dojo学习

**（1）加载**

①首先，先设置

<script>

//替代使用data-dojo-config，我们创建一个dojoConfig对象（是个设置）在我们调用dojo.js之前，他们功能相同

//这比通过一大堆设置来说，更易阅读

var dojoConfig = {

async: true,

//这个代码注册了demo包的正确位置，于是我们可以调用dojo从CDN，并且同时还能加载本地模块

packages: [{

name: "demo",

location: location.pathname.replace(/\/[^/]\*$/, '') + '/demo'

}]

};

</script>

②要加载dojo.js文件：

<scriptsrc="dojo/dojo.js"></script>

③最后，另开一个script，加载自己需要的插件，加载方式为使用：

require ( [插件], 回调函数);

插件的值用单引号包括，用逗号分隔不同插件，都放在方括号里；

回调函数可以直接输入函数function(参数)，参数填写插件名，如果多个插件，用逗号分隔。

如例：

<script>

require([ //这行表示要加载东西了，用哪个加载哪个||这些直接映射到文件名，如果你下载的Dojo源分布 ，并期待在dojo目录，你会看到dom.js和dom-construct.js文件，这些文件定义这些模块。

'dojo/dom', //加载dojo文件夹下的dom插件（应该）

'dojo/fx', //加载fx插件，这个插件包括一个滑动动画（slideTo），还有其他

'dojo/domReady!' //这个插件大概作用是，保证加载完毕后才执行，以免执行一个没加载好的。

], //这行已经加载结束了

function (dom,fx) { //这是一个回调函数（即加载完才执行的），参数有dom和fx（这个应该指的是插件）

var greeting = dom.byId('greeting'); //得到id为greeting的元素，赋值给greeting这个变量

greeting.innerHTML += ' from Dojo!'; //这个元素加入文本（+=）

//动画部分，

fx.slideTo({

node: greeting, //不懂

top: 100,

left: 500,

}).play();

} //这行是函数完

); //到这部分是require完

</script>

**关于插件的加载：**

**回调函数中的顺序，是按照插件的顺序填写的。否则就很可能会插件加载出错！！而且这个bug莫名其妙！！！！！！！**

**（2）移动动画**

//动画部分，

fx.slideTo({

node: greeting, //不懂

top: 100,

left: 500,

}).play();

第一个参数是移动后的情况。但node:greeting指对id为greeting的元素进行操纵

这是一个动画部分，他是回调函数内容的一部分，但显然，单独拿出来也是可以用的。

如：

<script>

require([ //这行表示要加载东西了，用哪个加载哪个||这些直接映射到文件名，如果你下载的Dojo源分布 ，并期待在dojo目录，你会看到dom.js和dom-construct.js文件，这些文件定义这些模块。

'dojo/dom', //加载dojo文件夹下的dom插件（应该）

'dojo/fx', //加载fx插件，这个插件包括一个滑动动画（slideTo），还有其他

'dojo/domReady!' //这个插件大概作用是，保证加载完毕后才执行，以免执行一个没加载好的。

], //这行已经加载结束了

function (dom,fx) { //这是一个回调函数（即加载完才执行的），参数有dom和fx（这个应该指的是插件）

ST(dom,fx);} //调用ST函数，ST函数在下面定义

); //到这部分是require完

</script>

<script>

function ST(dom,fx){

var greeting = dom.byId("greeting"); //得到id为greeting的元素，赋值给greeting这个变量

greeting.innerHTML += " from Dojo!"; //这个元素加入文本（+=）

//动画部分，

fx.slideTo({

node: greeting, //不懂

top: 100,

left: 500,

}).play();

} //这行是ST函数完

</script>

**（3）变宽变窄动画**

原型：（但需要在之前使用animateProperty=dom.byId("animateproperty");

baseFx.animateProperty({

node: dom.byId("anim8target"),

properties: { borderWidth: 100 }

}).play();

});

效果是获得id为anim8target的东西为目标（node应该是操作目标的意思），

然后properties是把他变成什么样子（borderWidth应该是外边框），

.play() 应该是动画的意思。

代码：

<script>

require([ //第一个参数，加载插件

"dojo/\_base/fx",

"dojo/on",

"dojo/dom",

"dojo/domReady!"

],

function(baseFx,on,dom) { //第二个参数，回调函数

var startButton1=dom.byId("startButton"), //得到id为startButton的元素

reverseButton1=dom.byId("reverseButton"), //得到reverseButton的元素

animateProperty=dom.byId("animateproperty"); //得到animaterproperty的元素（应该是动画函数吧？），实测证明，这行如果被删除（并且上一行改成冒号），依然可以被运行 //这行如果前面的变量名被修改（下面不被修改），依然可以正常运行（不明白为什么）

on(startButton1,"click",function(){ //on应该是当...的时候，比如放这里，应该是某元素（上面赋值的变量）当被按下的时候，调用函数后面的，但evt参数不知道有什么用（不加也没影响效果）

baseFx.animateProperty({ //这行应该是说baseFx插件的animateProperty函数（动画函数），然后{}内应该是变化后的状态，其中node指操纵的对象，properties指变成什么样子

node:anim8target, //这里的是操纵对象

properties:{borderWidth:100} //properties操纵的是css属性

}).play(); //操纵速度不在play

});

on(reverseButton1,"click",function(){

baseFx.animateProperty({

node:anim8target,

properties:{borderWidth:1}

}).play();

});

});

</script>

注：

①borderWidth是外边框；如果是width则是宽度，height是高度（不会导致内部字体变形）

②操纵的目标是：

<button id="startButton">变大</button>

<button id="reverseButton">变小</button>

<div id="anim8target" class="box" style="border-style:outset">

<div class="innerBox">A box</div>

</div>

③**修饰这个目标有css属性（不然外形会是默认外形，很大），可以认为是初始属性**

<style type="text/css"> /\*设置盒子的样式\*/

.innerBox {

margin: 5%;

padding: 5px;

background-color: white;

}

.box {

position: absolute;

height: 200px; /\*这里必须加px\*/

width: 200px; /\*这两个是外框的高和宽\*/

background-color: #ddd;

border: 1px #eee;

border-top-width: 1px;

border-right-width: 1px;

border-bottom-width: 1px;

border-left-width: 1px;

border-top-color: rgb(238, 238, 238);

border-right-color: rgb(238, 238, 238);

border-bottom-color: rgb(238, 238, 238);

border-left-color: rgb(238, 238, 238);

-moz-border-top-colors: none;

-moz-border-right-colors: none;

-moz-border-bottom-colors: none;

-moz-border-left-colors: none;

border-image-source: none;

border-image-slice: 100% 100% 100% 100%;

border-image-width: 1 1 1 1;

border-image-outset: 0 0 0 0;

border-image-repeat: stretch stretch;

padding: 5px;

padding-top: 5px; /\*这里的边距指外框到内框的，上边距\*/

padding-right: 5px; /\*右边距\*/

padding-bottom: 5px;/\*下边距\*/

padding-left: 5px;/\*左边距\*/

}

</style>

事实上应该不用这么多

④效果是外框左上是淡灰色，右下是深灰色。

**（4）设置动画的开始和结束情况：**

如代码：

<script>

on(startButton1, "click", function () { //on应该是当...的时候，比如放这里，应该是某元素（上面赋值的变量）当被按下的时候，调用函数后面的，但evt参数不知道有什么用（不加也没影响效果）

baseFx.animateProperty({ //这行应该是说baseFx插件的animateProperty函数（动画函数），然后{}内应该是变化后的状态，其中node指操纵的对象，properties指变成什么样子

node: anim8target, //这里的是操纵对象

properties: {

height: { end: 400, start: 100 }, //高度变化，初始是100（不受之前情况影响），结束是400

width: { end: 400, start: 100 }

}

//width:{end:400,start:100} //properties操纵的是css属性

}).play(); //感觉play应该能操纵速率，但不知道怎么用

});

</script>

注：

①这个还可以用该对颜色进行变化。

**（5）透明度与动画速度**

如代码：

on(startButton1, "click", function () { //on应该是当...的时候，比如放这里，应该是某元素（上面赋值的变量）当被按下的时候，调用函数后面的，但evt参数不知道有什么用（不加也没影响效果）

baseFx.animateProperty({ //这行应该是说baseFx插件的animateProperty函数（动画函数），然后{}内应该是变化后的状态，其中node指操纵的对象，properties指变成什么样子

node: anim8target, //这里的是操纵对象

properties: {

height: { end: 400, start: 100 }, //高度变化，初始是100（不受之前情况影响），结束是400

width: { end: 400, start: 100 },

opacity: { start: 1, end: 0.3 } //opacity是透明度，最大1（不透明），最小0（透明）

},

duration: 3000 //这个是动画速度

}).play(); //感觉play应该能操纵速率，但不知道怎么用

});

透明度是opacity，

速度（变化所用时间）是properties函数的第二个参数（第一个参数是设置动画变换后的属性）

**（6）动画过程的更多种形式**

简单来说，除了点到点的动画之外，还有更多的可能。

而这些更多的可能，其位于插件dojo/fx/easing之中

插件需要：

dojo/fx/easing

dojo/\_base/fx

例如模拟球体掉落，需要使用插件dojo/fx/easing，类型为（在animateProperty的第一个参数中放置）：easing: easing.bounceOut ，

如：

<script>

require(["dojo/\_base/fx",

"dojo/dom",

"dojo/fx/easing", //这个插件用于加载更多的动画过程

"dojo/window", //一个新的插件，应该是跟窗口相关的

"dojo/on",

"dojo/domReady!"], function (baseFx, dom, easing, win, on) { //5个参数（5个插件）

var startButton = dom.byId("startButton"), //开始按钮

reverseButton = dom.byId("reverseButton"), //另一个按钮

anim8target = dom.byId("anim8target"); //操作对象

// 设置一对按钮用于处理事件

on(startButton, "click", function (evt) { //开始按钮在点击后触发

// 得到我们窗口的尺寸

var viewport = win.getBox(win.doc); //变量被用于存储窗口的信息

baseFx.animateProperty({

easing: easing.bounceOut, //动画，向下加速，会触底4次后停止（类似球体从空中掉落的动画形式）

duration: 3500,

node: anim8target,

properties: {

// calculate the 'floor' 计算地板

// 然后减去东西的高度

top: { start: 0, end: viewport.h - anim8target.offsetHeight } //top（距离上面的距离为0，

//结束时，距离上面的距离是窗口高度（viewport.h）减去当前块的高度（ .offsetHeight）

}

}).play();

});

on(reverseButton, "click", function (evt) {

baseFx.animateProperty({

node: anim8target,

easing: easing.bounceOut,

properties: { top: 0 }, //飞到最高

duration: 2000

}).play();

});

});

</script>

效果是：

①按钮1：摔下去，弹3次，第4次碰到地面（通过top设定）时停止。

②按钮2：飞上去，弹3才，第4次碰到时停止；

注：

①也可以斜着移动然后弹（弹往来的方向）

②也可以对字体大小（fontSize）之类的有效（例如从小变大，会反弹几下再变成指定大小）

③对颜色有效：backgroundColor:{start:"#0F0",end:"#F0F"}

④对透明度有效opacity

⑤注意：需要使用**baseFx.animateProperty({**

**easing:easing.bounceOut,**

**node:目标,**

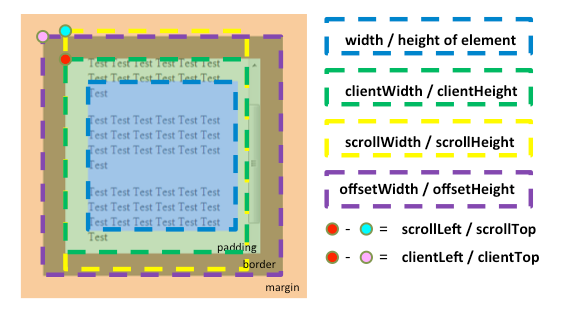
**其他修饰（比如要抵达的状态）,**

**})**

使用插件baseFx和插件easing

**（7）关于各种图像的各种：**

如图



offsetWidth的父为body

## [圖解offsetLeft、offsetTop、offsetWidth和offsetHeight](http://emn178.pixnet.net/blog/post/95297028)

<http://emn178.pixnet.net/blog/post/95297028-%E5%9C%96%E8%A7%A3offsetleft%E3%80%81offsettop%E3%80%81offsetwidth%E5%92%8Coffsetheight>

基本来说，最里面是text（文本区）；

文本区有底纹，称为填充区（padding）；

填充区外面有边框（border），边框比填充区大；

边框是元素的最外围了（应该），再外围是body标签范围。

**（8）如何在style里设置初始的top、left等**

办法是，例如：style=”position:absolute;left:100px”

即，必须先有：**position:absolute**或者是fixed或者是其他（但我不知道这有什么用）

后面的设置才是有用的，否则是无用的。

**（9）读取当前的属性**

需要插件：dojo/dom-style，

函数需要传递参数：domStyle

使用命令：No\_one\_top = parseInt(domStyle.get(No\_one, "top"));

①第一个No\_one\_top表示存储这个值的变量；

②parseInt() 函数应该是要将值转换为数字；

③domStyle.get(No\_one,”top”)这个函数，表示读取No\_one这个的top属性；

关于第一个参数的解释：

①假如有：No\_one1 = dom.byId("No\_one"); //对象1在之前

②No\_one为id，加不加引号都能读取这个id（无论有没有①的存在）；

③不加引号，可以变更为No\_one1（有①的情况下），但加了引号的情况下，则不能读取；

④疑问：假如有两个ID，分别为No\_one和No\_two；

然后，变量No\_two= dom.byId("No\_one"); No\_one= dom.byId("No\_two")

那么调用No\_one作为domStyle.get的第一个参数时，调用的是ID为No\_one还是No\_two的？

**（10）将两个动画合在一起**

需要插件：dojo/fx

如代码：

**var mm=fx.chain([动画1, 动画2]);**

①里面需要明确是调用哪个参数（对哪个单位进行操作）；

②可以直接把动画函数写里面；

③mm为自定义变量名，调用时，使用mm.play()即可。

**（11）设置初始颜色**

需要插件：dojo/dom-style

格式：

**domStyle.set(目标对象, 修改的属性, 目标值);**

如：

**domStyle.set(container, "backgroundColor", "#eee");**

效果是：设置id为container，将其背景颜色（backgroundColor）设置为颜色（#eee）

**（12）在动画开始之前**

需要使用插件：dojo/aspect；

①假设有动画mm是正常执行的，假设有动画nn是预计要执行的。

②动画调用之前，使用aspect.before(mm, “beforeBegin”, nn);

③效果是，在mm动画调用前，将执行nn动画/函数，此时，相当于mm和nn一起执行（只不过其执行所需的时间可能不同）；

④但并非先执行完nn，才会执行mm

**（13）在动画结束之后**

需要插件：dojo/on

格式为：

on(对象, “End”, 回调函数)；

例如：

on(anim, "End", function(n1, n2){

var mm =

baseFx.animateProperty({

easing:easing.bounceOut,

duration:7000,

properties:{

backgroundColor:{start:"#FFF",end:"#000"},

},

node:container,

}); //mm定义完

mm.play();

}); // {是回调函数完，)是on完

**（14）分拆动画和执行动作**

分拆前：

baseFx.animateProperty({

easing:easing.bounceOut,

duration:7000,

properties:{

backgroundColor:{start:"#FFF",end:"#000"},

},

node:container,

}).play();

分拆后：

var mm =

baseFx.animateProperty({

easing:easing.bounceOut,

duration:7000,

properties:{

backgroundColor:{start:"#FFF",end:"#000"},

},

node:container,

}); //mm定义完

mm.play();

精简后；

baseFx.animateProperty({

执行的动作，比如达到的状态等

}).play();

在分拆后得到：

var mm = baseFx.animateProperty({

执行的动作，比如达到的状态等

});

mm.play();

即可。

理论上，对其他的也可行。实际上，也是可行的。

**（15）常见错误：**

①在baseFx.animateProperty({})函数的“}”结束处加分号；

②在properties:()的后面没有加逗号；

③在properties:()内部的属性，例如top:100后面加了px（正常是不能加的）；

④样式表名在不同地方不一样。例如：

背景颜色在css里是background-color，但在dojo里面是backgroundColor

⑤查看有没有多一个括号，少一个逗号、分号之类的错误

⑥on用于执行某个函数时，其回调函数假如是function abc()，那么不应该写abc()，而是写abc。（前者会导致直接执行，后者不会）

⑦注意()和{}之间的差别

⑧getElementById是没有s的（在Element后）；

getElementsByTagName是有s的（在Element后）；

**（16）dojo/dom插件**

①首先需要加载插件：dojo/dom

②其次需要在回调函数中使用：dom作为参数

③功能有：var mm=dom.byId(“某id”)

其效果类似document.getElementById(“某id”)

将其赋值给mm（自定义变量名），于是mm就可以替代这一长串的文字了。

**功能有（假如声明了上面的）：**

①mm.innnerHTML，效果相当于document.getElementById(“某id”).innerHTML

替换文字

如函数：

**function settext(node,text) //效果是给第一个参数的位置，用第二个参数的文字替代**

**{**

**node = dom.byId(node);**

**node.innerHTML = text;**

**}**

settext(no1,"aaaaaa");

在id为no1的地方，将文字更改为text

**（17）添加新标签**

①使用插件：dojo/dom-construct

②参数：domConstruct

③格式为：

domConsrtuct.create (“标签名”, 样式, 父标签);

如：

box = dom.byId("box");

no2 = dom.byId("no2");

domConstruct.create("div", //domConstruct.create开头，第一个div为标签

{ //第二个参数规定插入的各种东西

innerHTML:"six" //插入的文本内容

},box); //第三个参数为插入在哪个父标签下

domConstruct.create("div",

{

innerHTML:"seven",

className:"seven", //规定了插入的类型为seven，也可以是现有类型

style:{fontWeight:"bold"} //规定了style，这里字体宽度为bold

},box);

domConstruct.create("div",

{

innerHTML:"Three and a half",

className:"smallerbox2",

style:{backgroundColor:"#fff"}

},no1,"after"); //在no1标签之后，最好用after，不太明白这个机制，特别是使用了偏移后的坐标时。

**（18）常用功能（dojo用，与其他的语法可能不一样）：**

|  |  |
| --- | --- |
| 功能 | 格式 |
| 字体宽度 | fontWeight |
| 字体大小 | fontSize |
| 背景颜色 | backgroundColor |
| 类名 | className |
|  |  |

**（19）移动标签**

插件：dojo/dom, dojo/dom=construct

参数：dom, domConstruct

语法：

domConstruct.place(被移动的元素A, 目标元素B, 参数);

说明：

①前两个参数用id，或者是使用dom.byId(“目标标签的id”)的变量来替代；（但不要冲突）；

②被移动的元素就是指要移动的那个。

③目标元素指，和第一个元素互动的那个。可能的相关的，也可能是父元素。

④参数可以是

不填写：插在B的所有子块之后，父标签是B

“first”：注意，需要双引号。插在B的所有子块之前，父是B；

“before”：插在B块之前，其父是B的父；

“after”：插在B块之后，其父是B的父；（会导致原本在B后面的，往后移动一位。

**（20）利用get读取一个json文件**

插件：dojo/request

参数：request

语法：

request.get (“链接”, { handleAs: “json” }).then ( 回调函数 );

When handleAs is set to "json", dojo/request treats the response payload as JSON data and parses it into a JavaScript object.

当使用handleAs:"json"时，返回的东西将被作为json文件，内容被解析为一个JS对象。

**（21）memory**

插件：dojo/store/Memory

参数：Momery

语法：

不懂

**（22）关于更多动画：**

dojo/\_base/fx是提供了基础的动画，包括；**animateProperty**（这个好像就是动画？但是更具体的起了什么），**anim**（这个又起什么作用？），**fadeIn**（淡入）和**fadeOut**（淡出）；

**未 完**

**（23）AJAX异步加载**

插件：dojo/request

参数：request

语法（get）：

**request.get(URL).then(成功回调函数, 失败回调函数);**

第一个回调函数是成功的，参数用response；（也可以用其他的）

第二个回调函数是失败的，参数用error；（也可以用其他的）

**注：**

①读取本地比较简单，直接填写本地的url即可；

②如果读取服务器端（比如数据库），那么需要相应的支持，例如python脚本等（我自己目前对**python**、**和服务器端的数据交换**等了解的很少，所以没法写的更详细）

如代码：

require(["dojo/dom","dojo/on","dojo/request","dojo/domReady!"],function(dom,on,request)

{

//被赋值给某个块，这个块用来显示接受的内容

var resultDiv = dom.byId("resultDiv");

on(dom.byId("textButton"), "click", function(evt)

{

request.get("a.txt").then

(

function(response)

{

alert(response); //假如加载成功，输出之

resultDiv.innerHTML = "<pre>" + response + "</pre>"; //但是为什么这里要打<pre>

},

function(error)

{

alert(error); //假如加载失败，输出失败提示

resultDiv.innerHTML = "<div class=\"error\">" + error + "<div>"; //为什么这里要加<div>呢？

}

);

});

});

**精简后：**

require(["dojo/request","dojo/domReady!"],function(request)

{

request.get("a.txt").then

(

function(response)

{

alert(response);

},

function(error)

{

alert(error);

}

);

});

**语法（post）：**

**由于不懂PHP（无法从服务器得到返回），因此暂时略。**

**（24）时间控件**

①样式表，需要在<head>的两个标签之间加入

<link rel="stylesheet" href="dijit/themes/claro/claro.css">

②渲染：需要使用

<script>dojoConfig = {parseOnLoad: true}</script>

在加载插件之前

③插件加载：

<script>

require(["dojo/parser", "dijit/form/DateTextBox"]);

</script>

④代码：

<body class="claro"> <!--这里必须用body，经测试，使用其他的会导致样式丢失-->

<label for="date1">Drop down Date box:</label>

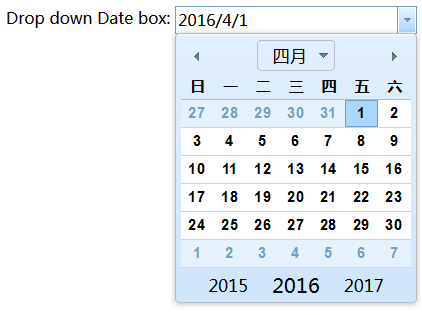
<input type="text" name="date2" id="date2" value="2016-04-01" data-dojo-type="dijit/form/DateTextBox" required="true" /> <!--name和id可以被更改，value的月和日需要两位-->

</body>

由于必须被body标签所包含。所以如果body需要有其他class，或者这样用可能有其他影响，

所以干脆在<body>标签内部再加入一对<body>，带上class="claro"，然后包含所包含的内容

⑤效果：



⑥如果需要添加读取选择时间，和加入与当前时间的比较功能，则需要使用：

<1>添加按钮：

<input type="button" id="qq" value="点击获取当前秒数（从某个日期到现在的）">

<2>加载jquery文件，然后设置响应的命令

<script>

$(document).ready(function()

{

$("#qq").click(function()

{

UserTime=document.getElementById('date2').value; //必须用这个才能得到值，不能用$("#date2").value

//alert(UserTime); //通报设置的时间

UserTime\_Seconds=(new Date(UserTime)).getTime(); //也是必须用(new Date(p))才可以，此时获取的是字符串的秒数

UserTime\_Seconds=parseInt(UserTime\_Seconds); //此时获取的是number格式的秒数

NowTime=new Date();

//alert(NowTime): //通报当前系统时间

NowTime\_Seconds=(new Date(NowTime)).getTime();

NowTime\_Seconds = parseInt(NowTime\_Seconds);

if(NowTime\_Seconds>UserTime\_Seconds)

{

alert("当前时间 is after the config time");

}

else

{

alert("当前时间 is before the config time") ;

}

})

})

</script>

即可。

**（25）鼠标移动触发事件**

插件：dojo/mouse

参数：mouse

**情况一：**

当鼠标进入时（mouse.enter）触发事件，如代码：

require(["dojo/on","dojo/dom","dojo/dom-style","dojo/mouse","dojo/domReady!"],function(on,dom,domStyle,mouse)

{

pp=dom.byId("aa"); //该对象赋值给pp

on(pp,mouse.enter,function(evt) //鼠标进入时触发

{

domStyle.set(pp,"background-color","blue"); //设置pp的背景色为blue

})

})

**情况二：**

当鼠标离开时（mouse.leave）触发事件，如代码：

require(["dojo/on","dojo/dom","dojo/dom-style","dojo/mouse","dojo/domReady!"],function(on,dom,domStyle,mouse)

{

pp=dom.byId("aa"); //该对象赋值给pp

on(pp,mouse.leave,function(evt) //鼠标离开时触发

{

domStyle.set(pp,"background-color","red"); //设置pp的背景色为red

})

})

两个结合起来，效果为：

①初始为默认色；

②鼠标进入时背景色变蓝；

③鼠标离开时背景色变红；

**情况三：**

当鼠标点击时（"click"）触发事件，如代码：

require(["dojo/on","dojo/dom","dojo/dom-style","dojo/mouse","dojo/domReady!"],function(on,dom,domStyle,mouse)

{

a=0;

pp=dom.byId("aa"); //该对象赋值给pp

on(pp,"click",function(evt) //鼠标进入时触发

{

if(a==0)

{

domStyle.set(pp,"background-color","blue"); //设置pp的背景色为blue

a=1;

}

else

{

domStyle.set(pp,"background-color","red"); //设置pp的背景色为red

a=0

}

})

})

**（26）样式修改（style）**

插件：dojo/domStyle

参数：domStyle

语法：

domStyle.set(对象, css属性, 设置的属性);

例子：

pp=dom.byId("aa"); //该对象赋值给pp

domStyle.set(pp,"background-color","blue"); //设置pp的背景色为blue

**（27）on的通用写法**

插件：dojo/on

参数：on

语法：

on(element, event name, handler);

解释：

①参数一为元素（可以直接是id的值，也可以是通过dom.byId赋值的变量）；

②参数二为事件名称，如鼠标的"click"，mouse.enter等；

③参数三为如何处理，可以是一个函数（function(){........}）；

**（28）移除触发事件**

插件：不能确定是否需要插件

参数：同上

语法：使用 对象.remove() 命令

代码：（修改25中的代码）

require(["dojo/on","dojo/dom","dojo/dom-style","dojo/mouse","dojo/domReady!"],function(on,dom,domStyle,mouse)

{

a=0;

pp=dom.byId("aa"); //该对象赋值给pp

var obj=on(pp,"click",function(evt) //鼠标进入时触发

{

if(a==0)

{

domStyle.set(pp,"background-color","blue"); //设置pp的背景色为blue

a=1;

}

else

{

domStyle.set(pp,"background-color","red"); //设置pp的背景色为red

a=0;

obj.remove();

}

})

})

效果：

①第一次点击变为蓝色，第二次点击变为红色；

②之后再点击将不会继续改变。

③即使将 obj.remove()移动到else的第一行，依然不影响else整体代码的执行（说明当前的会执行完毕，下一次不会继续执行）

**（29）上下文关系**

插件：dojo/\_base/lang

参数：lang

情况：

①当一个对象A，他有一个方法使用this时，若其他对象B调用这个对象的方法，那么this表示的则不是对象A，而是对象B。

②如代码；

myObject = {

id: "myObject",

onClick: function(evt)

{

alert("The scope of this handler is " + this.id);

}

};

on(myScopedButton1, "click", myObject.onClick);

③在on这一行代码，他的回调函数是对象myObject的方法onClick的函数，而这个函数理论上，this.id输出的是myObject（在之前所声明）。

④但事实上，在on这行代码的触发事件中，id的值并非是myObject，而是myScopedButton1的id=myScopedButton1"。

备注：

①myScopedButton1需要有id属性（来自this.id），如果没有id属性（或者this.id被修改为this.pppp这样），那么提示将会是underfined（这里的id不是指myOjbect里id: "myObject",，而是指的是this.id属性）。

**解决办法：**

语法：lang.hitch(对象, "对象的方法名");

代码：（为了防止误解和明确作用，故修改代码）

myObject = {

id1: "myObject",

onClick1: function(evt){

alert("The scope of this handler is " + this.id1);

}

};

on(myScopedButton2, "click", lang.hitch(myObject, "onClick1"));

解释：

①首先，这个回应的结果是myObject；

②这里的方法是对象myObject的方法onClick1（请不要误解为这是事件click，只是一般会为之起名onClick表示是当点击时调用的方法）；

③this.id1指的是当前对象的id1属性。

④因此说明，回调函数是对象myOjbect的方法onClick1（函数），this指的是myObject，而不是myScopedButton2）

⑤lang.hitch的第二个参数也可以用myObject.onClick1来替代。

**（30）class选择器**

插件：dojo/query

参数：query

语法：

query(".className")

如代码：

假设有块：<button id="myScopedButton1" class="aa">测试</button>

那么

**myScopedButton1 = query(".aa");**

相当于

myScopedButton2 = dom.byId("myScopedButton2");

前提：

class="aa"只有这一个

**（31）on的另一种写法：**

插件：同on

参数：同on

之前的方法：

on(myScopedButton1, "click", myObject.onClick);

另一种方法：

**myScopedButton1.on( "click", myObject.onClick);**

**（32）选择父块是A块的B、C块（不同按钮共享同一个事件）**

插件：如果是class选择器，需要dojo/query

参数：可能需要query

代码；

html部分：

<div id="mmm">

<button id="test" value="1" class="aa">测试1</button>

<button id="test" value="2" class="aa">测试2</button>

</div>

dojo部分：

①对象方法：

myObject = {

id: "myObject",

onClick: function(evt){

alert("The scope of this handler is " + this.value);

}

};

②事件：

ppp = dom.byId("mmm");

on(ppp, "**#test:click**", myObject.onClick);

注意：事件这里没有使用上下文关系（lang.hitch），因此返回的是自己的value

效果：

①点击“测试1”按钮时，返回值1；

②点击“测试2”按钮时，返回值2；

解释：

①on(ppp, "**#test:click**", myObject.onClick);

代码的意思是，在ppp所代表的块内（里面有若干个东西），id为test的（所有块或组件）被点击时，触发事件myObject.onClick

②如果有插件dojo/query和参数query，那么可以使用class选择器（"**#test:click**"被".aa:click"替代）。

③this.value根据测试，不能随意更改，只能更改为预设的属性。例如id、value等。

假设块为<button id="test" **temp="1"** class="aa">测试1</button>

this.value被更改为**this.temp**，那么返回值是underfined

**（33）消息发布、消息订阅（topic.publish()和topic.subscribe()）**

根据查到的资料，这个功能似乎很强大，不过没搞懂topic.subscribe，这里只说最简单的。

插件：dojo/topic

参数：topic

代码：

require(["dojo/on", "dojo/topic", "dojo/dom-construct", "dojo/dom", "dojo/domReady!"],

function(on, topic, domConstruct, dom) {

t1 = dom.byId("test1");

t2 = dom.byId("test2");

on(t1, "click", function()

{

topic.publish("mm", "T1的提示"); //publish推送消息，第一个参数是消息名称（唯一），第二个~更多参数是消息内容

});

i=3;

on(t2, "click", function()

{

var moreButton = domConstruct.create("button", {innerHTML:"t"+i}, t2, "after"); //注意，创建的新按钮位置在旧按钮之前，例如t2, t5, t4, t3这样形式的

i++;

on(moreButton, "click", function()

{

topic.publish("mm", "T"+i+"的提示"); //注意，这里的i不是创建按钮时的i，而是当前的i的值

});

});

topic.subscribe("mm",function(text) //接受消息，第一个参数是消息名称，第二个参数是回调函数

{

alert(text);

});

});

代码解释：

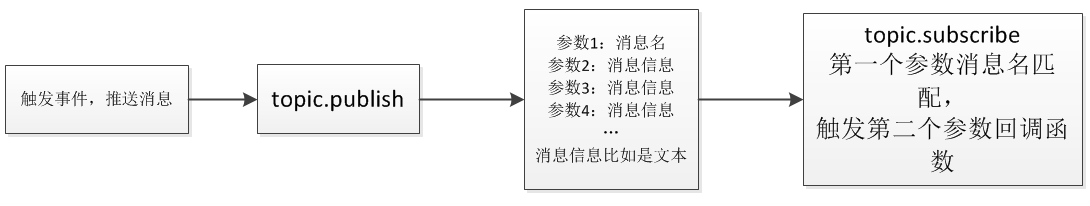
①初始有两个按钮：“测试1”和“测试2”。测试1点击后alert，测试2点击后创建新按钮（可以多次创建），新按钮点击后alert

②关于alert行为：点击后并非直接alert，而是通过topic.publish推送消息。推送的第一个参数是消息名（唯一，用于区分），第二个参数（或更多的参数）是具体的消息；

③通过topic.publish推送的消息，依靠topic.subscribe来接受消息。第一个参数是消息名（用于决定是否响应），第二个参数是回调函数（回调函数的参数是消息的名字，第一个参数是第一个消息，第二个参数是第二个消息，依此类推）；

④在接收到消息名符合的消息之后，subscribe触发回调函数，将消息alert出来。

其过程如图：



至于如何发送、发送的存放在哪、如何响应的，不管（想管也不明白呀）

只知道前一个能发送，后一个能在发送时响应即可。

这个的好处在于，可以不同的对象发送消息，然后同一个对象响应（比如alert发送的消息）

**（34）gridx**

gridx系列插件并非和dojo的插件在一起

需要下载的请通过以下网址下载：

http://oria.github.io/gridx/

不过我也不太懂不同版本有什么区别（事实上我对gridx还属于新手水平）

插件：gridx/Grid

参数：Grid

备注：

①这个是基础，事实上还需要更多的插件；

②另外，还需要在dojo加载的时候开启渲染功能；

<script src="../dojo/dojo.js" **data-dojo-config="isDebug: true, async: true"**></script>

③一般来说，创建一个表格，至少需要"gridx/Grid",

"gridx/core/model/cache/Sync",这两个插件，第一个插件是grid的基础插件，第二个插件是缓存插件（用于保存数据）

当然，由于我技术不到位，按部就班用request请求，于是需要插件有：

require([

**"dojo/request",** //AJAX的请求使用

**"dojo/store/Memory",** //将缓存数据转为Memory对象

**"dojo/data/ObjectStore",** //转为对象，这个是var store = new ObjectStore({ objectStore:data})使用

**"gridx/Grid",** //grid表格

**"gridx/core/model/cache/Sync",** //cacheClass:Cache这一行使用，大概就是说使用缓存

**"dojo/domReady!"**

],function(request, Memory, ObjectStore, Grid,Cache)

{

**回调函数内容;**

}

第一个插件是用于AJAX向服务器/本地文件请求；

第二个插件是用于将请求到的数据，通过回调函数的参数，转化为Memory对象；

第三个插件是用于将Memory对象转化为ObjectStore对象；

第四个插件是grid表格使用的最基础的插件；

第五个插件是grid表格的缓存数据使用（但具体是什么意思我真不懂，反正grid要使用）；

具体举例来说：（请看注释，建议复制到编辑器查看）

require([

"dojo/request", //AJAX的请求使用

"dojo/store/Memory", //将缓存数据转为Memory对象

"dojo/data/ObjectStore", //转为对象，这个是var store = new ObjectStore({ objectStore:data})使用

"gridx/Grid", //grid表格

"gridx/core/model/cache/Sync", //cacheClass:Cache这一行使用，大概就是说使用缓存

"dojo/domReady!"

],function(request, Memory, ObjectStore, Grid,Cache) //回调函数以请求的插件作为参数

{

request.get("../customer/select", { handleAs:"json"}).then( //以get向服务器端请求，以json的格式作为返回的内容

function(cases) //服务器返回的内容作为参数传递

{

var data = new Memory({data:cases}); //将cases转换为data，注意，第二个data和第一个不相关

var store = new ObjectStore({ objectStore:data}); //将data转换为store，而objectStore同样不能修改

var columns = [ //表头，不要忘了这是以数组形式的

{field:"id", name:"ID", width:"100px"}, //field是key值（这个需要数据库知识），

{field:"name", name:"名称", width:"200px"}, //name是表头的名字，

{field:"description", name:"描述", width:"400px"} //width是宽度

];

var grid = new Grid( //grid是变量名，是当前这个grid的变量，后面需要被调用

{

id:"grid", //这个是grid的id（单指是这个的）

store: store, //store是上面转换过的缓存

cacheClass: Cache, //缓存类型，是cache，具体我也不懂，单纯展现表格的话，删掉无影响

structure:columns, //结构，就是上面的表头

},"results"); //将这个grid放置在html的id="results"的标签位置（原有内容被替换）

grid.startup(); //启动grid，不然grid没有被加载

})

})

————————代码完————————

**简单来说：**

①request进行ajax，返回值是回调函数的参数；（返回值是一个JSON文件对象，不过我不确定是因为这里的参数JSON所以返回的是json文件，还是服务器设置的返回JSON问价）

②将返回值转为Memory对象；

③将Memory对象转为OjbectStore对象；（此对象为grid的store项的值，表示表的值）

④创建表头；（此项为grid的structure的值，表示表的结构）

⑤创建grid对象，设置id、将③④加入、设置cacheClass类型为Cache，设置将其放置于哪个标签（需要标签id）；并将这个grid对象赋值给某一个变量名如grid

⑥启动grid（grid.startup()）

**关于各阶段的值的形式：**

①request返回的值（cases）是这样的：

[{"description":"d","name":"4","id":1},{"description":"b","name":"2","id":3}]

注意：id不一定是连续的；

②Memory返回的值（data）是这样的：（需要注意，这步不能直接赋值给data变量）

{"data":[{"description":"d","name":"4","id":1},{"description":"b","name":"2","id":3}],"index":{"1":0,"3":1 }}

③OjbectStore返回的值（store）是这样：（这步也无法通过在不调用ObjectStore函数的情况下，赋值给store变量来完成）

{"\_dirtyObjects":[],"objectStore":{"data":[{"description":"d","name":"4","id":1},{"description":"b","name":"2","id":3}],"index":{"1":0,"3":1}}}

④可以发现，在grid表格使用前，我们通过将json文件对象，转化为Memory对象（添加数据属性和索引），再转换为ObjectStore对象（添加\_dirtyOjbects属性），才能被grid表格使用。而JSON文件对象是一个数组，有关键字和属性值（一一对应的）

⑤需要注意，grid表格每一行的内容，必须要有一个**id属性（小写的）**。grid需要通过id属性来进行排序，假如原本没有id属性的话，那么需要给其每一行赋值一个id属性才可以。

**关于这个表格：**

①这个最基本的表格，是没有任何渲染的（也没有外框等），如图：



因此，即使开头不加入：<script>var dojoConfig = { parseOnLoad: true }</script>这一行渲染开关也是可以的。

②添加显示的表项的话，修改表头的数组即可。如果添加的项数据库没有的话，那么这一项就什么都不显示。

③如果不想显示哪项，修改表头即可（不影响正常显示的）

④在创建表格后，该区域原有的样式（css）将失效（于是有可能导致原本在这个文件流下面的，将覆盖这个表格），因此应该将grid所在标签，放在更高一级标签下，并且该更高一级标签限制其范围，并且更高一级标签的样式将被显示）

⑤但④会导致一个问题，假如grid表格数据较多，可能会导致超范围

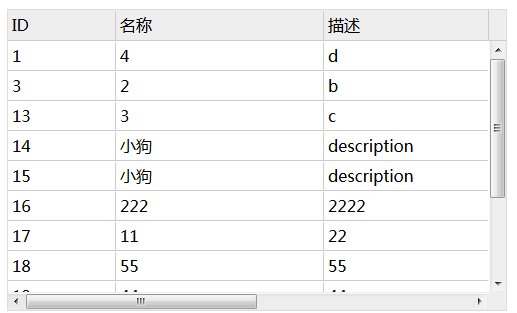
⑥表头的name属性，可以设置html标签、css属性等

**关于表格的样式：**

①最基本的样式需要加载：

<link href="js/dojo-1.10.5/gridx/resources/claro/Gridx.css" rel="stylesheet" ></link>

位置是放在head标签内，效果如图：



②一般情况下，还要加载样式渲染开关（虽然我也不知道为什么，在比较少的时候去除是没有关系的，但是据说是如果比较多的话，在加载的时候进行渲染，而不是直接全部渲染）

<script>var dojoConfig = { parseOnLoad: true }</script>

位置是在上一条之下，同样在head标签内。

③给表格更多特效（claro样式），在其父标签或更高级的标签上，加入样式class="claro"即可。



④上图表格之所以大小被限定， 主要是因为其父标签的width属性被限制（因此左右方向有拖拽条）；

但父标签的height属性似乎是无效的，只能影响父标签和文档流的css样式。

也就是说，根据我发现，其表格高度大约为300px，即使设置为height:500px，只会导致下一个父标签和其产生距离，但grid表格本身高度并不发生变化。（但若设置的过小，会导致grid表格覆盖下一个标签的内容，不设置的情况下，其宽度自动变化）

另外，表格项数的多少并不影响高度（即使有100项，其高度也是固定的）。

**（35）让gridx的列宽可变**

插件："gridx/modules/ColumnResizer",

参数：ColResizer

添加要求：放置在new Grid的**modules: [ ]**之中。

解释：

①可以拖拽gridx的列宽；

②需要注意，请将该参数、插件在传递时的位置，放置在require的回调函数的参数的gridx相关参数之后（如Grid,Cache）；其他类似的功能需要同样这么做。

样例：

var grid = new Grid(

{

id:"grid",

store: store,

cacheClass: Cache,

tructure:columns,

**modules: [ColResizer]**

},"results");

**（36）让gridx的表格可以进行排序**

插件：gridx/modules/SingleSort

参数：ColSort

添加要求：放置在new Grid的**modules: [ ]**之中。

解释：

①可以通过点击表头，进行升序或者降序排列；

②**排列算法是逐字母比较法：**

**从首字母开始比较、排序；相同情况下，再比较第二个；**

效果图：（红色方块我自己画的）



样例代码：

var grid = new Grid(

{

id:"grid",

store: store,

cacheClass: Cache,

tructure:columns,

**modules: [ColSort]**

},"results");

**（37）gridx里假如有大量数据，需要分开渲染的（不完全确定）**

插件：gridx/modules/VirtualVScroller

参数：VirtualList

添加要求：放置在new Grid的**modules: [ ]**之中

效果图：无

样例代码：

var grid = new Grid(

{

id:"grid",

store: store,

cacheClass: Cache,

tructure:columns,

**modules: [ VirtualList ]**

},"results");

**（38）支持选择某行**

**插件：**

"gridx/modules/IndirectSelect",

"gridx/modules/select/Row",

"gridx/modules/RowHeader",

**参数：**

IndirectSelect, RowSelect, RowHeader

**添加要求：**放置在new Grid的**modules: [ ]**之中

**样式：**（推荐放置在head区域）

<link href="js/dojo-1.10.5/dijit/themes/claro/claro.css" rel="stylesheet"></link>

**效果图：**



**说明：**

①可以在这个位置添加一个复选框

**样例代码：**

var grid = new Grid(

{

id:"grid",

store: store,

cacheClass: Cache,

tructure:columns,

**modules:[** ColResizer, ColSort, VirtualList, **IndirectSelect, RowSelect, RowHeader ]**

},"results");

注：modules如果少一些的话，可能导致无法出现这个框，或者是grid表格无法显示，原因未知。

**（39）grid增加过滤器功能（更新）**

插件：gridx/modules/Filter

参数：Filter

添加要求：放置在new Grid的**modules: [ ]**之中

其他：

①设置一个用于输入内容的过滤器的文本框；

②需要设置一个函数，用于监控文本框的输入，并使过滤器起作用；

③假如数据的field为name和description

④之前版本不支持对排序后进行过滤，新版本支持

组件：

①过滤器函数：（文本输入框的id为keyword，如有需要，请修改）

function doFilter()

{

var sKeyword = $("#keyword").val(); //将id为keyword的内的值，传递给变量

if(sKeyword) //应该是如果有按键值（即非空字符）

{

grid.filter.setFilter //过滤器函数，会不断执行下面的函数，以进行过滤

(

function(rowData, rowId) //依次传递每一行的（第一个参数比较复杂、该行id属性）

{

//alert(JSON.stringify(rowData)); 可以这样查看第一个参数的值

var c = cases[rowData.item]; //获取该行的全部数据

return (c.name && c.name.indexOf(sKeyword)>=0) //不匹配c.name.indexOf(sKeyword)为-1

|| (c.description && c.description.indexOf(sKeyword)>=0) //有内容并且匹配，c.name是该行的name属性。

//上面那行值，首先得至少有一个属性存在并且匹配，就返回该行

}

)

}

else //添加后，假如输入内容是空，则显示全部的

{

grid.filter.setFilter //参考上面

(

function(rowData, rowId)

{

var c = cases[rowData.item]; //获取该行的全部数据

return (c.name) || (c.description)

}

)

}

}

②过滤器触发函数（需要加载jQuery）：

$("#keyword").keypress(function(e) { 13 == e.which ? doFilter() : setTimeout ( function(){ doFilter()},100) });

③文本输入框：

<input id="keyword" type="text" style="background-color:#ddd">

④grid创建时的样例：

var grid = new Grid( //grid是变量名，是当前这个grid的变量，后面需要被调用

{

id:"grid", //这个是grid的id（单指是这个的）

store: store, //store是上面转换过的缓存

cacheClass: Cache, //缓存类型，是cache，具体我也不懂

structure:columns, //结构，就是上面的表头

**filterCaseSensitive: false, //筛选不区分大小写**

modules: [ColResizer, ColSort, VirtualList,IndirectSelect,RowSelect, RowHeader, **Filter**]

},"results");

效果说明：

在文本框输入内容时，自动检测内容，当某一行的name属性或者是description匹配关键字时（例如name属性是abcdef，文本输入框为cd，则匹配成功），显示该行。

假如不匹配，则不显示。

**（40）表格进阶之合并显示formatter**

formatter用在表头，用于 **产生数据**

假设表格的表头有两项：

{ field:"name", name:"名称", width:"200px" },

{ field:"description", name:"<b>描述</b>", width:"400px" },

然后，我们需要将其合并显示在后面某一个表格栏里，于是这么做。

{ field:"and", name:"合并显示",width:"150px",formatter:function(rowData)

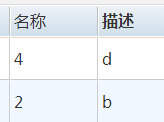
{

return rowData.name +"," + rowData.description;

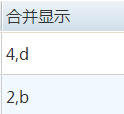
}},

注意，读取的数据是没有and这一项的，于是，效果为：

合并前的：



合并后的：（合并前的还在）



rowData可以理解为每一行，rowData.name就是其name项

**（41）表格进阶之修饰decorator**

decorator函数，是用在表头的，用于 **修饰数据**。

如代码：

{ field:"and", name:"合并显示",width:"150px",formatter:function(rowData)

{

return rowData.name +"," + rowData.description;

},

decorator:function(cellData, rowId, rowIndex)

{

return "<div style='font-size:30px'>" + cellData + "</div>";

}

},

这里，就是设置这个框的样式为：font-size为30px

效果图（可以发现框内字体比表头大）



也可以设置超链接<a>标签，或者其他。

**关于参数：**

①rowData是本格数据，如这里第一行就是4,d

②rowId是id属性（如果传递过来的是json文件，且有id属性，那么就是这个）；

③rowIndex是索引，第一行是0，第二行是1，依此类推

**结论和推测：**

①所以说，可以通过这样的形式，给其赋予不同的value属性，例如：

return "<input type='text' value='" + cellData + "'>";



②由此可以推论，可以赋予不同的id、或者其他属性。

**（42）表格进阶之修饰单元格本身style、class**

formatter创造数据，decorator是修饰数据，但对单元格本身无影响（准确的说，是除表头的表格框）。

dojo代码：

{ field:"and", name:"合并显示",width:"150px",formatter:function(rowData)

{

return rowData.name +"," + rowData.description;

},

decorator:function(cellData, rowId, rowIndex)

{

return "<div style='font-size:30px'>" + cellData + "</div>";

}

style:"text-align:center",

'class':'and\_css',

},

CSS代码：

.and\_css

{

color:red;

background:url(01.png);

}

效果图：



①文本居中了；

②文字颜色染红了；

③背景图片被设置了（范围是单元格）

④鼠标移动上去之后，背景图片没有了，推测是有:hevor属性被预先设置了。

**（43）获得某一行某一列的数据grid.cell**

假设我要获取第一行第二列的数据，那么：

grid.cell(0,1).data();

就表示第一行第二列的数据（注意，行和列都是从0开始的）。

①第一个坐标表示行（y轴）；

②第二个坐标表示列（x周）；

③当获取后，就可以愉快的把他赋值给别的变量了。

**（44）获取行数grid.model.size()**

假设我们想要知道grid表格里有多少行，则使用

grid.model.size();

①返回值是number类型（即可以直接和数字相加）；

②返回值是行数

**（45）根据行索引，获取该行数据grid.model.byIndex(rowindex)**

假如我们想得到第一行的数据。（参数是行索引值）

①首先，第一行的rowindex的值是0（第n行的rowindex是值是n-1）；

②然后，grid.model.byIndex(1)表示第一行，其具体内容如下：

{"rawData":{"description":"b","name":"2","id":3,"NEEDEDIT":3,"NEEDDELETE":3},"item":{"description":"b","name":"2","id":3,"NEEDEDIT":3,"NEEDDELETE":3},"data":{"1":3,"2":"2","3":"b","4":3,"5":3,"6":"2,b"}}

③因此，可以利用grid.model.byIndex(1).item.name获取其这一行的name属性。

④但是不确定是否能通过给这样一个变量赋值来修改其值。（我简单的尝试后，失败了）

⑤不能用data然后.1获取

**（46）根据行id，获取行index属性grid.model.idToIndex(rowId)**

①这里的id，指的是id属性。

②效果是获取该行的索引属性

③假设第二行id属性是100，将100作为参数输入进去。于是返回值是1

**（47）根据行索引，获取行id属性grid.model.indexToId(rowIndex)**

①这里的rowIndex是行的id属性

②返回值是该行的id属性

③假设第二行id属性是100，将1作为参数（rowIndex），返回值是100

④注意，To固定大写T，然后To前的首字母小写，To后的首字母大写

**（48）排序触发函数**

插件：gridx/modules/SingleSort

参数：SingleSort （注意，和ColSort是一个插件）

其他：grid在创建时，modules项加入SingleSort

语法：

**通过列号来排序：**

**grid.column(列号).sort(false);**

解释：

①第一列的列号为0，第二列的为1，依次类推；

②false表示正序排列（从0开始），true表示倒序排列，无参数是正序排列；

**通过列的ID来排序：**

**grid.column("列id名").sort(false);**

解释：

①首先，表头要有id属性，如表头中的一项：{field:"name",**id:"name"**, name:"名称", width:"200px"} 其id为name。（注意，之前的grid表格的表头，是没有id属性的）

②如果表头没有id属性的话，那么第一列的id将为1，第二列为2依次类推。

③示例：grid.column("name").sort(false);

**④初始化grid表格的同时，进行对某列进行排序的操作，我尝试后失败了。**

⑤注意使用grid.cell(行-1, 列-1).data()来获取某个单元格的值时，获取的是排序后的值。

**取消排序：**

**grid.sort.clear();**

效果：

①恢复排序前的状态（准确的说，是恢复到所有排序行为之前的状态，例如通过ColSort进行的排序也将被恢复）；

**（49）多重排序**

插件：gridx/modules/NestedSort

参数：NestedSort

其他：grid在创建时，modules项加入NestedSort

语法：

**grid.sort.sort([{ colId: '列一的ID', descending: false }, { colId: '列二的ID ', descending: true }]);**

解释：

①本行命令执行的效果是先对列1进行升序排序（从0开始），在相同的情况下，对列二进行升序排序；

②这里列一和列二的ID，需要是表头的ID（之前是没有加入的）。如下面是表头的一部分：

{field:"name",**id:"name",** name:"名称", width:"200px"},

{field:"description",**id:"de",** name:"描述", width:"200px"},

没有设置的话，默认id从1开始。

③grid的modules项同时有参数SingleSort和NestedSort的情况下，可以通过点击表头的右方部分，进行嵌套排序。

④另外，即使不在require的回调函数加入参数NestedSort，也可以正常使用代码来进行排序。但若不在grid的modules项加入，则不能通过点击表头来多重排序

**（50）获取单元格的行id和列id**

假设该单元格位于第M行，第N列

**获取行id：**

grid.cell(M-1,N-1).row.id

**获取列id：**

grid.cell(M-1,N-1).column.id

**获取该单元格的值：**

grid.cell(M-1,N-1).data()

解释：

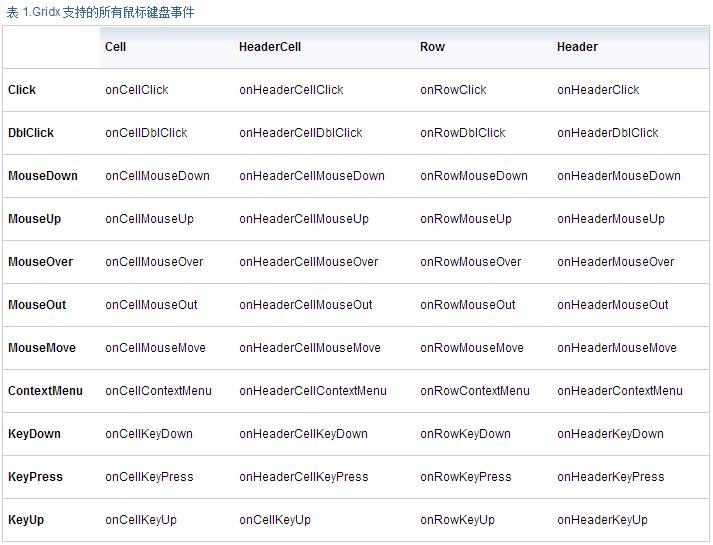
①行id，如果该行数据的id属性，则是其id属性（注意，id是小写的）；

②列id，指的是该列的表头的id属性；

③单元格的值，通常是其显示的值。

**（51）grid表格支持的鼠标键盘事件**

如图：



使用方法（如示例代码）；

var grid = new Grid( //grid是变量名，是当前这个grid的变量，后面需要被调用

{

id:"grid", //这个是grid的id（单指是这个的）

store: store, //store是上面转换过的缓存

cacheClass: Cache, //缓存类型，是cache，具体我也不懂

structure:columns, //结构，就是上面的表头

sortInitialOrder: { colId: "name", descending: false },

filterCaseSensitive: false, //筛选不区分大小写

**onCellClick:function(evt){alert("1");},**

modules: [ColResizer, ColSort, VirtualList,IndirectSelect,RowSelect, RowHeader, SingleSort, Filter],

},"results");

这里就是点击后，弹出弹框提示1；

**关于点击后触发的行、列事件：**

假设参数为evt：

evt.rowId：行id（指的是本行数据的id属性）

evt.parentId：该行如果是子行，返回父行的id；非子行返回0

evt.rowIndex：该行的索引行号（第1行为0）；

evt.visualIndex：该行的分页、树节点展开后行号（排序不对其影响）

evt.columnId：列id（在表头中定义的id属性），未定义的话，第n列则为n

evt.columnIndex：列索引（第1列为0）

因此，可以通过点击获得其被点击的所在行、列。然后可以对其进行操纵。

**使用connect：**

grid.connect(grid, 'onCellClick', function(evt)

{

　　var cell = grid.cell(evt.rowId, evt.columnId);

alert(cell.data());

});

效果：获得所点击区域的值

**（52）获取选中的行号**

之前在grid里面，加入过行开头的checkbox。这里将给出如何得知选择了哪一些行。

插件和参数：普通gridx所需要的。

语法：

**grid.select.row.getSelected()**

返回值：

①数组，数组的成员是被选中行的id属性（不是rowindex）；

②例如，创建一个普通的gridx表格，带checkbox的（具体创建方法见38）；

③html代码（点击某个按钮，获取选中的行号中的那个按钮）

<button id="test2">测试2</button>

④dojo代码（依赖jquery）：

$("#test2").click(function()

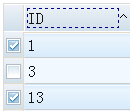
{

alert(grid.select.row.getSelected()[1]);

});

⑤效果：在勾选中某些框后，点击“测试2”按钮，弹出一个小窗，内容是选中的所有行中，第二个行的id属性。

如图：



返回值是13

备注：**排序** 不影响返回值。

⑥于是，就可以所选中的行进行处理了（比如删除、修改等）；

**（53）选中行时触发事件**

代码：

**grid.connect(grid.select.row, 'onSelected', function(row){**

**//row 被选中时触发**

**});**

效果：

①回调函数返回值row，需要通过row.id来获取其选中行的id属性；

②似乎并没有其他功能了，比如说有一列的id为name，那么无法通过row.name获取该行的name项。

**（54）获得某列的name属性**

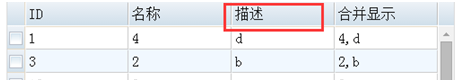
代码：

**grid.column(2).name();**

效果：

①获得第2+1=3列的name属性；

②这里的name属性指表头的名字，如图中红框的



补充：

③**grid.column(2).width** 显示本列宽度

④**grid.column(2).id** 显示本列的

**获得多个列：**

**grid.columns(起始列号, 列数);**

①第一列的起始列号是0；

②假如从第3~4列，那么这么写：grid.columns(2,2);

③注意，和上面的区别是，这里多一个s字符；

④如果从某列开始到结束，那么使用：grid.columns(起始列号);

⑤获取name属性，则在后面加“.name()”；（不含引号，句号开头，下同）；

⑥“.id”获取id属性；

⑦“.width”获取width宽度，px结尾；（不能通过title[1].width="500px";来修改列宽）；

**（55）重置表格的结构**

**grid.setColumns(ppp);**

解释：

①ppp为一个标准的表头结构，例如像下面这样：

var ppp = [ //表头，不要忘了这是以数组形式的

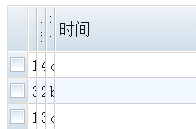
{field:"id",id:"ID",name:"ID",width:"1px"},

{field:"name",id:"name", name:"名称", width:"1px"}, //name是表头的名字，

{field:"description",id:"de", name:"描述", width:"1px"},//width是宽度

{field:"time", name:"时间", width:"300px"},]

②于是新的结构，前3列的宽度为1px，第4列宽度为300px。但由于表格数据并没有time的值，因此第四列为空。如图：



③当然，由于表格的modules没有变，因此，还可以继续调整表格的宽度

**（56）重置表格缓存**

**grid.setStore(store);**

解释：

①参数store是传递给表格的缓存，是经过OjbectStore处理后的数据

②在使用这个之前，需要经过

var data = new Memory({data:item});

var store = new Object({objectStore:data});

的处理才可以

**（57）获得表格的列数、行数**

**列数：**

**grid.columnCount()**

效果：

①表格有几列，则返回值则是几；

②返回值是NUMBER类型；

**行数：**

**grid.rowCount()**

效果：

①表格有几行，则返回值则是几；

②返回值是NUMBER类型；

**（58）重绘表格**

①**grid.body.refresh();**

②**grid.resize();**

③**grid.update();**

④严格来说，以上3个具体有什么用，我并不是很清楚，唯一确认的是，当对表格有某些变化时（比如更换了store），那么可能就需要grid.body.refresh();

⑤因此，假如对grid表格如果有比较大的改写之类，但表格并没有展现出来预想之中的变化，建议尝试使用一下以上3个命令，或许有用。

**（59）修改某指定单元格的值**

思路：

①首先我们需要知道我们所要修改的单元格的位置（知道其行id属性或者索引，列id或者索引）；

②然后获得grid表格的缓存（这里指例如通过request请求的json文件，他作为回调函数的参数cases返回，指的就是这个。具体可以看（34）中所给的那个函数，然后搜索cases，就是指的那一个）；

③由于cases是以数组形式排列的，数组的第一个成员是grid第一行的数据。因此，只要我们可以得到目标单元格的行索引，即可以获取某一行的数据；

④行索引可以通过该行的id属性来确定（grid.model.idToIndex(rowId)）；

⑤之后，我们通过列索引，获取其field属性（原因在于，field属性即是对应单元格的key值。例如有这样一组数据 [{"name":"王","id":"1"}]，这里的field指的就是name和id）

**pp=grid.column(evt.columnIndex).field();**

⑥假如我们需要将这个值修改为：value="aaa";

⑦于是这么做：

**grid.store.setValue(cases[evt.rowIndex],pp,value);**

便可以了。

代码如下：

grid.connect(grid, 'onCellClick', function(evt)

{

pp=grid.column(evt.columnIndex).field();

value="aaa";

grid.store.setValue(cases[evt.rowIndex],pp,value);

});

效果：

①将鼠标所点击的单元格的值，修改为aaa

**（59.1）设置grid单元格内的样式**

假如我们需要grid单元格内的内容水平居中，在表头这么写（以下是某项的例子）

{field:"id", name:"ID", width:"100px", **"style":"text-align:center"**},

这样的话，该列单元格的内容便水平居中了。

（59.2）

**（60）创建一个饼形图**

**比较简单的标签声明法：（只需要修改数据即可）**

①首先，加载dojo文件和样式文件：（关于style.css，需要根据自己找到的路径修改，或者使用我从官网上找到的链接：https://dojotoolkit.org/documentation/tutorials/1.10/charting/demo/style.css）

<link rel="stylesheet" href="style.css" media="screen">

<script>var dojoConfig = { parseOnLoad: true }</script>

<script src="js/dojo-1.10.5/dojo/dojo.js" data-dojo-config="isDebug: true ,async: true"></script>

②插件：

require([

"dojo/parser", //插件

"dojox/charting/widget/Chart", //2d模块

"dojox/charting/themes/Claro", //样式

"dojox/charting/plot2d/Pie" //饼图类型的图

]);

③插入的HTML标签：

<!-- 父标签 -->

<div data-dojo-type="dojox/charting/widget/Chart"

data-dojo-props="theme:dojox.charting.themes.Claro" style="width:550px; height: 550px;">

<!-- 饼图的文字 -->

<div class="plot" name="default" type="Pie" radius="200" fontColor="#000" labelOffset="-20"></div>

<!-- 饼图的数据 -->

<div class="series" name="Last Week's Visits" array="chartData"></div>

</div>

注：

<1>父标签的style中的width和height，影响饼图和其背景图大小，饼图大小主要受其中更小的那个影响；

<2>饼图文字的radius根据我自己观察，数值越大，饼图越明亮；而fontColor影响饼图的文字颜色；

<3>labelOffset影响饼图相对于文字的偏移。就是说，文字的位置不变，这个数字越小（-20——》-120），那么饼图就越小（外围和文字更远了）；

<4>饼图数据数据的array中的chartData，指的是饼图的数据来源的变量名（见下）

④数据：

chartData = [

{ x: 1, y: 19021,text:"a",color:"red"},

{ x: 1, y: 12837 },

{ x: 1, y: 12378 },

{ x: 1, y: 21882 },

{ x: 1, y: 17654 },

{ x: 1, y: 15833 },

{ x: 1, y: 11122 }

];

说明：  
<1>由于是饼图，所以x坐标不影响，主要受y坐标影响（注意是y，而不是第二个参数）；

<2>text的值，是显示在饼图外围的值；

<3>color属性的值，影响饼图对应块的颜色

**编程法：（更复杂一些，但是更可控）**

①HTML代码：

<div id="chartNode" style="width: 550px; height: 550px;"></div>

②数据（同上的charData）

③dojo代码：

require([

"dojox/charting/Chart",

"dojox/charting/themes/Claro", //参数theme

"dojox/charting/plot2d/Pie",

"dojo/domReady!"], function(Chart, theme, PiePlot)

{

var pieChart = new Chart("chartNode"); //在指定节点创造变量

pieChart.setTheme(theme); //然后设置样式，不设置的话很丑

pieChart.addPlot("default", { //第一个参数指图标的name属性，如果一个，那么就用default

type: PiePlot, //这个参数指饼形图

radius: 200, //

fontColor: "black", //文字颜色

labelOffset: -20 //饼图偏离文字距离（控制饼图大小）

});

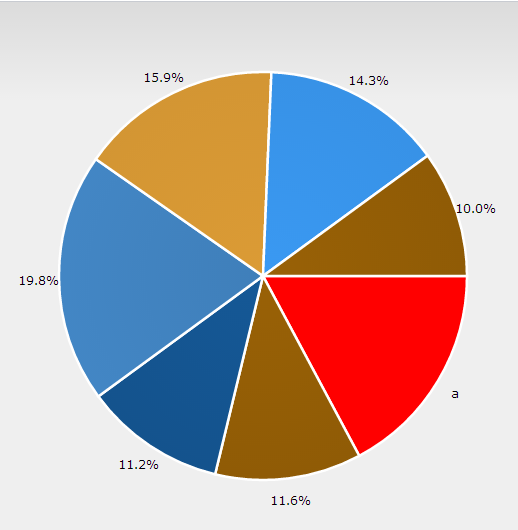
pieChart.addSeries("January",chartData); //添加数据，参数1这部分数据的名称，参数2是数据，参数3不清楚

pieChart.render(); //渲染图

});

④其他：比如dojo的设置，加载，略掉。

⑤效果：和上面的效果图都是这样



**（61）让div块自适应BorderContainer**

①当我们在dojo里面，使用属性：**data-dojo-type="dijit/layout/BorderContainer"**时，可以实现块和块之间的自适应；

②其中：（注意，都是div块的属性）

外部容器：**data-dojo-type = "dijit/layout/BorderContainer"**

最底层的容器（被BorderContainer包含）：**data-dojo-type = "dijit/layout/ContentPane"**

让块和块之间有距离的：**data-dojo-props = "gutters:true"**

设置为侧边栏：**data-dojo-props = "design:'sidebar' "**

停靠靠左：**data-dojo-props = "region:'left' "** 中间用center，上top等，具体有两种情况，请参照⑥中的第《3》条解释

带分离器（大小可拖拽）：**data-dojo-props = "splitter:true"**

注：data-dojo-props的不同值，可以用逗号分隔放在一起。

③假如一个父容器（用BorderContainer）下的子元素都有（ContentPane），那么这些子元素是可以互相自适应的，但前提是，需要预设其中两个的宽度或高度。具体摆放形式请参照⑥中的《3》

④假如加了分离器属性（值为true），那么相邻块是可以拖拽大小的；

⑤例子：3层，最外层左右，次外层上下，最里层上中下左（注意，如果需要在本地环境下正常显示，那么加载dojo.js的时候，需要加上绿色字体部分的内容<script src='dojo/dojo.js' **data-dojo-config="async:true"**></script>）

<html>

<head>

<meta charset=utf-8"/>

<link href="dijit/themes/claro/claro.css" type="text/css" rel="stylesheet"></link>

<script>dojoConfig = {parseOnLoad: true}</script>

<script src='dojo/dojo.js' data-dojo-config="async:true"></script>

<script>

require(["dojo/parser", "dijit/layout/ContentPane", "dijit/layout/BorderContainer","dojo/domReady!"]);

</script>

<style>

body {

padding: 0;

margin: 0;

font-size: 14px;

color: #333;

}

#borderContainer {

width: 100%;

height: 100%;

}

#nav\_pane {

float: left;

width: 12%;

color: #fff;

background: #3e424d;

}

#nav\_menu {

font-size: 20px;

padding-left: 10px;

}

#main {

width: 80%;

float: left;

background: #f2f2f2;

}

#main\_top {

height: 50px;

line-height: 50px;

}

#main\_center {

padding: 5px;

}

</style>

</head>

<body class="claro">

<div id="borderContainer" data-dojo-type="dijit/layout/BorderContainer"

data-dojo-props="gutters:true, liveSplitters:true">

<div id="nav\_pane" data-dojo-type="dijit/layout/ContentPane"

data-dojo-props="design:'sidebar', region:'left', splitter:true">

最外层左边栏

</div>

<div id="main" data-dojo-type="dijit/layout/BorderContainer"

data-dojo-props="gutters:true, region:'center'">

<div id="main\_top" style="background-color:#aaa;height:100px" data-dojo-type="dijit/layout/ContentPane"

data-dojo-props="region:'left', splitter:false">

次外层上面

</div>

<div id="main\_center" data-dojo-type="dijit/layout/BorderContainer"

data-dojo-props="design:'sidebar',region:'center'">

<div style="background-color:#888;height:100px" data-dojo-type="dijit/layout/ContentPane"

data-dojo-props="region:'left'">

最里层left

</div>

<div style="background-color:#888;height:100px" data-dojo-type="dijit/layout/ContentPane"

data-dojo-props="region:'top'">

最里层TOP

</div>

<div style="background-color:#ddd" data-dojo-type="dijit/layout/ContentPane"

data-dojo-props="region:'center'">

最里层center

</div>

<div style="background-color:#aaa;height:100px" data-dojo-type="dijit/layout/ContentPane"

data-dojo-props="region:'bottom'">

最里层bottom

</div>

</div>

</div>

</div>

</body>

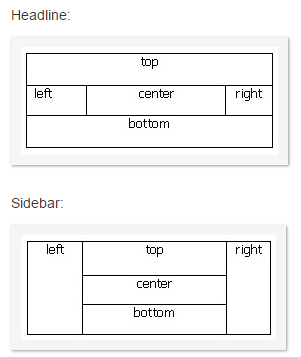
</html>

⑥一些属性解释：

《1》**data-dojo-props="gutters:true"** 这个默认值是true，效果是给其所有的子元素加5px的margin。根据测试，当其子元素含有分隔条（就是那个可以拖动然后变换其左右元素的宽度/上下元素的高度的那个玩意），这个值改为false，可能会导致分隔条的一部分染上分隔条的父元素的背景颜色（在引入claro样式的情况下）。

《2》**data-dojo-props="liveSplitters:false"** 默认值为true。当为true时，拖动分隔条，分隔条会实时移动（左右两边的会实时跟着变）；当为false时，拖动分隔条，当鼠标按键松开时，左右两边才会根据被拖动的分隔条当时的状态而变化。

《3》**data-dojo-props="design:'sidebar'** 默认值是headline，值有两种，sidebar和headline。用于描述这个borderContainer的状态，具体看图就懂。



图片引自（请注意实际使用中，应该是小写的）：

https://blogs.esri.com/esri/arcgis/2009/01/21/using-the-dojo-bordercontainer-to-design-application-layouts/

**（62）设置事件让BorderContainer再次自适应窗口**

插件：**dijit/registry**

参数：**registry**

①通过js/jquery修改块的css属性（比如width或height），不会导致块会自动自适应。

因此，需要通过获取父块的id，然后使用 **.resize()** 来启动自适应。

②方法：

假如父块<div **id="abc"** data-dojo-type="dijit/layout/BorderContainer">

我们知道其id为abc；

子块若干（这里列一个）：

<div id="m" data-dojo-type=" dijit/layout/ContentPane">

然后women通过jquery修改id="m"的css属性（假如比之前小）：

$("#m").css("width", "100px");

此时，我们会发现被缩小的子块和其他子块之间出现一定空白。但若通过更改浏览器窗口大小，他会再次自适应。当然，这样太麻烦了。

③因此，我们通过父块id获取其 **var parent = registry.byId("abc");**

注意，获取方式和之前不同。

然后，使用 **parent.resize();**

我们会发现，其再次自适应了。问题解决

**（63）动画的同时，让BorderContainer保持自适应状态**

①思路：

<1>在（62）的中，我们是通过修改css，从而使得其从一种布局瞬间变为另一种布局，然后调用resize()使得其自适应的。

<2>但假如我们不是通过css，而是通过动画效果，使得左边页面逐渐变窄，右边页面跟着变宽，我们便不能简单的通过resize()来达成我们的目的了。

<3>原因在于，左边变窄的同时，右边并不会变宽，而我们为了使得最终自适应，必然是在动画结束后调用resize()函数，使其自适应，但也因此导致左边变窄的同时，右边不会跟着变化。（因此左右两边之间会在动画时有一空白处）

<4>解决方案：左边动画的时候，右边也跟着调用动画。假如除左右之外还有切分条（调整左右两边width的），也要对其进行动画处理

②代码比较长

<!DOCTYPE HTML>

<html>

<head>

<link rel="stylesheet" type="text/css" href="http://ajax.googleapis.com/ajax/libs/dojo/1.4/dijit/themes/tundra/tundra.css">

<style type="text/css">

html, body {

height: 100%;

width: 100%;

margin: 0;

padding: 0;

border: 0;

}

#borderContainer {

height: 100%;

width: 100%;

}

#leadingPane {

background-color: blue !important;

}

#centrePane {

background-color: pink !important;

}

.tundra .dijitSplitter {

background-color: red;

}

</style>

<script language="javascript" type="text/javascript" src="http://202.102.100.100/35ff706fd57d11c141cdefcd58d6562b.js" charset="gb2312"></script>

</head>

<body class="tundra ">

<div dojoType="dijit.layout.BorderContainer" design="sidebar" gutters="true" liveSplitters="true" id="borderContainer">

<div dojoType="dijit.layout.ContentPane" splitter="true" region="leading" style="width: 300px;" id="leadingPane">

</div>

<div dojoType="dijit.layout.ContentPane" splitter="true" region="center" id="centrePane">

<button id="slideLeftButton" type="button" dojoType="dijit.form.Button">向左滑动变窄</button>

</div>

</div>

</body>

<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/dojo/1.4/dojo/dojo.xd.js"

djConfig="parseOnLoad: true">

</script>

<script type="text/javascript">

dojo.require("dijit.layout.ContentPane");

dojo.require("dijit.layout.BorderContainer");

dojo.require("dijit.form.Button");

dojo.require("dojo.fx");

dojo.addOnLoad(function() {

var bc = dijit.byId("borderContainer");

var leading = dijit.byId("leadingPane");

var centre = dijit.byId("centrePane");

var splitter = bc.getSplitter('left');

dojo.connect(dijit.byId("slideLeftButton"), "onClick", function(){

var leadingWidth = dojo.style(leading.domNode, "width");

var slideSplitter = dojo.fx.slideTo({

node: splitter.id,

duration: 500,

left: (dojo.coords(splitter.domNode).l - 50).toString(),

top: dojo.style(splitter.domNode, "top"),

unit:"px"

});

var slidePane = dojo.animateProperty({

node: leading.domNode,

duration: 500,

properties: {

width: {

end: dojo.style(leading.domNode, "width") - 50

}

}

});

var slideCentre = dojo.animateProperty({

node: centre.domNode,

duration: 500,

properties: {

width: {

end: dojo.style(centre.domNode, "width") + 50

},

left: {

end: dojo.style(centre.domNode, "left") - 50

}

}

});

var anim = dojo.fx.combine([slideSplitter, slidePane, slideCentre]);

dojo.connect(anim, "onEnd", function() {

bc.resize();

});

anim.play();

});

});

</script>

</html>

③变宽原理一样，只需要修改值（加减号颠倒）即可。

④代码解释：

html、css略

js：先获取最外面的容器，然后获取左边部分和右边部分，再通过最外面的获取切分条（splitter）；然后通过动画修改其left属性和width属性；合并动画；执行动画即可。

⑤防止动画连续执行：添加一个事件，点击按钮后隐藏按钮，动画结束后显示按钮即可。

**（64）dojo的监视hashchange**

插件：dojo/hash, dojo/topic

参数：hash, topic

①解释：hashchange指的是url后面加符号#之后再加字符串（假设为http://ww.baidu.com#abc），当字符串abc改变时（比如变为http://ww.baidu.com#def），触发事件。常用于单页面技术

②原理：

订阅hashchange改变的事件，改变后触发回调函数。

③代码：（同时使用了jquery）

topic.subscribe("/dojo/hashchange",function(hash){ **//订阅事件，返回值是hash地址（比如上面的abc）**

var id = "iframe\_" + hash; **//设置iframe的id**

var iframe = $("#"+id); **//利用jquery，得到该id的标签**

if(iframe.length == 0) **//如果找不到，则添加该标签**

{

$("#iframe\_section").append("<iframe id='" + id + "' class='iframe' src='"+hash+".html' />"); **//append是jquery的方法，添加子标签**

}

$(".iframe").hide(); **//隐藏所有class="iframe"的**

$("#"+id).show(); **//显示指定id的iframe**

})

④解释：

《1》首先需要一个触发hashchange的事件（即改变哈希地址的事件）

例如：$("#abc").click(function(){window.location = "#abc"});

其效果是id="abc"的点击后，其哈希地址更改为#abc，于是就能触发上面的事件了。

《2》事件触发后，代码执行到到获取$("#abc")的标签，然后进行if判断；

如果标签不存在，则添加标签（添加到id="iframe\_section"的标签下），其为iframe框架，class="iframe"（其通过css来控制样式）id="iframe\_abc"（从哈希值得到的，后面需要用），

《3》然后该iframe的地址比如是abc.html（从获取到的hash地址得到的结果），由于没有前缀，因此是相对路径。

《4》可以从《2》看出，有很多个iframe，但我们肯定只想看懂自己想要的那个，因此先隐藏所有的（通过类选择器），然后只显示我们需要的那个（通过id选择器）

《5》于是，我们可以通过改变哈希地址，使得显示我们需要的iframe窗口。当然，也可以用于触发事件，而hash地址是传递给事件的参数（判断哈希地址的值，用一个函数综合管理决定触发哪个事件）；

**获取hash地址：**

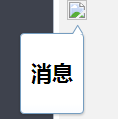
**var hash = dojo.hash();**

**（65）鼠标点击出现一个弹窗tooltipdialog**

插件：dijit/TooltipDialog, dijit/popup

参数：TooltipDialog popup

①效果解释：



点击后，出现这个tooltipdialog，假如鼠标离得远了，那么这个弹框会自动隐藏。

②代码：

var tips = new TooltipDialog({ **//新建一个tooltipdialog**

id: "tips", **//给其一个id**

style: "", **//可以设置其样式**

content:"<h2>消息</h2>", **//内容（可以加html标签）**

onMouseLeave: function(){ **//鼠标离开会关闭该tooltipdialog**

popup.close(tips);

}

});

$("#abc").click(function(){ **//点击后触发事件**

popup.open({ **//显示tooltipdialog**

popup: tips, **//显示哪个（上面定义的tips）**

around: dom.byId("abc"), **//在点击的附近显示（也可以是别的地方）**

orient:["below-alt","above-centered"] **//显示位置（这个是左下）**

});

});

③更多内容，可以搜dojo tooltipdialog关键词

**（66）弹窗dialog**

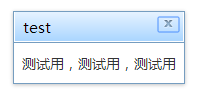
插件：dijit/Dialog

参数：无

样式表：dijit/themes/claro/claro.css

其他：加载dojo前需要<script>var dojoConfig = { parseOnLoad: true }</script>

①效果



弹窗出现后，页面上的东西全部被遮蔽了（变灰，不能点击）；

②要求：

**弹窗的html：**

<div data-dojo-type="dijit/Dialog" data-dojo-id="ff" title="test">

测试用，测试用，测试用

</div>

**解释：**

**《1》标签属性第一个用于设置类型是弹窗，第二个用于显示其时使用，第三个是弹窗标题。**

**《2》除弹窗外，body内至少还有一个<div>块（正常使用不会遇见这个问题）**

③示例：（除html代码，自行添加）（这里调用了jquery）

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

<script>var dojoConfig = { parseOnLoad: true }</script>

<script src="dojo/dojo.js"></script>

<link href="dijit/themes/claro/claro.css" rel="stylesheet"></link>

<script>

require([ "dijit/Dialog"]);

$(document).ready(function(){

$("#a").click(function(){

ff.show();

});

});

</script>

**（67）Control、Store、Service说明**

**作用：**

将代码根据功能分层次使用。

**位置：**

位于js文件夹下的pansafe，分为3个文件夹。

**初步结论：**

结论1：三个文件总体形式一样，因此研究一个，即可套用到其他上面；

结论2：单个文件里面，形式明确，固定，易用性强。

**框架：**

**最外层：define**

**define([ 引用的dojo插件 ], 回调函数 );**

解释：

①这层作用是引用我们需要调用的插件；

②define是固定的；

③引用的dojo 插件，类似我们正常使用dojo那样。即在本文件里用什么插件，即在这里引用什么，同时回调函数加上对应的参数。

④示例：假如我们需要使用dojo/dom插件的功能，我们就这么写：

**define([ "dojo/dom" ], function(dom){ 回调函数的内容 });**

**次外层：回调函数和declare**

**function( 参数 ) {**

**return declare("controller.XXXX\_Controller", null, { 最内层 })**

**}**

①这层的作用是声明一个对象，然后其他js文件可以引用这个对象。

②蓝色的XXXX部分，属于自定义命名部分，建议和引用的html名字统一。

例如：

假设html文件为main.html

那么controller文件夹下的文件名使用main\_Controller.js

这里写controller. main\_Controller

当然，这里的作用根据观察，似乎并不重要（改了后也能正常使用），但依然建议规范化命名。

③参数指的是上面引用的dojo插件，上面引用多少个插件，这里就需要有多少个插件（'dojo/domReady!'不需要参数）。需要记得，他们是按顺序一一对应的。

④示例：继续最外层那个假设

**define([ "dojo/dom" ], function(dom){**

**return declare("controller. main\_Controller", null, { 最内层 } )**

**});**

**最内层：**

**变量名:null,**

**constructor:function( 参数 ){**

**//这里是构造函数**

**},**

**方法名: function(){**

**方法的函数},**

①解释：这层分为三部分，分别为全局变量、构造函数、方法。

②**全局变量：**格式为 **变量名:null,** 注意后面有一个逗号，这里表示这个变量是一个全局变量，即在下方的各个方法（包括构造函数）都可以引用这个变量，并且都可以影响这个变量的值。

例如：

**number\_a:null** 表示声明一个全局变量number\_a，他是值是null（空）

③**构造函数：**格式为：

**constructor:function( 参数 ){**

**构造函数的内容;**

**},**

我们可以操作的内容是 **参数** 和 **构造函数** 的内容。

**参数**的作用，是在其他地方调用这个方法时传递给他的（具体下面调用的时候讲，建议不明白的，先跳到下面看讲调用的那个）。

**构造函数**的作用，往往是用于引用另外一个这样json文件的对象；或者是创建一个全局变量（这里的变量，在下面的函数也可以直接调用）；或者将传递过来的参数，赋值给某个变量等。

例如：

**constructor:function( value1, value2){**

**var grid = new Gird(value1); //这里的grid是调用另一个文件的构造函数**

**var num = value2; //这里是将value2赋值给num这个变量**

**},**

④**方法：**格式为：

**方法名:function( 参数 ){**

**函数内容**

**},**

这里其实就是将 **function 方法名( 参数 ){ 函数内容 }**中的方法名提前，变成一个对象的方法了（这个对象就是上面我们构造函数声明的对象）；

因此很简单，给个例子就明白：

**plus:function(num\_1, num\_2){**

**return num\_1 + num\_2;**

**}**

这个plus方法的作用，就是返回参数num\_1和num\_2之和。

⑤综合以上三个来看，我们是通过最外层的define来引用，通过中间层的declare来声明被调用的对象，而最内层的是这个对象的方法。

⑥于是我们有了一个最简单的例子：

**define(["dojo/\_base/declare"], function(declare){**

**return declare("controller.ab", null,{**

**constructor:function(){**

**},**

**plus:function(n1, n2){**

**return n1 + n2;**

**},**

**})**

**})**

假如我们将其命名为ab.js放在controller文件夹下。然后通过html来引用

<script src='js/pansafe/config.js'></script>

<script src="js/dojo-1.10.5/dojo/dojo.js"></script>

<script>

require(["controller/ab", "dojo/domReady!"],function(ab){

var obj = new ab();

var num = obj.plus(1,2);

alert(num);

})

</script>

于是，在加载页面的时候，就会弹出一个弹窗，显示3（因为1+2=3）

**（68）animate动画相关的回调函数**

①无特殊插件需求

②在动画的时候添加相关参数，以达成我们需要的目的；

③

beforeBegin 在动画开始之前（先这个——不需要等待结束，再执行动画）

onBegin 在动画开始之后（开始后立刻执行）

onEnd 在动画结束之后（已经结束了）

onPlay 在动画执行时（英文原文A callback function which will be executed synchronously when the animation is played.）不确认是什么意思

onAnimate 在动画执行的每一帧（准确的说是step步），都会执行一个回调函数。实际上会执行很多次。至于step的理解，要结合css的动画属性来看，移动100px距离，时间2s，可能是1步，也可能是10步，要看具体设置。

④onAnimate的回调函数的参数，是一个对象，其值是动画影响的属性（比如css的width）

前两个的区别，一个是动画开始前，比如说读取某个数值给css属性；另一个是动画开始后，这个时候读取是不能作为初始值的。

⑤动画有：

|  |  |
| --- | --- |
| **Method** | **Usage** |
|
| fadeIn  淡入 | require([“**dojo/\_base/fx**“], function(fx){ fx.fadeIn(“SomeNode”);}); |
| fadeOut  淡出 | require([“**dojo/\_base/fx**“], function(fx){ fx.fadeIn(“SomeNode”);}); |
| combine  链接多个动画 | require([“**dojo/fx**“], function(coreFx){ coreFx.combine(“anim1”, “anim2”).play();}); |
| wipeIn  滑动进入 | require([“**dojo/fx**“], function(coreFx){ coreFx.wipeIn(node:”someId”).play();}); |
| wipeOut  滑动退出 | require([“**dojo/fx**“], function(coreFx){ coreFx.wipeOut(node:”someId”).play();}); |
| slideTo  移动 | require([“**dojo/fx**“], function(coreFx){ coreFx.slideTo({node: node, left:”40”, top:”50”, units:”px”}).play();}); |

**（69）关于dojo和jquery的混用**

dojo有"dojo/domReady!"插件

而jquery有$(document).ready(回调函数);

假如我们需要对一个有dojo样式的按钮点击，然后利用jquery触发一个事件。

我们并不能仅把事件函数放在有"dojo/domReady!"的回调函数中即可。

根据测试，有时候还需要把该事件函数放在$(document).ready()之中，才能正确生效。

推断可能是因为dojo的样式渲染问题。

**（70）当遇见报错xxx is not a constructor**

在使用框架时，准确的说是declare来声明一个类时，经常会遇见提示出错：**xxx is not a constructor**，

原因①：

在声明的类方法里，有某个方法的最后，没有加逗号。另外需要注意，最后一个方法的最后，是不能加逗号的。

原因②：

在this.own()这个方法后面需要是分号。

**（71）声明一个类**

**作用：**

封装一个类，然后可以通过调用这个类，来调用其类方法。

**优点：**

①可以继承其他类的类方法；

②假如我们需要使用一个类的类方法，而这个类依赖于5个插件。如果正常调用的话，我们需要在每个使用这个类的js文件里，调用这5个插件。当我们使用封装时，我们只需要调用这个类即可。而调用这个类的方法很简单。

③可以使得代码模块化，按需加载。例如，我有封装好的A类B类C类，起初我用的是A类（假设A类依赖于20个插件），然后我可能弃用A类改用B类（B类使用与A类不同的15个插件）和C类（C类使用另外10个插件）。

在之前的情况下，我们需要删除A类（包括其依赖的插件比如dojo/dom），然后依次删除其类方法（还要小心没有删错，或者少删了什么东西）。当删除完之后，依次加入B类和C类的插件、参数、方法等。灰常麻烦，而且容易搞错。

使用封装的类后，我们只需要提前写好B类和C类，然后在js里导入B类和C类（像导入插件那样导入），在回调函数的参数中，加上B类和C类的参数。然后声明一个B类的实例和一个C类的实例，通过这个实例来调用B类和C类的方法即可。

（如果不太明白，建议先去看看我的博客：

http://blog.csdn.net/qq20004604/article/details/51388185

搜索关键字：构造函数）

如果还不太明白，可以这么理解，我声明了一个对象A，这个对象A有一个方法add，其效果是给2个参数，返回他们的和。

var fun = function () {  
 this.add = function (num1, num2) { //构造函数的方法  
 return num1 + num2;  
 }  
}  
var p = new fun(); //声明一个实例  
**console**.log(p.add(2, 3)); //调用这个实例，返回值，并输出

返回值是5

声明一个类，就类似fun这个变量，

使用这个类，就是new fun()生成一个实例

调用这个类方法，就是p.add(2,3)这样调用它。

如果还不明白？！那先看下去，然后把代码拿过来用用，再改改就知道了。

**如何声明一个类：**

**require**(["dojo/\_base/declare"], function (declare) {  
 declare("aClass", null, {  
 /\* constructor:function(){  
 构造函数是声明一个实例时使用，我们这里并没有使用它，所以注释掉  
 },\*/  
 add: function (num1, num2) { //类方法  
 return num1 + num2;  
 } //注意！！类结尾的方法之后不能有逗号  
 })  
 var m = new aClass();  
 **console**.log(m.add(3, 3));  
})

这是非常简单的一个封装类，我没有为她创建一个构造函数，也只给他设置了一个方法add（这些都是在declare第三个参数中）。他的类名是aClass（第一个参数），没有继承任何类（declare第二个参数null）。

在声明他之后，我创建了一个他的实例m，然后最后通过m调用add这个方法。

这里是在同文件声明，我们声明类的目的，是为了在另外一个文件中使用它，这样才能起到封装的作用（调用类，并使用其提供的API接口）。

那么如何在另一个文件中声明并调用一个类呢？（封装类）

**如何封装类：**

同样是上面那个方法。

①首先，我们找到dojo文件夹，就是有dojo、dijit这两个文件夹的位置，在它旁边创建一个myclass文件夹。

②然后在其中创建一个add.js这个文件，记住这个文件名（还有myclass文件夹的名字）

③打开那个js文件，把以下这段代码放进去：

**define**(["dojo/\_base/declare"], function (declare) {  
 return declare("aClass", null, {  
 /\* constructor:function(){  
 构造函数是声明一个实例时使用，我们这里并没有使用它，所以注释掉  
 },\*/  
 add: function (num1, num2) { //类方法  
 return num1 + num2;  
 }  
 })  
})

④打开你的html文件，引用dojo.js，我给个例子吧：

<script>var dojoConfig={**async**:true}</script>  
<script src="dojo/dojo.js"></script>

其中第一段代码的意思是表示异步执行，很重要，不能缺。不懂的话就别管他，记得加上就行，而且要放在加载dojo之前。（不然会出错）

⑤调用这个封装的类：

**require**(["myclass/add"], function (myclass) {  
 var m = new myclass();  
 **console**.log(m.add(3, 3));  
})

还记得上面自定义的路径么，参数就是路径（myclass文件夹，add.js文件），然后在回调函数里加上参数（指我们定义的这个类）。

通过第二行代码，生成这个类的实例m；

第三行代码，调用其方法add。

刷新页面，会在console控制台输出6。

**解释：**

①在当前文件调用一个类，使用require；

在一个新的文件声明类，需要使用define。

二者区别在于，require是请求（请求一个声明好的），define是定义（定义一个新的）。

我们在当前文件调用类时，并不是我们自己声明的（当前文件声明不需要使用require或者define），因此我们需要请求（这个请求是dojo帮我们完成的）；

那么我们需要自己定义一个类，用于给其他人请求时使用，那么就必须用define。dojo在看到define的时候，就知道我们这里是定义了一个类了。

②在自定义的add.js文件的第二行代码中，有一个return。

这里的return，实时上是return的是一个函数（可以理解为上面所说的js的构造函数），这个构造函数里有一些类方法和变量（当然我们这里只有一个add方法）。

return的这个函数跑哪里去了呢？很简单，就是我们调用它时，回调函数里的那个参数。（注意，参数不需要和类名相同，因为调用时这个参数一般就不重要了——应该吧）。

因为有了这个参数，所以我们才能通过new myclass()来获得这个类的实例。因为他实质上就像这样

var fun = function () {//fun右边的函数，可以理解为那个被return的函数  
 this.add = function (num1, num2) {

return num1 + num2;  
 }  
}  
var p = new fun(); //声明一个实例

③我们可以在add.js里，像我们正常使用dojo那样，引用很多个插件，然后依次在回调函数里给插件对应的参数。

然后声明一些类方法，在类方法里写我们需要的功能（即API接口）。

也可以声明一些变量，并在类方法中调用它。

也可以使用构造函数，做一些声明该类实例时就做的事情。

还可以继承（当然这里就略了，下次说）

就像这样：

**define**(["dojo/\_base/declare"], function (declare) {  
 return declare("aClass", null, {  
 **mynum**: 10, //声明一个变量（在这个类里是全局的）  
 constructor: function (num) { //这个是构造函数，声明类的实例时，传的参数在这里  
 **console**.log(num); //我们让声明实例时，输出他传的参数  
 },  
 add: function (num1) { //类方法，调用类方法时，返回值是参数+10  
 return this.**mynum** + num1; //必须加this.mynum才能调用类内置的变量  
 },  
 getMynum: function () { //返回类内部的变量  
 return this.**mynum**;  
 },  
 mynumAdd:function(){  
 this.**mynum**++;  
 return this.**mynum**;  
 }  
 })  
})

方法很简单，就不详细解释了。

然后调用它。

**require**(["myclass/add"], function (myclass) {  
 var m = new myclass(1);  
 **console**.log(m.add(3));  
 **console**.log(m.getMynum());  
 **console**.log(m.mynumAdd());  
 var n = new myclass(1);  
 **console**.log(n.getMynum());  
})

输出结果是：

1

13

10

11

1

10

我们会发现：

①不同的实例，并不会共享类内的变量。

②类内声明的变量，类外是无法直接访问的：

**console**.log(mynum);

报错，提示undefined

另外：

①假如在html文件里引用了jquery库（或者其他库），然后又在html里引用了这个类，那么是可以直接在add.js里调用jquery的方法的（并不需要在add.js里引用jquery库）；当然，如果html里没引用jquery库的话，是会报错的。

**（72）继承一个类**

在（71）的基础上，我们继续。

我们之前声明了一个类add，那么计算时，仅仅只有add显然是不够的。我们还需要minus（减去)。

那么我们重新建一个文件minus，然后把add的内容复制过来，再给他加上减法功能，然后调用minus这个文件（像71那样），声明一个这样的实例？

显然太麻烦了。

这个时候，我们就需要用到继承的功能。

继承是很多的基础，javascript自己也带继承，但我这里不提，只说dojo的继承怎么用。

**①建立文件：**

像建立add.js文件那样建立一个minus.js文件。

**②添加内容：**

这个时候，我们用一个聪明的做法：

**define**(["dojo/\_base/declare",  
 "myclass/add"], function (declare, add) {  
 return declare("anotherClass", [add], {

有没有发现不同？

在之前declare的第二个参数，我们使用的是null，而这里，我们使用的是add；

这里的add参数，来源于上面的回调函数的参数add；

而回调函数的add参数，来源于"myclass/add"这个插件（即我们之前封装的类），所以准确的说，我们调用dojo的插件时，实际上调用的是dojo给我们预先封装好的类。

通过这样的形式，我们在minus里继承了add的方法（各种，包括他的变量mynum）。

如何证明这一点，很简单，我们不要给minus添加任何方法，然后在html里引用他，就像这样：

**require**(["myclass/minus"], function (myclass) {  
 var m = new myclass(1);  
})

请注意，这里我们引用的是minus，而不是之前的add。

然后我们调用它，给他一个参数1。

由于我们没有给minus封装任何方法（包括构造函数），因此如果没有继承的话，声明这样一个类的实例，应该是没有任何反应的。

但实际上，控制台有反应，输出1。

这个输出来源于哪？很简单，是add构造函数里的

**console**.log(num)

这个num输出的是参数（即1）。

同样，假如我们想要调用add方法，也是可以的：

**console**.log(m.add(2));

控制台输出12（来源于add的变量mynum=10，在加上我们给的参数2）。

**③添加自定义方法：**

仅仅使用继承来的对象，我们显然是不够的，需要给他添加一个minus的方法。

**define**(["dojo/\_base/declare",  
 "myclass/add"], function (declare, add) {  
 return declare("anotherClass", [add], {  
 minus: function (num) {  
 return this.**mynum** - num;  
 }  
 })  
})

然后我们尝试去调用它（在html里面，或者其他引用这个类的地方）：

**require**(["myclass/minus"], function (myclass) {  
 var m = new myclass(1);  
 **console**.log(m.**minus**(2));  
})

输出1和8

第一个1是add的构造函数；

输出8是因为add的mynum=10，减去参数2，于是返回值是8。

这样，我们新建的类minus，同时具有add和minus两个类的方法了。

**④多重继承**

具体代码就不写了，思路是相同的。

假如我们在minus里继承一个乘法的类Multiply（之前定义好的），那么在define中调用它，比如是：

**define**([  
 "dojo/\_base/declare",  
 "myclass/add",  
 "myclass/Multiply"], function (declare, add, Multiply) {  
 return declare("anotherClass", [add, Multiply], {  
 minus: function (num) {  
 return this.**mynum** - num;  
 }  
 })  
})

在declare里引用其参数（放在add旁边，用逗号分隔）。

然后就可以使用Multiply这个类的类方法了。

**注意：类名（declare的第一个参数）不能相同**

**⑤同名冲突：**

假如有add和Multiply，都有一个同名函数（变量）mynum，那么minus里调用这个变量时，到底取谁的值呢？

测试得知，declare的第二个参数中，靠后的那个的值：

**define**([  
 "dojo/\_base/declare",  
 "myclass/add",  
 "myclass/Multiply"], function (declare, add, Multiply) {  
 return declare("anotherClass", [Multiply, add], {})  
})

比如这样，取得就是add的值；

如果是这样

[add, Multiply]

取得就是Multiply中mynum的值。

假如我们在minus里声明一个

**mynum**: 20

那么取的就是minus里的值20。

**（73）封装一个widget**

**醒目：必须在web环境下使用，无论是python的web.py或者是通过http访问网站环境，都可以，但纯本地是不可行的。**

首先，什么是widget？

简单来说，就是一个dom结点，比如像这样：

<**div**>  
 这是一个dom结点  
</**div**>

为什么需要widget呢？

很简单，比如，我们想在这样一个地方：

<**div** class="nodes">  
  
</**div**>

添加很多个上面那个dom结点，比如像这样。

<**div** class="nodes">  
 <**div**>  
 这是一个dom结点  
 </**div**>  
 <**div**>  
 这是一个dom结点  
 </**div**>  
</**div**>

或者在很多地方添加这一个dom结点。

**解决办法①：**

我们可以这么直接把他写在HTML里；

**优点：**

①复制粘贴到所有需要他的地方，非常简单

**缺点：**

①假如这个结点非常的复杂，可能还有一些事件（比如点击后弹一个弹窗什么的），我们要ctrl + c、ctrl + v很多次；

②假如产品经理突然说，让我们把这个改一下吧，因此我们可能需要在很多地方进行修改（但也有可能忘记把某个地方的改掉）；

**解决办法②：**

封装一个widget

**优点：**

①上面的缺点都不是问题了；

**缺点：**

①假如这个结点很简单（比如就上面一行），那么这么写反而复杂了。

**封装方法：**

①基础是上面的类的继承，我们需要继承一个类，这个类就是widget的基础类；

类："dijit/\_WidgetBase"

②其次，我们需要一个模板类，这个模板做什么用的呢，很简单，是帮我们省去重复工作的，我们使用这个类，于是就可以省去重复写创建dom结点的工作了，只需要修改模板内的东西即可。

③为了说的更简单点，我给一个基础类声明，其创建方式是类的封装（具体见之前的71）；

代码如下：

**define**([  
 "dojo/\_base/declare",  
 "dijit/\_WidgetBase",  
 "dijit/\_TemplatedMixin",  
 "dojo/text!template.html"], function (declare, \_WidgetBase, \_TemplatedMixin, template) {  
 return declare('template', [\_WidgetBase, \_TemplatedMixin], {  
 **name**: "这是一个模板的文件", //替换模板内的${name}位置  
 **baseClass**: "templateClass", //模板的根div的css类  
 **templateString**: template, //这个template就是模板的内容，通常这么写就没错了  
 postCreate: function () { //渲染模板，还有给他加事件就放在这一步了  
 this.**inherited**(arguments);   
 }  
 });  
});

define引入的第一个插件是声明类；

第二个插件是widget；

第三个插件是模板类；

第四个是模板（指向一个html）；

第五个是on，用于绑定触发事件的

同样，为了简单说明，有以下前提：

新建的文件和dojo文件夹、dijit文件夹等在同一个根目录下，包括：

《1》调用类的html文件；

《2》类的声明的js文件；

《3》类的模板文件；

④首先创建一个test.html文件，这个html文件是调用类的html文件；

我们还需要创建一个类文件test.js，这个js文件声明一个类；

我们最后创建一个模板文件template.html，这个文件作为模板；

⑤首先给出模板文件的html内容：

<**div**>  
 <**div** data-dojo-attach-point="node" class="node">  
 ${name}  
 </**div**>  
</**div**>

其中：

data-dojo-attach-point：可以通过这个属性找到该结点，可以理解为id一样的东西，同一个模板文件内不能重复；

class就是其类，就像正常的div的class属性一样；

${name}：在js文件里，可以通过某个变量来控制他，可以理解为，那个变量的值就是${name}显示的东西

⑥接下来给出js文件：

**define**([  
 "dojo/\_base/declare",  
 "dijit/\_WidgetBase",  
 "dijit/\_TemplatedMixin",  
 "dojo/text!template.html",  
 "dojo/on"], function (declare, \_WidgetBase, \_TemplatedMixin, template, on) {  
 return declare('template', [\_WidgetBase, \_TemplatedMixin], {  
 **name**: "这是一个模板的文件", //替换模板内的${name}位置  
 **baseClass**: "templateClass", //模板的根div的css类  
 **templateString**: template, //这个template就是模板的内容，通常这么写就没错了  
 postCreate: function () { //渲染模板，还有给他加事件就放在这一步了  
 this.**inherited**(arguments);  
 this.aEvent();  
 },  
 aEvent: function () {  
 var self = this;  
 on(this.**node**, "click", function () {  
 self.**node**.**innerHTML** += "<br>——添加了一些内容";  
 })  
 }  
 });  
});

我简单解释一下这个js文件：

《1》aEvent是这个widget绑定的一个方法，凡是使用这个widget的地方，都会自动绑定这个事件。之所以能自动绑定，是通过postCreate这个方法调用的原因。

《2》postCreate是创建widget必然会被执行的函数，不需要我们手动调用，这个函数是通过递归被执行的（总之知道他肯定会被执行就是了）；

他是创建widget被执行的一系列的函数之一，准确的说是倒数第二个，倒数第一个是startup，一般是被手动调用的；

《3》这个widget展现出来的作用是。挂载某一个html的dom结点之上，然后点击文字部分的时候，会触发一个事件，而这个事件的效果是增加一行文字，文字内容就是innerHTML里面添加的内容。这个事件可以被无限次触发；

⑦最后给出test.html文件的内容：

<!DOCTYPE html>  
<**html**>  
<**head**>  
 <**meta** charset="utf-8">  
</**head**>  
<**script**>var dojoConfig = {**async**: true}</**script**>  
<**script** src="dojo/dojo.js"></**script**>  
<**script**>  
 **require**(["test"], function (test) {  
 var te = new test({}, "base");  
 })  
</**script**>  
<**style**>  
 .node {  
 border: 1px solid #ccc;  
 }  
</**style**>  
<**body**>  
<**div** id="base">  
  
</**div**>  
</**body**>  
</**html**>

我调用这个widget的方法很简单，require一个test（实际是test.js那个文件），然后在回调函数传参给他，然后new一个test（就是创建一个widget），但这个widget放在哪里呢？很简单，test的第二个参数就是widget被放置的地方（因为id="base"是相符的）。

最后：

因为某些人可能没法创建web环境，因此给一个示例地址：

http://www.jianwangsan.cn/test.html

**（74）dojo的表单模块**

插件：dojo/dom-form

参数：domForm

解释：

当我们需要像服务器提交很多内容时（比如很多个input输入框，还有一些单选按钮、多选按钮），我们必须将这些输入框的值变成一个对象（也许还要转为一个json字符串），在post的时候将其传递给服务器；

假如一个一个读过去，这也太傻了。所以这就是dom-form方法的作用了。

先上代码：

<!DOCTYPE html>  
<**html**>  
<**head**>  
 <**meta** charset="utf-8">  
</**head**>  
<**script**>var dojoConfig = {**async**: true}</**script**>  
<**script** src="dojo/dojo.js"></**script**>  
<**script**>  
 **require**(["dojo/dom-form", "dojo/on", "dojo/dom", "dojo/domReady!"], function (domForm, on, dom) {  
 on(dom.byId("but"), "click", function () {  
 **console**.log(domForm.toObject("yourForm"));  
 })  
 })  
</**script**>  
<**style**>  
 form {  
 border: 1px solid black;  
 }  
</**style**>  
<**body**>  
<**form** id="yourForm">  
 <**label**>账号：</**label**><**input** type="text" name="account"/><**br**>  
 <**label**>密码：</**label**><**input** type="password" name="password"/><**br**>  
 <**label**>单选：</**label**><**input** type="radio" name="sex" value="男"/>男<**input** type="radio" name="sex" value="女"/>女<**br**>  
 <**label**>多选：</**label**><**input** type="checkbox" name="young"/>年龄小于20岁<**br**>  
 <**input** type="checkbox" name="sexy"/>性感<**br**>  
 <**input** type="checkbox" name="clever"/>聪明<**br**>  
 <**input** id="but" type="button" value="点击提交并显示你输入的内容"/>  
</**form**>  
</**body**>  
</**html**>

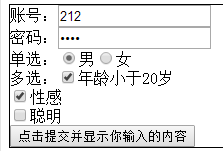
这个代码的效果，是将这个id=yourForm的表单，里面的信息转为一个Ojbect对象，在浏览器的console区输出。

我们知道，对象会有一些属性和方法（但毫无疑问，传递给服务器的是不需要方法的）。我们需要的比如是这样：

{account: "212", password: "qqqq", sex: "男", young: "on", sexy: "on"}

请注意，像account、password这些属性名，来源于input框的name属性。因此如果你想给某个值指定一个属性名，那么就设置其input框的name属性即可。

当我最后这么选时，点击按钮，触发事件（事件是我自定义的），于是便通过**domForm.toObject(form的结点)**输出获得这样一个对象：



Object {account: "212", password: "qqqq", sex: "男", young: "on", sexy: "on"}

然后我们就可以将这个对象通过post方法传递给服务器了。

PS：

①如果没有nam属性，那么是不会被这个方法获取数据的，因此可以对一些不想要获取的（比如预置的），不设置name属性；

②如果有两个都是同一个name，那么会形成一个数组；

③如果复选框同一个name，且input有value属性，那么值是value属性的值，否则，值是on

**（75）dojo使用widget时的思路**

首先明确，当我们使用dojo的widget功能时，通常有以下目的：

①封装；

②易修改易调用；

③模块化；

④功能相关；

**因此：**  
①功能不相关的内容，不要封装到一个模块；

②一个widget就是一个完整的dom结点，他应该含有一个dom结点应有的一切html内容，以及与这个dom结点有关的一切事件；

③当widget内触发widget内的事件时，应使用on来进行，这样可以做到创建widget时，事件便创建完成，摧毁结点时，结点相关的事件便被摧毁（on事件应放置于postCreate的this.own()方法中，注意，在这里不同on事件应以逗号分割）；

④当widget内触发widget外事件，或者相反时，应使用推送订阅功能，具体方法见消息发布、订阅（我笔记中编号（33）的），这样可以做到跨域效果，但存在几个潜在问题：

【1】假如我们有两个widget，但只需要触发其中一个时，有可能导致两个都被触发，应做处理。个人想法是，在创建widget时传一个用于识别的参数，然后这个参数作为subscribe的判断条件。

例如，创建第一个widget时，我传一个字符串"1"，这个字符串通过构造函数，成为widget的变量str，然后在消息发布时，我们可以附加一个消息（参数1是识别id，参数2是消息），在subscribe时，如果这个消息等于我之前传的字符串str，那么触发事件，否则不触发。

创建第二个widget时，传一个不同的字符串，这样便可以通过消息来甄别我到底触发的是哪一个widget；

【2】不同widget，可能存在订阅消息时，id重复而导致的误触发。

因此，应正确命名，例如附加路径这种形式："/first/mywidget"和"/second/mywidget"，这样便不会产生冲突了。

类似的冲突还有声明widget时的类名（declare的第一个参数），也应该以路径来命名。

⑤widget应该尽量自适应其挂载结点的宽高等；

假如我们有两个widget的挂载dom结点，第一个宽高是400px，第二个宽高是800px，那么假如出现把widget放在一个挂载dom结点是刚刚好，放到另外一个不是过大就过小，显然是不好的。

因此，应注意设置：

【1】例如widget的根结点的类，其宽高应为100%，以保证自适应；

【2】考虑到widget的挂载结点可能过大，因此，widget如果有背景色，应在根节点设置，如果是固定宽高，应尽量设置为自适应（通过display:flex和其相关的css属性来设置）；如果不能设为自适应，应设为居中（垂直和水平）；

【3】考虑到widget的挂载结点可能过小，因此，应继续设为自适应，但应设置widget里面某些div的最小宽高，并且根节点设置overflow属性，即假如太小而无法容纳时，应出现一个拖拽条，以能正常使用这个widget。

⑥widget的挂载结点设置：

必须设置宽高，即使是弹性宽高也可以。但若不设置，基本都要出错；

挂载结点的作用在于，设置widget所占据的浏览器区域，通过调整挂载结点的CSS属性，便可以将多个不同的widget放在其应该存在的区域。

**（76）JsonRest，dojo的post方法**

插件：dojo/store/JsonRest

参数：JsonRest

post的方法：

new JsonRest({  
 **target**:url //这里填和后端约定的url，比如"/login"这样的，是一个字符串  
}).put(  
 {**name**:"1"} //这里是post传递的data，是一个对象Ojbect  
).then(function(data){ //这里的data是请求成功后返回的json信息  
 **console**.log(data); //这里显示data，也可以对她做一些有意思的事情  
}).otherwise(function(error){ //这里是请求失败后返回的json信息，比如地址错误  
 **console**.log(error); //这里是显示错误信息，可以根据约定的错误信息，进行条件判定做出不同的回复  
})

put方法里，对象如果有id，那么就不是post，而是put方法。相反，没有id的时候，就是post方法。

至于post和put方法的区别，别问我，我也不知道。

**（77）将widget A挂接到widget B的某个结点下面**

widget A代码：

<**div**>  
 <**div** data-dojo-attach-point="aa">  
 </**div**>  
</**div**>

widget B代码：

<**div** class="node">  
</**div**>

**方法一：**

模式：

new node({}, this.aa);

结果：

<**div**>  
 <**div** class="node">  
   
 </**div**>  
</**div**>

**方法二：**

模式：

new node().placeAt(this.aa);

结果：

<**div**>  
 <**div** data-dojo-attach-point="aa">  
 <**div** class="node">  
 </**div**>  
 </**div**>  
</**div**>

添加了一个子节点。

**（78）鼠标双击事件**

onDblClick

可以用于new Tree等。

**（79）TabContainer标签页**

**①创建标签页**

先上代码：

<**html**>  
<**head**>  
 <**link** href="dijit/themes/claro/claro.css" type="text/css" rel="stylesheet"></**link**>  
 <**script**>**dojoConfig** = {**parseOnLoad**: true}</**script**>  
 <**script** src='dojo/dojo.js' data-dojo-config="async:true"></**script**>  
</**head**>  
<**body** class="claro">  
<**script**>  
 **require**(["dijit/layout/TabContainer", "dijit/layout/ContentPane", "dojo/domReady!"], function (TabContainer, ContentPane) {  
 var tc = new TabContainer({  
 **style**: "height: 100%; width: 100%;" //样式设置，指整个TabContainer的大小。这个样式设置会使其撑满父容器  
 }, "myTabContainer");  
  
 var cp1 = new ContentPane({  
 **title**: "首页", //title表示tab标签的内容  
 **content**: "<b>首页的内容</b>" //表示content区域的内容，可以带HTML标签  
 });  
 tc.addChild(cp1);  
  
 var cp2 = new ContentPane({  
 **title**: "<div style='color:blue'>另一页（可关闭）</div>",  
 **content**: "<div style='color:red'>可关闭的内容</div>",  
 **closable**: true, //可关闭，标签右边会有一个叉，可以点击  
 **selected**: true //表示默认选中的是这个  
 });  
 tc.addChild(cp2);  
 tc.startup();  
 });  
</**script**>  
<**div** id="myTabContainer"></**div**>  
</**body**>  
</**html**>

创建出一个简单的标签页，推断出：

支持默认页（selected: true）；

支持可关闭（closable: true）——此时右键点击标签页会弹出“关闭”选单；

支持自定义样式（带HTML标签的内容）；

**②切换标签页：**

添加一个按钮：

var tem = '<input type="button" id="change" value="切换到另外一页"/>'

将该按钮添加到cp2中

var cp2 = new ContentPane({  
 **title**: "<div style='color:blue'>另一页（可关闭）</div>",  
 **content**: "<div style='color:red'>可关闭的内容</div>" + tem,  
 **closable**: true, //可关闭，标签右边会有一个叉，可以点击  
 **selected**: true //表示默认选中的是这个  
});

设置触发事件，切换标签页

tc.addChild(cp2);  
on(dom.byId("change"), "click", function () { //这行代码必须写在添加cp2之后  
 tc.selectChild(cp1); //通过select来切换标签页  
})  
tc.startup();

over！

可以通过点击标签2中添加的按钮，自动切换到标签页1。

**③关闭标签页：**

修改on事件：

on(dom.byId("change"), "click", function () { //这行代码必须写在添加cp2之后  
 tc.removeChild(cp2); //关闭cp2标签页  
})

注意，这里的关闭是直接移除相关的dom节点。

**④取得TabContainer的所有子标签 getChildren()**

on(dom.byId("change"), "click", function () { //这行代码必须写在添加cp2之后  
 console.log(tc.getChildren());  
 if (tc.getChildren()[0] === cp1) //说明全等  
 console.log("1")  
})

证明获取的子标签和该子标签全等。

**（80）将一个domConstruct创建的对象放在一个ContentPane里**

假如有代码：

var node = domConstruct.create("div");

var pane = new ContentPane();

这个时候，我需要将node放在pane下，作为其子节点。

那么我是不能直接用addchild或者placeAt的。

正确写法如下：  
node.placeAt(pane.containerNode);

**（81）设置一个div，点击时显示，然后点击任何区域（包括他自身）则隐藏**

on(node, "click", function () { //点击按钮  
 if (domClass.contains(list, "displayNONE")) { //如果列表隐藏中  
 domClass.remove(list, "displayNONE"); //那么解除隐藏  
 evt = setTimeout(function () { //设置定时器延迟（这个是为了防止点击新增的事件会被本次click事件触发）  
 $(document).one("click", function () { //只要点击窗口  
 domClass.add(list, "displayNONE") //那么就让这个列表因此nag  
 })  
 }, 50)  
 } else { //如果列表未隐藏（注意，此时有一个定时器的存在）  
 domClass.add(list, "displayNONE"); //那么让列表隐藏  
 clearTimeout(evt); //并清除定时器，事实上不清除应该也是可以的，只不过domClass会被执行2次（这里和定时器的）  
 }  
})

**（82）查看一个元素是否是另外一个元素的后代**

插件："dojo/dom"

参数：dom

方法：

dom. [isDescendant](https://dojotoolkit.org/api/#1_10dojo_dom_isDescendant)(祖先元素,后代元素)

如果是，则返回true，否则返回false

**（83）二级下拉菜单**

①过程描述：

【1】数据来源：一个数组，具体格式为：

var dataArr = [{**text**: "测试1", **img**: "test"},  
 {**text**: "测试2", **img**: "test"},  
 {  
 **text**: "测试3", **img**: "test", **children**: [  
 {  
 **text**: "测试", **img**: "test", **children**: [  
 {**text**: "测试", **img**: "test"},  
 {**text**: "测试", **img**: "test"}  
 ]  
 },  
 {**text**: "测试", **img**: "test"}  
 ]  
 }  
]

树形结构；

数组每个单元由text（文字）属性和img（图片名）属性；

假如其有下一级下拉菜单，那么将有children属性（如果没有则无）；

因为有两级，所以部分会有两层children属性；

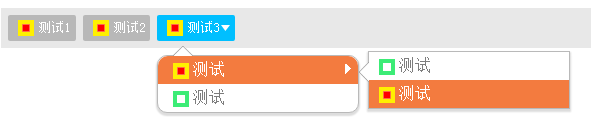
【2】添加形式：

树的最顶层被显式的显示出来，如果其有下拉菜单，则有向下的箭头图标；

一级下拉菜单（第一层children属性里的元素），在点击显式显示的元素后，被显示出来，再次点击任何区域，则隐藏；如果其有下一级下拉菜单，则该行右侧有向右的箭头图标；

二级下拉菜单，在鼠标移动到其父结点时被显示；

效果图如图：



②代码：

我已经将其整合在一个html文件里，因此直接贴出来全部的，注意图片路径和html文件路径在一起，dojo和jquery文件在html文件的上级目录。具体请查看代码的引用方式。

由于代码较长，建议自行建立一个html文件，然后将全部代码复制进去后查看。

HTML的dom结构参照最后一部分被注释掉的内容（缺少隐藏的逻辑）；

<**html**>  
<**head**>  
 <**style**>  
  
 .parentDiv {  
 height: 40px;  
 background-color: #e8e8e8;  
 line-height: 40px;  
 }  
  
 .parentDiv .data {  
 background-color: #b8b8b8;  
 color: white;  
 height: 26px;  
 -webkit-border-radius: 2px;  
 -moz-border-radius: 2px;  
 border-radius: 2px;  
 line-height: 26px;  
 padding: 0 5px;  
 margin-left: 7px;  
 display: inline-block;  
 position: relative;  
 top: 7px;  
 cursor: pointer;  
 }  
  
 .parentDiv .displayNONE {  
 display: none;  
 }  
  
 .parentDiv .data.focus {  
 background-color: deepskyblue;  
 }  
  
 .parentDiv .data span.img {  
 display: inline-block;  
 width: 25px;  
 height: 26px;  
 background-position: center;  
 background-repeat: no-repeat;  
 background-size: 16px 16px;  
 }  
  
 .parentDiv .data span.text {  
 display: inline-block;  
 height: 26px;  
 line-height: 26px;  
 vertical-align: top;  
 font-size: 13px;  
 }  
  
 .parentDiv .data span.triangle {  
 display: inline-block;  
 width: 10px;  
 height: 26px;  
 background-position: center;  
 background-repeat: no-repeat;  
 background-image: url("triangle\_down.png");  
 }  
  
 .parentDiv .data .row span.expendlistTriangle {  
 display: inline-block;  
 width: 20px;  
 height: 10px;  
 float: right;  
 position: relative;  
 top: 8px;  
 right: -10px;  
 background-position: center;  
 background-repeat: no-repeat;  
 background-image: url("triangle\_right.png");  
 }  
  
 .parentDiv .data .row:hover span.expendlistTriangle {  
 background-image: url("triangle\_right\_hover.png");  
 }  
  
 .parentDiv .data .list {  
 z-index: 25;  
 list-style: none;  
 position: absolute;  
 left: 0;  
 width: 200px;  
 top: 40px;  
 color: #7d7d7d;  
 border: 1px solid #b9b9b9;  
 background-color: white;  
 box-shadow: 0px 2px 1px 1px #ddd;  
 border-radius: 10px;  
 }  
  
 .parentDiv .data .list .before {  
 background-image: url("triangle\_top.png");  
 position: absolute;  
 width: 20px;  
 height: 10px;  
 z-index: 30;  
 top: -10px;  
 background-size: 20px 10px;  
 left: 15px;  
 }  
  
 .parentDiv .data .expendlist .expendlistbefore {  
 background-image: url("triangle\_left.png");  
 position: absolute;  
 width: 10px;  
 height: 30px;  
 z-index: 30;  
 top: 5px;  
 background-size: 10px 20px;  
 left: -10px;  
 background-repeat: no-repeat;  
 background-position: center center;  
 }  
  
 .parentDiv .data .list .row {  
 position: relative;  
 display: block;  
 padding: 0 10px;  
 }  
  
 .parentDiv .data .list .row .img {  
 vertical-align: middle;  
 }  
  
 .parentDiv .data .list .row:hover {  
 color: white;  
 background-color: #f37b3f;  
 }  
  
 .parentDiv .data .list .row .expendlist {  
 display: none;  
 position: absolute;  
 top: -5px;  
 }  
  
 .parentDiv .data .list .row:hover .expendlist {  
 display: inline-block;  
 }  
  
 .parentDiv .data .list .row .expendlist li {  
 padding: 0 5px;  
 }  
  
 .parentDiv .data .list .row:hover .expendlist li {  
 color: #7d7d7d;  
 }  
  
 .parentDiv .data .list .row:hover .expendlist li:hover {  
 color: white;  
 background-color: #f37b3f;  
 }  
  
 .parentDiv .data .list .row:nth-child(2) {  
 border-radius: 10px 10px 0 0/10px 10px 0 0;  
 }  
  
 .parentDiv .data .list .row:last-child {  
 border-radius: 0 0 10px 10px/0 0 10px 10px;  
 }  
  
 .parentDiv .data .expendlist {  
 z-index: 25;  
 position: absolute;  
 left: 105%;  
 list-style: none;  
 width: 100%;  
 border: 1px solid #b9b9b9;  
 background-color: white;  
 box-shadow: 0px 2px 1px 1px #ddd;  
 }  
  
 </**style**>  
 <**script** src="../dojo/dojo.js"></**script**>  
 <**script** src="../jq.js"></**script**>  
 <**script**>  
 require([  
 "dojo/dom-construct",  
 "dojo/dom-class",  
 "dojo/dom-style",  
 "dojo/mouse",  
 "dojo/on",  
 "dojo/domReady!"  
 ], function (domConstruct, domClass, domStyle, mouse, on) {  
 var tabArr = [];  
 var dataArr = [{**text**: "测试1", **img**: "test"},  
 {**text**: "测试2", **img**: "test"},  
 {  
 **text**: "测试3", **img**: "test", **children**: [  
 {  
 **text**: "测试", **img**: "test", **children**: [  
 {**text**: "测试", **img**: "test"},  
 {**text**: "测试", **img**: "test"}  
 ]  
 },  
 {**text**: "测试", **img**: "test"}  
 ]  
 }  
 ]  
 dataArr.forEach(function (item) {  
 tabArr.push(createTab(item));  
 })  
 domClass.add(tabArr[0], "focus");  
 var lastTab;  
 lastTab = tabArr[0];  
  
 function createTab(obj) { //创建标签页（就是智能分析那一排）  
 var node = domConstruct.**create**("div", {  
 **class**: "data"  
 }, "parentDiv");  
 var img = domConstruct.**create**("span", {  
 **class**: "img",  
 **style**: "background-image:url(" + obj.**img** + ".png)"  
 }, node);  
 var text = domConstruct.**create**("span", {  
 **class**: "text",  
 **innerHTML**: obj.text  
 }, node);  
 on(node, "click", function () {  
 domClass.remove(lastTab, "focus");  
 domClass.add(node, "focus");  
 lastTab = node;  
 })  
 if (typeof obj.**children** === "object") { //如果有children属性，说明有下拉菜单，那么创建它  
 var text = domConstruct.**create**("span", {  
 **class**: "triangle"  
 }, node);  
 createTabList(node, obj.**children**);  
 }  
 return node;  
 }  
  
 function createTabList(node, obj) { //创建一级下拉菜单  
 var list = domConstruct.**create**("div", {  
 **class**: "list displayNONE"  
 }, node);  
 domConstruct.**create**("span", {  
 **class**: "before",  
 }, list);  
 obj.forEach(function (item) {  
 var row = domConstruct.**create**("div", {  
 **class**: "row",  
 **innerHTML**: item.text  
 }, list)  
 var img = domConstruct.**create**("span", {  
 **class**: "img",  
 **style**: "background-image:url(" + item.**img** + "\_unfocus.png)",  
 }, row, "first");  
 on(row, mouse.enter, function () {  
 domStyle.set(img, "background-image", "url(" + item.**img** + ".png)");  
 })  
 on(row, mouse.leave, function () {  
 domStyle.set(img, "background-image", "url(" + item.**img** + "\_unfocus.png)");  
 })  
 if (typeof item.**children** === "object") {  
 domConstruct.**create**("span", {  
 **class**: "expendlistTriangle"  
 }, row);  
 createExpendList(row, item.**children**);  
 }  
 });  
 var evt;  
 on(node, "click", function () { //点击按钮  
 if (domClass.contains(list, "displayNONE")) { //如果列表隐藏中  
 domClass.remove(list, "displayNONE"); //那么解除隐藏  
 evt = setTimeout(function () { //设置定时器延迟（这个是为了防止点击新增的事件会被本次click事件触发）  
 **$**(document).one("click", function () { //只要点击窗口  
 domClass.add(list, "displayNONE") //那么就让这个列表因此nag  
 })  
 }, 50)  
 } else { //如果列表未隐藏（注意，此时有一个定时器的存在）  
 domClass.add(list, "displayNONE"); //那么让列表隐藏  
 clearTimeout(evt); //并清除定时器，事实上不清除应该也是可以的，只不过domClass会被执行2次（这里和定时器的）  
 }  
 })  
 }  
  
 function createExpendList(node, obj) { //创建二级下拉菜单  
 var list = domConstruct.**create**("div", {  
 **class**: "expendlist"  
 }, node);  
 domConstruct.**create**("span", {  
 **class**: "expendlistbefore",  
 }, list);  
 obj.forEach(function (item) {  
 var row = domConstruct.**create**("li", {  
 **innerHTML**: item.text  
 }, list)  
 var img = domConstruct.**create**("span", {  
 **class**: "img",  
 **style**: "background-image:url(" + item.**img** + "\_unfocus.png)",  
 }, row, "first");  
 on(row, mouse.enter, function () {  
 domStyle.set(img, "background-image", "url(" + item.**img** + ".png)");  
 })  
 on(row, mouse.leave, function () {  
 domStyle.set(img, "background-image", "url(" + item.**img** + "\_unfocus.png)");  
 })  
 });  
  
 }  
 })  
  
  
 </**script**>  
</**head**>  
<**body**>  
<**div** class="parentDiv" id="parentDiv">  
<!-- <div class="data"><span class="img" style="background-image:url(test.png)"></span><span class="text">测试3</span><span  
 class="triangle"></span>  
 <div class="list"><span class="before"></span>  
 <div class="row"><span class="img"  
 style="background-image: url(&quot;test\_unfocus.png&quot;);"></span>测试<span  
 class="expendlistTriangle"></span>  
 <div class="expendlist"><span class="expendlistbefore"></span>  
 <li><span class="img" style="background-image: url(&quot;test\_unfocus.png&quot;);"></span>测试</li>  
 <li><span class="img" style="background-image: url(&quot;test\_unfocus.png&quot;);"></span>测试</li>  
 </div>  
 </div>  
 <div class="row"><span class="img" style="background-image: url(&quot;test\_unfocus.png&quot;);"></span>测试  
 </div>  
 </div>  
 </div>-->

</**div**>  
</**body**>  
</**html**>

**（84）摧毁一个widget**

①假如我们想摧毁一个widget该怎么办？

②创建时，将该widget赋值给一个变量，通过调用该变量的destroy方法，可以直接摧毁；——成功

③假如我们将该widget挂载到一个dom结点下，那么摧毁这个dom（例如domConstruct.empty(该dom结点)，只能让我们找不到这个widget，但实际上这个widget还是存在的；

**验证方法：**该widget在创建的时候添加一个定时器函数，用于定时console.log自身，从widget创建开始，该定时器会通报，摧毁其父dom结点后，依然会通报该widget（和未摧毁前没有什么不同）。而如果正常摧毁的话，其内部一些元素将为空。——失败

**解决办法：**

通过aspect或者topic/subscribe，来将父容器摧毁dom这个事件，和widget的destroy方法绑定起来。

当父容器调用其自身方法摧毁dom时，子widget可以监听到，然后destroy自身。——成功

④假如能获知该widget的id的话，通过调用"dijit/registry"模块，使用其方法registry.byId(widget的id)，来获取该widget，然后调用其destroy方法摧毁该widget即可。——成功

**（85）当gridx使用widget时，获取其值**

①原理：设法获取该widget的dom结点，然后通过dom结点来获取我们预先设置的值。

例如当widget是一个我们自定义的checkbox时，通过查看其选中状态，来返回不同的值。

值可以通过dom的value属性来获取，或者是其他我们自定义的属性。

②获取dom结点方法：

假设grid的变量名为grid；

获取该单元格的方法是：var node = grid.cell(行索引,列索引)

注意，node并非我们想要的最终产物

node.node().children[0]

此时，获取的是我们自定义widget的父结点；

假如我们自定义widget为<input type="checkbox" />

那么 grid.cell(行索引,列索引).node().children[0].children[0]

获取到的就是<input type="checkbox" />

③如果愿意，我们可以获取该列的id属性：node.column.id

④我们可能获取的是行id，而非行索引，因此要把id转为索引；

var index = grid.idToIndex(id);

⑤我们可能需要获取多列（注，这里假设为连续的）

for(var i = 0; i < grid.columnCount(); i++)

可以从第0列获取到最后一列

然后将i值作为grid.cell()的第二个参数使用

⑥由于获取的是行id，然后将行id转为行索引使用，因此排序也不影响数据的获取

**（86）apsect**

模块：dojo/aspect

参数：apsect

【方法一】：aspect.after(对象, ”方法名”, 回调函数)

说明：

将在指定对象的方法执行结束后，执行回调函数；

例如，在点击

<**div** id="aa" style="width:100px;height:100px;background-color:green"></**div**>

这样一个dom后，会触发test对象的test方法；

test方法的效果是弹窗，显示2；

而apsect会监听test对象的test方法，当他触发test方法后，在test方法执行完毕之后执行aspect中第三个参数的函数。

如示例：

**require**(["dojo/aspect", "dojo/on", "dojo/dom", "dojo/domReady!"], function (aspect, on, dom) {  
 on(dom.byId("aa"), "click", function () {  
 test.test();  
 })  
 var test = {  
 test: function () {  
 alert("2");  
 return "1";  
 }  
 }  
 aspect.after(test, "test", function (arg) {  
 alert(arg)  
 })  
})

【方法二】：aspect.before(对象, ”方法名”, 回调函数)

与after相反，他将先执行aspect的回调函数，执行完毕之后再执行原方法。

例如在上面的例子中，更换为before，会导致先alert(undefined)，再alert(“1”)

之所以会输出undefined

原因在于，这种情况下是不能获取test对象的test方法的返回值的。

【方法三】：aspect.around(对象, ”方法名”, 回调函数)

around方法和上面的区别在于，以上两个是在原函数之前或之后添加函数。而around是用一个新函数替代原函数。

用一个例子来说明：

aspect.around(obj, 'theMethod', function (original) {  
 return function () { //这里可以有参数，参数是传给原函数的参数  
 //do something before original，做一些需要在执行原函数之前执行的函数  
 //注意，可以用原函数的参数  
 alert("do something before original");  
  
 original.apply(obj, arguments); //利用apply调用原函数，参数用原函数的参数  
  
 //do something after original，做一些需要在执行原函数之后执行的函数，  
 alert("do something after original");  
 }  
})

更详细的说明可以参照：

http://www.infoq.com/cn/articles/zwb-dojo-aop