浅谈自执行函数（立即调用的函数表达式） - 简书

笔记本: 程序员
创建时间: 2020/8/20 星期四 10:12
标签: Android
URL: https://www.jianshu.com/p/c64bfbcd34c3

浅谈自执行函数（立即调用的函数表达式）

celineWong7

1 2018.12.27 15:21:35 字数 2,023 阅读 12,357

> 在JavaScript中，会遇到自执行匿名函数：`(function () {/*code*/} ) ()`。
> 这个结构大家并不陌生，但若要说：为什么要括弧起来？它的应用场景有哪些？.....就会有点模糊。
> 此处作个小结。

本文篇幅比较长，但例子都很简单，可以跳跃式阅读。

一、函数的声明与执行

我们先来看下最初的函数声明与执行：

```
// 声明函数fun0
function fun0(){
    console.log("fun0");
}

//执行函数fun0
fun0(); // fun0
```

除了上面这种最常见的函数声明方式，还有变量赋值方式的，如下：

```
// 声明函数fun1 - 变量方式
var fun1 = function(){
    console.log("fun1");
}

// 执行函数fun1
fun1(); // fun1
```

二、 函数的一点猜想

既然函数名加上括号`fun1()`就是执行函数。
思考：直接取赋值符号右侧的内容直接加个括号，是否也能执行?
试验如下，直接加上小括弧：

```
function(){
    console.log("fun");
}();
```

以上会报错 line1：`Uncaught SyntaxError: Unexpected token (`。
分析：`function`是声明函数关键字，若非变量赋值方式声明函数，默认其后面需要跟上函数名的。

加上函数名看看：

```
function fun2(){
    console.log("fun2");
}();
```

以上会报错 line3：`Uncaught SyntaxError: Unexpected token )`。
分析： 声明函数的结构花括弧后面不能有其他符号（比如此处的小括弧）。

不死心的再胡乱试一下，给它加个实参（表达式）：

```
function fun3(){
    console.log("fun3");
}(1);
```

不会报错，但不会输出结果`fun3`。
分析： 以上代码相当于在声明函数后，又声明了一个毫无关系的表达式。相当于如下代码形式：

```
    function fun3(){
        console.log("fun3");
    }

(1);

// 若此处执行fun3函数，可以输出结果
fun3(); //"fun3"
```

三、自执行函数表达式

1. 正儿八经的自执行函数

想要解决上面问题，可以采用小括弧将要执行的代码包含住（方式一），如下：

```
// 方式一
    (function fun4(){
        console.log("fun4");
    }()); // "fun4"
```

分析：因为在JavaScript语言中，`()`里面不能包含语句（只能是表达式），所以解析器在解析到`function`关键字的时候，会把它们当作function表达式，而不是正常的函数声明。
除了上面直接整个包含住，也可以只包含住函数体（方式二），如下：

```
// 方式二
    (function fun5(){
        console.log("fun5");
    })();// "fun4"
```

写法上建议采用方式一（这是参考文的建议。但实际上，我个人觉得方式二比较常见）。

2. “歪瓜裂枣”的自执行函数

除了上面 `()` 小括弧可以把 `function` 关键字作为函数声明的含义转换成函数表达式外，JavaScript的 `&&` 与操作、 `||` 或操作、 `,` 逗号等操作符也有这个效果。

```
    true && function () { console.log("true &&") } (); // "true &&"
    false || function () { console.log("true ||") } (); // "true ||"
    0, function () { console.log("0,") } (); // "0,"

// 此处要注意： &&, || 的短路效应。即： false && (表达式1) 是不会触发表达式1；
// 同理，true || (表达式2) 不会触发表达式2
```

如果不在意返回值，也不在意代码的可读性，我们甚至还可以使用一元操作符（ `!` `~` `-` `+` ），函数同样也会立即执行。

```
    !function () { console.log("!"); } (); //"!"
    ~function () { console.log("~"); } (); //"~"
    -function () { console.log("-"); } (); //"-"
    +function () { console.log("+"); } (); //"+"
```

甚至还可以使用 `new` 关键字：

```
// 注意：采用new方式，可以不要再解释花括弧 `}` 后面加小括弧 `()`
new function () { console.log("new"); } //"new"

// 如果需要传递参数
new function (a) { console.log(a); } ("newwwwwwwww"); //"newwwwwwwww"
```

嗯，最好玩的是赋值符号 `=` 同样也有此效用（例子中的 `i` 变量方式）：

```
//此处 要注意区分 i 和 j 不同之处。前者是函数自执行后返回值给 i ；后者是声明一个函数
    var i = function () { console.log("output i:"); return 10; } (); // "outpu
    var j = function () { console.log("output j:"); return 99;}
    console.log(i); // 10
    console.log(j); // ƒ () { console.log("output j:"); return 99;}
```

上面提及到，要注意区分 `var i` 和 `var j` 不同之处（前者是函数自执行后返回值给 `i` ；后者是声明一个函数，函数名为 `j` ）。如果是看代码，我们需要查看

代码结尾是否有没有`()`才能区分。一般为了方便开发人员阅读，我们会采用下面这种方式：

```
    var i2 = (function () { console.log("output i2:"); return 10; } ()); // "
    var i3 = (function () { console.log("output i3:"); return 10; }) (); // "
// 以上两种都可以，但依旧建议采用第一种 i2 的方式。（个人依旧喜欢第二种i3方式）
```

四、自执行函数的应用

1. for循环 + setTimeout 例子

直接来看一个例子。`for`循环里面通过延时器输出索引`i`

```
for( var i=0;i<3;i++){
    setTimeout(function(){
        console.log(i);
    }
    ,300);
}
// 输出结果 3,3,3
```

输出结果并不是我们所预想的`1,2,3`。当然，这个要涉及到setTimeout 的原理了，即使把300ms改成0ms，同样也会输出`3,3,3`。具体可以查看博文 setTimeout(0) 的作用 。这里摘取其中一段说明。

> JavaScript是单线程执行的，无法同时执行多段代码。当某段代码正在执行时，后续任务都必须等待，形成一个队列。只有当前任务执行完毕，才会从队列中取出下一个任务——也就是常说的“阻塞式执行”。

上面代码中设定了一个`setTimeout`，那浏览器会在合适时间（此处是300ms后）把代码插入任务队列，等待当前的`for`循环代码执行完毕再执行。（注意：setTimeout 虽然指定了延时的时间，但并不能保证执行的时间与设定的延时时间一直，是否准确取决于 JavaScript 线程是拥挤还是空闲。）

上面说了那么多，都是在分析为什么会输出`3,3,3`。那怎么样才能输出`1,2,3`呢?

看看下面的方式（写法一）：把`setTimeout`代码包含在匿名自执行函数里面，就可以实现“锁住”索引`i`，正常输出索引值。

```
for( var i=0;i<3;i++){
    (function(lockedIndex){
        setTimeout(function(){
            console.log(lockedIndex);
        }
        ,300);
    })(i);
}
// 输出 "1,2,3"
```

分析：尽管循环执行结束，`i`值已经变成了3。但因遇到了自执行函数，当时的`i`值已经被`lockedIndex`锁住了。也可以理解为 自执行函数属于for循环一部分，每次遍历`i`，自执行函数也会立即执行。所以尽管有延时器，但依旧会保留住立即执行时的`i`值。

*上面的分析有点模糊和牵强，也可以从 **闭包** 角度出发分析的。但鄙人“闭包”概念模糊，先遗憾下，以后再补充分析了。QAQ*

除了上面的写法，也可以直接在`setTimeout`第一个参数做自执行（写法二），如下。

注意: 写法二 会比 写法一 先执行。原因不明。

```
for( var i=0;i<3;i++){
    setTimeout((function(lockedInIndex){
        console.log(lockedInIndex);
    })(i)
    ,300);
}
```

关于 自执行函数参数`lockedInIndex`，补充说明以下几点。

注意：自执行函数在 setTimeout 和在 setTimeout 里在第2、3中情况有区别（原因不明，后续再补）。

```
// 1. lockedInIndex变量，也可以换成i，因为和外面的i不在一个作用域
for( var i=0;i<3;i++){
    (function(i){
        setTimeout(function(){
            console.log(i); // 1,2,3
        }
```

```
        ,300);
    })(i);
}

for( var i=0;i<3;i++){
    setTimeout((function(i){
        console.log(i); // 1,2,3
    })(i)
    ,300);
}

// 2. 自执行函数不带入参数
for( var i=0;i<3;i++){
    (function(){
        setTimeout(function(){
            console.log(i); // 3,3,3
        }
        ,300);
    })();
}

for( var i=0;i<3;i++){
    setTimeout((function(){
        console.log(i); // 1,2,3
    })()
    ,300);
}

// 3. 自执行函数只有实参没有写形参
for( var i=0;i<3;i++){
    (function(){
        setTimeout(function(){
            console.log(i); // 3,3,3
        }
        ,300);
    })(i);
}

for( var i=0;i<3;i++){
    setTimeout((function(){
        console.log(i); // 1,2,3
    })(i)
    ,300);
}

// 4. 自执行函数只有形参没有写实参，这种情况不行。因为会导致输出 undefined。
for( var i=0;i<3;i++){
    (function(i){
        setTimeout(function(){
            console.log(i); // undefined,undefined,undefined
        }
        ,300);
    })();
}
```

```
61
62  for( var i=0;i<3;i++){
63      setTimeout((function(i){
64          console.log(i); // undefined,undefined,undefined
65      })()
66      ,300);
67  }
```

2. html元素绑定事件

假设要对页面上的元素安装点击相同的点击事件。我们会考虑如下方式。

```
1   <div id="demo">
2       <p>p1</p>
3       <p>p2</p>
4       <p>p3</p>
5       <p>p4</p>
6       <p>p5</p>
7   </div>
8   <script type="text/javascript">
9       var oDiv = document.getElementById("demo");
10      var eles = oDiv.getElementsByTagName("p");
11
12      for ( var k=0; k < eles.length; k++){
13          eles[k].addEventListener('click',function(e){
14              alert("index is: " + k + ", and this ele is: " + eles[k]); // inde
15          });
16
17          /** 安装事件方式也可以用 onclick 方式。不过这种方式安装多个onclick触发事件
18          // eles[k].onclick = function(){
19          //  alert("index is: " + k + ", and this ele is: " + eles[k]);
20          // }
21      }
22  </script>
```

我们期望点击某个 `p` 元素，能得到该元素所在的索引，但实际是，点击每个 `p`，索引值都是 `5`，而对应的元素都是 `undefined`。
分析：这种现象和上面的延时器类似，JavaScript在执行 `for` 循环语句时，负责给元素安装点击事件，但当用户点击元素触发事件时，`for` 循环语句早就执行完毕了，此时的 `i` 自然是 `5` 了。

一样的，我们也希望“锁住”索引 `i`。所以可以如上采用自执行函数方式（在addEventListener外部）：

```
/** 1. 自执行函数方式一 */
for ( var k=0; k < eles.length; k++){
        (function(k){
            eles[k].addEventListener('click',function(e){
                alert("index is: " + k + ", and this ele is: " + eles[k].inner
            });
        })(k);
    }
```

也可以 **在addEventListener里面** 的处理函数使用自执行函数表达式，具体如下。不过上面的方式更具有可读性。

```
        /** 2. 自执行函数方式二 */
    for ( var k=0; k < eles.length; k++){
            eles[k].addEventListener('click',function(k){
                return function(e){
                    alert("index is: " + k + ", and this ele is: " + eles[k].i
                }
            }(k));
    }
```

当然，除了自执行函数表达式，我们还有一种讨巧的解决办法：

```
    /** 3. 讨巧的解决方案 */
    for ( var k=0; k < eles.length; k++){
        eles[k].index = k;
        eles[k].addEventListener('click',function(e){
            alert("index is: " + this.index + ", and this ele is: " + eles[thi
        });
    }
// 把索引 k 保存在元素的属性中。在点击元素触发事件时，巧用 this 关键字去取出当前点击
```

四、自执行与立即执行

最后来唠嗑下命名方式。

文中对 `(function () {/*code*/} ) ()` 这种表达式，称作为 **自执行匿名函数**（Self-executing anonymous function）；而参考的英文博文中作者更建议称它为 **立即调用的函数表达式**（Immediately-Invoked Function Expression）。

以下是截取该参考博文的例子：

```
// 自执行函数。自己调用自己（递归）
function foo() { foo(); }

// 自执行的匿名函数。
var foo = function () { arguments.callee(); };

// 立即执行匿名函数。但我们习惯称其为：自执行的匿名函数。
(function () { /* code */ } ());

// 立即执行函数。加一个标示名称，可以方便Debug
(function foo() { /* code */ } ());

// 立即调用的函数表达式（IIFE）也可以自执行，不过可能不常用罢了
(function () { arguments.callee(); } ());
(function foo() { foo(); } ());
```

注意：arguments.callee在ECMAScript 5 strict mode里被废弃了。

个人愚见：上面例子中把 **自执行** 解释成 “自己调用自己”，当然和 **立即执行** 相差很大了。但如果把 **自执行** 解释成 “自动执行”，就和 **立即执行** 异曲同工了。命名方式绝对统一也没必要，重要的是能深入了解并应用它们。

参考内容：

1. 深入理解JavaScript系列（4）： 立即调用的函数表达式
2. Immediately-Invoked Function Expression (IIFE)
3.