

call 和 apply

```
1 function show(a,b){
2     alert('this: '+this+'\n'+
3         'a: '+a+'\n'+
4         'b: '+b
5     );
6 }
7 //show(12,5); // this: window a: 12 b:
8 show.call('abc',12,5); // this:abc a: 12 b: 5
9
10 function show(a,b){
11     alert('this: '+this+'\n'+
12         'a: '+a+'\n'+
13         'b: '+b
14     );
15 }
16 show.apply('abc',[12,5]); // this: abc a: 12 b : 5
```

继承

```
1 function Person(name,age){
2     this.name='^_'+name;
3     this.age=age;
4 }
5 Person.prototype.showName=function(){
6     return this.name;
7 };
8 Person.prototype.showAge=function(){
9     return this.age;
10 };
11
```

```

12 //=====
13 function Worker(name,age,job){
14     Person.apply(this,arguments); // 属性继承
15
16     this.job=job;
17 };
18 Worker.prototype=Person.prototype; // 方法继承 1
19 Worker.prototype.showAge=function(){
20     return this.name;
21 };
22 Worker.prototype.showJob=function(){
23     return this.job;
24 };
25
26 var p1=new Person('abc',30);
27 var w1=new Worker('strive',16,'打杂的');
28

```

```

1 function Person(name,age){
2     this.name='^_'+name;
3     this.age=age;
4 }
5 Person.prototype.showName=function(){
6     return this.name;
7 };
8 Person.prototype.showAge=function(){
9     return this.age;
10 };
11
12 //=====
13 function Worker(name,age,job){
14     Person.apply(this,arguments);
15
16     this.job=job;
17 };
18
19 for(var name in Person.prototype){
20     Worker.prototype[name]=Person.prototype[name]; // 方法继承。拷贝出来
21 }
22

```

```
23 Worker.prototype.showJob=function(){
24     return this.job;
25 };
26
27 var p1=new Person('abc',30);
28 var w1=new Worker('strive',16,'打杂的');
```

```
1 function Person(name,age){
2     this.name='^_'+name;
3     this.age=age;
4 }
5 Person.prototype.showName=function(){
6     return this.name;
7 };
8 Person.prototype.showAge=function(){
9     return this.age;
10 };
11
12 //=====
13 function Worker(name,age,job){
14     Person.apply(this,arguments);
15
16     this.job=job;
17 };
18
19 Worker.prototype=new Person(); // 实例继承
20 Worker.prototype.constructor=Worker;
21
22 Worker.prototype.showJob=function(){
23     return this.job;
24 };
25
26 var p1=new Person('abc',30);
27 var w1=new Worker('strive',16,'打杂的');
28
29 alert(w1 instanceof Person); // true
30 alert(w1.constructor==Worker); // true
```

箭头函数

```
1 (function (param) {  
2     console.log(arguments)  
3 })(1) // 1  
4  
5 ((param) => {  
6     console.log(arguments)  
7 })(1) // Uncaught ReferenceError: arguments is not defined  
8
```

箭头函数并不暴露 `arguments` 对象.在箭头函数中使用`arguments`指向的是其外层的`arguments`对象

箭头函数的`this`指向的对象和其外层函数是一致的

```
1 function Person(){  
2     this.age = 0;  
3  
4     setInterval(() => {  
5         this.age++; // 现在|this|和上下文的this一致了  
6     }, 1000);  
7 }  
8  
9 var p = new Person();
```

```
1 (x)=>{ return x * x }
2 如果箭头函数只有一个输入的话，可以把前面的括号省了，于是它变成了
3 x=>{ return x * x }
4 如果只有一个返回值，可以连后面的return和大括号都省了，变成
5 x=> x * x
6 于是现在它神似python里的lambda，我们可以这样用它啦：
7 [1, 2, 3].map( x => x * x ) // [1, 4, 9]
```

Promise的含义

<https://juejin.im/entry/56c46015c24aa800528da4d5>

```
1 下面是一个Promise对象的简单例子。
2 function timeout(ms) {
3   return new Promise((resolve, reject) => {
4     setTimeout(resolve, ms, 'done');
5   });
6 }
7
8 timeout(100).then((value) => {
9   console.log(value);
10 });
11
```

Promise是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大

Promise 对象有以下两个特点

(1) 对象的状态不受外界影响。**Promise** 对象代表一个异步操作，有三种状态：**Pending**（进行中）、**Resolved**（已完成，又称Fulfilled）和**Rejected**（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是**Promise**这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

(2) 一旦状态改变，就不会再变，任何时候都可以得到这个结果。**Promise** 对象的状态改变，只有两种可能：从**Pending** 变为 **Resolved** 和从 **Pending** 变为 **Rejected**。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对 **Promise** 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了 `Promise` 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，`Promise` 对象提供统一的接口，使得控制异步操作更加容易。

`Promise` 也有一些缺点。首先，无法取消 `Promise`，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，`Promise` 内部抛出的错误，不会反应到外部。第三，当处于 `Pending` 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

如果某些事件不断地反复发生，一般来说，使用stream模式是比部署 `Promise` 更好的选择。

flex水平居中

```
1 <div id='container'>
2   <div id = 'center'>#center</div>
3 </div>
4 // 父容器
5 #container{
6   height:300px;
7   background:#d6d6d6;
8   display:flex;
9   justify-content:center;
10  align-items: center;
11 }
12 // 子
13 #center{
14   width:100px;
15   height:100px;
16   background-color:#666;
17 }
```

闭包和原型链

作用域：

全局变量：在哪里都能使用，生存周期长，直到浏览器关闭，占资源

局部变量：只能在函数内部使用，生存周期短，几乎不占

闭包：子函数可以使用父函数局部变量，如果子函数得不到释放，整条作用域链上面局部变量都会保留

作用域链:子函数如果使用了一个变量, 先会找自身, 自己父级, 父级的父级...直到到全局, 还没有, 报错

对象组成:

1. 属性 特征 变量

变量是自由的, 属性是有所属者

2. 方法 行为(功能) 函数

函数是自由的, 方法是有所属者

原型: prototype 给一类物体去添加东西

1. 属性 放到构造函数里面

2. 方法 挂到原型上

```
1 // 构造函数
2 //工厂模式
3 function CreatePerson(name,age){
4     //如果外面加了new, 系统会自动帮你创建一个空白对象, 并且赋值给this
5     //var this=new Object(); //空白对象
6     //加工
7     //添加属性
8     this.name=name;
9     this.age=age;
10    //添加方法
11    this.showName=function(){
12        return this.name;
13    };
14    this.showAge=function(){
15        return this.age;
16    };
17
18    //如果外面加了new, 系统会自动帮你返回this
19    //return obj;
20 }
21
22 var p1=new CreatePerson('strive',16);
23 var p2=new CreatePerson('blue',48);
24
25 alert(p2.showAge());
```

```

1 //构造函数/原型混合模式 constructor/prototype hybrid mode
2 function CreatePerson(name,age){
3     //如果外面加了new, 系统会自动帮你创建一个空白对象, 并且赋值给this
4     //var this=new Object(); //空白对象
5     //加工
6     //添加属性
7     this.name=name;
8     this.age=age;
9
10    //如果外面加了new, 系统会自动帮你返回this
11    //return obj;
12 }
13
14 CreatePerson.prototype.showName=function(){
15     return this.name;
16 };
17 CreatePerson.prototype.showAge=function(){
18     return this.age;
19 };
20
21 var p1=new CreatePerson('strive',16);
22 var p2=new CreatePerson('blue',48);
23
24 alert(p2.showAge==p1.showAge);

```

如何判断Array是构造函数

JS中判断一个对象的类型时, 通常使用typeof, 这时候问题就来了, 因为typeof()辨别数组的时候返回的是object,所以JS

中判断一个对象是不是数组需要一些特殊的处理方式

```

1 function isArray(obj){
2     if(Array.isArray){
3         return Array.isArray(obj);
4     }else{
5         return Object.prototype.toString.call(obj)=="[object Array]";
6     }

```



```
7 }
```

(1) 方法一：利用toString方法

通过调用toString()方法试着将该变量转化为代表其类型的string。该方法对于真正的array可行；参数对象转化为string时

返回[object Arguments]会转化失败；此外，对于含有数字长度属性的object类也会转化失败。

方法如下

```
1 <script>
2     function isArrayOne(arr){
3         return <span style="color:#cc0000;">Object.prototype.toString.call(arr) ===
4     }
5     var obj = {"k1":"v1"};
6     var arr = [1,2];
7     console.log("对象的结果："+isArrayOne(obj));
8     console.log("数组的结果："+isArrayOne(arr));
9 </script>
```

top ▼ ☐ Preserve log

对象的结果: false

数组的结果: true

(2) 方法二：通过isArray

ECMAScript 5 的方法。不兼容IE9

```
1 Array.isArray(obj);    //obj是待检测的对象
```

返回true或false，如果为true则为数组

(3) 方法三：通过instanceof运算符来判断，

注意：instanceof运算符左边是子对象（待测对象），右边是父构造函数（这里是Array），


即：子对象 instanceof 父构造函数

instance: 实例：凡是用new 构造函数()创建出的对象，都称为是构造函数的实例

```

1 <script>
2   var obj = {"k1":"v1"};
3   var arr = [1,2];
4   console.log("Instanceof处理对象的结果: "+(obj instanceof Array));
5   console.log("Instanceof处理数组的结果: "+(arr instanceof Array));
6 </script>

```

 top ☐ Preserve log
 Instanceof处理对象的结果: false
 Instanceof处理数组的结果: true

(4) 使用isPrototypeOf()函数

原理：检测一个对象是否是Array的原型（或处于原型链中，不但可检测直接父对象，还可检测整个原型链上的所有父对象）

使用方法：parent.isPrototypeOf(child)来检测parent是否为child的原型；

需注意的是isPrototypeOf()函数实现的功能和instanceof运算符非常类似；

```

1 Array.prototype.isPrototypeOf(arr) //true表示是数组，false不是数组

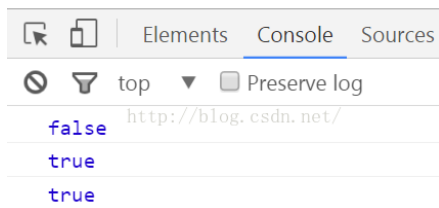
```

(5) 利用构造函数constructor

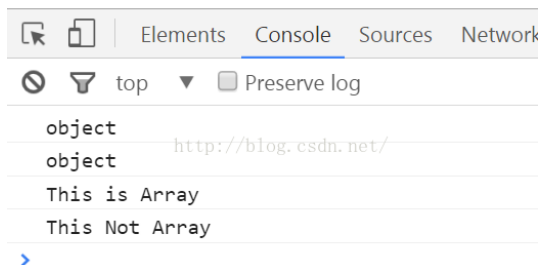
```

1 <script>
2   var obj = {'k':'v'};
3   var t1 = new Array(1);
4   var t2 = t1;
5   console.log(obj.constructor == Array);
6   console.log(t1.constructor == Array);
7   console.log(t2.constructor == Array);
8 </script>

```



(6) 使用typeof(对象)+类型名结合判断:



这种方法其实有局限性，有的同学可能一下就破解了，那就是要是要是对象中不巧定义了这属性怎么办

```
var obj = {'concat': 'Teast me?'};
```

我只能说哥你赢了!!!

跨域

jsonp: 跨域取数据

步骤:

1. 动态创建script
2. 定义全局函数（函数定义好是用来传递数据的）

```
1 function json2url(json){
2     var arr=[];
3     for(var name in json){
4         arr.push(name+'='+json[name]);
5     }
6     alert(arr.join('&'))
7     return arr.join('&');
8 }
9 function jsonp(json){
10     json=json || {};
11     json.data=json.data || {};
12     json.cbName=json.cbName || 'cb';
13
14     var fnName='jsonp_'+Math.random();
15     fnName=fnName.replace('.', '');
16
17     window[fnName]=function(data){
18         json.success && json.success(data);
19
20         //删除
21         oHead.removeChild(oS);
22     };
23
24     json.data[json.cbName]=fnName;
25
26
27     var oS=document.createElement('script');
28     oS.src=json.url+'?'+json2url(json.data);
29     alert(oS.src)
30     var oHead=document.getElementsByTagName('head')[0];
31     oHead.appendChild(oS);
32 }
33
```

```

34 // 调用试例子
35 window.onload=function(){
36     var oT=document.getElementById('t1');
37
38     oT.onkeyup=function(){
39         jsonp({
40             url:'http://sug.so.360.cn/suggest',
41             data:{
42                 word:oT.value
43             },
44             cbName:'callback',
45             success:function(json){
46                 alert(json.s);
47             }
48         });
49     };
50 };
51

```

JSONP 也叫填充式JSON，是应用JSON的一种新方法，只不过是包含在函数调用中的JSON，例如：

```
callback({"name","trigkit4"});
```

JSONP由两部分组成：**回调函数和数据**。回调函数是当响应到来时应该在页面中调用的函数，而数据就是传入回调函数中的JSON数据。

在js中，我们直接用 **XMLHttpRequest** 请求不同域上的数据时，是不可以的。但是，**在页面上引入不同域上的js脚本文件却是可以的，jsonp正是利用这个特性来实现的**。例如：

```

<script type="text/javascript">
    function dosomething(jsondata){
        //处理获得的json数据
    }
</script>

```

```
<script src="http://example.com/data.php?callback=dosomething"></script>
```

js文件载入成功后会执行我们在url参数中指定的函数，并且会把我们需要的json数据作为参数传入。所以jsonp是需要服务器端的页面进行相应的配合的。

```
<?php
$callback = $_GET['callback'];//得到回调函数名
$data = array('a','b','c');//要返回的数据
echo $callback.'('.json_encode($data).')';//输出
?>
```

最终，输出结果为：`dosomething(['a','b','c']);`

JSONP的优缺点

JSONP的优点是：它不像 `XMLHttpRequest` 对象实现的Ajax请求那样受到同源策略的限制；它的兼容性更好，在更加古老的浏览器中都可以运行，不需要XMLHttpRequest或ActiveX的支持；并且在请求完毕后可以通过调用callback的方式回传结果。

JSONP的缺点则是：它只支持GET请求而不支持POST等其它类型的HTTP请求；它只支持跨域HTTP请求这种情况，不能解决不同域的两个页面之间如何进行 `JavaScript` 调用的问题。

什么是跨域？

<https://segmentfault.com/a/1190000000718840>

概念：只要协议、域名、端口有任何一个不同，都被当作是不同的域。

1	同一域名下	允许
---	-------	----

```
2 http://www.a.com/a.js
3 http://www.a.com/b.js
4 同一域名下不同文件夹 允许
5 http://www.a.com/lab/a.js
6 http://www.a.com/script/b.js
7 同一域名, 不同端口 不允许
8 http://www.a.com:8000/a.js
9 http://www.a.com/b.js
10 同一域名, 不同协议 不允许
11 http://www.a.com/a.js
12 https://www.a.com/b.js
13 域名和域名对应ip 不允许
14 http://www.a.com/a.js
15 http://70.32.92.74/b.js
16 主域相同, 子域不同 不允许
17 http://www.a.com/a.js
18 http://script.a.com/b.js
19 同一域名, 不同二级域名 (同上) 不允许 (cookie这种情况下也不允许访问)
20 http://www.a.com/a.js
21 http://a.com/b.js
22 不同域名 不允许
23 http://www.cnblogs.com/a.js
24 http://www.a.com/b.js
```

跨域资源共享 (CORS)

CORS (Cross-Origin Resource Sharing) 跨域资源共享, 定义了必须在访问跨域资源时, 浏览器与服务器应该如何沟通。CORS 背后的基本思想就是使用自定义的HTTP头部让浏览器与服务器进行沟通, 从而决定请求或响应是应该成功还是失败。

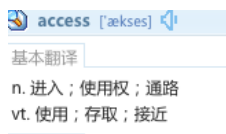
```
1 <script type="text/javascript">
2     var xhr = new XMLHttpRequest();
3     xhr.open("GET", "/trigkit4", true);
4     xhr.send();
5 </script>
```

以上的 `trigkit4` 是相对路径, 如果我们要使用 CORS, 相关 Ajax 代码可能如下所示:

```
<script type="text/javascript">
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "http://segmentfault.com/u/trigkit4/", true);
  xhr.send();
</script>
```

代码与之前的区别就在于相对路径换成了其他域的绝对路径，也就是你要跨域访问的接口地址。

服务器端对于 **CORS** 的支持，主要就是通过设置 **Access-Control-Allow-Origin** 来进行的。如果浏览器检测到相应的设置，就可以允许Ajax进行跨域的访问。



CORS和JSONP对比

CORS与JSONP相比，无疑更为先进、方便和可靠。

- 1、JSONP只能实现GET请求，而CORS支持所有类型的HTTP请求。
- 2、使用CORS，开发者可以使用普通的XMLHttpRequest发起请求和获得数据，比起JSONP有更好的错误处理。
- 3、JSONP主要被老的浏览器支持，它们往往不支持CORS，而绝大多数现代浏览器都已经支持了CORS）。

3.nginx反向代理解决跨域

<https://www.cnblogs.com/gabrielchen/p/5066120.html>

上面已经说到，禁止跨域问题其实是浏览器的一种安全行为，而现在的大多数解决方案都是用标签可以跨域访问的这个漏洞或者是技巧去完成，但都少不了目标服务器做相应的改变，而我最近遇到了一个需求是，目标服务器不能给予我一个header，更不可以改变代码返回个script，所以前5种方案都被我否决掉。最后因为我的网站是我自己的主机，所以我决定搭建一个nginx并把相应代码部署在它的下面，由页面请求本域名的一个地址，转由nginx代理处理后返回结果给页面，而且这一切都是同步的。

关于nginx的一些基本配置和安装请看我的另一篇博客，下面直接讲解如何配置一个反向代理。

首先找到nginx.conf或者nginx.conf.default 或者是default里面的这部份

```
#gzip on;

server {
    listen      8080;
    server_name localhost;

    #charset koi8-r;

    #c access_log /var/log/host.access.log;

    location / {
        root html;
        index index.html index.htm;
    }
}
```

其中server代表启动的一个服务，location 是一个定位规则。

1	location / {	#所有以/开头的地址，实际上是所有请求
2		
3	root html	#去请求../html文件夹里的文件,其中..的路径在nginx里面有定义，安装的时候会有默认路径，详见另一篇博客
4		
5	index index.html index.htm	# 首页响应地址
6		
7	}	

从上面可以看出location是nginx用来路由的入口，所以我们接下来要在location里面完成我们的反向代理。

假如我们我们是www.a.com/html/msg.html 想请求www.b.com/api/?method=1¶=2;

我们的ajax:

1	var url = 'http://www.b.com/api/msg?method=1¶=2';
2	 \$.ajax({
3	type: "GET",
4	url:url,
5	success: function(res){..},
6
7	})

上面的请求必然会遇到跨域问题，这时我们需要修改一下我们的请求url，让请求发在nginx的一个url下。

1	var url = 'http://www.b.com/api/msg?method=1¶=2';
	var proxyurl = 'msg?method=1¶=2';

```

2 //假如实际地址是 www.c.com/proxy/html/api/msg?method=1&para=2; www.c.com是nginx主机
3 地址
4 $.ajax({
5     type: "GET",
6     url: proxyurl,
7     success: function(res){..},
8     ....
9 })

```

再在刚才的路径中匹配到这个请求，我们在location下面再添加一个location。

```

1 location ^~/proxy/html/{
2     rewrite ^/proxy/html/(.*)$ /$1 break;
3     proxy_pass http://www.b.com/;
4 }

```

以下做一个解释：

1. '^~/proxy/html/ '

就像上面说的一样是一个匹配规则，用于拦截请求，匹配任何以 /proxy/html/ 开头的地址，匹配符合以后，停止往下搜索正则。

2. `rewrite ^/proxy/html/(.*)$ /$1 break;`

代表重写拦截进来的请求，并且只能对域名后边的除去传递的参数外的字符串起作用，例如 `www.c.com/proxy/html/api/msg?method=1¶=2` 重写。只对 `/proxy/html/api/msg` 重写。

`rewrite` 后面的参数是一个简单的正则 `^/proxy/html/(.*)$`，`$1` 代表正则中的第一个()`$2` 代表第二个()的值，以此类推。

`break` 代表匹配一个之后停止匹配。

3. `proxy_pass`

既是把请求代理到其他主机，其中 `http://www.b.com/` 写法和 `http://www.b.com` 写法的区别如下：

不带/

```

1 location /html/
2 {
3     proxy_pass http://b.com:8300;
4 }

```

带/

```

1 location /html/

```

```
2 {  
3     proxy_pass http://b.com:8300/;  
4 }
```

上面两种配置，区别只在于proxy_pass转发的路径后是否带“/”。

针对情况1，如果访问url = http://server/html/test.jsp，则被nginx代理后，请求路径会变为http://proxy_pass/html/test.jsp，将test/ 作为根路径，请求test/路径下的资源。

针对情况2，如果访问url = http://server/html/test.jsp，则被nginx代理后，请求路径会变为http://proxy_pass/test.jsp，直接访问server的根资源。

修改配置后重启nginx代理就成功了。

数组去重

数组 slice 和 splice

slice

slice是指定在一个数组中的元素创建一个新的数组，即原数组不会变

```
1 var color = new Array('red','blue','yellow','black');  
2 var color2 = color.slice(1,2);  
3 alert(color);    //输出    red,blue,yellow,black  
4 alert(color2);   //输出    blue;注意：这里只有第二项一个值
```

splice

splice()方法是直接修改原数组的

splice是JS中数组功能最强大的方法，它能够实现对数组元素的删除、插入、替换操作，**返回值为被操作的值。**

splice删除：**color.splice(1,2)**（删除color中的1、2两项）；

splice插入：**color.splice(1,0,'brown','pink')**（在color键值为1的元素前插入两个值）；

splice替换：**color.splice(1,2,'brown','pink')**（在color中替换1、2元素）

```
1 var color = new Array('red','blue','yellow','black');
2 var color2 = color.splice(2,3,'brown','pink'); // 原数组变为 ["red", "blue", "brown"
3 color.splice(1,2) // 原数组变为 ["red", "black"]
4 color.splice(1,0,'brown','pink') // 原数组变为 ["red", "brown", "pink", "blue", "ye
```

语法: array.splice(starti,n);

starti 指的是从哪个位置开始(不包含starti)

n指的是需要删除的**个数**

```
1 <script>
2     var array=[1,2,3,4,5];
3     var deletes =array.splice(3,2);
4     console.log(deletes); // [4,5]
5     console.log(array) // 结果: [1,2,3]
6 </script>
```

```
1
```

语法: array.splice(starti,0,值1, 值2...);

starti: 在哪个位置插入, 原来starti位置的值**向后顺移**

0: 表示删除0个元素, 因为插入和替换都是由删除功能拓展的。

值1, 值2: 需要插入的值

```
1 <script>
2   var array=[1,2,3,4,5];
3   array.splice(2,0,123,456);
4   console.log(array);
5 </script> // 结果: [1,2,123,456,3,4,5]
```

语法: `array.splice(starti,n,值1, 值2);`

原理和插入的用法相同

实际是就是: 在starti的位置删除n个元素, 然后在这个位置插入值1, 值2, 可以起到替换

原来被删除的值

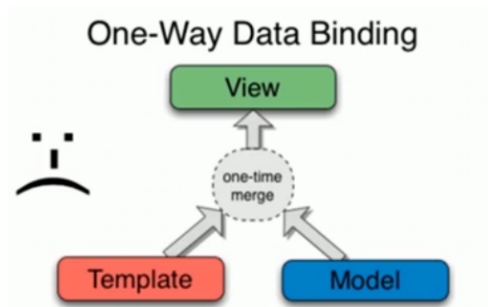
```
1 <script>
2   var array=[1,2,3,4,5];
3   array.splice(2,2,123,456);
4   console.log(array);
5 </script> // 结果: [1,2,123,456,5]
6
```

双向数据绑定和单向数据绑定的区别

单向数据绑定

单向数据绑定，带来单向数据流。。

指的是我们先把模板写好，然后把模板和数据（数据可能来自后台）整合到一起形成HTML代码，然后把这段HTML代码插入到文档流里面。适用于整体项目，并于追溯。



优点：

1. 所有状态的改变可记录、可跟踪，源头易追溯；
2. 所有数据只有一份，组件数据只有唯一的入口和出口，使得程序更直观更容易理解，有利于应用的可维护性；
3. 一旦数据变化，就去更新页面(data-页面)，但是没有(页面-data)；
4. 如果用户在页面上做了变动，那么就手动收集起来(双向是自动)，合并到原有的数据中。

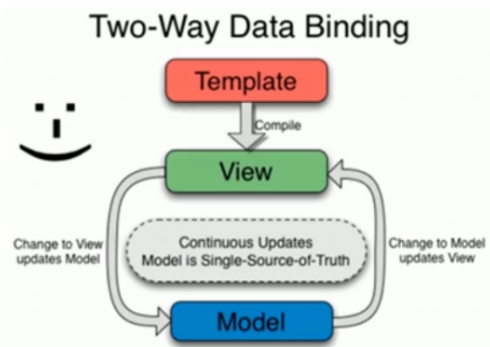
缺点：

1. HTML代码渲染完成，无法改变，有新数据，就须把旧HTML代码去掉，整合新数据和模板重新渲染；
2. 代码量上升，数据流转过程变长，出现很多类似的样板代码；
3. 同时由于对应用状态独立管理的严格要求(单一的全局store)，在处理局部状态较多的场景时(如用户输入交互较多的“富表单型”应用)，会显得啰嗦及繁琐。

双向数据绑定

双向数据绑定，带来双向数据流。**AngularJS2**添加了**单向数据绑定**

数据模型（Module）和视图（View）之间的双向绑定。无论数据改变，或是用户操作，都能带来互相的变动，自动更新。适用于项目细节，如：UI控件中(通常是类表单操作)。



优点：

1. 用户在视图上的修改会自动同步到数据模型中去，数据模型中值的变化也会立刻同步到视图中去；
2. 无需进行和单向数据绑定的那些CRUD（Create, Retrieve, Update, Delete）操作；
3. 在表单交互较多的场景下，会简化大量业务无关的代码。

缺点：

1. 无法追踪局部状态的变化；
2. “暗箱操作”，增加了出错时 debug 的难度；
3. 由于组件数据变化来源入口变得可能不止一个，数据流转方向易紊乱，若再缺乏“管制”手段，血崩

HTML5新增了哪些内容或API，使用过哪些

在API层，HTML5 增加了更多多样化的 应用程序接口

canvas

离线：想起 Cache Manifest，和 Cache APIs。再加上 Service Worker 的特性，用户体验能提升不少

网络存储：sessionStorage & localStorage，这个应该不陌生，保存一些稍大的数据，或者不适合放在Cookie 的，就用网络存储。

元素与属性

<section> 写文章的时候会经常用到

<video> 和 <audio>：视频和音频

<footer> 和 <header>：之前 <div class='footer OR header'></div> 的写法换成标签

<footer> 和 <header> 就行了，为了 语义化，推荐

<nav> 表示导航等等，还要更多的标签

input 和 textarea 的区别

input

- 可以指定 type 为 url, email, 可以方便地检测用户输入。还是指定为 file, submit, button, checkbox, radio, datetime, color等, 改变 input 的样式和行为。
- 可以通过 value 属性指定初始值
- 宽高只能通过 CSS 指定

textarea

- 可以输入多行文字
- 输入值初始化需要用标签对包裹, 并可以夹杂 HTML 代码, 而不会被浏览器解析

```
<textarea><h1>h1</h1></textarea>
```

- 宽高能用 CSS 或 rows, cols 指定

相同:

- 都可以使用 maxlength, minlength 等限制输入

忽略页面中的电话号码

```
<meta name="format-detection" content="telephone=no" />
```

detection

英 [dɪ'tekʃn] ㄉㄧˋ ㄊㄜˊ ㄎㄢˋ 美 [dɪ'tekʃən] ㄉㄧˋ ㄊㄜˊ ㄎㄢˋ

n. 侦查; 检查; 察觉; 发觉

左右布局: 左边定宽、右边自适应, 不少于3种方法

absolute + padding

splice

splice

splice

splice

HTML5新增了哪些内容或API，使用过哪些

```
1 <div class='example-1 auto-width'>
2   <div class='left'>left</div>
3   <div class='right'>right</div>
4 </div>
5
6 <style>
7   .example-1 {
8     position: relative;
9     height: 100px;
10    padding-left: 100px;
11  }
12  .example-1 .left {
13    position: absolute;
14    width: 100px;
15    left: 0;
16    height: 100%;
17    background: #0f0;
18  }
19  .example-1 .right {
20    background: #f00;
21    width: 100%;
22    height: 100%;
23  }
24 </style>
```

flex, 未来趋势, 推荐!

```
1 <div class='example-2 auto-width'>
2
3   <div class='left'>left</div>
4   <div class='right'>right</div>
5 </div>
6 <style>
7   .example-2 {
```

```
8     display: flex;
9     height: 100px;
10  }
11  .example-2 .left {
12      width: 100px;
13      height: 100%;
14      background: #0f0;
15  }
16  .example-2 .right {
17      background: #f00;
18      height: 100%;
19      flex: 1;
20  }
21  </style>
```

table

```
1 <div class='example-3 auto-width'>
2   <div class='left'>left</div>
3   <div class='right'>right</div>
4 </div>
5
6 <style>
7   .example-3 {
8       display: table;
9       height: 100px;
10      width: 100%;
11  }
12  .example-3 .left {
13      width: 100px;
14      height: 100%;
15      background: #0f0;
16      display: table-cell;
17  }
18  .example-3 .right {
19      background: #f00;
20      height: 100%;
21      display: table-cell;
22  }
23 </style>
```

```
1 <div class='example-4 auto-width'>
2   <div class='left'>left</div>
3   <div class='right'>right</div>
4 </div>
5
6 <style>
7   .example-4 {
8     height: 100px;
9     width: 100%;
10  }
11  .example-4 .left {
12    float: left;
13    width: 100px;
14    height: 100%;
15    background: #0f0;
16  }
17  .example-4 .right {
18    background: #f00;
19    overflow: hidden;
20    height: 100%;
21  }
22 </style>
```

CSS3用过哪些新特性

- `border-radius` 圆角, `@font-face` 字体, `box-shadow` `text-shadow` 文本和框的阴影

`border-radius`: 上 右 下 左

- * `border-top-left-radius`
- * `border-top-right-radius`
- * `border-bottom-right-radius`
- * `border-bottom-left-radius`

```

1 <style>
2 @font-face
3 {
4   font-family: myFirstFont;
5   src: url('Sansation_Light.ttf'),
6        url('Sansation_Light.eot'); /* IE9+ */
7 }
8
9 div
10 {
11   font-family:myFirstFont;
12 }
13 </style>

```

box-shadow: *h-shadow v-shadow blur spread color inset*;

box-shadow: 10px 10px 5px #888888;

值	描述
<i>h-shadow</i>	必需。水平阴影的位置。允许负值。
<i>v-shadow</i>	必需。垂直阴影的位置。允许负值。
<i>blur</i>	可选。模糊距离。
<i>spread</i>	可选。阴影的尺寸。
<i>color</i>	可选。阴影的颜色。请参阅 CSS 颜色值。
<i>inset</i>	可选。将外部阴影 (outset) 改为内部阴影。

text-shadow: *h-shadow v-shadow blur color*;

text-shadow: 5px 5px 5px #FF0000;

值	描述
<i>h-shadow</i>	必需。水平阴影的位置。允许负值。
<i>v-shadow</i>	必需。垂直阴影的位置。允许负值。
<i>blur</i>	可选。模糊的距离。
<i>color</i>	可选。阴影的颜色。参阅 CSS 颜色值 。

- `word-wrap` , `background-size` , `background-origin` , `border-image` , `box-sizing` , `calc` , `linear-gradient` 等等

`word-wrap:break-word`

```
哈哈 this a long  
worddddddddddddddddddddddddddd  
dddddd!
```

`word-break:break-all`

```
哈哈 this a long worddddddddddd  
dddddddddddddddddddddd!
```

`word-wrap` 是用来决定允不允许单词内断句的，如果不允许的话长单词就会溢出。最重要的一点是它还是会首先尝试挪到下一行，看看下一行的宽度够不够，不够的话就进行单词内的断句。而`word-break:break-all`则更变态，因为它断句的方式非常粗暴，它不会尝试把长单词挪到下一行，而是直接进行单词内的断句。这也可以解释为什么说它的作用是决定用什么方式来断句的，那就是——用了`word-break:break-all`，就等于使用粗暴方式来断句了。总之一句话，如果您想更节省空间，那就用`word-break:break-all`就对了！

`word-wrap:break-word`与`word-break:break-all`共同点是都能把长单词强行断句，不同点是`word-wrap:break-word`会首先起一个新行来放置长单词，新的行还是放不下这个长单词则会对长单词进行强制断句；而`word-break:break-all`则不会把长单词放在一个新行里，当这一行放不下的时候就直接强制断句了。

`background-size` 属性规定背景图像的尺寸。
`background-size: length|percentage|cover|contain;`

值	描述
<i>length</i>	设置背景图像的高度和宽度。 第一个值设置宽度，第二个值设置高度。 如果只设置一个值，则第二个值会被设置为 "auto"。
<i>percentage</i>	以父元素的百分比来设置背景图像的宽度和高度。 第一个值设置宽度，第二个值设置高度。 如果只设置一个值，则第二个值会被设置为 "auto"。
cover	把背景图像扩展至足够大，以使背景图像完全覆盖背景区域。 背景图像的某些部分也许无法显示在背景定位区域中。
contain	把图像扩展至最大尺寸，以使其宽度和高度完全适应内容区域。

background-origin 属性规定 background-position 属性相对于什么位置来定位。

```
background-origin: padding-box|border-box|content-box;
```

值	描述
padding-box	背景图像相对于内边距框来定位。
border-box	背景图像相对于边框盒来定位。
content-box	背景图像相对于内容框来定位。

box-sizing 属性允许您以特定的方式定义匹配某个区域的特定元素。

- `transform` 转换
 - 2D 转换
 - `rotate` 旋转，图片转个90或180度什么的
 - `translate` 位置移动
 - `scale` , `skew` , `matrix` 等
 - 3D 转换
 - `rotate(XYZ)` 根据x,y,z轴旋转
 - `translate(XYZ)` , `scale(XYZ)` 同理
 - `perspective` 透视，这个很多3D效果都要设置一下，不然3D还是只会有"2D"的效果

- `transition`: 过渡, 简单的动画 (如: 移个位置, 变个长短), 直接用这个属性就能搞定。
- `animation`: 动画, 3D可以调用硬件渲染。
- 新的长度单位: `rem`, `ch`, `vw`, `vh`, `vmax`, `vmin` 等。
- `clip-path`: 绘制路径, 类似 SVG 技术。国外炫酷产品。
- `flex`: `flex` 布局, 继 `table` 和 `div` 后的趋势, 不了解或不熟悉的可以参考[cssreference](#)。
- 伪类选择器: 如: `:target`, `:enabled`, `:disabled`, `:first-child`, `last-child` 等等
- `@media` 媒体查询, 适用于一些响应式布局中
- `columns`: 分栏布局。
- `will-change`: 改善渲染性能, 参考[使用CSS3 will-change提高页面滚动、动画等渲染性能](#)。

一.iconfont使用场景（优缺点）；

1.优点；

轻量（文件体积小）、灵活（样式可改变图标）、兼容性好（IE8+）。

2.缺点；

图标只能单色调（太复杂的多色图标无法实现）、生成图标字体相对花时间。

PC端页面优化, 可将部分雪碧图换成小图标, 字体图标比雪碧图的http请求少、体积小; （加载一个页面时分模块开发关系可能有多张雪碧图, 但使用字体图标, 文件一个就够）

PC端做换肤业务时, 使用了字体图标实现起来更加的优雅、方便。（之前做页面换皮肤功能时发现换肤时小图标得多出一套雪碧图略麻烦, 如果是字体图标直接更新颜色样式就OK）

<https://segmentfault.com/a/1190000010299811>

金额格式化函数

1:

```
1 function toThousands(num) {
2     var result = [ ], counter = 0;
3     num = (num || 0).toString().split('');
4     for (var i = num.length - 1; i >= 0; i--) {
5         counter++;
6         result.unshift(num[i]);
7         if (!(counter % 3) && i != 0) { result.unshift(','); }
8     }
9     return result.join('');
10 }
```

11 方法一的执行过程就是把数字转换成字符串后，打散为数组，再从末尾开始，逐个把数组中的元素插入到新数组（

2:

```
1 function formatNumber(num, precision, separator) {
2     var parts;
3     // 判断是否为数字
4     if (!isNaN(parseFloat(num)) && isFinite(num)) {
5         // 把类似 .5, 5. 之类的数据转化成0.5, 5, 为数据精度处理做准，至于为什么
6         // 不在判断中直接写 if (!isNaN(num = parseFloat(num)) && isFinite(num))
7         // 是因为parseFloat有一个奇怪的精度问题，比如 parseFloat(12312312.1234567119)
8         // 的值变成了 12312312.123456713
9         num = Number(num);
10        // 处理小数点位数
11        num = (typeof precision !== 'undefined' ? num.toFixed(precision) : num).toS
12        // 分离数字的小数部分和整数部分
13        parts = num.split('.');
14        // 整数部分加[separator]分隔，借用一个著名的正则表达式
15        parts[0] = parts[0].toString().replace(/(\d)(?=(\d{3})+(?!\d))/g, '$1' + (s
16
17        return parts.join('.');
18    }
19    return NaN;
20 }
21 formatNumber(10000)
22 "10,000"
```



```

23 formatNumber(10000, 2)
24 "10,000.00"
25 formatNumber(10000.123456, 2)
26 "10,000.12"
27 formatNumber(10000.123456, 2, ' ')
28 "10 000.12"
29 formatNumber(.123456, 2, ' ')
30 "0.12"
31 formatNumber(56., 2, ' ')
32 "56.00"
33 formatNumber(56., 0, ' ')
34 "56"
35 formatNumber('56.')
36 "56"
37 formatNumber('56.a')
38 NaN

```

computed 和 watch 的对比

- ①从属性名上，`computed` 是计算属性，也就是依赖其它的属性计算所得出最后的值。`watch` 是去监听一个值的变化，然后执行相对应的函数。
 - ②从实现上，`computed` 的值在 `getter` 执行后是会缓存的，只有在它依赖的属性值改变之后，下一次获取 `computed` 的值时才会重新调用对应的 `getter` 来计算。`watch` 在每次监听的值变化时，都会执行回调。其实从这一点来看，都是在依赖的值变化之后，去执行回调。很多功能本来就很多属性都可以用，只不过有更适合的。如果一个值依赖多个属性（多对一），用 `computed` 肯定是更加方便的。如果一个值变化后会引起一系列操作，或者一个值变化会引起一系列值的变化（一对多），用 `watch` 更加方便一些。
 - ③`watch` 的回调里面会传入监听属性的新旧值，通过这两个值可以做一些特定的操作。`computed` 通常就是简单的计算。
 - ④`watch` 和 `computed` 并没有哪个更底层，`watch` 内部调用的是 `vm.$watch`，它们的共同之处就是每个定义的属性都单独建立了一个 `Watcher` 对象
- `computed` 可以监控自定义属性 如果一个值依赖多个属性（多对一）
`watch` 可以监控路由 一般用于监控data里的值

```

1 watch: {
2     // 如果路由有变化，会再次执行该方法
3     '$route': 'fetchData'
4 }
5 methods: {
6     fetchData() {
7         let that = this;

```

```

8      let query = that.$store.state.route.query;
9
10     // 解决后退时，高亮与数据不统一
11     that.areaId = query.areaId;
12     that.zoneId = query.zoneId;
13   }
14 }

```

css 选择器权重

行内 > id > 类、属性、伪类选择器 > 元素、伪元素

vue 生命周期

Lifecycle hooks

生命周期钩子应该算 vue 这次升级中 broken changes 最多的一部分了，对照 1.0 的文档和 [release note](#)，作了下面这张表

vue 1.0+	vue 2.0	Description
init	beforeCreate	组件实例刚被创建，组件属性计算之前，如 data 属性等
created	created	组件实例创建完成，属性已绑定，但 DOM 还未生成， <code>\$el</code> 属性还不存在
beforeCompile	beforeMount	模板编译/挂载之前
compiled	mounted	模板编译/挂载之后
ready	mounted	模板编译/挂载之后（不保证组件已在 document 中）
-	beforeUpdate	组件更新之前
-	updated	组件更新之后
-	activated	for <code>keep-alive</code> ，组件被激活时调用
-	deactivated	for <code>keep-alive</code> ，组件被移除时调用
attached	-	不用了还说啥哪...
detached	-	那就不说了吧...
beforeDestory	beforeDestory	组件销毁前调用
destoryed	destoryed	组件销毁后调用

http://blog.csdn.net/sexy_squirrel

垂直水平居中

1:

2、绝对定位+margin反向偏移

</style>

<style type="text/css">

```
.wrp2 { position: relative; }
.box2 {
  position: absolute;
  top: 50%; left: 50%;
  margin-left: -100px; /* (width + padding)/2 */
  margin-top: -75px; /* (height + padding)/2 */
}
```

</style>

<div class="wrp wrp2">

```
<div class="box box2">
  <h3>完全居中方案二：</h3>
  <h3>开发工具 【 WeX5 】： 高性能轻架构、开源免费、跨端、可视化</h3>
</div>
```

</div>

3、绝对定位+transform反向偏移

<style type="text/css">

```
.wrp3 { position: relative; }
.box3 {
  margin: auto;
  position: absolute;
  top: 50%; left: 50%;
  -webkit-transform: translate(-50%, -50%);
  -ms-transform: translate(-50%, -50%);
  transform: translate(-50%, -50%);
}
```

</style>

<div class="wrp wrp3">

```
<div class="box box3">
  <h3>完全居中方案三：</h3>
  <h3>开发工具 【 WeX5 】： 高性能轻架构、开源免费、跨端、可视化</h3>
```

</div>

4、display : tabel

<style type="text/css">

```
.wrp4 { display: table; }
.subwrp4 {
    display: table-cell;
    vertical-align: middle;
}
.box4 {
    margin: auto;
    overflow-wrap: break-word;
    height: auto;
    max-height: 80%;
    max-width: 80%;
}
```

</style>

<div class="wrp wrp4">

```
<div class="subwrp4">
    <div class="box box4">
        <h3>完全居中方案四: </h3>
    </div>
</div>
```

</div>

6、display: flex-box

<style type="text/css">

```
.wrp6 {
  display: -webkit-flex;
  display: -moz-box;
  display: -ms-flexbox;
  display: -webkit-box;
  display: flex;
  -webkit-box-align: center;
  -moz-box-align: center;
  -ms-flex-align: center;
  -webkit-align-items: center;
  align-items: center;
  -webkit-box-pack: center;
  -moz-box-pack: center;
  -ms-flex-pack: center;
  -webkit-justify-content: center;
  justify-content: center;
}

.box6 {
  width: auto;
  height: auto;
  max-width: 90%;
  max-height: 90%;
}
```

</style>

<div class="wrp wrp6">

```
<div class="box box6">
  <h3>完全居中方案六: </h3>
  <h3>开发工具 【 WeX5 】: 高性能轻架构、开源免费、跨端、可视化</h3>
</div>
```

</div>

数组方法：改变自身的方法， 不改变自身， 遍历方法

1：改变自身的方法

array.pop()

删除一个数组中的最后一个元素，并且返回这个元素

array.push(element1, ...elementN)

添加一个或多个元素到数组的末尾，并返回数组新的长度

array.reverse()

前后颠倒数组中元素的位置，第一个元素会成为最后一个

```
array.shift()
```

删除数组的第一个元素，并返回这个元素

```
array.unshift(element1, ...elementN)
```

在数组的开头插入一个或多个元素，并返回数组的新长度

```
array.sort([function(a, b)])
```

对数组的元素做原地的排序，并返回这个数组。sort可能不稳定，默认按照字符串的unicode码位点排序

记a和b是两个将要被比较的元素：

- 如果函数function (a, b) 返回值小于0，则a会排在b之前
- 如何函数返回值等于0，则a和b的相对位置不变（并不被保证）
- 如果函数返回值大于0，则a会排在b之后
- 比较函数输出结果必须稳定，否则排序的结果将是不确定的

```
array.splice(start, deleteCount[, item1[, item2...])
```

在任意的位置给数组添加或删除任意个元素（拼接），返回被删除的元素组成的数组，没有则返回空数组

2: 不改变自身的方法

```
array.concat(value1, value2.....)
```

将传入的数组或非数组值与原数组合并，组成一个新的数组并返回

注意：concat方法在拷贝原数组的过程中，

- 对象引用（非对象直接量）：concat方法会复制对象引用放到组合的新数组里，**原数组和新数组中的对象引用都指向同一个实际的对象**，所以，当实际的对象被修改时，两个数组也同时被修改
- 字符串和数字（是原始值，而不是包装原始值的string和number对象）：concat方法会复制字符串和数字的值放到新数组里

一个栗子：

```
1 var arr1 = [1, 2, {a: 'test'}]
2 var arr2 = ['a', 'b', 'c']
3
4 var output = arr1.concat(arr2)
5 console.log(output) // output[2].a == 'test'
6 setTimeout(function(){
7     arr1[2].a = 'has changed'
8     console.warn(output) //output[2].a == 'has changed'
```

```

9  }, 5000)
10
11 另一个栗子:
12  var arr1 = [1, 2, 3]
13  var arr2 = ['a', 'b', 'c']
14
15  var output = arr1.concat(arr2)
16  console.log(output)
17  setTimeout(function(){
18      arr1[2] = 99
19      console.warn(output)      //output值并不会改变
20  }, 5000)
21
22 将非数组值合并到数组里:
23
24  var alpha = ['a', 'b', 'c']
25  var output = alpha.concat(1, [2, 3]) //['a', 'b', 'c', 1, 2, 3]

```

`array.includes(searchElement, [, fromIndex])` [实验性质, es7, 可能会改变或删除]

用来判断当前数组是否包含某指定的值, 如果是, 则返回true, 否则false

`array.join([separator = ','])`

将数组中的所有元素连接成一个字符串(默认用逗号作为分隔符, 如果separator是一个空字符串, 那么数组中的所有元素将被直接连接)

如果元素是undefined或者null, 则会转化成空字符串

`array.slice([begin = 0 [, end = this.length - 1]])`

把数组中一部分的浅复制 (shallow copy) 存入一个新的数组对象中, 并返回这个新的数组
不修改原数组, 只会返回一个包含了原数组中提取的部分元素的一个新数组

`array.toString()`

返回一个字符串, 该字符串由数组中的每个元素的toString () 返回值经调用join () 方法连接 (由逗号隔开) 组成。

`array.toLocaleString()`

返回一个字符串表示数组中的元素。数组中的元素将使用各自的toLocaleString方法转化成字符串, 这些字符串将使用一个特定语言环境的字符串 (例如逗号) 隔开

`array.indexOf(searchElement[, fromIndex = 0])`

返回指定元素能在数组中找到的第一个索引值, 否则返回-1

fromIndex可以为负, 表示从倒数第n个开始 (此时仍然从前向后查询数组)

使用“严格相等” (===) 进行匹配

`array.lastIndexOf(searchElement[, fromIndex = arr.length - 1])`

返回指定元素在数组中的最后一个的索引, 如果不存在则返回-1, 从数组的后面向前查找

3: 遍历方法

`array.forEach((v, i, a) => {})`

让数组的每一项都执行一次给定的函数

v表示当前项的值, i表示当前索引, a表示数组本身

forEach遍历的范围在第一次调用 callback前就会确定。调用forEach后添加到数组中的项不会被 callback访问到。如果已经存在的值被改变, 则传递给 callback的值是 forEach遍历到他们那一刻的值。已删除的项不会被遍历到。

`array.entries()`

返回一个Array Iterator对象, 该对象包含数组中每一个索引的键值对

```
1 var arr = ["a", "b", "c"];
2 var eArr = arr.entries();
3
4 console.log(eArr.next().value); // [0, "a"]
5 console.log(eArr.next().value); // [1, "b"]
6 console.log(eArr.next().value); // [2, "c"]
7
```

`array.every(callback(v, i, a){})`

callback只会为那些已经被赋值的索引调用, 不会为那些被删除或从来没有被赋值的索引调用
和forEach函数类似

注意: array.every()返回一个布尔值, 即对每个元素的callback函数结果作逻辑“&”操作

`array.some()`

使用方法同上,

注意: 对每个元素的callback函数结果作逻辑“||”操作

`array.filter((v, i, a) => {})`

使用指定的函数测试所有元素, 并创建一个包含所有测试通过的元素的新数组

callback函数返回一个布尔值, true即通过测试

callback只会在已经赋值的索引上被调用, 对于那些已经被删除或者从未被赋值的索引不会被调用
不会改变原数组

`array.find((v, i, a) => {})` 【有兼容性问题目前】

返回数组中满足测试条件的第一个元素, 如果没有满足条件的元素, 则返回undefined

`array.keys()`

返回一个数组索引的迭代器 (类似于array.entries()方法)

`array.map((v, i, a) => {})`

返回一个由原数组中的每个元素调用一个指定方法后的返回值组成的新数组

map 不修改调用它的原数组本身 (当然可以在 callback 执行时改变原数组)

`array.reduce(callback[, initialValue])`

该方法接收一个函数作为累加器（accumulator），数组中的每个值（从左到右）开始合并，最终为一个值
callback参数：

previousValue:上一次调用回调返回的值，或者是提供的初始值（initialValue）

currentValue: 数组中当前被处理的元素

index: index

array: 调用的数组

如果 initialValue 在调用 reduce 时被提供，那么第一个 previousValue 等于 initialValue，并且currentValue 等于数组中的第一个值；如果initialValue 未被提供，那么previousValue 等于数组中的第一个值，currentValue等于数组中的第二个值。

一个例子

```
[0, 1, 2, 3, 4, 5].reduce((p, v, i, a) => {  
  return p + v  
})  
  
//15
```

```
var flattened = [[0, 1], [2, 3], [4, 5]].reduce((a, b) => {  
  return a.concat(b)  
})  
//flattened is [0, 1, 2, 3, 4, 5]
```

还有style-loader Sass-loader 顺序

```
1 module: {  
2   rules: [  
3     { test: /\.css$/, use: ['style-loader', 'css-loader', 'sass-loader'] }  
4   ]  
5 }
```

CSS3会出现卡顿，必须重新 硬刷新

Css 如何计算优先级

!important > id > class > 元素&伪元素 > *

ES6 新特性

let const 字符串模板 promise 解构赋值

箭头函数

箭头函数根本就没有绑定自己的 `this`，在箭头函数中调用 `this` 时，仅仅是简单的沿着作用域链向上寻找，找到最近的一个 `this` 拿来使用罢了；

箭头函数根本就没有绑定自己的 `this`，在箭头函数中调用 `this` 时，仅仅是简单的沿着作用域链向上寻找，找到最近的一个 `this` 拿来使用罢了；

实际上箭头函数中并不只是 `this` 和普通函数有所不同，箭头函数中没有任何像 `this` 这样自动绑定的局部变量，包括：
`this`，`arguments`，`super`(ES6)，`new.target`(ES6).....

GET一般用于获取/查询资源信息，而POST一般用于更新资源信息。

GET请求的数据会附在URL之后

POST把提交的数据则放置在是HTTP包的包体中。

POST的安全性要比GET的安全性高