## Bootstrapjs里栅格系统的布局

概要，栅格系统针对pc,pad,移动端开发出响应式web页面，根据不同屏幕分辨率有针对不同的解决方法。  
(0.1, 屏幕设备尺寸大于1200px 选择col-lg   
(0.2. 屏幕设备尺寸在970px到1200px 选择col-md   
(0.3. 屏幕设备尺寸在768px到970px 选择col-sm   
(0.4. 屏幕设备尺寸小于768px 选择col-xs

1.栅格系统把页面分为12栏（最多12栏），如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | <!DOCTYPE html>  <html lang="zh-cn">  <head>   <meta charset="UTF-8">   <meta name="viewport" content="width=device-width,initial-scale=1,maxinum-scale,user-scalable=no">   <title>栅格系统</title>   <link rel="stylesheet" href="library/bootstrap.min.css">   <style>         .a{          height: 50px;          border: 1px red solid;          background: pink;         }   </style>  </head>  <body>   <div class="container a">    <div class="row">     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>     <div class="col-md-1 a">1</div>    </div>      <div class="row">     <div class="col-md-3 a">3</div>     <div class="col-md-9 a">9</div>    </div>   </div>      <script src="library/jq.js"></script>  <script src="library/bootstrap.min.js"></script>  </body>  </html> |

(2.1,  col-md-1为一栏，合计12栏布满一个"横排"，md后尾随的数字为分配的栏数，（col-lg,col-sm,col-xs同理）

3.在不同屏幕分辨率的设备下，所呈现的页面为对应的"栅格式栏数页面"，从而实现响应式布局,如下代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | <!DOCTYPE html>  <html lang="zh-cn">  <head>   <meta charset="UTF-8">   <meta name="viewport" content="width=device-width,initial-scale=1,maxinum-scale,user-scalable=no">   <title>栅格系统</title>   <link rel="stylesheet" href="library/bootstrap.min.css">   <style>         .a{          height: 50px;          border: 1px red solid;          background: pink;         }   </style>  </head>  <body>   <div class="container">    <div class="row">     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>     <div class="col-lg-3 col-md-4 col-sm-6 col-xs-12 a">1</div>    </div>   </div>    <script src="library/jq.js"></script>  <script src="library/bootstrap.min.js"></script>  </body>  </html> |

（3.1，上图代码表示在屏幕设备尺寸大于1200px时，一个横排有四大栏，一个大栏有三小栏，小栏共计12栏，   
（3.2，表示在屏幕设备尺寸在970px到1200px时（可以先看为浏览器缩小至这个阶段时），一个横排有三大栏，一个大栏有四小栏，小栏共计12栏，   
（3.2，表示在屏幕设备尺寸在768px到970px时（可以先看为浏览器缩小至这个阶段时），一个横排有二大栏，一个大栏有六小栏，小栏共计12栏，   
（3.2，表示在屏幕设备尺寸小于768px时（可以先看为浏览器缩小至这个阶段时），一个横排有一大栏，一个大栏有十二小栏，小栏共计12栏，

4、栅格系统里的列偏移，嵌套和交换位置  
（4.1，列偏移

|  |  |
| --- | --- |
| 1  2  3  4 | <div class="row">     <div class="col-md-8 a">8</div>     <div class="col-md-3 col-md-offset-1 a">3</div> <!-- 列向右偏移一位 -->   </div> |

（4.2，嵌套

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <div class="row">  <!-- 嵌套 -->    <div class="col-md-9 a" style="padding:0;">     <div class="col-md-4 a"></div>     <div class="col-md-4 a"></div>     <div class="col-md-4 a"></div>    </div>    <div class="col-md-3 a">3</div>  </div> |

（4.3，交换位置

|  |  |
| --- | --- |
| 1  2  3  4 | <div class="row">   <!-- 交换位置 -->    <div class="col-md-9 col-md-push-3 a">9</div>   <!-- push,向右移 -->    <div class="col-md-3 col-md-pull-9 a">3</div>   <!-- pull，向左移 -->  </div> |

## Angular控制器

# Angular控制器

 AngularJS 控制器 **控制** AngularJS 应用程序的数据。

 AngularJS 控制器是常规的 **JavaScript 对象**。

## Angular控制器

AngularJS 应用程序被控制器控制。

**ng-controller** 指令定义了应用程序控制器。

控制器是 **JavaScript 对象**，由标准的 JavaScript **对象的构造函数** 创建。

## AngularJS 实例

<div ng-app="myApp" ng-controller="myCtrl">  
  
名: <input type="text" ng-model="firstName"><br>  
姓: <input type="text" ng-model="lastName"><br>  
<br>  
姓名: {{firstName + " " + lastName}}  
  
</div>  
  
<script>  
var app = angular.module('myApp', []);  
app.controller('myCtrl', function($scope) {  
    $scope.firstName = "John";  
    $scope.lastName = "Doe";  
});  
</script>

应用解析：

AngularJS 应用程序由 **ng-app** 定义。应用程序在 <div> 内运行。

**ng-controller="myCtrl"** 属性是一个 AngularJS 指令。用于定义一个控制器。

**myCtrl** 函数是一个 JavaScript 函数。

AngularJS 使用**$scope** 对象来调用控制器。

在 AngularJS 中， $scope 是一个应用对象(属于应用变量和函数)。

控制器的 **$scope** （相当于作用域、控制范围）用来保存AngularJS Model(模型)的对象。

控制器在作用域中创建了两个属性 (**firstName** 和 **lastName**)。

**ng-model** 指令绑定输入域到控制器的属性（firstName 和 lastName）。

## 控制器方法

上面的实例演示了一个带有 lastName 和 firstName 这两个属性的控制器对象。

控制器也可以有方法（变量和函数）：

## AngularJS 实例

<div ng-app="myApp" ng-controller="personCtrl">  
名: <input type="text" ng-model="firstName"><br>  
姓: <input type="text" ng-model="lastName"><br>  
<br>  
姓名: {{fullName()}}  
</div>  
<script>  
var app = angular.module('myApp', []);  
app.controller('personCtrl', function($scope) {  
    $scope.firstName = "John";  
    $scope.lastName = "Doe";  
    $scope.fullName = function() {  
        return $scope.firstName + " " + $scope.lastName;  
    }  
});  
</script>

## 外部文件中的控制器

在大型的应用程序中，通常是把控制器存储在外部文件中。

只需要把 <script> 标签中的代码复制到名为 [personController.js](http://www.runoob.com/try/demo_source/personController.js" \t "http://www.runoob.com/angularjs/_blank) 的外部文件中即可：

## AngularJS 实例

<div ng-app="myApp" ng-controller="personCtrl">  
  
First Name: <input type="text" ng-model="firstName"><br>  
Last Name: <input type="text" ng-model="lastName"><br>  
<br>  
Full Name: {{firstName + " " + lastName}}  
  
</div>  
  
<script src="personController.js"></script>

## 其他实例

以下实例创建一个新的控制器文件:

angular.module('myApp', []).controller('namesCtrl', function($scope) { $scope.names = [ {name:'Jani',country:'Norway'}, {name:'Hege',country:'Sweden'}, {name:'Kai',country:'Denmark'} ]; });

保存文件为  [namesController.js](http://www.runoob.com/try/demo_source/namesController.js" \t "http://www.runoob.com/angularjs/_blank):

然后，在应用中使用控制器文件:

## AngularJS 实例

<div ng-app="myApp" ng-controller="namesCtrl">  
<ul>  
  <li ng-repeat="x in names">  
    {{ x.name + ', ' + x.country }}  
  </li>  
</ul>  
</div>  
<script src="namesController.js"></script>

## Angular.js控制器之间的通信

AngularJS 本身已经提供了像指令 Directive 和 服务 Service 一类的方式，来实现数据和代码的共享和复用，但在实际的项目开发中，或许是处于懒惰，亦或是为了便利，总会想在两个控制器之间，直接进行数据的共享通信，或者是函数与方法的调用，这里我们就看看有哪些方法可以满足这个要求。

**单例服务**

单例服务是 AngularJS 本身支持的数据和代码共享方式，因为是单例的，所有的控制器访问的便是同一份数据。比如，下面的 Service 便可以实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | angular   .module('app')   .service('ObjectService', [ObjectService]);  function ObjectService() {   var list = {};   return {    get: function(id){     return list[id];    },    set: function(id, v){     list[id] = v;    }   };  } |

在一个控制器中，调用 ObjectService.set('i', 1) 设置的数据，在其它控制器中，便可以通过 ObjectService.get('i') 来获取。

**广播与事件**

AngularJS 中在触发事件和发送广播时，都可以传递参数，可以通过这一特性，来实现数据的共享。与事件和广播相关的，共有三个方法，分别是：

**1.$emit()：**触发事件，它可以向上传递数据，比如，子控制器向父控制器，还有控制器向 **$rootScope**

**2.$broadcast()：**发送广播，它可以向下传递数据，比如，父控制器向子控制器传递数据，或者 **$rootScope** 向任意控制器传递数据

**3.$on()：**监听事件与广播，可以捕获 **$emit** 和 **$broadcast**

可以将控制器之间的通信，分为三种情形：

**1.无直接关联的控制器：**使用 **$rootScope.$emit()、$rootScope.$boardcast()** 或 **$scope.$emit** 来发出数据，通过 $rootScope.$on() 来获取数据

**2.父控制器到子控制器：**父控制器使用 **$scope.$boradcast()** 来发送数据，子控制器通过 **$scope.$on()** 来获取数据

**3.子控制器至父控制器：**子控制器使用 **$scope.$emit()** 来发送数据，父控制器通过 **$scope.$on()** 来获取数据

下面是简单的用法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // one controller  angular   .module('app')   .controller('OneController', ['$scope', OneController]);  function OneController($scope){   var data = {value: 'test'};   $rootScope.$broadcast('open.notice.editor', data);  }    // other controller  angular   .module('app')   .controller('AnotherController', ['$scope', AnotherController]);  function AnotherController($scope){   $scope.$on('open.notice.editor', function(event, data){    $scope.open(data);    $scope.$emit('notice.editor.opened');   });  } |

**父控制器**

如果两个控制器共同拥有同一个父控制器，则可以通过父控制器来进行数据共享和通信。比如：

|  |  |
| --- | --- |
| 1  2  3  4 | <div ng-controller="ParentController">   <div ng-controller="ChildOneController"></div>   <div ng-controller="ChildTwoController"></div>  </div> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | // 父控制器  angular   .module('app')   .controller('ParentController', ['$scope', ParentController]);  function ParentController($scope){   // 用于传递数据的变量   $scope.data = null;  }    // 子控制器  angular   .module('app')   .controller('ChildOneController', ['$scope', ChildOneController]);  function ChildOneController($scope){   // 数据设置   $scope.$parent.data = 1;  }    // 子控制器  angular   .module('app')   .controller('ChildTwoController', ['$scope', '$timeout', ChildTwoController]);  function ChildTwoController($scope, $timeout){   $timeout(function(){    // 数据读取    console.log($scope.$parent.data);   }, 1000);  } |

**全局或共用的变量**

AngularJS 提供了对 **window** 和 **localStorage** 两个变量的封装，**$window** 和 **$localStorage** ，通过修改和监听这两个值，可以达到在控制器之间数据共享和通信的目的。方法如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | // one controller  angular   .module('app')   .controller('OneController', ['$scope', '$window', OneController]);  function OneController($scope, $window){   // 数据设置   $window.data = 1;  }    // other controller  angular   .module('app')   .controller('AnotherController', ['$scope', AnotherController]);  function AnotherController($scope){   // 监听修改   $scope.$watch(function(){    return $window.data;   }, function(n){    $scope.windowData = n;   });  } |

其实，这种监听修改的方式，也可以用在其它通信方式中。

**元素绑定**

AngularJS 中，可以通过一个元素，来获取其上的控制器实例。通过这种方式便可以快速的获取  
修改某个控制器中的数据，或者调用这个控制器中的方法。比如：

|  |  |
| --- | --- |
| 1  2  3 | <div ng-controller="AppController">   <div id="div-a"></div>  </div> |

可以通过以下的方法，来获取控制器实例：

|  |  |
| --- | --- |
| 1 | var instance = angular.element(document.getElementById('div-a')).scope(); |

接着，便可以通过这个 **instance** 来调用控制器的方法，获取和修改值了。无法是元素本身绑定有控制器，还是元素的父级元素绑定有控制器，都可以成功的获取。

## 4.[Ajax关于readyState（状态值）和status（状态码）的研究](http://www.cnblogs.com/liu-fei-fei/p/5618782.html)

var getXmlHttpRequest = function () {

try{

//主流浏览器提供了XMLHttpRequest对象

return new XMLHttpRequest();

}catch(e){

//低版本的IE浏览器没有提供XMLHttpRequest对象，IE6以下

//所以必须使用IE浏览器的特定实现ActiveXObject

return new ActiveXObject("Microsoft.XMLHTTP");

}

};var xhr = getXmlHttpRequest();// readyState 0=>初始化 1=>载入 2=>载入完成 3=>解析 4=>完成

// console.log(xhr.readyState); 0

xhr.open("TYPE", "URL", true);// console.log(xhr.readyState); 1xhr.send();// console.log(xhr.readyState); 1

xhr.onreadystatechange = function () {

// console.log(xhr.status); //HTTP状态吗

// console.log(xhr.readyState); 2 3 4

if(xhr.readyState === 4 && xhr.status === 200){

alert(xhr.responseText);

}

};

1.Ajax：readyState（状态值）和status（状态码）的区别  
readyState，是指运行AJAX所经历过的几种状态，无论访问是否成功都将响应的步骤，可以理解成为AJAX运行步骤，使用“ajax.readyState”获得  
status，是指无论AJAX访问是否成功，由HTTP协议根据所提交的信息，服务器所返回的HTTP头信息代码，使用“ajax.status”获得  
总体理解：可以简单的理解为state代表一个整体的状态。而status是这个大的state下面具体的小的状态。

2.什么是readyState  
readyState是XMLHttpRequest对象的一个属性，用来标识当前XMLHttpRequest对象处于什么状态。  
readyState总共有5个状态值，分别为0~4，每个值代表了不同的含义

|  |
| --- |
| 0：初始化，XMLHttpRequest对象还没有完成初始化  1：载入，XMLHttpRequest对象开始发送请求  2：载入完成，XMLHttpRequest对象的请求发送完成  3：解析，XMLHttpRequest对象开始读取服务器的响应  4：完成，XMLHttpRequest对象读取服务器响应结束 |

3.什么是status  
status是XMLHttpRequest对象的一个属性，表示响应的HTTP状态码  
在HTTP1.1协议下，HTTP状态码总共可分为5大类

|  |
| --- |
| 1xx：信息响应类，表示接收到请求并且继续处理  2xx：处理成功响应类，表示动作被成功接收、理解和接受  3xx：重定向响应类，为了完成指定的动作，必须接受进一步处理  4xx：客户端错误，客户请求包含语法错误或者是不能正确执行  5xx：服务端错误，服务器不能正确执行一个正确的请求    100——客户必须继续发出请求  101——客户要求服务器根据请求转换HTTP协议版本  200——交易成功  201——提示知道新文件的URL  202——接受和处理、但处理未完成  203——返回信息不确定或不完整  204——请求收到，但返回信息为空  205——服务器完成了请求，用户代理必须复位当前已经浏览过的文件  206——服务器已经完成了部分用户的GET请求  300——请求的资源可在多处得到  301——删除请求数据  302——在其他地址发现了请求数据  303——建议客户访问其他URL或访问方式  304——客户端已经执行了GET，但文件未变化  305——请求的资源必须从服务器指定的地址得到  306——前一版本HTTP中使用的代码，现行版本中不再使用  307——申明请求的资源临时性删除  400——错误请求，如语法错误  401——请求授权失败  402——保留有效ChargeTo头响应  403——请求不允许  404——没有发现文件、查询或URl  405——用户在Request-Line字段定义的方法不允许  406——根据用户发送的Accept拖，请求资源不可访问  407——类似401，用户必须首先在代理服务器上得到授权  408——客户端没有在用户指定的饿时间内完成请求  409——对当前资源状态，请求不能完成  410——服务器上不再有此资源且无进一步的参考地址  411——服务器拒绝用户定义的Content-Length属性请求  412——一个或多个请求头字段在当前请求中错误  413——请求的资源大于服务器允许的大小  414——请求的资源URL长于服务器允许的长度  415——请求资源不支持请求项目格式  416——请求中包含Range请求头字段，在当前请求资源范围内没有range指示值，请求也不包含If-Range请求头字段  417——服务器不满足请求Expect头字段指定的期望值，如果是代理服务器，可能是下一级服务器不能满足请求  500——服务器产生内部错误  501——服务器不支持请求的函数  502——服务器暂时不可用，有时是为了防止发生系统过载  503——服务器过载或暂停维修  504——关口过载，服务器使用另一个关口或服务来响应用户，等待时间设定值较长  505——服务器不支持或拒绝支请求头中指定的HTTP版本 |

 4.思考问题：为什么onreadystatechange的函数实现要同时判断readyState和status呢？

第一种思考方式：只使用readyState

var getXmlHttpRequest = function () {

if (window.XMLHttpRequest) {

return new XMLHttpRequest();

}

else if (window.ActiveXObject) {

return new ActiveXObject("Microsoft.XMLHTTP");

}

};var xhr = getXmlHttpRequest();

xhr.open("get", "1.txt", true);

xhr.send();

xhr.onreadystatechange = function () {

if (xhr.readyState === 4) {

alert(xhr.responseText);

}

};

服务响应出错了，但还是返回了信息，这并不是我们想要的结果  
如果返回不是200，而是404或者500，由于只使用readystate做判断，它不理会放回的结果是200、404还是500，只要响应成功返回了，就执行接下来的javascript代码，结果将造成各种不可预料的错误。所以只使用readyState判断是行不通的。

第二种思考方式：只使用status判断

var getXmlHttpRequest = function () {

try{

return new XMLHttpRequest();

}catch(e){

return new ActiveXObject("Microsoft.XMLHTTP");

}

};var xhr = getXmlHttpRequest();

xhr.open("get", "1.txt", true);

xhr.send();

xhr.onreadystatechange = function () {

if (xhr.status === 200) {

alert("readyState=" + xhr.readyState + xhr.responseText);

}

};

事实上，结果却不像预期那样。响应码确实是返回了200，但是总共弹出了3次窗口！第一次是“readyState=2”的窗口，第二次是“readyState=3”的窗口，第三次是“readyState=4”的窗口。由此，可见onreadystatechange函数的执行不是只在readyState变为4的时候触发的，而是readyState（2、3、4）的每次变化都会触发，所以就出现了前面说的那种情况。可见，单独使用status判断也是行不通的。

1. 由上面的试验，我们可以知道判断的时候readyState和status缺一不可。那么readyState和status的先后判断顺序会不会有影响呢？我们可以将status调到前面先判断，代码如 xhr.status === 200 && xhr.readyState === 4  
   事实上，这对于最终的结果是没有影响的，但是中间的性能就不同了。由试验我们知道，readyState的每次变化都会触发onreadystatechange函数，假如先判断status，那么每次都会多判断一次status的状态。虽然性能上影响甚微，不过还是应该抱着追求极致代码的想法，把readyState的判断放在前面。  
   xhr.readyState === 4 && xhr.status === 200

## 5.给同一个div下的多个p元素同时添加一img元素

<div id="div">

<p></p>

<p></p>

<p></p>

<p></p>

<p></p>

<p></p>

<p></p>

</div>

<script>

var ppp=document.getElementById('div').getElementsByTagName('p');

for(var i=0;i<ppp.length;i++){

var aaa=document.createElement('a');

aaa.innerHTML="我是a!";

ppp[i].appendChild(aaa);

}

</script>

## $(“#div”)==document.getElementById(‘div’)

False

以前一直认为jquery中的$("#id")和document.getElementByIdx\_x("id")得到的效果是一样的，今天做特效的时候才发现并不是这么一回事，通过测试得到:

1、alert($("#div"))得到的是[object Object]

2、alert(document.getElementById("div"))得到的是[object HTMLDivElement]

3、alert($("#div")[0])或者alert($("#div").get(0))得到的是[object HTMLDivElement]

原因解读：

document.getElementById()返回的是DOM对象，而$()返回的是jQuery对象