewModel);</script>

</body>

[复制代码](javascript:void(0);)

先来看看效果：



**代码释疑：通过添加监控依赖属性  ko.dependentObservable() 将Des属性的值能同时监控到Name和Profession两个的变化，其中任何一个发生变化，Des绑定的标签都会触发改变，这样做的最大好处就是避免了我们js去操作dom的麻烦，有点意思吧。**

**3.3、ObservableArray；监控数组**

除了上面两种，ko还支持对数组对象的监控。我们来看一个例子：

[复制代码](javascript:void(0);)

<head>

<meta name="viewport" content="width=device-width" />

<title>Index3</title>

<script src="~/scripts/jquery-1.9.1.min.js"></script>

<script src="~/scripts/knockout/knockout-3.4.0.min.js"></script>

</head>

<body>

<div><select data-bind="options:deptArr,

optionsText:'Name'"></select>

</div>

<div>

<input type="text" id="txt\_testobservable" /><input type="button" id="btn\_test" value="新增部门" />

</div>

<script type="text/javascript">

var deptArr = ko.observableArray([

{ id: 1, Name: '研发部' },

{ id: 2, Name: '行政部' },

{ id: 3, Name: '人事部' }

]);

var viewModel = {

deptArr: deptArr,

};

ko.applyBindings(viewModel);

var i=4;

$(function () {

$("#btn\_test").on("click", function () {

deptArr.push({ id: i++, Name: $("#txt\_testobservable").val() });

});

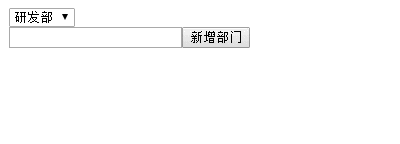
});

</script>

</body>

[复制代码](javascript:void(0);)

看看效果：



**代码释疑：以上通过ko.observableArray()这个方法添加了对数组对象的监控，也就是说，js里面任何地方只要对deptArr数组对象做了数组的改变，都会触发UI给出相应。需要注意的一点是，监控数组实际上是监控的数组对象本身，对于数组对象里面的子对象属性发生变化，是无法监控到的。比如我们将点击事件改成这样：**

$(function () {

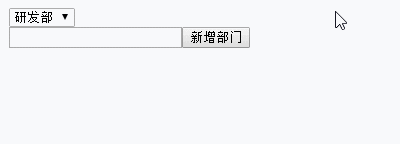
$("#btn\_test").on("click", function () {

deptArr[1].Name = "aaa";

});

});

效果：



**由此说明数组监控实际上监控的是数组对象本身，对于数组里面元素的属性变化不会监控。如果确实需要对数据里面对象的属性变化进行监控，需要再对数据里面对象属性使用ko.observable()，两者联合使用。有兴趣的可以试试。**

[回到顶部](http://www.cnblogs.com/landeanfen/p/5400654.html" \l "_labelTop)

**4、ko里面常见的data-bind属性**

上文中，我们使用了多个data-bind属性，那么在knockout里面，到底有多少个这种data-bind的属性呢。这里我们列出一些常用的属性。

**4.1、text和inputText**

text，顾名思义就是文本的意思，这个绑定属性一般用于<label>、<span>、<div>等标签显示文本，当然，如果你愿意，任何标签都可以使用这个绑定。它是使用基本上没什么好说的。如果没有使用ko.observable()，则是静态绑定，否则是动态绑定。

inputText，input标签的文本，相当于input标签的value属性。

　　<div>

姓名：<label data-bind="text:Name"></label><br />

职业：<input type="text" data-bind="textinput:Profession" />

</div>

[复制代码](javascript:void(0);)

　　　　 //1.定义ViewModel

var myViewModel = {

Name: ko.observable("Lilei"),

Profession: "软件工程师",

};

//2.激活绑定

ko.applyBindings(myViewModel);

[复制代码](javascript:void(0);)

**4.2、value**

这个绑定属性一般用于input标签，和上面的inputText基本相似。只不过value更加规范。

和value一起使用的还有一个参数valueUpdate，它表示界面做一个什么操作的时候更新该value。valueUpdate主要取值有change/keyup/keypress/afterkeydown等。分别表示文本变化、键盘缩起、键盘按下、键盘按下之后等操作时候更新value对应的viewmodel的值。

姓名：<input type="text" data-bind="value:Name,valueUpdate:'keyup'" /><br />

　　　　var myViewModel = {

Name: ko.observable("Lilei"),

};//2.激活绑定

ko.applyBindings(myViewModel);

上述代码表示键盘收起的时候更新文本框的value属性和myViewModel的Name属性。

**4.3、checked**

checked绑定一般用于checkbox、radio等可以选中的表单元素，它对应的值是bool类型。和value的用法基本相似，就不做重复说明。

**4.4、enable**

enable绑定一般用于是否启用标签元素，一般用于表单元素的启用和禁用。和disabled相反，对应的值也是bool类型。

[复制代码](javascript:void(0);)

　　<div>

<input type="text" data-bind="enable:IsMen"/>

</div>

<script type="text/javascript">

//1.定义ViewModel

var myViewModel = {

Name: ko.observable("Lilei"),

Profession: ko.observable("软件工程师"),

Age: ko.observable(40),

IsMen:ko.observable(true)

};

//2.激活绑定

ko.applyBindings(myViewModel);

myViewModel.IsMen(false);

</script>

[复制代码](javascript:void(0);)

由于IsMen属性变成了false，所有对应的文本框会显示禁用状态。

**4.5、disabled**

和enable相反，用法和enable类似。

**4.6、options**

上文中在使用select的绑定时候使用过options，它表示select标签的option的集合，对应的值为一个数组，表示这个下拉框的数据源。可以使用observableArray启用对这个数据源的监控。用法见上面。

**4.7、html**

text绑定实际上是对标签innerText的设置和取值，那么同理，html绑定也是对innerHTML的设置和取值。它对应的值为一段html标签。

**4.8、css**

css绑定是添加或删除一个或多个样式（class）到DOM元素上。使用格式：

<style type="text/css">

.testbold {

background-color:powderblue;

}

</style>

<div data-bind="css:{testbold:myViewModel.Name()=='Lilei'}">aaaa</div>

var myViewModel = {

Name: ko.observable("Lilei"),

Profession: ko.observable("软件工程师"),

Age:ko.observable(40)

};

该div会显示背景色。

如果需要增加或移除多个样式，只要稍微改下即可，比如：

<div data-bind="css:{testbold:myViewModel.Name()=='Lilei',testborder:myViewModel.Profession()=='PHP工程师'}">aaaa</div>

**4.9、style**

如果css绑定的作用是向标签动态添加或移除class样式，那么style绑定的作用就是想标签动态添加或移除某一个样式。比如：

<div data-bind="css:{background-color:myViewModel.Name()=='Lilei'?'red':'white'}">aaaa</div>

如果是添加或者移除多个，同css绑定的用法

**4.10、attr**

attr绑定主要用于向标签添加移除某一个或多个属性（包括自定义属性），永和和css类似。

**4.11、click**

click绑定表示在对应的DOM元素上面添加点击事件的执行方法。可以在任意元素上面使用。

<div>

<input type="button" value="测试click绑定" data-bind="click:ClickFunc" />

</div>

[复制代码](javascript:void(0);)

　　　　var myViewModel = {

ClickFunc:function(){

alert($(event.currentTarget).val());

}

};

ko.applyBindings(myViewModel);

[复制代码](javascript:void(0);)

event.currentTarget表示当前点击的DOM元素。有时为了简便，我们直接使用匿名函数的方式绑定，比如：

<div>

<input type="button" value="测试click绑定" data-bind="click:function(){

alert('点击了');

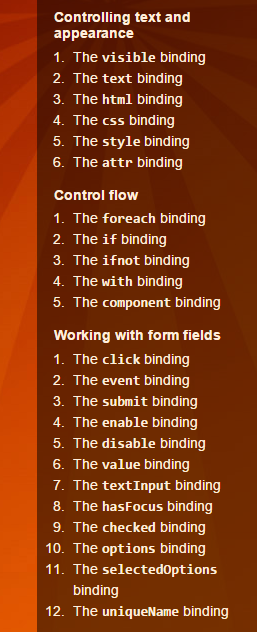
}" />

</div>

但是这种将js揉到html里面的写法让博主难以接受，并且觉得维护起来相对不方便，尤其是点击事件里面的逻辑略复杂时。所以，如非必须，不建议直接写这种匿名函数的方式。

**4.12、其他**

关于data-bind的所有绑定，可以看官网上面的介绍，这里就不一一列举了。需要用的时候去官网上查下就好了。看看官网上面列举的所有绑定：



[回到顶部](http://www.cnblogs.com/landeanfen/p/5400654.html#_labelTop)

**5、Json对象和监控属性的转化及关系**

我们知道，为了避免不同语言直接的展现方式，一般情况下我们前端和后端交互的时候统一使用Json格式的数据，我们通过http请求从后端取到的数据模型，而要使用我们的ko的一些特性，必须要将这些普通的数据模型转换成ko的监控属性；反过来，我们使用ko的监控属性，有时又需要把这些属性转换为普通的json数据传到后台，那么如何实现这个转换呢？

**5.1、JSON对象转换成ViewModel**

比如我们从后台取到一个Json对象，然后把它变成到我们的viewmodel，然后绑定到我们的界面DOM。

[复制代码](javascript:void(0);)

$.ajax({

url: "/Home/GetData",

type: "get",

data: {},

success: function (data, status) {

var oJson = data;

}

});

[复制代码](javascript:void(0);)

我们发送一个请求到后端，取到一个json对象，赋值到oJson，然后我们把oJson转换成viewmodel，最直观的方式就是手动转换了。比如我们可以这样：

[复制代码](javascript:void(0);)

　　　　var myViewModelJson = {

DeptName: ko.observable(),

DeptLevel: ko.observable(),

DeptDesc:ko.observable()

};

ko.applyBindings(myViewModelJson);

[复制代码](javascript:void(0);)

然后在ajax请求的success里面

[复制代码](javascript:void(0);)

success: function (data, status) {

var oJson = data;

myViewModelJson.DeptName(oJson.DeptName);

myViewModelJson.DeptLevel(oJson.DetpLevel);

myViewModelJson.DeptDesc(oJson.DeptDesc);

}

[复制代码](javascript:void(0);)

这样，通过手动绑定，实现了json对象到viewmodel的绑定。这样做的好处就是灵活，坏处显而易见，手工代码量太大。

还好，有我们万能的开源，总有人想到更好的办法，我们使用knockout.Mapping组件就能很好地帮助我们界面json对象到viewmodel的转换。

knockout.Mapping开源地址：[下载](https://github.com/SteveSanderson/knockout.mapping)

下面来简单看看它如何使用，还是上面的例子，我们不用实现定义任何viewmodel，首先需要引用knockout.mapping.js

<script src="~/scripts/knockout/knockout-3.4.0.min.js"></script>

<script src="~/scripts/knockout/extensions/knockout.mapping-latest.js"></script>

**注意：这里knock.mapping-lastest.js必须要放在knockout-3.4.0.min.js的后面，否则调用不到ko.mapping。**

然后直接在success函数里面这样使用

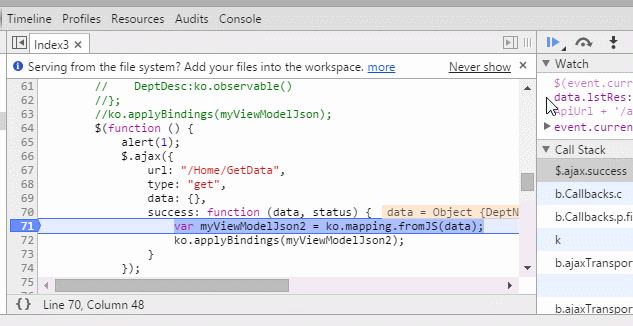
　　　　　　　　　success: function (data, status) {

var myViewModelJson2 = ko.mapping.fromJS(data);

ko.applyBindings(myViewModelJson2);

}

我们来看效果：



**代码释疑：通过ajax请求从后台取到的json对象，通过ko.mapping.fromJS()，很方便地将其转换成了viewmodel，是不是猴犀利！当然除了这种用法，还可以更新已经存在viewmodel，使用如下：**

[复制代码](javascript:void(0);)

　　　　var myViewModelJson = {

DeptName: ko.observable(),

DeptLevel: ko.observable(),

DeptDesc:ko.observable()

};

ko.applyBindings(myViewModelJson);

$(function () {

$.ajax({

url: "/Home/GetData",

type: "get",

data: {},

success: function (data, status) {

ko.mapping.fromJS(data, myViewModelJson)

}

});

});

[复制代码](javascript:void(0);)

**在success里面，根据data的值去更新myViewModelJson这个viewmodel。**

**5.2、ViewModel转换成JSON对象**

 上面说了JSON对象转化为viewmodel，那么反过来，如果我们需要将viewmodel转换为Json对象传递到后端，怎么办呢？

knockout里面提供了两个方法：

* ko.toJS()：将viewmodel转换为JSON对象
* ko.toJSON()：将viewmodel转换为序列化过的Json string。

比如我们的代码如下：

[复制代码](javascript:void(0);)

　　　　 $(function () {

var oJson1 = ko.toJS(myViewModelJson);

var oJson2 = ko.toJSON(myViewModelJson);

});

var myViewModelJson = {

DeptName: ko.observable("研发部"),

DeptLevel: ko.observable("2"),

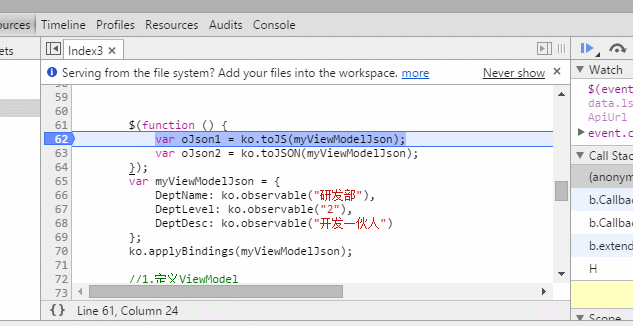
DeptDesc: ko.observable("开发一伙人")

};

ko.applyBindings(myViewModelJson);

[复制代码](javascript:void(0);)

那么我们来监控下oJson1和oJson2的值：



**代码释疑：通过上面这张图，很容易理解两个方法的区别，这里需要说明一点的是，这两个方法是ko自带的，并不需要mapping组件的支持。**

[回到顶部](http://www.cnblogs.com/landeanfen/p/5400654.html#_labelTop)

**6、创建自己的data-bind属性**

上面讲了那么多，都是介绍knockout里面的一些绑定和监控，那么，有些时候，我们需要自定义我们的data-bind，型如： <label data-bind="myBind:Name"></label> ，这种需求再做组件封装的时候尤其有用，是否可以实现呢？当然可以。

在knockout里面，提供了ko.bindingHandlers属性，用于自定义data-bind属性，它的语法如下：

[复制代码](javascript:void(0);)

ko.bindingHandlers.MySelect = {

init: function (element, valueAccessor, allBindingsAccessor, viewModel) {

},

update: function (element, valueAccessor, allBindingsAccessor, viewModel) {

}

};

[复制代码](javascript:void(0);)

就这样申明一下，然后在我们的html标签里面就可以使用自定义data-bind了。

[复制代码](javascript:void(0);)

<div>

<select data-bind="MySelect:$root">

<option id="1">研发部</option>

<option id="2">人事部</option>

<option id="3">行政部</option>

</select>

</div>

[复制代码](javascript:void(0);)

MySelect就是我们自定义的绑定属性，$root暂且可以理解为初始化（虽然这样解释并不严谨，如果有更加合理的解释欢迎指正）。

**代码释疑：通过上面的ko.bindingHandlers就能简单实现自定绑定属性，需要说明两点：**

* **init，顾名思义初始化自定义绑定，它里面包含多个参数，一般使用较多的是前两个参数，第一个参数表示初始化自定义绑定的DOM元素，第二个参数一般用来传递初始化的参数。**
* **update，更新回调，当对应的监控属性变化时，会进入到这个方法。如果不需要回调，此方法可以不声明。**

 在此博主就结合原来分享过的一个下拉框组件MutiSelect来简单说明下自定义绑定的使用。

**6.1、最简单的MutiSelect**

一般情况下，如果我们需要使用ko去封装一些通用组件，就需要用到我们的ko.bindingHandlers，下面博主就结合MutiSelect组件来说说如何使用。

首先声明自定义的ko.bindingHandlers，在init方法里面初始化我们的select标签

[复制代码](javascript:void(0);)

ko.bindingHandlers.MySelect = {

init: function (element, valueAccessor, allBindingsAccessor, viewModel) {

$(element).multiselect();

},

update: function (element, valueAccessor, allBindingsAccessor, viewModel) {

}

};

[复制代码](javascript:void(0);)

然后在页面标签里面使用

[复制代码](javascript:void(0);)

<div style="text-align:center;">

<select data-bind="MySelect:$root">

<option id="1">研发部</option>

<option id="2">人事部</option>

<option id="3">行政部</option>

</select>

</div>

[复制代码](javascript:void(0);)

最后第三部，激活绑定

$(function () {

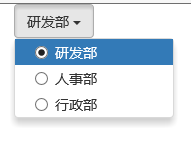
var MultiSelect = {};

ko.applyBindings(MultiSelect);

});

如果不需要传递参数，这里只需要绑定一个空的viewmodel即可。有人疑惑了，第三部感觉没啥实际意义呢。博主的理解是，DOM元素需要使用data-bind去绑定数据，必须要启用ko的绑定，也就是这里的ko.applyBindings()。

得到效果：



**6.2、参数传递**

 第一步还是自定义ko.bindingHandlers

[复制代码](javascript:void(0);)

ko.bindingHandlers.MySelect = {

init: function (element, valueAccessor, allBindingsAccessor, viewModel) {

var oParam = valueAccessor();

$(element).multiselect(oParam);

},

update: function (element, valueAccessor, allBindingsAccessor, viewModel) {

}

};

[复制代码](javascript:void(0);)

第二步和上面相同，在html标签里面使用这个自定义绑定。

第三步，在激活绑定的时候传入参数

[复制代码](javascript:void(0);)

$(function () {

var MultiSelect = {

enableClickableOptGroups: true,//收起分组

onChange: function (option, checked) {

alert("选择改变");

}

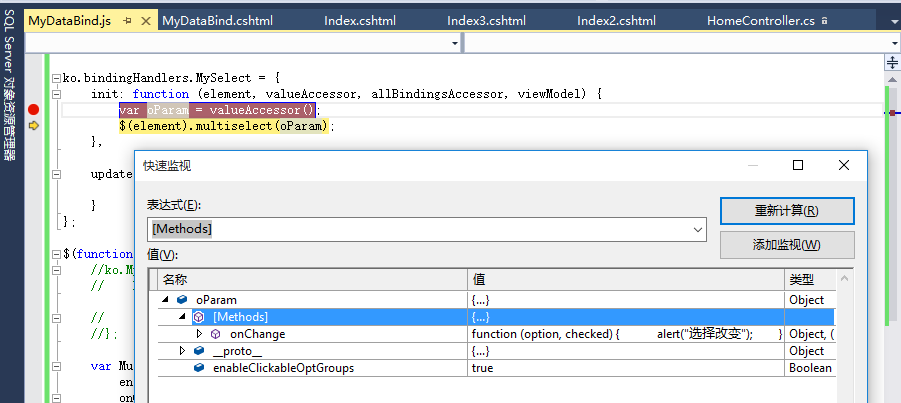
};

ko.applyBindings(MultiSelect);

});

[复制代码](javascript:void(0);)

通过这三步即可将参数传到我们的MutiSelect的初始化里面：



**代码释疑：init事件的第二个参数，我们说了，它主要作用是获取我们viewmodel里面传过来的参数，只不过这里要把它当做方法使用，为什么会这么用，还有待研究！**

[回到顶部](http://www.cnblogs.com/landeanfen/p/5400654.html#_labelTop)

**二、第一个增删改查实例**

至此基础的东西终于是铺垫完了，本来打算一篇搞定的，可以没料到基础的东西展开来这么多篇幅！增删改查的示例放到下篇吧。欢迎学习交流，当然也欢迎**推荐**！

PS：下篇已经完成，有需要的可以看看：[JS组件系列——BootstrapTable+KnockoutJS实现增删改查解决方案（二）](http://www.cnblogs.com/landeanfen/p/5407221.html)

# <http://www.cnblogs.com/landeanfen/p/5407221.html>

# [JS组件系列——BootstrapTable+KnockoutJS实现增删改查解决方案（二）](http://www.cnblogs.com/landeanfen/p/5407221.html)

**阅读目录**

* [一、效果预览](http://www.cnblogs.com/landeanfen/p/5407221.html#_label0)
* [二、代码示例](http://www.cnblogs.com/landeanfen/p/5407221.html#_label1)
  + [1、表格初始化](http://www.cnblogs.com/landeanfen/p/5407221.html#_label1_0)
  + [2、按钮操作](http://www.cnblogs.com/landeanfen/p/5407221.html#_label1_1)
* [三、总结](http://www.cnblogs.com/landeanfen/p/5407221.html#_label2)

**正文**

前言：上篇 [JS组件系列——BootstrapTable+KnockoutJS实现增删改查解决方案（一）](http://www.cnblogs.com/landeanfen/p/5400654.html) 介绍了下knockout.js的一些基础用法，由于篇幅的关系，所以只能分成两篇，望见谅！昨天就觉得应该快点完成下篇，要不然有点标题党的感觉，思及此，博主心有不安，于是加班赶出了下篇。如果你也打算用ko去做项目，且看看吧！

[回到顶部](http://www.cnblogs.com/landeanfen/p/5407221.html#_labelTop)

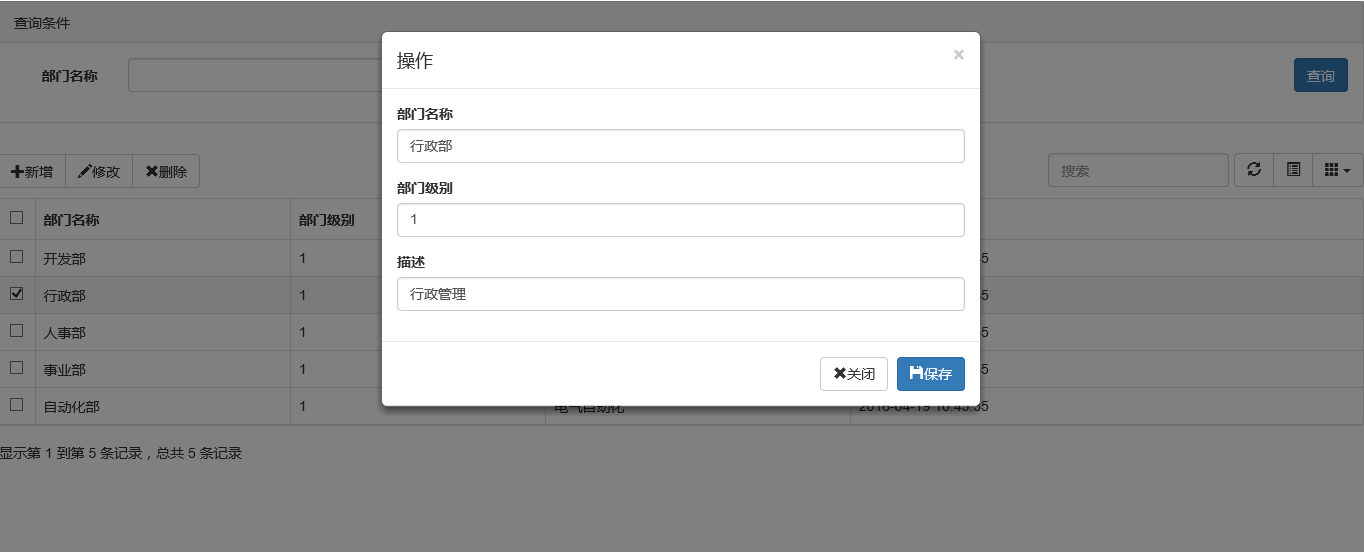
**一、效果预览**

其实也没啥效果，就是简单的增删改查，重点还是在代码上面，使用ko能够大量节省界面DOM数据绑定的操作。下面是整个整个增删改查逻辑的js代码：



页面效果：





[回到顶部](http://www.cnblogs.com/landeanfen/p/5407221.html#_labelTop)

**二、代码示例**

好了，进入重点吧！博主打算分两块介绍，第一部分是表格初始化部分，第二部分是按钮操作增删改部分。

[回到顶部](http://www.cnblogs.com/landeanfen/p/5407221.html#_labelTop)

**1、表格初始化**

**1.1、准备工作**

首先看看需要引用的js和css文件

[复制代码](javascript:void(0);)

<link href="~/Content/bootstrap/css/bootstrap.min.css" rel="stylesheet" />

<link href="~/Content/bootstrap-table/bootstrap-table.min.css" rel="stylesheet" />

<script src="~/scripts/jquery-1.9.1.min.js"></script>

<script src="~/Content/bootstrap/js/bootstrap.min.js"></script>

<script src="~/Content/bootstrap-table/bootstrap-table.min.js"></script>

<script src="~/Content/bootstrap-table/locale/bootstrap-table-zh-CN.js"></script>

<script src="~/scripts/knockout/knockout-3.4.0.min.js"></script>

<script src="~/scripts/knockout/extensions/knockout.mapping-latest.js"></script>

<script src="~/Content/bootstrap-table/knockout.bootstraptable.js"></script>

<script src="~/scripts/Department.js"></script>

[复制代码](javascript:void(0);)

都是一些常用的css和js文件，我们自定义的js文件主要有两个: knockout.bootstraptable.js 和 Department.js 。上篇我们介绍过使用ko可以自定义我们的data-bind。同样，这里对于table的绑定，我们也定义一个自定义的绑定，代码 knockout.bootstraptable.js 里面。

[复制代码](javascript:void(0);)

//添加ko自定义绑定

ko.bindingHandlers.myBootstrapTable = {

init: function (element, valueAccessor, allBindingsAccessor, viewModel) {

//这里的oParam就是绑定的viewmodel

var oViewModel = valueAccessor();

var $ele = $(element).bootstrapTable(oViewModel.params);

//给viewmodel添加bootstrapTable方法

oViewModel.bootstrapTable = function () {

return $ele.bootstrapTable.apply($ele, arguments);

}

},

update: function (element, valueAccessor, allBindingsAccessor, viewModel) {}

};

//初始化

(function ($) {

//向ko里面新增一个bootstrapTableViewModel方法

ko.bootstrapTableViewModel = function (options) {

var that = this;

this.default = {

search: true, //是否显示表格搜索，此搜索是客户端搜索，不会进服务端，所以，个人感觉意义不大

strictSearch: true,

showColumns: true, //是否显示所有的列

cache:false,

showRefresh: true, //是否显示刷新按钮

minimumCountColumns: 2, //最少允许的列数

clickToSelect: true, //是否启用点击选中行

showToggle: true,

};

this.params = $.extend({}, this.default, options || {});

//得到选中的记录

this.getSelections = function () {

var arrRes = that.bootstrapTable("getSelections")

return arrRes;

};

//刷新

this.refresh = function () {

that.bootstrapTable("refresh");

};

};

})(jQuery);

[复制代码](javascript:void(0);)

**代码释疑：这个js文件主要做了两件事**

1. **自定义data-bind属性myBootstrapTable。对于ko.bindingHandlers.myBootstrapTable里面的update方法，如非必须，可以不用定义。**
2. **通过向ko对象里面添加bootstrapTableViewModel来封装bootstrapTable。**

**1.2、html标签启动绑定**

[复制代码](javascript:void(0);)

<table id="tb\_dept" data-bind="myBootstrapTable:$root">

<thead>

<tr>

<th data-checkbox="true"></th>

<th data-field="Name">部门名称</th>

<th data-field="Level">部门级别</th>

<th data-field="Des">描述</th>

<th data-field="strCreatetime">创建时间</th>

</tr>

</thead>

</table>

[复制代码](javascript:void(0);)

**代码释疑：定义一个table标签，使用自定义绑定myBootstrapTable，上篇说过，$root可以理解为初始化的意思。为了简单，所有的colums就直接在<th>里面写了。**

**1.3、激活ko的绑定**

在页面加载完成之后，启动ko的绑定：

[复制代码](javascript:void(0);)

//初始化

$(function () {

//1、初始化表格

tableInit.Init();

//2、注册增删改事件

operate.operateInit();

});

//初始化表格

var tableInit = {

Init: function () {

//绑定table的viewmodel

this.myViewModel = new ko.bootstrapTableViewModel({

url: '/Department/GetDepartment', //请求后台的URL（\*）

method: 'get', //请求方式（\*）

toolbar: '#toolbar', //工具按钮用哪个容器

queryParams: function (param) {

return { limit: param.limit, offset: param.offset };

},//传递参数（\*）

pagination: true, //是否显示分页（\*）

sidePagination: "server", //分页方式：client客户端分页，server服务端分页（\*）

pageNumber: 1, //初始化加载第一页，默认第一页

pageSize: 10, //每页的记录行数（\*）

pageList: [10, 25, 50, 100], //可供选择的每页的行数（\*）

});

ko.applyBindings(this.myViewModel, document.getElementById("tb\_dept"));

}

};

[复制代码](javascript:void(0);)

**代码释疑：页面加载完成之后，调用上面封装的bootstrapTableViewModel对象合并传递的参数，最后激活绑定，将this.myViewModel作为绑定的viewmodel激活。调试代码可知，当执行到 ko.applyBindings(this.myViewModel, document.getElementById("tb\_dept")); 这一句的时候，自定义绑定才会生效，程序才会进入到 ko.bindingHandlers.myBootstrapTable 对象的init方法去初始化bootstrapTable。这里需要说明一点：**

[复制代码](javascript:void(0);)

init: function (element, valueAccessor, allBindingsAccessor, viewModel) {

//这里的oParam就是绑定的viewmodel

var oViewModel = valueAccessor();

var $ele = $(element).bootstrapTable(oViewModel.params);

//给viewmodel添加bootstrapTable方法

oViewModel.bootstrapTable = function () {

return $ele.bootstrapTable.apply($ele, arguments);

}

}

[复制代码](javascript:void(0);)

**上文中的init方法，通过第二个参数valueAccessor，我们得到的是当前绑定的viewmodel，也就是我们上面的this.myViewModel这个对象，博主觉得这一点有利于你理解自定义绑定的逻辑。基本上执行到 var $ele = $(element).bootstrapTable(oViewModel.params); 这一句的时候，我们表格的初始化就完成了。后台对应的方法博主随便定义了一个集合，为了完整，这里还是贴出来：**

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifDepartmentController

[回到顶部](http://www.cnblogs.com/landeanfen/p/5407221.html#_labelTop)

**2、按钮操作**

上面通过bootstrapTable的初始化完成了我们的自定义data-bind的使用。下面的按钮操作我们来体验一把使用监控属性的“爽歪歪”。

**2.1、view页面**

首先在view页面上面定义我们的增删改按钮

[复制代码](javascript:void(0);)

<div id="toolbar" class="btn-group">

<button id="btn\_add" type="button" class="btn btn-default">

<span class="glyphicon glyphicon-plus" aria-hidden="true"></span>新增

</button>

<button id="btn\_edit" type="button" class="btn btn-default">

<span class="glyphicon glyphicon-pencil" aria-hidden="true"></span>修改

</button>

<button id="btn\_delete" type="button" class="btn btn-default">

<span class="glyphicon glyphicon-remove" aria-hidden="true"></span>删除

</button>

</div>

[复制代码](javascript:void(0);)

为了简便，博主使用了一个隐藏的弹出框用来包含新增和编辑的文本框。当然，一般情况下，可能这里用的是部分视图，你的项目里面可能会有一个Edit.cshtml，但这里博主将这些都放在一个页面上面，因为这不是文本的重点。

[复制代码](javascript:void(0);)

<div class="modal fade" id="myModal" tabindex="-1" role="dialog" aria-labelledby="myModalLabel">

<div class="modal-dialog" role="document">

<div class="modal-content">

<div class="modal-header">

<button type="button" class="close" data-dismiss="modal" aria-label="Close"><span aria-hidden="true">&times;</span></button>

<h4 class="modal-title" id="myModalLabel">操作</h4>

</div>

<div class="modal-body">

<div class="form-group">

<label for="txt\_departmentname">部门名称</label>

<input type="text" name="txt\_departmentname" data-bind="value:Name" class="form-control" id="txt\_departmentname" placeholder="部门名称">

</div>

<div class="form-group">

<label for="txt\_departmentlevel">部门级别</label>

<input type="text" name="txt\_departmentlevel" data-bind="value:Level" class="form-control" id="txt\_departmentlevel" placeholder="部门级别">

</div>

<div class="form-group">

<label for="txt\_des">描述</label>

<input type="text" name="txt\_des" data-bind="value:Des" class="form-control" id="txt\_des" placeholder="描述">

</div>

</div>

<div class="modal-footer">

<button type="button" class="btn btn-default" data-dismiss="modal"><span class="glyphicon glyphicon-remove" aria-hidden="true"></span>关闭</button>

<button type="button" id="btn\_submit" class="btn btn-primary" data-dismiss="modal"><span class="glyphicon glyphicon-floppy-disk" aria-hidden="true"></span>保存</button>

</div>

</div>

</div>

</div>

[复制代码](javascript:void(0);)

**2.2、JS初始化按钮操作**

[复制代码](javascript:void(0);)

//操作

var operate = {

//初始化按钮事件

operateInit: function () {

this.operateAdd();

this.operateUpdate();

this.operateDelete();

this.DepartmentModel = {

id: ko.observable(),

Name: ko.observable(),

Level: ko.observable(),

Des: ko.observable(),

CreateTime: ko.observable()

};

},

//新增

operateAdd: function(){

$('#btn\_add').on("click", function () {

$("#myModal").modal().on("shown.bs.modal", function () {

var oEmptyModel = {

id: ko.observable(),

Name: ko.observable(),

Level: ko.observable(),

Des: ko.observable(),

CreateTime: ko.observable()

};

ko.utils.extend(operate.DepartmentModel, oEmptyModel);

ko.applyBindings(operate.DepartmentModel, document.getElementById("myModal"));

operate.operateSave();

}).on('hidden.bs.modal', function () {

ko.cleanNode(document.getElementById("myModal"));

});

});

},

//编辑

operateUpdate: function () {

$('#btn\_edit').on("click", function () {

$("#myModal").modal().on("shown.bs.modal", function () {

var arrselectedData = tableInit.myViewModel.getSelections();

if (!operate.operateCheck(arrselectedData)) { return; }

//将选中该行数据有数据Model通过Mapping组件转换为viewmodel

ko.utils.extend(operate.DepartmentModel, ko.mapping.fromJS(arrselectedData[0]));

ko.applyBindings(operate.DepartmentModel, document.getElementById("myModal"));

operate.operateSave();

}).on('hidden.bs.modal', function () {

//关闭弹出框的时候清除绑定(这个清空包括清空绑定和清空注册事件)

ko.cleanNode(document.getElementById("myModal"));

});

});

},

//删除

operateDelete: function () {

$('#btn\_delete').on("click", function () {

var arrselectedData = tableInit.myViewModel.getSelections();

$.ajax({

url: "/Department/Delete",

type: "post",

contentType: 'application/json',

data: JSON.stringify(arrselectedData),

success: function (data, status) {

alert(status);

//tableInit.myViewModel.refresh();

}

});

});

},

//保存数据

operateSave: function () {

$('#btn\_submit').on("click", function () {

//取到当前的viewmodel

var oViewModel = operate.DepartmentModel;

//将Viewmodel转换为数据model

var oDataModel = ko.toJS(oViewModel);var funcName = oDataModel.id?"Update":"Add";

$.ajax({

url: "/Department/"+funcName,

type: "post",

data: oDataModel,

success: function (data, status) {

alert(status);

tableInit.myViewModel.refresh();

}

});

});

},

//数据校验

operateCheck:function(arr){

if (arr.length <= 0) {

alert("请至少选择一行数据");

return false;

}

if (arr.length > 1) {

alert("只能编辑一行数据");

return false;

}

return true;

}

}

[复制代码](javascript:void(0);)

**代码释疑：说说这里的执行逻辑，首先在$(function(){})方法里面调用 operate.operateInit(); 。在operateInit()方法里面注册页面上面按钮的点击事件，同时也定义 this.DepartmentModel 作为我们新增编辑的viewmodel，这个viewmodel里面定义了和页面元素对应的监控属性。还记得上面隐藏的弹出框里面的一些data-bind吗，没错，里面对应的value值就是和这里的监控属性对应，这样设置绑定之后，js里面所有的导致 this.DepartmentModel 里面监控的变化，都会触发界面上面这些绑定标签的value值变化，反之，界面上面的所有标签的Value值的变化，也势必会引起它的监控属性值的变化，此之所谓双向绑定。下面具体看看双向绑定的执行。**

**2.3、新增操作**

[复制代码](javascript:void(0);)

　　　　$('#btn\_add').on("click", function () {

$("#myModal").modal().on("shown.bs.modal", function () {

var oEmptyModel = {

id: ko.observable(),

Name: ko.observable(),

Level: ko.observable(),

Des: ko.observable(),

CreateTime: ko.observable()

};

ko.utils.extend(operate.DepartmentModel, oEmptyModel);

ko.applyBindings(operate.DepartmentModel, document.getElementById("myModal"));

operate.operateSave();

}).on('hidden.bs.modal', function () {

ko.cleanNode(document.getElementById("myModal"));

});

});

[复制代码](javascript:void(0);)

当我们界面触发新增操作的时候，首先会弹出上面说的隐藏模态框。在模态框显示的时候，首先定义一个空的viewmodel，然后调用 ko.utils.extend(operate.DepartmentModel, oEmptyModel); 这一句，将全局的operate.DepartmentModel被空的viewmodel覆盖。ko.utils.extend()这个方法的作用和jquery里面的$.extend()作用类似，都是根据后面对象合并前面对象，合并之后，使用新的viewmodel激活绑定。激活绑定之后，注册保存按钮的click事件。这样新增的时候，弹出模态框，由于viewmodel里面的监控属性都是空的，对应界面元素的value也会被清空，所以新增我们看到是这样：



当弹出框关闭后，我们通过关闭的事件，执行 ko.cleanNode(document.getElementById("myModal")); 这一句，这个很重要，因为对于同一个dom，ko只能绑定一次，如果需要再次绑定，需要先清空绑定，并且cleanNode()这个方法，它不仅会清空绑定，还是会dom里面注册的事件也会清空，使用的时候需要注意下！

**2.4、编辑操作**

[复制代码](javascript:void(0);)

　　　　$('#btn\_edit').on("click", function () {

$("#myModal").modal().on("shown.bs.modal", function () {

var arrselectedData = tableInit.myViewModel.getSelections();

if (!operate.operateCheck(arrselectedData)) { return; }

//将选中该行数据有数据Model通过Mapping组件转换为viewmodel

ko.utils.extend(operate.DepartmentModel, ko.mapping.fromJS(arrselectedData[0]));

ko.applyBindings(operate.DepartmentModel, document.getElementById("myModal"));

operate.operateSave();

}).on('hidden.bs.modal', function () {

//关闭弹出框的时候清除绑定(这个清空包括清空绑定和清空注册事件)

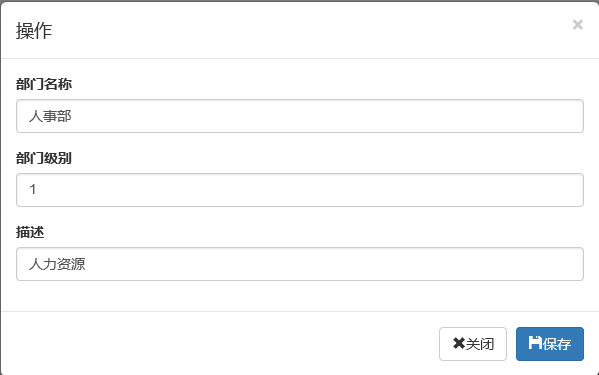
ko.cleanNode(document.getElementById("myModal"));

});

});

[复制代码](javascript:void(0);)

当我们触发编辑操作的时候，界面还是弹出框。在弹出框的弹出事件里面，我们取到当前选中的行，然后校验是否选中了一行。最好通过 ko.mapping.fromJS(arrselectedData[0]) 这一句，将普通的Json对象转换为带有监控属性的viewmodel，上篇说过，这个方法需要 knockout.mapping-latest.js 这个js文件的支持。转换之后，还是通过ko.utils.extend()方法更新viewmodel，然后激活绑定。由于viewmodel被当前选中行的数据更新了，所以得到结果：



**2.5、保存操作**

在新增和编辑弹出框之后，修改相关信息后点击保存，就会触发保存事件。

[复制代码](javascript:void(0);)

$('#btn\_submit').on("click", function () {

//取到当前的viewmodel

var oViewModel = operate.DepartmentModel;

//将Viewmodel转换为数据model

var oDataModel = ko.toJS(oViewModel);

var funcName = oDataModel.id?"Update":"Add";

$.ajax({

url: "/Department/"+funcName,

type: "post",

data: oDataModel,

success: function (data, status) {

alert(status);

tableInit.myViewModel.refresh();

}

});

});

[复制代码](javascript:void(0);)

当触发保存事件的时候，我们首先取到页面绑定的viewmodel，即operate.DepartmentModel，然后使用ko.toJS()方法将带有监控属性的viewmodel转换为纯数据的Json对象，这个方法是ko内置的，不需要其他js支持。得到json对象之后，发送ajax请求，去新增或者编辑数据。这样就很好地体现了双向绑定，界面上面所有文本框的value发生了变化之后，也会触发operate.DepartmentModel的变化。

**2.6、删除操作**

删除操作没什么好说的，和ko关系不大。

[回到顶部](http://www.cnblogs.com/landeanfen/p/5407221.html#_labelTop)

**三、总结**

以上通过一个简单的增删改查操作，介绍了下ko和bootstrapTable的联合使用。ko可以让你从DOM中解放出来，把关注点放在viewmodel上面。纵观整个js代码，几乎看不到jquery的val()、text()等对界面dom做取值和赋值的操作，是不是看着干净清爽，并且高大上了呢~~当然，这或许只是ko的一些比较基础的用法，毕竟博主学习ko才3天，更多高级用法还有待摸索，等过段时间用熟了，再将它的一些高级用法分享给大家。如果你觉得本文能够帮助你理解ko的原理以及它的一些用法，不妨**推荐**下，博主一定继续努力！