**JS变量提升**

会提升变量，但并不会提升它的赋值

'use strict';

**function** foo() {

**var** x = 'Hello, ' + y;

alert(x);

**var** y = 'Bob';

}

foo();

会输出Hello，undefined，这就是因为先把所有的变量扫描了一遍，发现存在y，但没有提升其赋值，所以还是undefined

**JS局部作用域，块级作用域**

Let来定义块级作用域，这样定义的变量，即使是在函数体内部也无法被块之外的地方调用，比如for循环块，let定义的i在循环体外无法被访问。

在函数体外的变量就是全局的，当然其实也是最大的方法window的一个属性。

可以用const来定义一个常量，它是不可改变的。

**JS的this变量**

'use strict';

**var** xiaoming = {

name: '小明',

birth: 1990,

age: **function** () {

**function** getAgeFromBirth() {

**var** y = **new** Date().getFullYear();

**return** y - **this**.birth;

}

**return** getAgeFromBirth();

}

};

xiaoming.age();

报错了！原因是this指针只在age方法的函数内指向xiaoming，在函数内部定义的函数，this又指向undefined了！（在非strict模式下，它重新指向全局对象window！）

要保证this指向正确，必须用obj.xxx()的形式调用！

修复的办法也不是没有，我们用一个that变量首先捕获this：

'use strict';

**var** xiaoming = {

name: '小明',

birth: 1990,

age: **function** () {

**var** that = **this**; *// 在方法内部一开始就捕获this*

**function** getAgeFromBirth() {

**var** y = **new** Date().getFullYear();

**return** y - that.birth; *// 用that而不是this*

}

**return** getAgeFromBirth();

}

};

xiaoming.age(); *// 25*

用var that = this;，你就可以放心地在方法内部定义其他函数，而不是把所有语句都堆到一个方法中。

apply方法可以为函数指定一个this对象

第一个参数就是需要绑定的this变量，第二个参数是Array，表示函数本身的参数。

**function** getAge() {

**var** y = **new** Date().getFullYear();

**return** y - **this**.birth;

}

**var** xiaoming = {

name: '小明',

birth: 1990,

age: getAge

};

xiaoming.age(); *// 25*

getAge.apply(xiaoming, []); *// 25, this指向xiaoming, 参数为空*

apply实现装饰者模式

利用apply()，我们还可以动态改变函数的行为。

JavaScript的所有对象都是动态的，即使内置的函数，我们也可以重新指向新的函数。

现在假定我们想统计一下代码一共调用了多少次parseInt()，可以把所有的调用都找出来，然后手动加上count += 1，不过这样做太傻了。最佳方案是用我们自己的函数替换掉默认的parseInt()：

**var** count = 0;

**var** oldParseInt = parseInt; *// 保存原函数*

window.parseInt = **function** () {

count += 1;

**return** oldParseInt.apply(null, arguments); *// 调用原函数*

};

*// 测试:*

parseInt('10');

parseInt('20');

parseInt('30');

count; *// 3*

**Map & Reduce**



**闭包**

关于闭包我自己的几个理解

首先是作用的顺序

**function** count() {

**var** arr = [];

**for** (**var** i=1; i<=3; i++) {

arr.push(**function** () {

**return** i \* i;

});

}

**return** arr;

}

**var** results = count();

**var** f1 = results[0];

**var** f2 = results[1];

**var** f3 = results[2];

这段代码的输出并非1、4、9，而是16、16、16.就是因为先使用count()时，返回的函数并没有被调用，而是将返回函数之外的部分执行完，也就是for循环转完，此时的i为4。然后results是获得count返回值的函数，也就是results是一个函数，它内部引用的i为4，这时我们就能理解，再执行返回的函数，也就是平方的话得到的当然都是16，跟输入参数没关系了。

**返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。**

**JS类型**

总结一下，有这么几条规则需要遵守：

* 不要使用new Number()、new Boolean()、new String()创建包装对象；
* 用parseInt()或parseFloat()来转换任意类型到number；
* 用String()来转换任意类型到string，或者直接调用某个对象的toString()方法；
* 通常不必把任意类型转换为boolean再判断，因为可以直接写if (myVar) {...}；
* typeof操作符可以判断出number、boolean、string、function和undefined；
* 判断Array要使用Array.isArray(arr)；
* 判断null请使用myVar === null；
* 判断某个全局变量是否存在用typeof window.myVar === 'undefined'；
* 函数内部判断某个变量是否存在用typeof myVar === 'undefined'。

最后有细心的同学指出，任何对象都有toString()方法吗？null和undefined就没有！确实如此，这两个特殊值要除外，虽然null还伪装成了object类型。

更细心的同学指出，number对象调用toString()报SyntaxError：

123.toString(); // SyntaxError

遇到这种情况，要特殊处理一下：

123..toString(); // '123', 注意是两个点！

(123).toString(); // '123'

**创建对象的一个好方法**

**function** Student(props) {

**this**.name = props.name || '匿名'; *// 默认值为'匿名'*

**this**.grade = props.grade || 1; *// 默认值为1*

}

Student.prototype.hello = **function** () {

alert('Hello, ' + **this**.name + '!');

};

**function** createStudent(props) {

**return** **new** Student(props || {})

}

这个createStudent()函数有几个巨大的优点：一是不需要new来调用，二是参数非常灵活，可以不传，也可以这么传：

**var** xiaoming = createStudent({

name: '小明'

});

xiaoming.grade; *// 1*

如果创建的对象有很多属性，我们只需要传递需要的某些属性，剩下的属性可以用默认值。由于参数是一个Object，我们无需记忆参数的顺序。如果恰好从JSON拿到了一个对象，就可以直接创建出xiaoming。

**浏览器对象**

window

window对象不但充当全局作用域，而且表示浏览器窗口。

window对象有innerWidth和innerHeight属性，可以获取浏览器窗口的内部宽度和高度。内部宽高是指除去菜单栏、工具栏、边框等占位元素后，用于显示网页的净宽高。

兼容性：IE<=8不支持。

navigator

navigator对象表示浏览器的信息，最常用的属性包括：

* navigator.appName：浏览器名称；
* navigator.appVersion：浏览器版本；
* navigator.language：浏览器设置的语言；
* navigator.platform：操作系统类型；
* navigator.userAgent：浏览器设定的User-Agent字符串。
* 正确的方法是充分利用JavaScript对不存在属性返回undefined的特性，直接用短路运算符||计算：
* **var** width = window.innerWidth || document.body.clientWidth;

screen

screen对象表示屏幕的信息，常用的属性有：

* screen.width：屏幕宽度，以像素为单位；
* screen.height：屏幕高度，以像素为单位；
* screen.colorDepth：返回颜色位数，如8、16、24。

### location

location对象表示当前页面的URL信息。例如，一个完整的URL：

http:*//www.example.com:8080/path/index.html?a=1&b=2#TOP*

可以用location.href获取。要获得URL各个部分的值，可以这么写：

location.protocol; // 'http'

location.host; // 'www.example.com'

location.port; // '8080'

location.pathname; // '/path/index.html'

location.search; // '?a=1&b=2'

location.hash; // 'TOP'

要加载一个新页面，可以调用location.assign()。如果要重新加载当前页面，调用location.reload()方法非常方便。

### document

document对象表示当前页面。由于HTML在浏览器中以DOM形式表示为树形结构，document对象就是整个DOM树的根节点。

document的title属性是从HTML文档中的<title>xxx</title>读取的，但是可以动态改变

**Promise**

new Promise(function (resolve, reject) {

log('start new Promise...');

var timeOut = Math.random() \* 2;

log('set timeout to: ' + timeOut + ' seconds.');

setTimeout(function () {

if (timeOut < 1) {

log('call resolve()...');

resolve('200 OK');

}

else {

log('call reject()...');

reject('timeout in ' + timeOut + ' seconds.');

}

}, timeOut \* 1000);

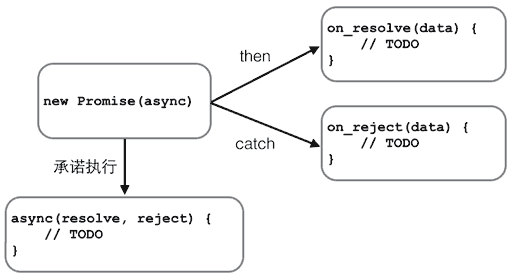
}).then(function (r) {

log('Done: ' + r);

}).catch(function (reason) {

log('Failed: ' + reason);

});

可见Promise最大的好处是在异步执行的流程中，把执行代码和处理结果的代码清晰地分离了：

JS的单线程是指一个浏览器进程中只有一个JS的执行线程，同一时刻内只会有一段代码在执行（你可以使用IE的标签式浏览试试看效果，这时打开的多个页面使用的都是同一个JS执行线程，如果其中一个页面在执行一个运算量较大的function时，其他窗口的JS就会停止工作）。  
而异步机制是浏览器的两个或以上常驻线程共同完成的，例如异步请求是由两个常驻线程：JS执行线程和事件触发线程共同完成的，JS的执行线程发起异步请求（这时浏览器会开一条新的HTTP请求线程来执行请求，这时JS的任务已完成，继续执行线程队列中剩下的其他任务），然后在未来的某一时刻事件触发线程监视到之前的发起的HTTP请求已完成，它就会把完成事件插入到JS执行队列的尾部等待JS处理。又例如定时触发（settimeout和setinterval）是由浏览器的定时器线程执行的定时计数，然后在定时时间把定时处理函数的执行请求插入到JS执行队列的尾端（所以用这两个函数的时候，实际的执行时间是大于或等于指定时间的，不保证能准确定时的）。  
所以，所谓的JS的单线程和异步更多的应该是属于浏览器的行为，他们之间没有冲突，更不是同一种事物，没有什么区别不区别的。

[css文件](https://www.baidu.com/s?wd=css%E6%96%87%E4%BB%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWuWKhPyn3n19brHF-nhf0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtYPHfYPjT1P1c4P1nzPWmdPWR)引用使用<link href="你的[css文件](https://www.baidu.com/s?wd=css%E6%96%87%E4%BB%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWuWKhPyn3n19brHF-nhf0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtYPHfYPjT1P1c4P1nzPWmdPWR)[相对地址](https://www.baidu.com/s?wd=%E7%9B%B8%E5%AF%B9%E5%9C%B0%E5%9D%80&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWuWKhPyn3n19brHF-nhf0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtYPHfYPjT1P1c4P1nzPWmdPWR)" rel="stylesheet" type="text/css" />  
js文件引用<script type="text/javascript" src="你的js[相对地址](https://www.baidu.com/s?wd=%E7%9B%B8%E5%AF%B9%E5%9C%B0%E5%9D%80&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWuWKhPyn3n19brHF-nhf0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtYPHfYPjT1P1c4P1nzPWmdPWR)"></script>  
  
并把这两个放在<head></head>标签中，不会影响你的排版，不过css是样式文件，就是负责网页的表现的，可能会影响你的页面显示。  
  
如果有jQuery.js这个文件 ，需要把它的引用放在其他js文件引用之前。

**DOM操作**

在操作一个DOM节点前，我们需要通过各种方式先拿到这个DOM节点。最常用的方法是document.getElementById()和document.getElementsByTagName()，以及CSS选择器document.getElementsByClassName()。

由于ID在HTML文档中是唯一的，所以document.getElementById()可以直接定位唯一的一个DOM节点。document.getElementsByTagName()和document.getElementsByClassName()总是返回一组DOM节点。要精确地选择DOM，可以先定位父节点，再从父节点开始选择，以缩小范围。

*// 返回ID为'test'的节点：*

**var** test = document.getElementById('test');

*// 先定位ID为'test-table'的节点，再返回其内部所有tr节点：*

**var** trs = document.getElementById('test-table').getElementsByTagName('tr');

*// 先定位ID为'test-div'的节点，再返回其内部所有class包含red的节点：*

**var** reds = document.getElementById('test-div').getElementsByClassName('red');

*// 获取节点test下的所有直属子节点:*

**var** cs = test.children;

*// 获取节点test下第一个、最后一个子节点：*

**var** first = test.firstElementChild;

**var** last = test.lastElementChild;

第二种方法是使用querySelector()和querySelectorAll()，需要了解selector语法，然后使用条件来获取节点，更加方便：

*// 通过querySelector获取ID为q1的节点：*

**var** q1 = document.querySelector('#q1');

*// 通过querySelectorAll获取q1节点内的符合条件的所有节点：*

**var** ps = q1.querySelectorAll('div.highlighted > p');

**修改DOM**

拿到一个DOM节点后，我们可以对它进行更新。

可以直接修改节点的文本，方法有两种：

一种是修改innerHTML属性，这个方式非常强大，不但可以修改一个DOM节点的文本内容，还可以直接通过HTML片段修改DOM节点内部的子树：

// 获取<p id="p-id">...</p>

var p = document.getElementById('p-id');

// 设置文本为abc:

p.innerHTML = 'ABC'; // <p id="p-id">ABC</p>

// 设置HTML:

p.innerHTML = 'ABC <span style="color:red">RED</span> XYZ';

// <p>...</p>的内部结构已修改

用innerHTML时要注意，是否需要写入HTML。如果写入的字符串是通过网络拿到了，要注意对字符编码来避免XSS攻击。

第二种是修改innerText或textContent属性，这样可以自动对字符串进行HTML编码，保证无法设置任何HTML标签：

// 获取<p id="p-id">...</p>

var p = document.getElementById('p-id');

// 设置文本:

p.innerText = '<script>alert("Hi")</script>';

// HTML被自动编码，无法设置一个<script>节点:

*// <p id="p-id">&lt;script&gt;alert("Hi")&lt;/script&gt;</p>*

两者的区别在于读取属性时，innerText不返回隐藏元素的文本，而textContent返回所有文本。另外注意IE<9不支持textContent。

修改CSS也是经常需要的操作。DOM节点的style属性对应所有的CSS，可以直接获取或设置。因为CSS允许font-size这样的名称，但它并非JavaScript有效的属性名，所以需要在JavaScript中改写为驼峰式命名fontSize：

// 获取<p id="p-id">...</p>

var p = document.getElementById('p-id');

// 设置CSS:

p.style.color = '#ff0000';

p.style.fontSize = '20px';

p.style.paddingTop = '2em';

**插入DOM**

当我们获得了某个DOM节点，想在这个DOM节点内插入新的DOM，应该如何做？

如果这个DOM节点是空的，例如，<div></div>，那么，直接使用innerHTML = '<span>child</span>'就可以修改DOM节点的内容，相当于“插入”了新的DOM节点。

如果这个DOM节点不是空的，那就不能这么做，因为innerHTML会直接替换掉原来的所有子节点。

有两个办法可以插入新的节点。一个是使用appendChild，把一个子节点添加到父节点的最后一个子节点。例如：

*<!-- HTML结构 -->*

<p id="js">JavaScript</p>

<div id="list">

<p id="java">Java</p>

<p id="python">Python</p>

<p id="scheme">Scheme</p>

</div>

把<p id="js">JavaScript</p>添加到<div id="list">的最后一项：

**var**

js = document.getElementById('js'),

**list** = document.getElementById('list');

**list**.appendChild(js);

现在，HTML结构变成了这样：

*<!-- HTML结构 -->*

<div id="list">

<p id="java">Java</p>

<p id="python">Python</p>

<p id="scheme">Scheme</p>

<p id="js">JavaScript</p>

</div>

因为我们插入的js节点已经存在于当前的文档树，因此这个节点首先会从原先的位置删除，再插入到新的位置。

更多的时候我们会从零创建一个新的节点，然后插入到指定位置：

**var**

**list** = document.getElementById('list'),

haskell = document.createElement('p');

haskell.id = 'haskell';

haskell.innerText = 'Haskell';

**list**.appendChild(haskell);

这样我们就动态添加了一个新的节点：

*<!-- HTML结构 -->*

<div id="list">

<p id="java">Java</p>

<p id="python">Python</p>

<p id="scheme">Scheme</p>

<p id="haskell">Haskell</p>

</div>

动态创建一个节点然后添加到DOM树中，可以实现很多功能。举个例子，下面的代码动态创建了一个<style>节点，然后把它添加到<head>节点的末尾，这样就动态地给文档添加了新的CSS定义：

**var** d = document.createElement('style');

d.setAttribute('type', 'text/css');

d.innerHTML = 'p { color: red }';

document.getElementsByTagName('head')[0].appendChild(d);

可以在Chrome的控制台执行上述代码，观察页面样式的变化。

insertBefore

如果我们要把子节点插入到指定的位置怎么办？可以使用parentElement.insertBefore(newElement, referenceElement);，子节点会插入到referenceElement之前。

还是以上面的HTML为例，假定我们要把Haskell插入到Python之前：

*<!-- HTML结构 -->*

<div id="list">

<p id="java">Java</p>

<p id="python">Python</p>

<p id="scheme">Scheme</p>

</div>

可以这么写：

**var**

**list** = document.getElementById('list'),

ref = document.getElementById('python'),

haskell = document.createElement('p');

haskell.id = 'haskell';

haskell.innerText = 'Haskell';

**list**.insertBefore(haskell, ref);

新的HTML结构如下：

*<!-- HTML结构 -->*

<div id="list">

<p id="java">Java</p>

<p id="haskell">Haskell</p>

<p id="python">Python</p>

<p id="scheme">Scheme</p>

</div>

可见，使用insertBefore重点是要拿到一个“参考子节点”的引用。很多时候，需要循环一个父节点的所有子节点，可以通过迭代children属性实现：

**var**

i, c,

**list** = document.getElementById('list');

**for** (i = 0; i < **list**.children.length; i++) {

c = **list**.children[i]; *// 拿到第i个子节点*

}

**删除DOM**

删除一个DOM节点就比插入要容易得多。

要删除一个节点，首先要获得该节点本身以及它的父节点，然后，调用父节点的removeChild把自己删掉：

*// 拿到待删除节点:*

**var** **self** = document.getElementById('to-be-removed');

*// 拿到父节点:*

**var** **parent** = **self**.parentElement;

*// 删除:*

**var** removed = **parent**.removeChild(**self**);

removed === **self**; *// true*

注意到删除后的节点虽然不在文档树中了，但其实它还在内存中，可以随时再次被添加到别的位置。

当你遍历一个父节点的子节点并进行删除操作时，要注意，children属性是一个只读属性，并且它在子节点变化时会实时更新。

例如，对于如下HTML结构：

<div id="parent">

<p>First</p>

<p>Second</p>

</div>

当我们用如下代码删除子节点时：

**var** **parent** = document.getElementById('parent');

**parent**.removeChild(**parent**.children[0]);

**parent**.removeChild(**parent**.children[1]); *// <-- 浏览器报错*

浏览器报错：parent.children[1]不是一个有效的节点。原因就在于，当<p>First</p>节点被删除后，parent.children的节点数量已经从2变为了1，索引[1]已经不存在了。

因此，删除多个节点时，要注意children属性时刻都在变化。

可以使用[addEventListener()](http://www.softwhy.com/forum.php?mod=viewthread&tid=6414)和[attachEvent()](http://www.softwhy.com/forum.php?mod=viewthread&tid=6414)为onload事件绑定事件处理函数，下面分别介绍一下：  
addEventListener()是当前标准的一种事件处理函数绑定方式，但是IE8和IE8以下的浏览器并不支持此方式，实例如下:

<!DOCTYPE html>

<html>

<head>

<meta charset=" utf-8">

<meta name="author" content="<http://www.softwhy.com/>" />

<title>window.onload用法-蚂蚁部落</title>

<style type="text/css">

#bg{

  width:100px;

  height:100px;

  border:2px solid red;

}

</style>

<script type="text/javascript">

window.addEventListener("load",bg,false);

window.addEventListener("load",changeW,false);

function bg(){

  document.getElementById("bg").style.backgroundColor="#F90";

}

function changeW(){

  document.getElementById("bg").style.width="200px";

}

</script>

</head>

<body>

  <div id="bg"></div>

</body>

</html>

**addEventListener("type",函数名,false)**

关于localStorage



**多参数时利用JSON.stringify(a[i]) == JSON.stringify(obj)判断类是否相等**

**forEach方法中的function回调有三个参数：第一个参数是遍历的数组内容，第二个参数是对应的数组索引，第三个参数是数组本身**

**因此：**

**[].forEach(function(value,index,array){**

**//code something**

**});**

**等价于：**

**$.each([],function(index,value,array){**

**//code something**

**})**

**写一个例子；**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6** | **var arr = [1,2,3,4];**  **arr.forEach(function(value,index,array){**  **array[index] == value;    //结果为true**  **sum+=value;**  **});**  **console.log(sum);    //结果为 8** |

**JavaScript同源策略**

同源的定义：同源指两个网页，它们的协议（protocal）、端口（port）和主机（host）一致。

比如下面这两个网页：

http://www.example.com/sam.html

https://www.example.com/sam.html

它们的协议不一，一个是 http，一个是 https，所以不同源。同理可推端口、主机。

同源策略限制了一个域的文档或脚本如何与另一个域的资源交互

因为 http://localhost 与 http://www.zfanw.com 不同源，所以**浏览器会禁止**来自 localhost 的脚本访问跨域资源。Chrome 控制台中提到的 Access-Control-Allow-Origin，正是我们邀请其它域的脚本访问的方法。跨域写入的情况可以参照读取部分的说明，这也是时下非常常见的，比如我有一个 API 服务器部署在另一个域名下，但我依然可以通过 ajax 请求将数据存储到 API 服务器中，只要我们在 Access-Control-Allow-Origin 中许可了来自其它源的请求。出于安全考虑，同源策略默认不允许源 A 的脚本读取源 B 的资源，但却允许执行源 B 的资源。这个概念也有些拗⼝。简单说，我这个博客，调用了 Google CDN 提供的 jQuery，它的源显然与我的博客不同，但我却可以用它来操作我的博客页面 DOM，它也可以读取我的 cookie、localStorage 等。假设一个场景，Google CDN 上的 jQuery 被注入了恶意代码，或者被劫持，指向另一个带有恶意代码的 jQuery，则引用它的网站就很危险了，因为这恶意 jQuery 现在可以读取我们网站的内容，并且向其它地方写入。  
4.跨域漏洞  
尽管各个浏览器都遵循都远策略，可是通过一些奇淫巧计会绕过同源的限制进行跨域。举个例子[UC浏览器跨本地域XSS可盗取任意域Cookie](https://link.zhihu.com/?target=http%3A//www.wooyun.org/bugs/wooyun-2013-023920) ，一段js代码本来是普通http域的权限，可是用过下载页面的一个xss，从而具有了file域的权限，而file域对权限限制也不严格，允许从file域跨到任意一个http域权限去执行js代码，经过了两次跨域操作，最终实现了普通http域向任意一个域注射js的目的，同源策略被完全破坏。  
4.最严重的跨域——跨特权域  
所有跨域行为中最严重的莫过于跨特权域。在客户端中，有一些域的权限比较高，例如file://这个协议域，ie中允许调用ActiveX执行cmd，Webkit中可以读取磁盘上的文件。另外还有客户端自定义的特权域，例如Chrome的chrome://downloads/域，有着执行exe的权限。这事儿太严重了，这不仅意味着任意一个网站都可以浏览/修改你的浏览器配置，而且可以不经你同意上传文件，甚至会在你的电脑上安装木马。跨特权域最经典的例子莫过于黑哥去年讲的 《跨过的那些客户端》。

从所周知，闭包函数可以访问包含函数的作用域，所以，回调函数可以访问包含函数的变量，甚至是全局变量。

轮询类中的属性

function logStuff (userData) {

if ( typeof userData === "string")

{

console.log(userData);

}

else if ( typeof userData === "object")

{

for (var item in userData) {

console.log(item + ": " + userData[item]);

}

}

}

使用含有this对象的回调函数的问题  
当回调函数是一个含有this对象的方法时，我们必须修改执行回调函数的方法以保护this对象的内容。否则this对象将会指向全局的window对象(如果回调函数传递给了全局函数)，或指向包含函数。让我们看看下面的代码：

// Define an object with some properties and a method

// We will later pass the method as a callback function to another function

var clientData = {

id: 094545,

fullName: "Not Set",

// setUserName is a method on the clientData object

setUserName: function (firstName, lastName) {

// this refers to the fullName property in this object

this.fullName = firstName + " " + lastName;

}

}

function getUserInput(firstName, lastName, callback) {

// Do other stuff to validate firstName/lastName here

// Now save the names

callback (firstName, lastName);

}

在下面的示例代码中，当clientData.setUserName被执行时，this.fullName不会设置clientData 对象中的属性fullName，而是设置window 对象中的fullName，因为getUserInput是一个全局函数。出现这种现象是因为在全局函数中this对象指向了window对象。

getUserInput ("Barack", "Obama", clientData.setUserName);

console.log (clientData.fullName);// Not Set

// The fullName property was initialized on the window object

console.log (window.fullName); // Barack Obama

使用Call或Apply函数保护this对象

我们可以通过使用 Call 或 Apply函数来解决前面示例中的问题。到目前为止，我们知道JavaScript中的每一个函数都有两个方法:Call和Apply。这些方法可以被用来在函数内部设置this对象的内容，并内容传递给函数参数指向的对象。

这听起来很复杂，但让我们看看Apply和Call的使用是多么容易。为解决前面例子中出现的问题，我们使用Apply函数如下：

//Note that we have added an extra parameter for the callback object, called "callbackObj"

function getUserInput(firstName, lastName, callback, callbackObj) {

// Do other stuff to validate name here

// The use of the Apply function below will set the this object to be callbackObj

callback.apply (callbackObj, [firstName, lastName]);

}

通过Apply函数正确地设置this对象，现在我们可以正确地执行回调函数并它正确地设置clientData对象中的fullName属性。

// We pass the clientData.setUserName method and the clientData object as parameters. The clientData object will be used by the Apply function to set the this object

 getUserInput ("Barack", "Obama", clientData.setUserName, clientData);

// the fullName property on the clientData was correctly set

console.log (clientData.fullName); // Barack Obama

**关于JS中的apply与call方法**

1、每个函数都包含两个非继承而来的方法：apply()和call()。   
2、他们的用途相同，都是在特定的作用域中调用函数。   
3、接收参数方面不同，apply()接收两个参数，一个是函数运行的作用域(this)，另一个是参数数组。  
call()方法第一个参数与apply()方法相同，但传递给函数的参数必须列举出来。

**call方法:**   
语法：call([thisObj[,arg1[, arg2[,   [,.argN]]]]])   
定义：调用一个对象的一个方法，以另一个对象替换当前对象。   
说明：   
call 方法可以用来代替另一个对象调用一个方法。call 方法可将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。   
如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。

1. function add(a,b)
2. {
3. alert(a+b);
4. }
5. function sub(a,b)
6. {
7. alert(a-b);
8. }
10. add.call(sub,3,1);

**apply方法：**   
语法：apply([thisObj[,argArray]])   
定义：应用某一对象的一个方法，用另一个对象替换当前对象。   
说明：   
如果 argArray 不是一个有效的数组或者不是 arguments 对象，那么将导致一个 TypeError。   
如果没有提供 argArray 和 thisObj 任何一个参数，那么 Global 对象将被用作 thisObj， 并且无法被传递任何参数。

　例1：

复制代码代码如下:

window.firstName = "diz";   
window.lastName = "song";   
var myObject = { firstName: "my", lastName: "Object" };   
function HelloName() {   
　　console.log("Hello " + this.firstName + " " + this.lastName, " glad to meet you!");   
}   
HelloName.call(window); //huo .call(this);   
HelloName.call(myObject);

　　运行结果为：   
Hello diz song glad to meet you!   
Hello my Object glad to meet you!   
　　例2：

复制代码代码如下:

function sum(num1, num2) {   
return num1 + num2;   
}   
console.log(sum.call(window, 10, 10)); //20   
console.log(sum.apply(window,[10,20])); //30

　　分析：在例1中，我们发现apply()和call()的真正用武之地是能够扩充函数赖以运行的作用域，如果我们想用传统的方法实现，请见下面的代码：

复制代码代码如下:

window.firstName = "diz";   
window.lastName = "song";   
var myObject = { firstName: "my", lastName: "Object" };   
function HelloName() {   
console.log("Hello " + this.firstName + " " + this.lastName, " glad to meet you!");   
}   
HelloName(); //Hello diz song glad to meet you!   
myObject.HelloName = HelloName;   
myObject.HelloName(); //Hello my Object glad to meet you!

　　见加红的代码，我们发现，要想让HelloName()函数的作用域在对象myObject上，我们需要动态创建myObject的HelloName属性，此属性作为指针指向HelloName()函数，这样，当我们调用myObject.HelloName()时，函数内部的this变量就指向myObjecct，也就可以调用该对象的内部其他公共属性了。   
　　通过分析例2，我们可以看到call()和apply()函数的真正运用之处，在实际项目中，还需要根据实际灵活加以处理！

# jQuery

## 选择器

层级选择器：$('ancestor descendant')只要具有上下层级关系即可，不需要一定是直属的。

子选择器： $('parent>child')类似层级选择器，但是限定了层级关系必须是父子关系，就是<child>节点必须是<parent>节点的直属子节点。

过滤器：过滤器一般不单独使用，它通常附加在选择器上，帮助我们更精确地定位元素。观察过滤器的效果：

$('ul.lang li'); // 选出JavaScript、Python和Lua 3个节点

$('ul.lang li:first-child'); // 仅选出JavaScript

$('ul.lang li:last-child'); // 仅选出Lua

$('ul.lang li:nth-child(2)'); // 选出第N个元素，N从1开始

$('ul.lang li:nth-child(even)'); // 选出序号为偶数的元素

$('ul.lang li:nth-child(odd)'); // 选出序号为奇数的元素

表单选择器：

针对表单元素，jQuery还有一组特殊的选择器：

* :input：可以选择<input>，<textarea>，<select>和<button>；
* :file：可以选择<input type="file">，和input[type=file]一样；
* :checkbox：可以选择复选框，和input[type=checkbox]一样；
* :radio：可以选择单选框，和input[type=radio]一样；
* :focus：可以选择当前输入焦点的元素，例如把光标放到一个<input>上，用$('input:focus')就可以选出；
* :checked：选择当前勾上的单选框和复选框，用这个选择器可以立刻获得用户选择的项目，如$('input[type=radio]:checked')；
* :enabled：可以选择可以正常输入的<input>、<select> 等，也就是没有灰掉的输入；
* :disabled：和:enabled正好相反，选择那些不能输入的。

此外，jQuery还有很多有用的选择器，例如，选出可见的或隐藏的元素：

$('div:visible'); // 所有可见的div

$('div:hidden'); // 所有隐藏的div

## 查找与过滤

最常见的查找是在某个节点的所有子节点中查找，使用find()方法，它本身又接收一个任意的**选择器**。

*<!-- HTML结构 -->*

<ul class="lang">

<li class="js dy">JavaScript</li>

<li class="dy">Python</li>

<li id="swift">Swift</li>

<li class="dy">Scheme</li>

<li name="haskell">Haskell</li>

</ul>

用find()查找：

**var** ul = $('ul.lang'); *// 获得<ul>*

**var** dy = ul.find('.dy'); *// 获得JavaScript, Python, Scheme*

**var** swf = ul.find('#swift'); *// 获得Swift*

**var** hsk = ul.find('[name=haskell]'); *// 获得Haskell*

如果要从当前节点开始向上查找，使用parent()方法：

**var** swf = $('#swift'); *// 获得Swift*

**var** **parent** = swf.**parent**(); *// 获得Swift的上层节点<ul>*

**var** a = swf.**parent**('div.red'); *// 从Swift的父节点开始向上查找，直到找到某个符合条件的节点并返回*

对于位于同一层级的节点，可以通过next()和prev()方法，例如：

当我们已经拿到Swift节点后：

var swift = $('#swift');

swift.**next**(); // Scheme

swift.**next**('[name=haskell]'); // Haskell，因为Haskell是后续第一个符合选择器条件的节点

swift.prev(); // Python

swift.prev('.js'); // JavaScript，因为JavaScript是往前第一个符合选择器条件的节点

**过滤**

和函数式编程的map、filter类似，jQuery对象也有类似的方法。

filter()方法可以过滤掉不符合选择器条件的节点：

**var** langs = $('ul.lang li'); *// 拿到JavaScript, Python, Swift, Scheme和Haskell*

**var** a = langs.filter('.dy'); *// 拿到JavaScript, Python, Scheme*

或者传入一个函数，要特别注意函数内部的this被绑定为DOM对象，不是jQuery对象：

**var** langs = $('ul.lang li'); *// 拿到JavaScript, Python, Swift, Scheme和Haskell*

langs.filter(**function** () {

**return** **this**.innerHTML.indexOf('S') === 0; *// 返回S开头的节点*

}); *// 拿到Swift, Scheme*

map()方法把一个jQuery对象包含的若干DOM节点转化为其他对象：

**var** langs = $('ul.lang li'); *// 拿到JavaScript, Python, Swift, Scheme和Haskell*

**var** arr = langs.map(**function** () {

**return** **this**.innerHTML;

}).get(); *// 用get()拿到包含string的Array：['JavaScript', 'Python', 'Swift', 'Scheme', 'Haskell']*

此外，一个jQuery对象如果包含了不止一个DOM节点，first()、last()和slice()方法可以返回一个新的jQuery对象，把不需要的DOM节点去掉：

**var** langs = $('ul.lang li'); *// 拿到JavaScript, Python, Swift, Scheme和Haskell*

**var** js = langs.first(); *// JavaScript，相当于$('ul.lang li:first-child')*

**var** haskell = langs.last(); *// Haskell, 相当于$('ul.lang li:last-child')*

**利用jq操作DOM**

无参数调用text()是获取文本，传入参数就变成设置文本，HTML也是类似操作

修改css：要高亮显示动态语言，调用jQuery对象的css('name', 'value')方法，我们用一行语句实现

var div = $('#test-div');

div.css('color'); // '#000033', 获取CSS属性

div.css('color', '#336699'); // 设置CSS属性

div.css('color', ''); // 清除CSS属性

如果要修改class属性，可以用jQuery提供的下列方法：

**var** div = $('#test-div');

div.hasClass('highlight'); *// false， class是否包含highlight*

div.addClass('highlight'); *// 添加highlight这个class*

div.removeClass('highlight'); *// 删除highlight这个class*

显示/隐藏DOM：要隐藏一个DOM，我们可以设置CSS的display属性为none，利用css()方法就可以实现。不过，要显示这个DOM就需要恢复原有的display属性，这就得先记下来原有的display属性到底是block还是inline还是别的值。

考虑到显示和隐藏DOM元素使用非常普遍，jQuery直接提供show()和hide()方法，我们不用关心它是如何修改display属性的，总之它能正常工作：

var a = $('a[target=\_blank]');

a.hide(); // 隐藏

a.show(); // 显示

获取DOM信息：attr()和removeAttr()方法用于操作DOM节点的属性：

// <div id="test-div" name="Test" start="1">...</div>

var div = $('#test-div');

div.attr('data'); // undefined, 属性不存在

div.attr('name'); // 'Test'

div.attr('name', 'Hello'); // div的name属性变为'Hello'

div.removeAttr('name'); // 删除name属性

div.attr('name'); // undefined

操作表单：val()

修改DOM：要添加新的DOM节点，除了通过jQuery的html()这种暴力方法外，还可以用append()方法。append()把DOM添加到最后，prepend()则把DOM添加到最前。拿到jQuery对象后直接调用remove()方法就可以了。



## jQuery的事件

首先举个例子

/\* HTML:

\*

\* <a id="test-link" href="#0">点我试试</a>

\*

\*/

// 获取超链接的jQuery对象:

var a = $('#test-link');

a.on('click', function () {

alert('Hello!');

});

on方法用来绑定一个事件，我们需要传入时间名称和对应的处理参数。

jQuery能够绑定的事件主要包括：

鼠标事件

click: 鼠标单击时触发； dblclick：鼠标双击时触发； mouseenter：鼠标进入时触发； mouseleave：鼠标移出时触发； mousemove：鼠标在DOM内部移动时触发； hover：鼠标进入和退出时触发两个函数，相当于mouseenter加上mouseleave。

键盘事件

键盘事件仅作用在当前焦点的DOM上，通常是<input>和<textarea>。

keydown：键盘按下时触发； keyup：键盘松开时触发； keypress：按一次键后触发。

其他事件

focus：当DOM获得焦点时触发； blur：当DOM失去焦点时触发； change：当<input>、<select>或<textarea>的内容改变时触发； submit：当<form>提交时触发； ready：当页面被载入并且DOM树完成初始化后触发。

其中，ready仅作用于document对象。由于ready事件在DOM完成初始化后触发，且只触发一次，所以非常适合用来写其他的初始化代码。

由于ready事件使用非常普遍，所以可以这样简化：

$(document).ready(**function** () {

*// on('submit', function)也可以简化:*

$('#testForm).submit(function () {

alert('submit!');

});

});

甚至还可以再简化为：

$(**function** () {

*// init...*

});

上面的这种写法最为常见。如果你遇到$(function () {...})的形式，牢记这是document对象的ready事件处理函数。

事件参数

有些事件，如mousemove和keypress，我们需要获取鼠标位置和按键的值，否则监听这些事件就没什么意义了。所有事件都会传入Event对象作为参数，可以从Event对象上获取到更多的信息：

$(**function** () {

$('#testMouseMoveDiv').mousemove(**function** (e) {

$('#testMouseMoveSpan').text('pageX = ' + e.pageX + ', pageY = ' + e.pageY);

});

});

取消绑定

一个已被绑定的事件可以解除绑定，通过off('click', function)实现：

**function** hello() {

alert('hello!');

}

a.click(hello); *// 绑定事件*

*// 10秒钟后解除绑定:*

setTimeout(**function** () {

a.off('click', hello);

}, 10000);

需要特别注意的是，下面这种写法是无效的：

*// 绑定事件:*

a.click(**function** () {

alert('hello!');

});

*// 解除绑定:*

a.off('click', **function** () {

alert('hello!');

});

这是因为两个匿名函数虽然长得一模一样，但是它们是两个*不同*的函数对象，off('click', function () {...})无法移除已绑定的第一个匿名函数。

为了实现移除效果，可以使用off('click')一次性移除已绑定的click事件的所有处理函数。

同理，无参数调用off()一次性移除已绑定的所有类型的事件处理函数。

**事件触发条件**

一个需要注意的问题是，事件的触发总是由用户操作引发的。例如，我们监控文本框的内容改动：

**var** input = $('#test-input');

input.change(**function** () {

console.log('changed...');

});

当用户在文本框中输入时，就会触发change事件。但是，如果用JavaScript代码去改动文本框的值，将**不会**触发change事件：

var input = $('#test-input');

input.val('change it!'); // 无法触发change事件

有些时候，我们希望用代码触发change事件，可以直接调用无参数的change()方法来触发该事件：

var input = $('#test-input');

input.val('change it!');

input.change(); // 触发change事件

input.change()相当于input.trigger('change')，它是trigger()方法的简写。

为什么我们希望手动触发一个事件呢？如果不这么做，很多时候，我们就得写两份一模一样的代码。

例子：

练习

对如下的Form表单：

*<!-- HTML结构 -->*

<form id="test-form" action="test">

<legend>请选择想要学习的编程语言：</legend>

<fieldset>

<p><label class="selectAll"><input type="checkbox"> <span class="selectAll">全选</span><span class="deselectAll">全不选</span></label> <a href="#0" class="invertSelect">反选</a></p>

<p><label><input type="checkbox" name="lang" value="javascript"> JavaScript</label></p>

<p><label><input type="checkbox" name="lang" value="python"> Python</label></p>

<p><label><input type="checkbox" name="lang" value="ruby"> Ruby</label></p>

<p><label><input type="checkbox" name="lang" value="haskell"> Haskell</label></p>

<p><label><input type="checkbox" name="lang" value="scheme"> Scheme</label></p>

<p><button type="submit">Submit</button></p>

</fieldset>

</form>

绑定合适的事件处理函数，实现以下逻辑：

当用户勾上“全选”时，自动选中所有语言，并把“全选”变成“全不选”；

当用户去掉“全不选”时，自动不选中所有语言；

当用户点击“反选”时，自动把所有语言状态反转（选中的变为未选，未选的变为选中）；

当用户把所有语言都手动勾上时，“全选”被自动勾上，并变为“全不选”；

当用户手动去掉选中至少一种语言时，“全不选”自动被去掉选中，并变为“全选”。

窗体顶端

'use strict';

var

form = $('#test-form'),

langs = form.find('[name=lang]'),

selectAll = form.find('label.selectAll :checkbox'),

selectAllLabel = form.find('label.selectAll span.selectAll'),

deselectAllLabel = form.find('label.selectAll span.deselectAll'),

invertSelect = form.find('a.invertSelect');

// 重置初始化状态:

form.find('\*').show().off();

form.find(':checkbox').prop('checked', false).off();

deselectAllLabel.hide();

// 拦截form提交事件:

form.off().submit(function (e) {

e.preventDefault();

alert(form.serialize());

});

// TODO:绑定事件

// 当用户勾上“全选”时，自动选中所有语言，并把“全选”变成“全不选”；

// 当用户去掉“全不选”时，自动不选中所有语言；

selectAll.click(function () {

if(selectAll.is(':checked')){

langs.prop('checked', true);

selectAllLabel.hide();

deselectAllLabel.show();

} else {

langs.prop('checked', false);

selectAllLabel.show();

deselectAllLabel.hide();

}

});

// 当用户点击“反选”时，自动把所有语言状态反转（选中的变为未选，未选的变为选中）；

invertSelect.click(function () {

langs.map(function () {

$(this).prop('checked', !$(this).is(':checked'))

});

});

// 当用户把所有语言都手动勾上时，“全选”被自动勾上，并变为“全不选”

// 当用户手动去掉选中至少一种语言时，“全不选”自动被去掉选中，并变为“全选”。

langs.click(function () {

var langsChecked = form.find('[name=lang]:checked');

if (langsChecked.length === langs.length) {

selectAll.prop('checked', true);

selectAllLabel.hide();

deselectAllLabel.show();

} else {

selectAll.prop('checked', false);

selectAllLabel.show();

deselectAllLabel.hide();

}

});

jQuery的动画

为什么有的动画没有效果

你可能会遇到，有的动画如slideUp()根本没有效果。这是因为jQuery动画的原理是逐渐改变CSS的值，如height从100px逐渐变为0。但是很多不是block性质的DOM元素，对它们设置height根本就不起作用，所以动画也就没有效果。

此外，jQuery也没有实现对background-color的动画效果，用animate()设置background-color也没有效果。这种情况下可以使用CSS3的transition实现动画效果。

练习

在执行删除操作时，给用户显示一个动画比直接调用remove()要更好。请在表格删除一行的时候添加一个淡出的动画效果：

窗体顶端

练习

在执行删除操作时，给用户显示一个动画比直接调用remove()要更好。请在表格删除一行的时候添加一个淡出的动画效果：

窗体顶端

'use strict';

function deleteFirstTR() {

var tr = $('#test-table>tbody>tr:visible').first();

tr.fadeOut(500, () => $(this).remove());

}

deleteFirstTR();

对于默认值的处理，我们用了一个简单的&&和||短路操作符，总能得到一个有效的值。

*// 要考虑到各种情况:*

*// options为undefined*

*// options只有部分key*

**var** bgcolor = options && options.backgroundColor || '#fffceb';

**var** color = options && options.color || '#d85030';

另一种方法是使用jQuery提供的辅助方法$.extend(target, obj1, obj2, ...)，它把多个object对象的属性合并到第一个target对象中，遇到同名属性，总是使用靠后的对象的值，也就是越往后优先级越高：

*// 把默认值和用户传入的options合并到对象{}中并返回:*

**var** opts = $.extend({}, {

backgroundColor: '#00a8e6',

color: '#ffffff'

}, options);

紧接着用户对highlight2()提出了意见：每次调用都需要传入自定义的设置，能不能让我自己设定一个缺省值，以后的调用统一使用无参数的highlight2()？

也就是说，我们设定的默认值应该能允许用户修改。

那默认值放哪比较合适？放全局变量肯定不合适，最佳地点是$.fn.highlight2这个函数对象本身。

于是最终版的highlight()终于诞生了：

$.fn.highlight = **function** (options) {

*// 合并默认值和用户设定值:*

**var** opts = $.extend({}, $.fn.highlight.defaults, options);

**this**.css('backgroundColor', opts.backgroundColor).css('color', opts.color);

**return** **this**;

}

*// 设定默认值:*

$.fn.highlight.defaults = {

color: '#d85030',

backgroundColor: '#fff8de'

}

这次用户终于满意了。用户使用时，只需一次性设定默认值：

$.fn.highlight.defaults.color = '#fff';

$.fn.highlight.defaults.backgroundColor = '#000';

然后就可以非常简单地调用highlight()了。

窗体底端

窗体底端

窗体底端