Service , Factory 的区别：

如果不仔细研究，感觉他们没有区别。如果没有区别又为何存在呢？

细细道来：

Service : 相当于是 函数 function, 所以当第一次引用时，angular 会自动使用 new 来实例化。 但是只实例化一次，且对象是全局共享。 以后再被引用也不会实例化了。 Service 不需要 return , 如果有 return , return xxx, return this;

也不会出错。

Factory: 本身就是一个单独的对象， 所以必须要求返回此对象, 如果没有返回return, 则提示出错， 对象不一定是 object , {} , 也可以是值类型，虽然是独立的对象， 但是由于第一次引用时就会创建此对象， 并且全局共享使用。 以后再被其他地方引用也不会实例化。

<script src="https://code.angularjs.org/1.6.0/angular.js"></script>

<script src="https://code.angularjs.org/1.6.0/angular-route.js"></script>

<script>

angular.module("LWHTable", [])

.service("myservice", function(){

this.count = 0;

this.show = function() {

this.count++;

alert("Service Count: " + this.count);

}

console.log("myservice init");

}).factory("myfact", function(myservice){

var fact = {};

fact.count = 0;

fact.hello = function() {

this.count++;

myservice.count++;

alert("Factory Count: " + this.count + " Service Count:" + myservice.count);

}

console.log("myfact init");

return fact;

});

</script>

上面我们定义了 一个 service - myservice, 一个factory – myfact

1. 定义只是定义， 不会初始化实例对象。
2. 由于在 factory 里引用了 myservice , 所以会有 myservice 的实例。
3. service , factory 在同一个HTML 页面里，不管被引用多少次，都只实例化一次。也就是说这个实例化的对象在整个页面周期里都是共享使用的。

我们看看这个例子： myservice 被引用了 3 次： 但是只会被实例化一次

<script>

angular.module("LWHTable", [])

.service("myservice", function(){

this.count = 0;

this.show = function() {

this.count++;

alert("Service Count: " + this.count);

}

console.log("myservice init count: " + this.count);

}).factory("myfact", function(myservice){

var fact = {};

fact.count = 0;

fact.hello = function() {

this.count++;

myservice.count++;

alert("Factory Count: " + this.count + " Service Count:" + myservice.count);

}

myservice.count++;

console.log("myfact init count: " + myservice.count);

return fact;

}).directive("mytext", function (myservice) {

return {

restrict: "E",

replace: true,

scope: {

},

template: [

'<span>Count = {{getCount()}}</span>'

].join(''),

controller: function ($scope, myservice) {

myservice.count++;

console.log("directive controller count: " + myservice.count);

$scope.getCount = function() {

return myservice.count;

}

},

link: function (sc, el, attr) {

}

}

});

angular.module("LWHPhone",["LWHTable"])

.factory("hisFact", function(myservice){

var fact = {};

fact.world = function() {

myservice.count++;

}

myservice.count++;

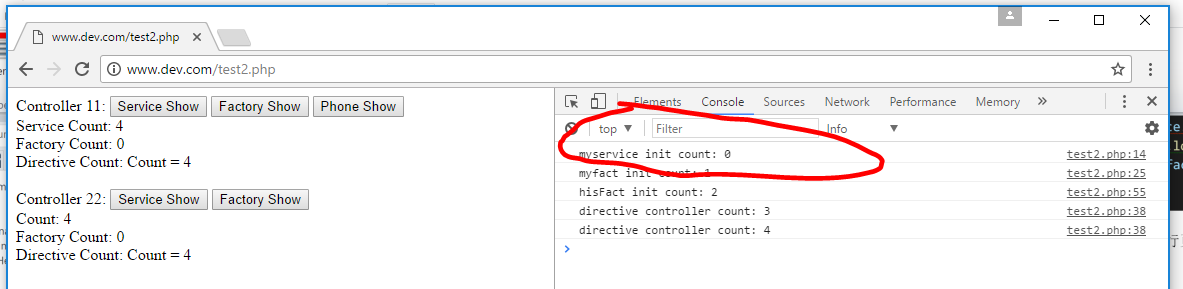
console.log("hisFact init count: " + myservice.count);

return fact;

});

</script>

我们在 Browser 运行页面会看到， myservice 会被实例化一次。



我们用实例说明， 不管是 service, factory 不管引用多少次，只实例化一次，而且对象是全局共享使用， 里面定义的对象属性自然也是全局共享， 这样就可以实现各个不同 controller之间共享数据，传递数据。

我们在一个 ng-app=”myApp” 里定义两个控制器 controller, 分别都引用 myservioce, myfact, hisFact, mytext directive(里面引用myservice)

<script>

var app = angular.module("myApp", ["LWHTable", "ngRoute", "LWHPhone"]);

app.controller("ct11", function ($scope, myservice, myfact, hisFact) {

$scope.serviceme1 = function() {

myservice.show();

}

$scope.serviceCount1 = function() {

return myservice.count;

}

$scope.factme1 = function() {

myfact.hello();

}

$scope.factCount1 = function() {

return myfact.count;

}

$scope.phoneme1 = function() {

hisFact.world();

}

});

app.controller("ct22", function($scope, myservice, myfact) {

$scope.serviceme2 = function() {

myservice.show();

}

$scope.serviceCount2 = function() {

return myservice.count;

}

$scope.factme2 = function() {

myfact.hello();

}

$scope.factCount2 = function() {

return myfact.count;

}

});

</script>

HTML 内容如下：

<body ng-app="myApp">

<div ng-controller="ct11">

Controller 11:

<button ng-click="serviceme1()">Service Show</button>

<button ng-click="factme1()">Factory Show</button>

<button ng-click="phoneme1()">Phone Show</button>

<br>

Service Count: {{ serviceCount1() }}

<br>

Factory Count: {{ factCount1() }}

<br>

Directive Count: <mytext></mytext>

</div>

<br>

<div ng-controller="ct22">

Controller 22:

<button ng-click="serviceme2()">Service Show</button>

<button ng-click="factme2()">Factory Show</button>

<br>

Count: {{ serviceCount2() }}

<br>

Factory Count: {{ factCount2() }}

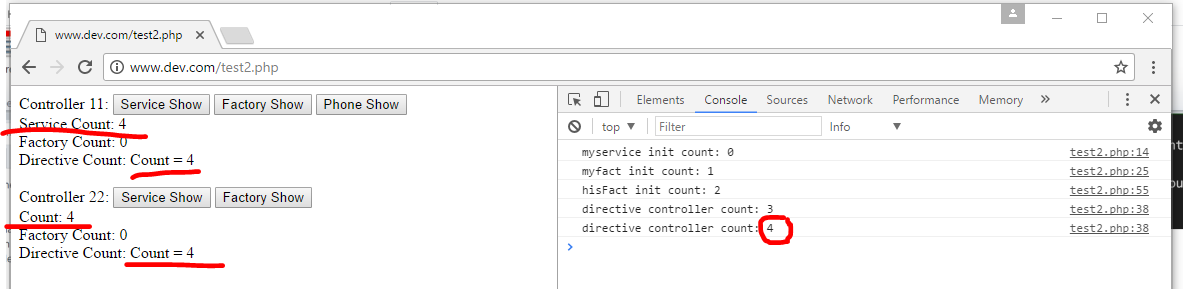
<br>

Directive Count: <mytext></mytext>

</div>

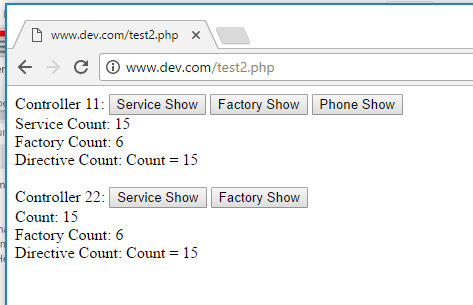
</body>

看看页面初始化时的结果：



分别引用的 myservice, myfact , hisFact 都是实例化一次， 而myservice.count 是全局共享， 每次增加1， 都会共享。

例如，在不同的 controller 里对 myservice, myfact 的公共属性 count 作增量操作。 最终他们的结果值都是最后赋予的值。



Factory 可以返回值类型

我们都知道 factory 本身就是个一独立的对象， 必须要返回一个对象， 但是这个对象不一定是 object , 也可以是值类型的，如 return 1000; return new Date();

angular.module("LWHPhone",["LWHTable"])

.factory("hisFact", function(myservice){

return new Date();

});

$scope.phoneme1 = function() {

alert("his:" + hisFact);

}

所以直接输出 Factory – hisFact 输出值类型

Service 可以返回值类型吗？？

.service("scalar", function() {

return 5000;

});

$scope.phoneme1 = function() {

alert("his:" + scalar);

console.log(scalar);

}

我们看看返回的是一个 空的 object {} , 相当于 new Function()｛｝

**Service , Factory 的另外一种写法**：可以很清楚他们区别在哪里

<script>

**//Javascript 定义函数**

var myfunc = function() {

this.count = 0;

console.log("myfunc init count: " + this.count);

}

myfunc.prototype = {

show: function() {

this.count++;

alert("myfunc count: " + this.count);

},

hello: function() {

this.count++;

alert("myfunc hello count: "+ this.count);

}

};

angular.module("LWHTable", [])

.service("myservice", myfunc ) - **service 使用的是函数，没有返回return**

.factory("myfact", function(myservice){

var fact = new myfunc(); - **factory 使用的是函数的实例对象**

return fact; - **factory 必须要使用return**

}).directive("mytext", function (myservice) {

return {

restrict: "E",

replace: true,

scope: {

},

template: [

'<span>Count = {{getCount()}}</span>'

].join(''),

controller: function ($scope, myservice) {

myservice.count++;

console.log("directive controller count: " + myservice.count);

$scope.getCount = function() {

return myservice.count;

}

},

link: function (sc, el, attr) {

}

}

})

angular.module("LWHPhone",["LWHTable"])

.factory("hisFact", function(myservice){

return { count: myservice.count++ };

});

</script>

**我们来看看容易引起误会的例子：**

angular.module("LWHPhone",["LWHTable"])

.factory("hisFact", function(myservice){

return { count: myservice.count++ };

});

我们输出hisFact 的属性值：

$scope.phoneme1 = function() {

alert(hisFact.count);

}

不管输出多少次：永远输出为 0 ， 并不会增加1. 为什么？

1. 因为我们返回的是一个独立的对象且 hisFact = { count : 0 }
2. 如果想实现对 hisFact.count 实现增量1， 代码如下：

$scope.phoneme1 = function() {

alert(hisFact.count++);

}

1. 如果想实现对myservice.count 实现增量，需要修改 factory – hisFact 的代码：

angular.module("LWHPhone",["LWHTable"])

.factory("hisFact", function(myservice){

return {

count: myservice.count++,

getCount: function() { - **这样才能对共享的myservice对象实现修改访问**

return ++myservice.count;

}

};

});

输出代码：

$scope.phoneme1 = function() {

alert(hisFact.getCount());

}

* **$provider – constant, value, service, factory, provider, decorator**

最近，在项目中遇到需要在 config 阶段中注入一些service的情况，然而 factory，service 还有 value 是不能在 config 中注入的，先看一个清单:

| **服务/阶段** | **provider** | **factory** | **service** | **value** | **constant** |
| --- | --- | --- | --- | --- | --- |
| config阶段 | Yes | No | No | No | Yes |
| run 阶段 | Yes | Yes | Yes | Yes | Yes |

注意，provider 在config阶段，注入的时候需要加上 provider 后缀，可以调用非 $get 返回的方法在 run 阶段注入的时候，无需加 provider 后缀，只能调用 $get 返回的方法

* **constant**

**定义常量： 有两种方法： 第一 config($provide) ; 第二 appModule.constant**

var app = angular.module("myApp", ["LWHTable", "ngRoute", "LWHPhone"]);

app.config( function($provide, Rate, UU){

$provide.constant("PI", Rate \* 10);

});

app.constant("Rate", 5.20);

**1）Constant 可以注入到 .config 里， 如上 Rate 可以注入到config 里**

**2）$provide 不能注入到 controller 里**

app.controller("ct11", function ($scope, $provide) { - **出错$provide不能注入到controller**

$provide.constant("PI11", 5699);

}

app.controller("ct11", function ($scope, PI11) { **- 这样注入也出错， PI11Provider unknown**

app.constant("PI11", 5699);

}

**注意：**

1. **Constant 定义在 module , config 里， 不能在controller 里定义**
2. **constant 是常量，会单独复制给不同的controller, 但是他们不是共享的，是各自独立的。**

**常量的值是可以修改的, 在各自独立的作用域里是公共的。**

**看看下面的例子说明：**

var app = angular.module("myApp", ["LWHTable", "ngRoute", "LWHPhone"]);

app.config( function($provide, Rate, UU){

Rate = 2;

$provide.constant("PI", Rate \* 10);

});

app.constant("Rate", 5.20);

app.controller("ct11", function ($scope, PI) {

PI += 20000;

}

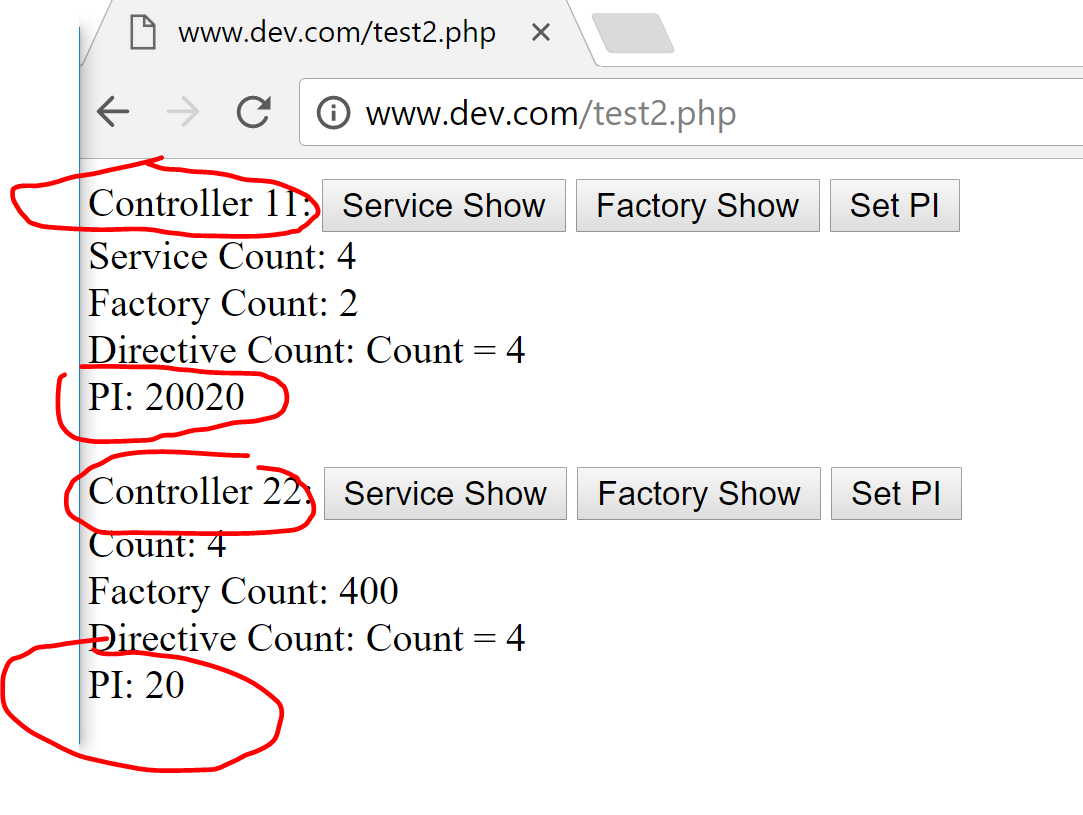
app.controller("ct22", function($scope, PI) {

}

**PI 在config 里的值是： 2 \* 10 = 20**

**PI 在ct11 里的值是： 20 + 20000 = 20020**

**PI 在ct22 里的值是： 20**



* **value**

**定义值：可以有两种方法 1）config($provide) 2) app.value(name, value);**

app.config( function($provide){

$provide.value("PI", 10);

});

app.value("Rate", 5.20);

**value 不能被注入到 config 中**

var app = angular.module("myApp", ["LWHTable", "ngRoute", "LWHPhone"]);

app.config( function($provide, Rate){ - **Rate 出错unknown provider: Rate**

$provide.value("PI", Rate \* 10);

});

app.value("Rate", 5.20);

* **service**

**service 定义： 1) app.service() 2) config(function($provide){})**

.config( function($provide){

$provide.service("myservice", function(){

this.count = 0;

this.show = function() {

this.count++;

console.log("config myservice count: " + this.count);

}

});

});

**具体应用如前面介绍的service 一样**

* **factory:**

.config( function($provide){

$provide.service("myservice", function(){

this.count = 0;

this.show = function() {

this.count++;

console.log("config myservice count: " + this.count);

}

});

$provide.factory("myfact", function(){

var fact = new myfunc();

fact.count += 300;

return fact;

});

});

**具体应用如前面介绍的factory 一样**

* **decorator – 修饰，其实就是对原来的定义进行修改，或者覆盖，有点类似 c# 里的 override**

$provide.decorator("TAX", function($delegate){

$delegate.money = 60000;

$delegate.good = 90000;

return $delegate;

});

$provide.decorator("myservice", function($delegate){

console.log("decorator:");

console.log($delegate);

$delegate.count = 50000;

$delegate.show = function() {

this.count +=1000;

console.log("delete count: " + this.count);

}

return $delegate;

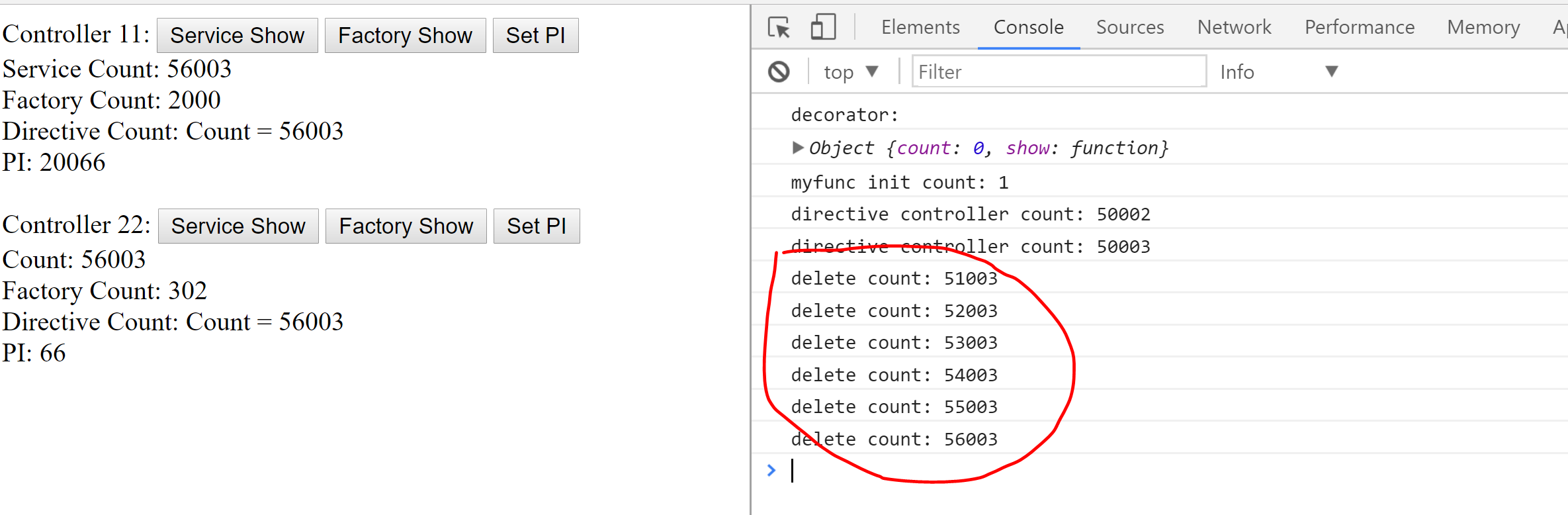
});

app.value("TAX", {money: 5.20, good:900});

$scope.factCount1 = function() {

return TAX.money + TAX.good; - 输出 60000 + 90000 = 150000 , 而不是 5.2 + 900

}



**而myservice 的输出也是使用新的逻辑**

1. **decorator 不能修饰constant之外， 其他的都可以修饰。**
2. **decorator 里的 $delegate 是代表要修饰的对象。 最后要 return $delegate 当然也可以是其他对象。**

$provide.decorator("TAX", function($delegate){

$delegate.money = 60000;

$delegate.good = 90000;

return **2000**;

});

app.value("TAX", {money: 5.20, good:900});

$scope.factCount1 = function() {

return TAX; - **输出 2000**

}

**扩展service, factory 的方法**

$provide.decorator("myservice", function($delegate){

console.log("decorator:");

console.log($delegate);

$delegate.count = 50000;

$delegate.show = function() {

this.count +=1000;

console.log("delete count: " + this.count);

}

$delegate.age = 30; **- 扩展的属性**

$delegate.saygood = function() { ***- 扩展的函数***

this.age += 5;

alert("decorator say good age:" + this.age);

}

return $delegate;

});

调用 myservice.saygood();这些扩展的方法与属性都是共享的共有属性与函数

**我们也可以在 controller 里扩展service, factory 的属性与函数，同名覆盖时，优先级别高过 decorator 的定义。而且是公共共享， 可以被跨过controller 调用访问**

app.controller("ct11", function ($scope, myservice, myfact, hisFact, scalar, PI, UU, TAX)

{

myservice.saygood = function( ) { - **优先级高过 decorator**

alert("Say Good");

}

}

**也可以用这种书写：**

app.decorator("TAX", function($delegate){

$delegate.money = 60000;

$delegate.good = 90000;

return $delegate;

});

app.value("TAX", {money: 5.20, good:900});

* **provider** provider是他们的老大，上面的几乎(除了constant)都是provider的封装，provider必须有一个$get方法，当然也可以说provider是一个可配置的factory

**先理解javascript function 的使用形式：**

1. **作为函数使用：**

var CC = function(a) {

this.age = 100 + a;

this.good = "good morning";

var gg = "News";

};

var lwh = CC(); **- 函数没有返回值， 所以 lwh = undefined**

console.log(lwh);

var CC = function(a) {

this.age = 100 + a;

this.good = "good morning";

var gg = "News";

return {

a: 100,

b: 200

}

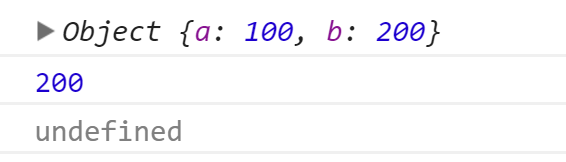
};

var lwh = CC(); **- 没有new**

console.log(lwh); **- 返回 { a: 100, b: 200 }**

console.log(lwh.b);

console.log(lwh.age); **- undefined 不在返回值里，虽然使用this**



var CC = function(a) {

this.age = 100 + a;

this.good = "good morning";

var gg = "News";

return this;

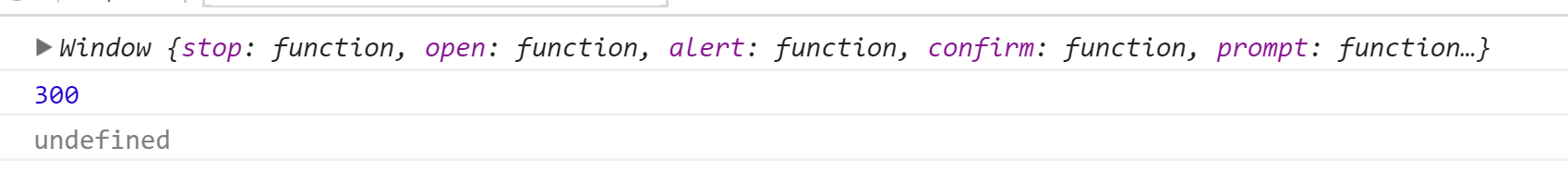
};

var lwh = CC(200); - **返回的是window 对象**

console.log(lwh);

console.log(lwh.age)

console.log(lwh.gg)



1. **对函数使用 new 返回一个对象**

var CC = function(a) {

this.age = 100 + a;

this.good = "good morning";

var gg = "News";

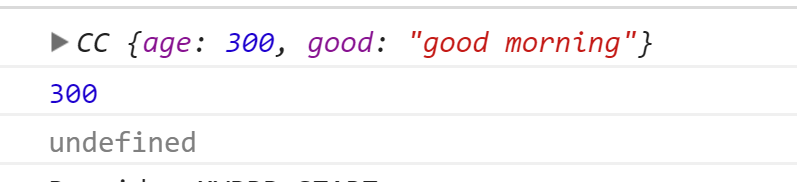
};

var lwh = new CC(200);

console.log(lwh); - 所有this的属性都是对象的属性，

console.log(lwh.age)

console.log(lwh.gg) – **var 的作用范围是function, 出了function 就不能访问**



var CC = function(a) {

this.age = 100 + a;

this.good = "good morning";

var gg = "News";

return {

a: 7000,

b: "Youtube"

}

};

var lwh = new CC(200);

console.log(lwh); - 返回 { a: 7000, b: “youtube” }

console.log(lwh.a);

console.log(lwh.age); - **undefined**

console.log(lwh.gg); - **undefined**

**如果使用return ， 则只接受 return 的对象**

**书写形式：**

var BB = function() {

this.ta = 3000;

this.sz = "Hello BB";

}

var CC = function(a) {

this.age = 100 + a;

this.good = "good morning";

var gg = "News";

var dd = new BB();

return dd;

};

1. **Function(){ xxx; return this; } - 不使用 new 直接执行函数，函数里又直接返回 this.**

**此时的 this 是指 window 对象， 因为函数的定义都是放在 window 对象里的。**

console.log( (function(){ this.a = 2000; return this; })() );

1. **复杂的嵌套function**

var CC = function(a) {

this.a\_val = 10 + a;

var \_self = this;

console.log(this);

return function() {

this.dodo = 10000 + \_self.a\_val;

this.zero = "Victoria Day";

return this;

}

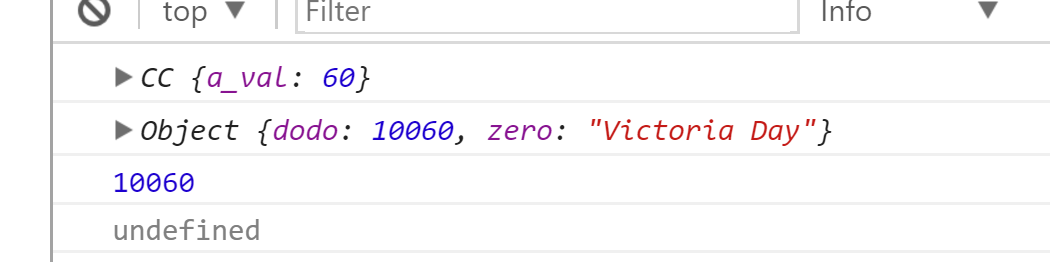
};

var lwh = new (new CC(50))();

console.log(lwh);

console.log(lwh.dodo);

console.log(lwh.a\_val); **- undefined**



var CC = function(a) {

this.a\_val = 10 + a;

console.log(this);

return function(p) {

this.dodo = 10000 + p;

this.zero = "Victoria Day";

return this;

}(1200);

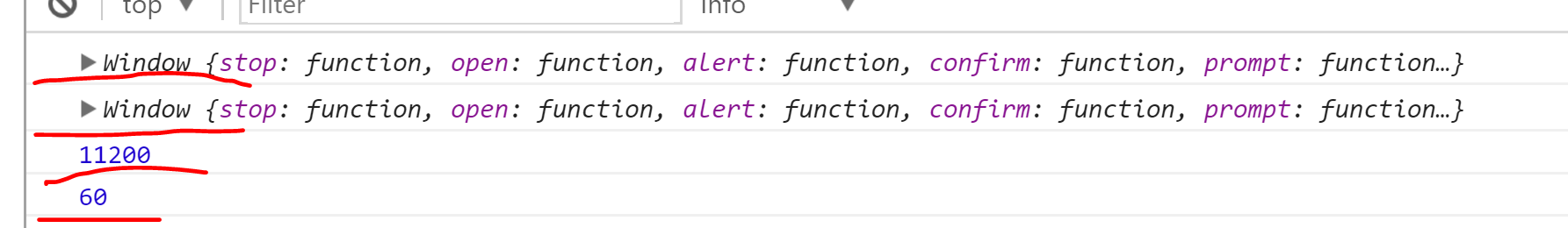
};

var lwh = CC(50);

console.log(lwh);

console.log(lwh.dodo);

console.log(lwh.a\_val);



**由于都是在 window 对象下， 所以 lwh.dodo lwh.a\_val 都有效。**

var CC = function(a) {

this.a\_val = 10 + a;

console.log(this);

return new function(p) {

this.dodo = 10000 + p;

this.zero = "Victoria Day";

}(1200);

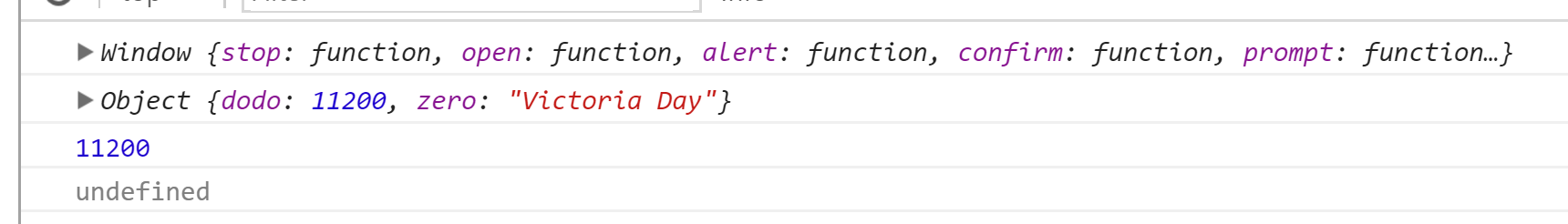
};

var lwh = CC(50);

console.log(lwh);

console.log(lwh.dodo);

console.log(lwh.a\_val);



**我们可以看到 lwh 实际返回的是 对象 { dodo: 11200, zero: “Victoria Day”}**

* **provider – 对function的以上理解， 我们才能更好使用provider**

1. 当你使用Provider创建一个service时，唯一的可以在你的控制器中访问的属性和方法是通过$get()函数返回内容。
2. config中只能注入供应商（两个例外是$provide和$injector），自定义的供应商用驼峰命名法写成 xxxxProvider，必须记得在config 中为自定义的供应商名字加上Provider, 这是Angular命名约定， Angular就会帮你注入它的供应商。
3. provider是他们的老大，上面的几乎(除了constant)都是provider的封装，provider必须有一个$get方法，当然也可以说provider是一个可配置的factory
4. **$get 方法所返回的数据，就是将来注入到控制器中此供应商所能提供的属性与方法。**
5. **执行顺序：**
   * **Provider definition -> app.provider**
   * **App Config -> app.config**
   * **如果controller 引用了provider 则执行 provider $get 并把return 的数值赋予自定义的provider**
   * **然后就是 controller function(){ …}**
6. provider 是一个service, provider $get是一个可配置的factory, 如何理解呢？

**定义provider: app.provider(“SQL”, function() { });**

angular.module("MYProvider", [])

.provider("SQL",

function(){

console.log("Provider SQL Definition");

var dd = "Provider Can you see me ??";

this.cc = "First Connection!!";

this.$get = function(){

console.log("Provider $get");

return function(param){

this.connect = "SQL Connection: " + param;

this.call = function() { alert( this.connect); }

};

}

}

);

</script>

<script>

var app = angular.module("myApp", ["ngRoute", "MYProvider"]);

app.config( function($provide, SQLProvider){

console.log("App Config Init");

console.log( SQLProvider );

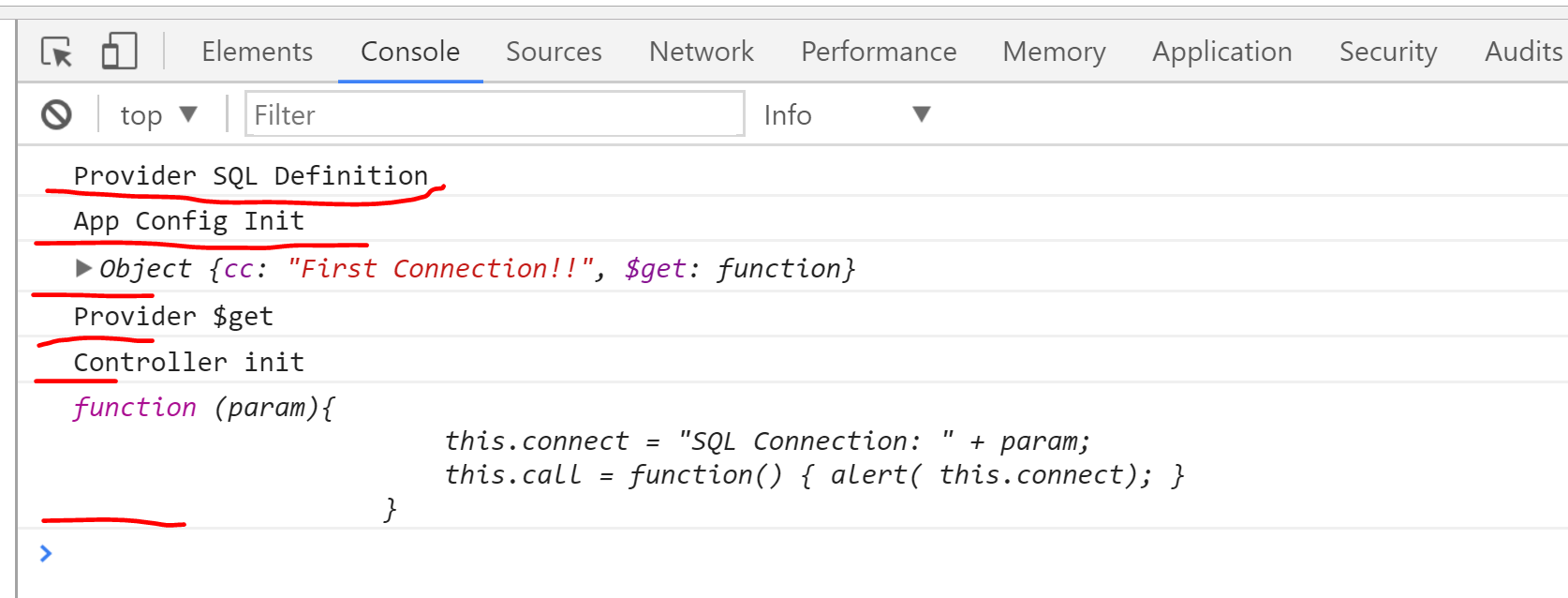
});

app.controller("ct11", function ($scope, SQL) {

console.log("Controller init");

console.log(SQL);

});



**我们看到， function() { this.cc , this.$get }**

**在app.config 里返回给 SQLProvider 是自动经过 new 实例化的对象 {cc: "First Connection!!", $get: function},**

angular.module("MYProvider", [])

.provider("SQL",

function(){

console.log("Provider SQL Definition");

var dd = "Provider Can you see me ??";

this.cc = "First Connection!!";

this.$get = function(){

console.log("Provider $get");

return function(param){

this.connect = "SQL Connection: " + param;

this.call = function() { alert( this.connect); }

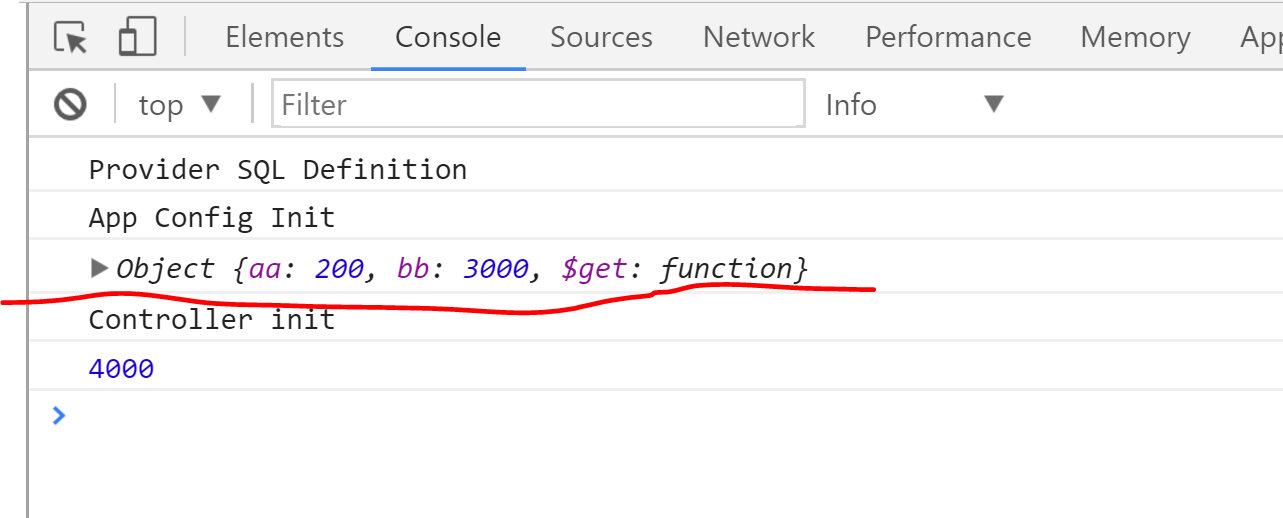
};

}

return { aa: 200, bb: 3000, $get: function(){ return 4000; } };

}

);



**如果返回新的对象， 则看到app.config SQLProvider对象如上：{aa: bb: $get: function }**

**在看看如果返回的是函数**

angular.module("MYProvider", [])

.provider("SQL",

function(){

console.log("Provider SQL Definition");

var dd = "Provider Can you see me ??";

this.cc = "First Connection!!";

this.$get = function(){

console.log("Provider $get");

return function(param){

this.connect = "SQL Connection: " + param;

this.call = function() { alert( this.connect); }

};

}

//return { aa: 200, bb: 3000, $get: function(){ return 4000; } };

return function() {

this.aaa = 1000;

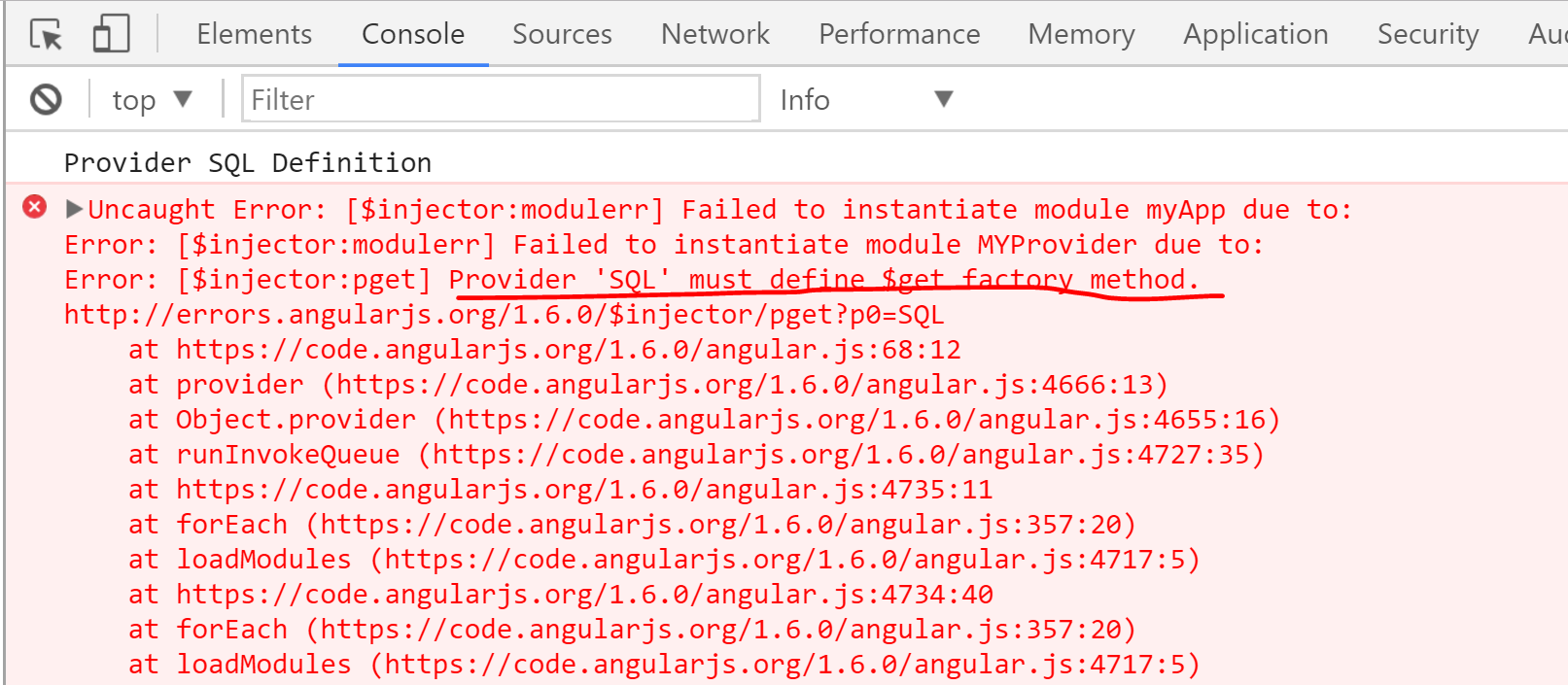
this.bbb = 2000;

this.$get = function() {return 6000};

};

}

);



**根据sevice 的原理，service函数里的返回值是 function(){} ，并不是一个对象， 所以就没有 $get 这个方法 , 类似以下情况：**

var CCC = function() {

this.aa = 100;

return function() {

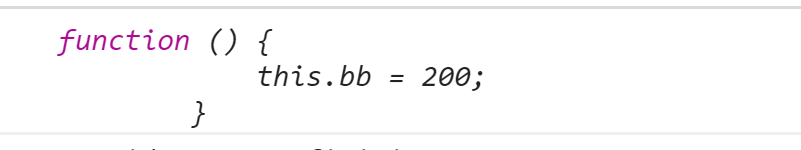
this.bb = 200;

};

};

var ccc = new CCC();

console.log(ccc);



**实际不是对象， 而是一个函数, 可以再进一步使用**

var ccc = new CCC(); - **ccc = function(){this.bb=200};**

var ddd = new ccc();

console.log(ddd); - **返回 { bb: 200 }**

**修改方案：**

angular.module("MYProvider", [])

.provider("SQL",

function(){

console.log("Provider SQL Definition");

var dd = "Provider Can you see me ??";

this.cc = "First Connection!!";

this.$get = function(){

console.log("Provider $get");

return function(param){

this.connect = "SQL Connection: " + param;

this.call = function() { alert( this.connect); }

};

}

return new function() {

this.aaa = 1000;

this.bbb = 2000;

this.$get = function() {return 6000};

}();

}

);

var app = angular.module("myApp", ["ngRoute", "MYProvider"]);

app.config( function($provide, SQLProvider){

console.log("App Config Init");

console.log( SQLProvider );

});

**SQLProvider: Object {aaa: 1000, bbb: 2000, $get: function}**

**$get: function – 返回的对象必须有**

**引用 provider – 就是引用 $get 返回的对象， 可以是任何类型.**

**例子一： 返回的是string**

angular.module("MYProvider", [])

.provider("SQL",

function(){

console.log("Provider SQL Definition");

var dd = "Provider Can you see me ??";

this.cc = "First Connection!!";

this.$get = function(){

console.log("Provider $get method");

return "GET : " + this.cc;

}

}

);

var app = angular.module("myApp", ["ngRoute", "MYProvider"]);

app.config( function($provide, SQLProvider){

SQLProvider.cc = "Change to Config";

console.log("App Config Init");

console.log( SQLProvider );

});

app.controller("ct11", function ($scope, SQL) {

console.log("Controller init");

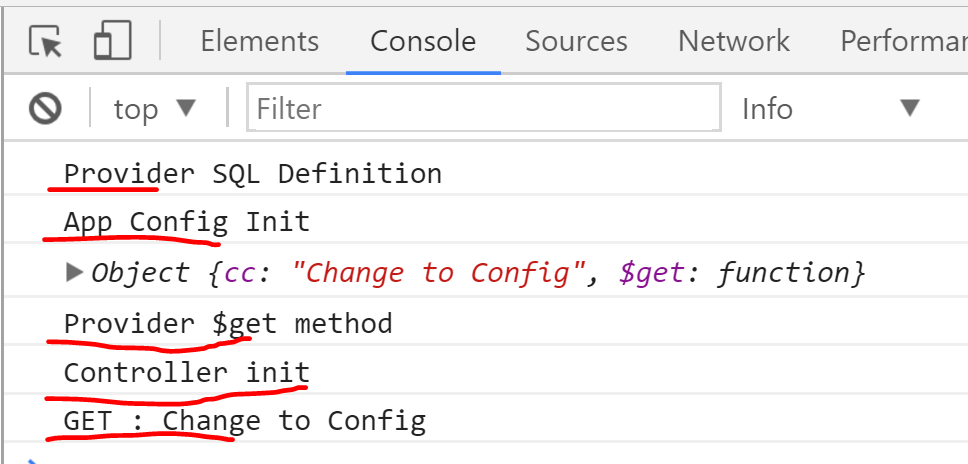
console.log(SQL);

$scope.call1 = function() {

alert(SQL);

}

});



## 从上面例子可以看到 controller SQL 的值是字符串: “GET: Change to Config”

## 例子二： 返回对象{ }

angular.module("MYProvider", [])

.provider("SQL",

function(){

console.log("Provider SQL Definition");

var dd = "Provider Can you see me ??";

this.cc = "First Connection!!";

this.$get = function(){

return {

connect : "SQL Connection: " + this.cc,

call: function() { alert( this.connect ) }

}

}

}

);

var app = angular.module("myApp", ["ngRoute", "MYProvider"]);

app.config( function($provide, SQLProvider){

SQLProvider.cc = "Change to Config"; - 通过配置修改供应商的值

console.log("App Config Init");

console.log( SQLProvider );

});

app.controller("ct11", function ($scope, SQL) {

console.log("Controller init");

console.log(SQL);

$scope.call1 = function() {

SQL.connect += " from controller"; **-直接修改属性值 connect, 注意SQL.cc 不是对象属性**

SQL.call();

}

});

## 

## 使用函数构造器：

angular.module("MYProvider", [])

.provider("SQL",

function(){

console.log("Provider SQL Definition");

var dd = "Provider Can you see me ??";

this.cc = "First Connection!!";

this.$get = function(){

return new function(a) {

this.connect = "SQL Connection: " + a,

this.call = function() { alert( this.connect ) }

}(this.cc)

}

}

);

## 例子三： 返回函数

angular.module("MYProvider", [])

.provider("SQL",

function(){

console.log("Provider SQL Definition");

var dd = "Provider Can you see me ??";

this.cc = "First Connection!!";

this.$get = function(){

return function(a) {

this.connect = "SQL Connection: " + this.cc + a,

this.call = function() { alert( this.connect ) }

}

}

}

);

## 注意：this.cc = undefined this是指函数本身

var \_self = this;

this.$get = function(){

return function(a) {

this.connect = "SQL Connection: " + \_self.cc + a,

this.call = function() { alert( this.connect ) }

}

}

## 正确的做法是： var \_self = this; 函数里使用 \_sefl.cc

app.controller("ct11", function ($scope, SQL) {

console.log("Controller init");

console.log(SQL);

$scope.call1 = function() {

var tt = new SQL(" from HELLO");

tt.call();

}

});

## 

## 

## 使用函数内全局变量var + 函数构造器

angular.module("MYProvider", [])

.provider("SQL",

function(){

var title = "First Connection!!";

this.set = function(tt) {

title = tt;

}

this.$get = function(){

return new function(a) {

this.connect = "SQL Connection: " + a,

this.call = function() { alert( this.connect ) }

}(title);

}

}

);

app.controller("ct11", function ($scope, SQL) {

$scope.call1 = function() {

SQL.call();

}

});

## provider 里的注入：

angular.module("MYWAY", [])

.value("PORT", 2000)

.service("EXECUTE", function(){ this.batch = "SQLBatch"; });

angular.module("MYProvider", ["MYWAY"])

.constant("CONSTR", "192.168.1.200")

.provider("SQL",

function(CONSTR){ **- 这里只能注入provider, 跟config里允许的注入一样**

var title = CONSTR;

this.set = function(tt) {

title += ":" + tt;

}

this.$get = function($http, PORT, EXECUTE){ **- 这里可以注入**

return new function(a) {

this.connect = "SQL Connection: " + PORT + " => " +

EXECUTE.batch + " -> " + a,

this.call = function() { alert( this.connect ) }

}(title);

}

}

);

## 可以使用config $provide定义provider

angular.module("MYWAY", [])

.value("PORT", 2000)

.service("EXECUTE", function(){ this.batch = "SQLBatch"; });

angular.module("MYProvider", ["MYWAY"])

.constant("CONSTR", "192.168.1.200")

.config( function($provide, CONSTR){ - **可以在这里注入CONSTR**

**$provide.provider**("SQL", function(CONSTR){ - **也可以在这里注入CONSTR**

var title = CONSTR;

this.set = function(tt) {

title += ":" + tt;

}

this.$get = function($http, PORT, EXECUTE){

return new function(a) {

this.connect = "SQL Connection: "+PORT+"=>"+EXECUTE.batch+"->" + a,

this.call = function() { alert( this.connect ) }

}(title);

}

})

});

## Provider 可以被修饰：

app.decorator("SQL", function($delegate){

$delegate = { call : function() { alert("Overwrite"); } }

return $delegate;

});

## 

## 总结

可以看出，从 service 到 factory 再到 provider，复杂程度依次递增，那么问题来了。

### 什么时候使用 provider 而不用 factory ？

provider 是 factory 的加强版。当需要一个可配置的 factory 的时候，使用 provider。

简单介绍一下 AngularJS 运行应用的过程，分两个阶段，config 阶段和 run 阶段。config 阶段是设置任何的 provider 的阶段。也是设置任何的指令，控制器，过滤器以及其它东西的阶段。在 run 阶段，AngularJS 会编译你的 DOM 并启动应用。

**引言**

看了很多文章可能还是不太说得出AngularJS中的几个创建供应商(provider)的方法(factory(),service(),provider())到底有啥区别，啥时候该用啥，之前一直傻傻分不清楚，现在来总结一下。

下文中泛指统一用中文，英文即为特指$provide方法中对应方法创建出的东东

供应商==>泛指provider

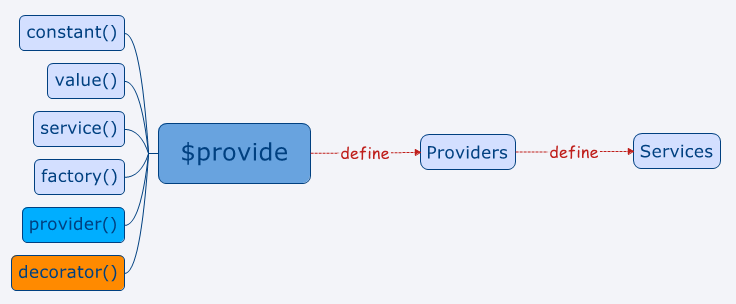
服务==>泛指service

provider==>provider()方法创建的东东

service==>service()方法创建的东东

**先说说供应商（$provide）**

$provide服务负责告诉Angular如何创造一个新的可注入的东西：即服务。服务会被叫做供应商的东西来定义，你可以使用$provide来创建一个供应商。你需要使用$provide中的provider()方法来定义一个供应商，同时你也可以通过要求$provide被注入到一个应用的config函数中来获得$provide服务。

上面的描述是官方wiki的翻译版，如果有些绕的话，看下这张图：  


* $provide是一个服务，在Auto模块中
* 这个服务下面有好多方法，是用来定义供应商
* 而供应商是用来提供服务的，被注入来注入去的东西就是供应商们提供的服务了

下面是一个例子：

myMod.config(function($provide) {

$provide.provider('greeting', function() {

this.$get = function() {

return function(name) {

alert("Hello, " + name);

};

};

});

});

这个例子可以说是最终形态，先大概看下

**定义供应商的方法们**

AngularJS用$provide去定义一个供应商,这个$provide有5个用来创建供应商的方法：

* constant
* value
* service
* factory
* provider
* decorator ***我没有说我也是，我只是路过o(╯□╰)o***

**Constant**

定义常量用的，这货定义的值当然就不能被改变，它可以被注入到任何地方，但是不能被装饰器(decorator)装饰

var app = angular.module('app', []);

app.config(function ($provide) {

$provide.constant('movieTitle', 'The Matrix');

});

app.controller('ctrl', function (movieTitle) {

expect(movieTitle).toEqual('The Matrix');

});

语法糖：

app.constant('movieTitle', 'The Matrix');

**Value**

这货可以是string,number甚至function,它和constant的不同之处在于，它可以被修改，不能被注入到config中，但是它可以被decorator装饰

var app = angular.module('app', []);

app.config(function ($provide) {

$provide.value('movieTitle', 'The Matrix')

});

app.controller('ctrl', function (movieTitle) {

expect(movieTitle).toEqual('The Matrix');

})

语法糖：

app.value('movieTitle', 'The Matrix');

**Service**

它是一个可注入的构造器，在AngularJS中它是单例的，用它在Controller中通信或者共享数据都很合适

var app = angular.module('app' ,[]);

app.config(function ($provide) {

$provide.service('movie', function () {

this.title = 'The Matrix';

});

});

app.controller('ctrl', function (movie) {

expect(movie.title).toEqual('The Matrix');

});

语法糖：

app.service('movie', function () {

this.title = 'The Matrix';

});

在service里面可以不用返回东西，因为AngularJS会调用new关键字来创建对象。但是返回一个自定义对象好像也不会出错。

**Factory**

它是一个可注入的function，它和service的区别就是：factory是普通function，而service是一个构造器(constructor)，这样Angular在调用service时会用new关键字，而调用factory时只是调用普通的function，所以factory可以返回任何东西，而service可以不返回(可查阅new关键字的作用)

var app = angular.module('app', []);

app.config(function ($provide) {

$provide.factory('movie', function () {

return {

title: 'The Matrix'

}

});

});

app.controller('ctrl', function (movie) {

expect(movie.title).toEqual('The Matrix');

});

语法糖：

app.factory('movie', function () {

return {

title: 'The Matrix'

}

});

factory可以返回任何东西，它实际上是一个只有$get方法的provider

**Provider**

provider是他们的老大，上面的几乎(除了constant)都是provider的封装，provider必须有一个$get方法，当然也可以说provider是一个可配置的factory

var app = angular.module('app', []);

app.provider('movie', function () {

var version;

return {

setVersion: function (value) {

version = value;

},

$get: function () {

return {

title: 'The Matrix' + ' ' + version

}

}

}

});

app.config(function (movieProvider) {

movieProvider.setVersion('Reloaded');

});

app.controller('ctrl', function (movie) {

expect(movie.title).toEqual('The Matrix Reloaded');

});

注意这里config方法注入的是movieProvider，上面定义了一个供应商叫movie，但是注入到config中不能直接写movie，因为前文讲了注入的那个东西就是服务，是供应商提供出来的，而config中又只能注入供应商（两个例外是$provide和$injector），所以用驼峰命名法写成movieProvider，Angular就会帮你注入它的供应商。（更详细可参考文末官方wiki翻译版中的配置provider）

**Decorator**

这个比较特殊，它不是provider,它是用来装饰其他provider的，而前面也说过，他不能装饰Constant，因为实际上Constant不是通过provider()方法创建的。

var app = angular.module('app', []);

app.value('movieTitle', 'The Matrix');

app.config(function ($provide) {

$provide.decorator('movieTitle', function ($delegate) {

return $delegate + ' - starring Keanu Reeves';

});

});

app.controller('myController', function (movieTitle) {

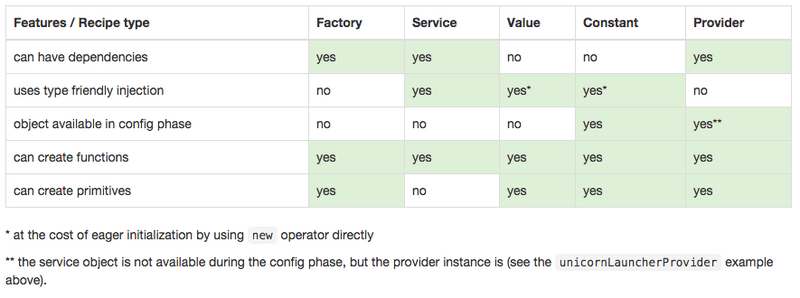
expect(movieTitle).toEqual('The Matrix - starring Keanu Reeves');

});

**总结**

* 所有的供应商都只被实例化一次，也就说他们都是单例的
* 除了constant，所有的供应商都可以被装饰器(decorator)装饰
* value就是一个简单的可注入的值
* service是一个可注入的构造器
* factory是一个可注入的方法
* decorator可以修改或封装其他的供应商，当然除了constant
* provider是一个可配置的factory

最后来看一张对比图：



文章出处

<https://segmentfault.com/a/1190000003096933>

# [深究AngularJS——自定义服务详解(factory、service、provider)](http://blog.csdn.net/zcl_love_wx/article/details/51404390)

标签： [angularjs](http://www.csdn.net/tag/angularjs)[factory](http://www.csdn.net/tag/factory)[service](http://www.csdn.net/tag/service)[provider](http://www.csdn.net/tag/provider)

2016-05-14 09:56 11659人阅读 [评论](http://blog.csdn.net/zcl_love_wx/article/details/51404390#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zcl_love_wx/article/details/51404390#report)

http://static.blog.csdn.net/images/category_icon.jpg分类：

AngularJS（17） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：博主地址：http://blog.csdn.net/zcl\_love\_wx

目录[(?)[+]](http://blog.csdn.net/zcl_love_wx/article/details/51404390)

# 前言

1. 3种创建自定义服务的方式。
   1. Factory
   2. Service
   3. Provider
2. 大家应该知道，AngularJS是后台人员在工作之余发明的，他主要应用了后台早就存在的分层思想。所以我们得了解下分层的作用，如果你是前端人员不了解什么是分层，那么你最好问问你后台的小伙伴。   
   **dao层**：就是Model层，在后台时，这一层的作用，就要是写与数据库交互数据的一层，在angularJS里就主要是写ajax的。   
   **service层**：主查写逻辑代码的，但在angularJS里也可以持久化数据（充当数据容器），以供不同的controller高用。   
   **controller层**：即控制层，在angularJS里就是写控制器的。控制器里尽量不要写那些不必要的逻辑，尽量写在service层里。   
   所以，就有了创建自定义服务的三种方式。

# ****factory****

factory方式创建的服务，作用就是返回一个有属性有方法的对象。相当于：var f = myFactory();

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<script src="http://apps.bdimg.com/libs/angular.js/1.4.6/angular.min.js"></script>

</head>

<body>

<div ng-app="myApp" ng-controller="myCtrl">

<p>{{r}}</p>

</div>

<script>

//创建模型

var app = angular.module('myApp', []);

//通过工厂模式创建自定义服务

app.factory('myFactory', function() {

var service = {};//定义一个Object对象'

service.name = "张三";

var age;//定义一个私有化的变量

//对私有属性写getter和setter方法

service.setAge = function(newAge){

age = newAge;

}

service.getAge = function(){

return age;

}

return service;//返回这个Object对象

});

//创建控制器

app.controller('myCtrl', function($scope, myFactory) {

myFactory.setAge(20);

$scope.r =myFactory.getAge();

alert(myFactory.name);

});

</script>

</body>

</html>

在自定义服务里**注入服务示例**，但不能注入$scope作用域对象。

<script>

var app = angular.module('myApp', []);

app.factory('myFactory', function($http,$q) {

var service = {};

service.name = "张三";

//请求数据

service.getData = function(){

var d = $q.defer();

$http.get("url")//读取数据的函数。

.success(function(response) {

d.resolve(response);

})

.error(function(){

d.reject("error");

});

return d.promise;

}

return service;

});

app.controller('myCtrl', function($scope, myFactory) {

//alert(myFactory.name);

myFactory.getData().then(function(data){

console.log(data);//正确时走这儿

},function(data){

alert(data)//错误时走这儿

});;

});

</script>

# ****service****

通过service方式创建自定义服务，相当于new的一个对象：var s = new myService();，只要把属性和方法添加到this上才可以在controller里调用。

<div ng-app="myApp" ng-controller="myCtrl">

<h1>{{r}}</h1>

</div>

<script>

var app = angular.module('myApp', []);

app.service('myService', function($http,$q) {

this.name = "service";

this.myFunc = function (x) {

return x.toString(16);//转16进制

}

this.getData = function(){

var d = $q.defer();

$http.get("ursl")//读取数据的函数。

.success(function(response) {

d.resolve(response);

})

.error(function(){

alert(0)

d.reject("error");

});

return d.promise;

}

});

app.controller('myCtrl', function($scope, myService) {

$scope.r = myService.myFunc(255);

myService.getData().then(function(data){

console.log(data);//正确时走这儿

},function(data){

alert(data)//错误时走这儿

});

});

</script>

# ****3.provder****

只有provder是能传 .config() 函数的 service。如果想在 service 对象启用之前，先进行模块范围的配置，那就应该选择 provider。需要注意的是：在config函数里注入provider时，名字应该是：providerName+Provider.   
使用Provider的优点就是，你可以在Provider对象传递到应用程序的其他部分之前在app.config函数中对其进行修改。   
当你使用Provider创建一个service时，唯一的可以在你的控制器中访问的属性和方法是通过$get()函数返回内容。

<body>

<div ng-app="myApp" ng-controller="myCtrl">

</div>

<script>

var app = angular.module('myApp', []);

//需要注意的是：在注入provider时，名字应该是：providerName+Provider

app.config(function(myProviderProvider){

myProviderProvider.setName("大圣");

});

app.provider('myProvider', function() {

var name="";

var test={"a":1,"b":2};

//注意的是，setter方法必须是(set+变量首字母大写)格式

this.setName = function(newName){

name = newName

}

this.$get =function($http,$q){

return {

getData : function(){

var d = $q.defer();

$http.get("url")//读取数据的函数。

.success(function(response) {

d.resolve(response);

})

.error(function(){

d.reject("error");

});

return d.promise;

},

"lastName":name,

"test":test

}

}

});

app.controller('myCtrl', function($scope,myProvider) {

alert(myProvider.lastName);

alert(myProvider.test.a)

myProvider.getData().then(function(data){

//alert(data)

},function(data){

//alert(data)

});

});

</script>

</body>

# ****4.在过滤器中注入自定义服务****

<body>

<div ng-app="myApp">

在过滤器中使用服务:

<h1>{{255 | myFormat}}</h1>

</div>

<script>

var app = angular.module('myApp', []);

app.service('hexafy', function() {

this.myFunc = function (x) {

return x.toString(16);

}

});

app.filter('myFormat',['hexafy', function(hexafy) {

return function(x) {

return hexafy.myFunc(x);

};

}]);

</script>

</body>

<http://blog.csdn.net/zcl_love_wx/article/details/51404390>