**AngularJS不同controller作用域之间通信的方式**

本文主要谈angularjs多个作用域之间如何互相通信。我们经常遇到这样的需求：A作用域这里有一个值改变了，如何通知作用域B相应值去改变。方法总结如下：

## **1.$rootscope**

大家都知道$scope是html和单个controller之间的桥梁，数据绑定就靠他了。而$rootscope可以被认为是全局$scope, 在各个controller里面都可以显示，也都可以修改。

下例展示了如何在$rootscope上创建一个对象和使用其中的数据：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  1 | angular.module('myApp', [])  .run(function($rootScope) {      $rootScope.test = new Date();  })  .controller('myCtrl', function($scope, $rootScope) {    $scope.change = function() {          $scope.test = new Date();      };        $scope.getOrig = function() {          return $rootScope.test;      };  })  .controller('myCtrl2', function($scope, $rootScope) {      $scope.change = function() {          $scope.test = new Date();      };        $scope.changeRs = function() {          $rootScope.test = new Date();      };        $scope.getOrig = function() {          return $rootScope.test;      };  }); |

优点:

* 简单易懂

缺点:

* 全局变量污染

适用范围:

频繁的使用$rootscope会造成全局变量污染，但我也反对部分代码洁癖者完全拒绝$rooScope的作风。一些特别频繁调用的方法，完全可以该放在$rootscope里。对于少量一旦登录就会到处显示，并且不太容易变化的变量，完全可以使用$rootScope来保存。例如系统的登录用户名，一般登录以后就基本不会变，还要在各个作用域中显示。那么对于少量此类变量，为何不用$rootScope来储存呢？除此以外，$rootScope还有一个特别有用的特性，那就是它处于所有scope的最顶层，在事件传播中有妙用，在一个通用的订阅/发布模式的angularjs通信模块中，几乎少不了使用$rootScope。这一点在后文中会有详细描述。

## 2.作用域继承

作用域嵌套带来的父子作用域的继承关系也可以算是一种父子作用域之间的通信方式。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <div ng-controller="Parent">    <div ng-controller="Child">      <div ng-controller="ChildOfChild">         <button ng-click="someParentFunctionInScope()">Do</button>      </div>    </div>  </div> |

优点:

* 对于从祖先到子孙的数据传递效果很好

缺点:

* 从子孙到祖先的数据传递效果不好，子 Scope 的属性会隐藏（覆盖）了父 Scope 中的同名属性，对子 Scope 属性的更改并不更新父 Scope同名属性的值。（这个行为实际上不是 AngularJS 特有的，JavaScript 本身的原型链就是这样工作的。）
* 不能进行兄弟作用域的数据传递，除非用一个共同祖先，例如$rootScope
* 调用祖先函数意味着祖先与子孙之间的紧密耦合，当程序复杂到一定程度时修改起来会导致牵一发动全身的悲惨结局

适用范围:

从上面的优缺点分析中我们可以看到作用域继承方法有很大的局限性。故而作用域继承通常只用在简单、小型的模块中，例如directive指令的书写中。

## 3.作用域继承+$watch

为了解决作用域继承不能解决的从子孙到祖先的数据传递问题，可以用$scope.$watch函数来监视数据变化。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | //父作用域监视子作用域  .controller("Parent", function($scope){    $scope.VM = {a: "a", b: "b"};    $scope.$watch("VM.a", function(newVal, oldVal){      // react    });  }    //子作用域监视父作用域  .controller("child", function($scope){      $scope.$parent.$watch($scope.VM.a, function(){         //react      });  } |

angularjs的指令书写模式中，还有一种指定指令的scope的方式，本质上与此相通，诸如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  1 | $compileProvider.directive('anrow', function ($compile) {          return {              require:    '^angrid',              restrict:   'E',              transclude: true,              scope:      {                  anrowData:     "=anrowData",                  selectedItems: "=selects",                  searchFilter:  "=searchFilter"              },              template: '',              replace:    true,                          link: function(scope, element, attrs, angridCtrl){                          }                   }    }); |

优点:

* 适合用在非controller生成的子作用域中，例如ng-repeat生成的大量自作用域中

缺点:

* 对于one-time events（一次性事件）没什么效果
* $watch函数类似eval函数，写出来的代码不宜读

适用范围:

$watch函数功能很强大，配合$scope.$parent和原本的继承关系可以实现父子作用域的各种数据传递。偶尔还会需要使用$scope.$eval方法来将字符串变为对象。这种方法用在整体架构的controller中或许会导致代码难以读懂，但是用在一些j较为独立的angularjs指令或插件中确是极好的。例如我写的angularjs 表格插件[angrid](https://github.com/zhangdiwaa/anGrid" \t "http://blog.csdn.net/lai_xu/article/details/_blank)就是用这个方法写的。表格中包含大量ng-repeat生成的自作用域，几乎都是用这种方式来实现。

## 4.消息机制

异步回调响应式通信—事件机制是javascript解决模块通信的最常用手段。在angularjs中此方法表现为由$scope下定义的三个函数$broadcast, $emit, $on组成的事件隧道通信机制。这里我援引 破狼的博客  [Angularjs Controller 间通信机制](http://www.cnblogs.com/whitewolf/archive/2013/04/16/3024843.html" \t "http://blog.csdn.net/lai_xu/article/details/_blank) 来简单地说明这个方法怎么用：

## Angularjs在scope中为我们提供了冒泡和隧道机制，$broadcast会把事件广播给所有子controller，而$emit则会将事件冒泡传递给父controller，$on则是angularjs的事件注册函数，有了这一些我们就能很快的以angularjs的方式去解决angularjs controller之间的通信，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <div ng-app="app" ng-controller="parentCtr">      <div ng-controller="childCtr1">name :          <input ng-model="name" type="text" ng-change="change(name);" />      </div>      <div ng-controller="childCtr2">Ctr1 name:          <input ng-model="ctr1Name" />      </div>  </div> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | angular.module("app", []).controller("parentCtr",  function ($scope) {      $scope.$on("Ctr1NameChange",        function (event, msg) {          console.log("parent", msg);          $scope.$broadcast("Ctr1NameChangeFromParrent", msg);      });  }).controller("childCtr1", function ($scope) {      $scope.change = function (name) {          console.log("childCtr1", name);          $scope.$emit("Ctr1NameChange", name);      };  }).controller("childCtr2", function ($scope) {      $scope.$on("Ctr1NameChangeFromParrent",        function (event, msg) {          console.log("childCtr2", msg);          $scope.ctr1Name = msg;      });  }); |

这里childCtr1的name改变会以冒泡传递给父controller，而父controller会对事件包装再广播给所有子controller，而childCtr2则注册了change事件，并改变自己。注意父controller在广播时候一定要改变事件name。

jsfiddle链接：[http://jsfiddle.net/whitewolf/5JBA7/15/](http://jsfiddle.net/whitewolf/5JBA7/15/" \t "http://blog.csdn.net/lai_xu/article/details/_blank)

优点:

* 对一次性事件的效果很好
* 事件机制可以有效降低controller之间的耦合度

缺点:

* 由于DOM树事件响应机制等原因，angularjs里的事件机制也是采取冒泡+广播的方式，不能像C语言中那样定义事件触发和响应槽这样的直接响应关系。这个因素直接导致如下两个问题：
* 不能进行兄弟作用域的数据传递，除非用一个共同祖先，例如$rootScope
* 相比$emit冒泡方法，$broadcast广播方法要消耗更多的资源，因为广播事件会深入到该作用域的所有子孙作用域，跟单路径冒泡的$emit消耗的资源完全不是数量级。故而对于包含成数千子作用域又要追求较高性能的情况，可能需要考虑一下是否弃用$broadcast方法。这里有一个对比测试，$rootScope.emit() vs $rootScope.$broadcast, performance tests ([http://jsperf.com/rootscope-emit-vs-rootscope-broadcast](http://jsperf.com/rootscope-emit-vs-rootscope-broadcast" \t "http://blog.csdn.net/lai_xu/article/details/_blank))。

适用范围:

事件隧道机制可以解决绝大部分事件通信问题，也是这里非常推荐的方式。

不过，当模块复杂到一定程度，可能就要援引一些设计模式方面的知识才能解决问题，而事件隧道机制、$rootScope、scope继承+$watch方式 都是成为了实现设计模式的基本手段。

## 5.专用service

前文已经提到，可以专门构建service来处理作用域间的通信问题。如果controller之间有较强依赖，例如都会操作同一个数据集，那么创建一个专门的service模块来处理此类事务，比直接用事件隧道机制在逻辑上更清晰。一个最简单的服务模块的例子如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var myApp = angular.module('myApp', []);    myApp.factory('Data', function () {    return { message: "I'm data from a service" };  });    function FirstCtrl($scope, Data) {    $scope.data = Data;  }    function SecondCtrl($scope, Data) {    $scope.data = Data;  } |

然而在实际应用中，仅仅把数据存取抽取出来是不足够的，我们还需要触发机制，以保证ctrl1使数据变化后，ctrl2的数据也能跟着改变。对此有两种办法，其一是使用消息机制，其二是使用$watch方法来检测数据变化。

使用$watch来监控数据变化的例子，这里我找了一个比较典型的，因为比较长所以我只贴链接：http://jsbin.com/rifob/1/edit?html,js,output

下面这个例子则是使用事件通信机制：代码来自：http://jsfiddle.net/simpulton/XqDxG/。

|  |  |
| --- | --- |
| 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 | var myModule = angular.module('myModule', []);  myModule.factory('mySharedService', function($rootScope) {      var sharedService = {};        sharedService.message = '';        sharedService.prepForBroadcast = function(msg) {          this.message = msg;          this.broadcastItem();      };        sharedService.broadcastItem = function() {          $rootScope.$broadcast('handleBroadcast');      };        return sharedService;  });    function ControllerZero($scope, sharedService) {      $scope.handleClick = function(msg) {          sharedService.prepForBroadcast(msg);      };        $scope.$on('handleBroadcast', function() {          $scope.message = sharedService.message;      });  }    function ControllerOne($scope, sharedService) {      $scope.$on('handleBroadcast', function() {          $scope.message = 'ONE: ' + sharedService.message;      });  }    function ControllerTwo($scope, sharedService) {      $scope.$on('handleBroadcast', function() {          $scope.message = 'TWO: ' + sharedService.message;      });  }    ControllerZero.$inject = ['$scope', 'mySharedService'];    ControllerOne.$inject = ['$scope', 'mySharedService'];    ControllerTwo.$inject = ['$scope', 'mySharedService']; |

## 6.发布/订阅模式

显然程序员并不以以上几种方式此为满足。前文讲事件已经提到，当模块复杂到一定程度，如果仅仅使用消息机制，同级作用域的交互都需要经过父作用域来传递消息，并且组件之间广播消息意味着它们需要多少知道一些其它组件编码的细节，这样就限制了它们的模块化和重用。这个时候就要引用一些设计模式的方法来解决问题。而实现他们的手段，就是以上提到的 $rootScope， scope继承+$watch, 消息机制 和 自定义service。考虑我们所要的需求，一方面要保证多个controller(模块)的数据一致性，一方面还要保证controller(或模块)的模块化和重用性，那么在这种场合使用观察者模式或其变种就非常合适。

在我查阅的资料中有很多国外程序员推荐使用使用[发布/订阅模式](http://blog.csdn.net/tjvictor/article/details/5223309" \t "http://blog.csdn.net/lai_xu/article/details/_blank)。发布/订阅模式是观察者模式的一个变种，也是消息队列模式的一类，是解决多模块操作同一数据集时的一种常用方案。  订阅发布模式定义了一种一对多的依赖关系，让多个订阅者对象同时监听某一个主题对象。这个主题对象在自身状态变化时，会通知所有订阅者对象，使它们能够自动更新自己的状态。可以有效地实现模块间的解耦，提高可维护性。

有个国外程序员就此做了一个demo并且写了篇博客介绍他的实现，现在他的这篇文章已经有了[翻译后的版本](http://www.lupaworld.com/article-243546-1.html" \t "http://blog.csdn.net/lai_xu/article/details/_blank)。这种方法从本质上说是同构构建service来处理作用域间的通信问题，还是用$rootScope来做顶级父作用域，并且做了事件的发布和接收全部封装了起来。大家有兴趣的话可以直接到[angularjs-pubsub](https://github.com/lavinjj/angularjs-pubsub" \t "http://blog.csdn.net/lai_xu/article/details/_blank) 上下载代码，研究他是怎么做的。

不过由于这位程序员完成他的demo较早，我个人感觉其中还有很多可以改进的地方。如下例是一个较为简单的[发布/订阅模式](http://blog.csdn.net/tjvictor/article/details/5223309" \t "http://blog.csdn.net/lai_xu/article/details/_blank)的实现：此方法通过$rootScope定义一个简单的发布/订阅者模式，并通过消息机制来进行发布和订阅。其中尽量避免了$broadcast的使用。此代码来自：http://jsfiddle.net/brendanowen/ADukg/47/

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  2 | var myApp = angular.module('myApp', []);    myApp.service('messageService', ['$rootScope', function($rootScope) {        return {          publish: function(name, parameters) {              $rootScope.$emit(name, parameters);          },          subscribe: function(name, listener) {              $rootScope.$on(name, listener);          }      };  }]);    myApp.controller('MyCtrl', ['$scope', 'messageService', function ($scope, messageService) {      $scope.showDialog = false;      $scope.name = 'Superhero';        $scope.show = function() {          messageService.publish('dialog', {show: true});      };        messageService.subscribe('dialog', function(event, parameters) {          $scope.showDialog = parameters.show;      });    }]);    myApp.controller('Dialog', ['$scope', 'messageService', function ($scope, messageService) {        $scope.hide = function() {          messageService.publish('dialog', {show: false});      };  }]); |

此外还有不使用消息机制，纯粹用自定义的消息队列来实现的发布/订阅模式的代码范例。注意其中使用了jquery。下面的代码来自：https://gist.github.com/floatingmonkey/3384419

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  2 | 'use strict';    (function() {      var mod = angular.module("App.services", []);  //register other services here...      /\* pubsub - based on https://github.com/phiggins42/bloody-jquery-plugins/blob/master/pubsub.js\*/      mod.factory('pubsub', function() {          var cache = {};          return {              publish: function(topic, args) {                  cache[topic] && $.each(cache[topic], function() {                      this.apply(null, args || []);                  });              },              subscribe: function(topic, callback) {                  if(!cache[topic]) {                      cache[topic] = [];                  }                  cache[topic].push(callback);                  return [topic, callback];              },              unsubscribe: function(handle) {                  var t = handle[0];                  cache[t] && d.each(cache[t], function(idx){                      if(this == handle[1]){                          cache[t].splice(idx, 1);                      }                  });              }          }      });      return mod;  })(); |

最后是我在书写这篇博客前，在查阅大量资料的基础上，总结而成的一个angularjs的 发布/订阅 模式 服务模块，目前下面的代码正应用于我的项目。此代码较好地解决了三个问题：

* 结构清晰易读，比$watch方式容易理解和使用
* 不使用$broadcast，只用$emit来发布事件，效率较高。
* 允许用户控制cache，在controller了生命周期结束后自动解除$rootscope上的事件绑定，低功耗无污染

|  |  |
| --- | --- |
| 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  3 | define([      '../../app'  ], function (app) {      // 这是一个通用的 发布订阅模块      //参考：https://gist.github.com/turtlemonvh/10686980/038e8b023f32b98325363513bf2a7245470eaf80      app.factory('pubSubService', ['$rootScope', function ($rootScope) {          // private notification messages          var \_DATA\_UPDATED\_ = '\_DATA\_UPDATED\_';          /\*           \* @name: publish           \* @description: 消息发布者，只用$emit冒泡进行消息发布的低能耗无污染方法           \* @param: {string=}: msg, 要发布的消息关键字，默认为'\_DATA\_UPDATED\_'指数据更新           \* @param: {object=}: data，随消息一起传送的数据，默认为空           \* @example:           \*         pubSubService.publish('config.itemAdded', {'id': getID()});           \*         更一般的形式是：           \*      pubSubService.publish();           \*/          var publish = function (msg, data) {              msg = msg || \_DATA\_UPDATED\_;              data = data || {};              $rootScope.$emit(msg, data);          };          /\*           \* @name: subscribe           \* @description: 消息订阅者           \* @param: {function}: 回调函数，在订阅消息到来时执行           \* @param: {object=}: 控制器作用域，用以解绑定,默认为空           \* @param: {string=}: 消息关键字，默认为'\_DATA\_UPDATED\_'指数据更新           \* @example:           \*         pubSubService.subscribe(function(event, data) {           \*        $scope.power = data.power;           \*            $scope.mass = data.mass;           \*        },  $scope, 'data\_change');           \*        更一般的形式是：           \*        pubSubService.subscribe(function(){});           \*/          var subscribe = function (func, scope, msg) {              if (!angular.isFunction(func)) {                  console.log("pubSubService.subscribe need a callback function");                  return;              }              msg = msg || \_DATA\_UPDATED\_;              var unbind = $rootScope.$on(msg, func);              //可控的事件反绑定机制              if (scope) {                  scope.$on('$destroy', unbind);              }          };            // return the publicly accessible methods          return {              publish:        publish,              subscribe:      subscribe          };      }])  }); |

优点:

* 我们可以降低组件之间的耦合度，并将它们的之间通信的细节封装起来。在不影响主体功能的情况下提高模块的重用性。

缺点:

* 缺点是增加了代码复杂度，所以一定要写足够给力的接口说明文档，否则时间久了自己都不知道发生了什么事情。

适用范围:

该方案推荐给内部含有大量作用域通信，并且特别强调代码重用性的场合使用。

发布/订阅模式已经能解决大部分复杂多模块的通信问题了。但是如果模块很多，复杂度继续上升，那么会造成消息种类过多。这种时候有必要使用责任链模式来替换普通的发布/订阅模式。

设计模式的种类有很多，但是引入前端设计的并不多。很多人觉得设计模式难是因为不存在一个“完美”的模式，必须根据实际情况来选用相应的模式，或者说，完美是不断适应新情况的能力。这种随机应变的能力才是真正考验代码设计者的问题。