[笔记][LIKE-JS][8-JS进阶-闭包]

JavaScript

[笔记][LIKE-JS][8-JS进阶-闭包]
- 211. 闭包的基本概念(了解)
- 212. 闭包场景1-封闭作用域(了解)
- 213. 闭包场景2-作用域链条(了解)
- 214. 闭包场景3-高级排他(了解)
- 215. 闭包场景4-参数传递(了解)
- 216. 闭包场景5-函数节流(了解)

211. 闭包的基本概念(了解)

一、闭包技术详解

1.1 什么是闭包？

- 闭包实际上是一种函数，所以闭包技术也是函数技术的一种；闭包能做的事情函数几乎都能做。
- 闭包技术花式比较多，用法也比较灵活，一般开发人员在学习闭包的时候都会遇到瓶颈，主要是因为闭包技术的分界线并不明显。几乎无法用一个特点去区分。
- 当一个内部函数被其外部函数之外的变量引用时，就形成了一个闭包。

```
function A(){
   function B(){
       console.log("Hello YJH!");
   }
   return B;
}
var b = A();
b();//Hello YJH!
```

- 闭包的最大用处有两个：一个是可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

```
<script>
    // 1. 可以读取函数内部的变量
    function test1() {
```

```
        var age = 19;  // age 是局部变量
        return function () {
            console.log(age);
        }
    }

    var func = test1();
    func();
</script>
```

> 其实这跟 `Python` 当中的闭包是一样的。
> 个人总结如下：
> 在 `Python` 中，闭包的定义是一个引用了外部作用域变量的函数。
> 而闭包的表现形式有以下三点：
>
> - 存在函数的嵌套
> - 内部的函数引用了外部函数的变量
> - 外部函数的返回值是内部函数的名字

---

212. 闭包场景1-封闭作用域(了解)

1.2 封闭作用域

- **JavaScript的GC机制**
  - 在javascript中，如果一个对象不再被引用，那么这个对象就会被GC回收，否则这个对象一直会保存在内存中。
- **封闭作用域**
  - 封闭作用域又称值为封闭空间，还有一个昵称叫小闭包，以及匿名函数自调。
  - 基本结构：

```
(function(){})();
;(function(){})();
+(function(){})();
-(function(){})();
?(function(){})();
```

- **技法最大目的: 全局变量私有化**
- **技术优点：**
  - 不污染全局空间！
  - 内部所有的临时变量执行完毕都会释放不占内存。
  - 可以保存全局数据。
  - 更新复杂变量。

> 封闭作用域、封闭空间、小闭包、匿名函数自调是一回事！
> 一般写第一个形式就好

封装框架、函数以及分割文件的时候会用到这个。

> 问号貌似不支持！

个人理解
匿名函数自调，前面把匿名函数包围起来的括号主要是为了让解释器识别对匿名函数的调用。

通过封闭空间可以解决当程序比较大的时候，变量的重名问题。

---

213. 闭包场景2-作用域链条(了解)

1.3 作用域链

- 嵌套之间的函数会形成作用域链，每次对变量的访问实际上都是对整条作用域链的遍历查找。先查找最近的作用域，最后再查找全局作用域。如果在某个作用域找到了对量就会结束本次查找过程。
- **思考？**
  - **对于作用域全局作用域查找快，还是局部作用域查找快？**
  - 局部作用域查找要远远大于全局作用域查找的速度。所以高级的程序设计一般是尽量避免全局查找。
  - 每次访问都是对作用域链的一次遍历查找其中全局作用域是最耗费时间的。
- **解决方案：**
  - 当前目标使用完以后，在退出作用域之前储存这个目标，就可以在下次取到上一次的目标。
- **补充：**

> 变量的生命周期 任何一个变量在内存中都是一个引用，这个变量是有自己的生命周期。周期结束意味着被销毁。 一个变量在它当前的作用域内被声明那一刻相当于变量出生，整个当前作用域执行完毕并退出作用域相当于变量的寿命终止。

1.4 保存作用域

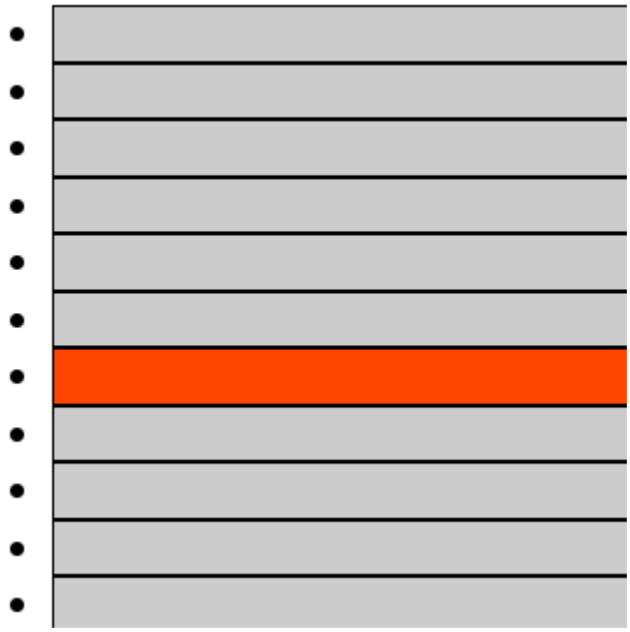
- 保存作用域是一种更高级的闭包技术，如果函数嵌套函数，那么内部的那个函数将形成作用域闭包。简单的说，这种闭包能够达到的好处就是让指令能够绑定一些全局数据去运行；
- **基本结构：**

```
var A=function(){
    return function(){};
}
```

- **优点：**
  - 全局数据隐藏化
  - 可以让某个指令运行时候绑定一些隐藏的全局数据在身上。
- **一句话： 将数据绑定在指令上运行，让指令不再依赖全局数据。**

---

214. 闭包场景3-高级排他(了解)



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <style>
        li{
            background-color: #cccccc;
            border: 1px solid #000;
            cursor: pointer;
        }

        .current{
            background-color: orangered;
        }
    </style>
</head>
<body>
<ul>
    <li class="current"></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
```

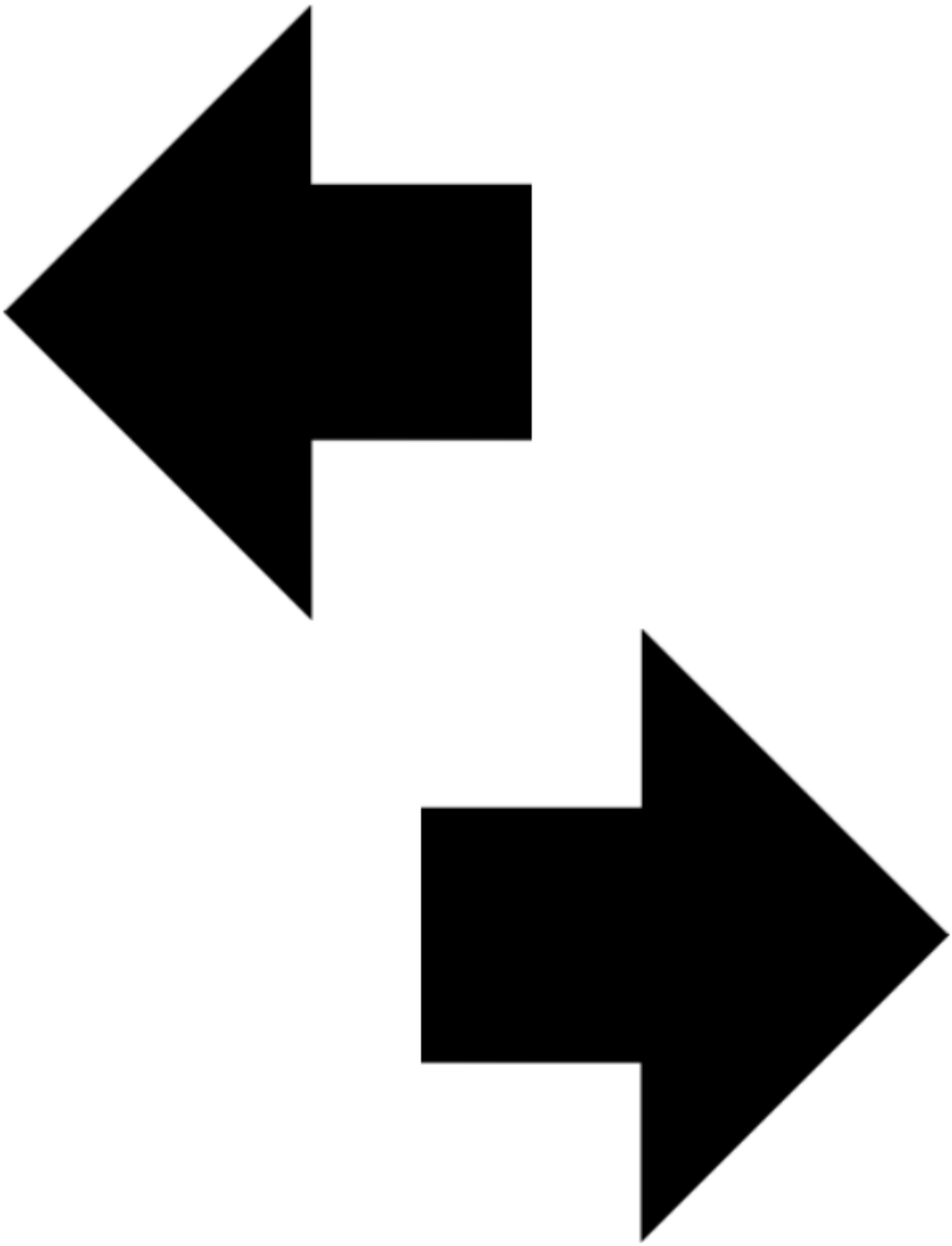
```
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
    </ul>
    <script>
        var allLis = document.getElementsByTagName("li");
        for (var i = 0; i < allLis.length; i++) {
            allLis[i].onmouseover = function () {
                // 排他
                for (var j = 0; j < allLis.length; j++) {
                    allLis[j].className = "";
                }
                // 设置当前样式
                this.className = "current";
            }
        }
    </script>
    </body>
    </html>
```

高级排他

```
    <script>
        var allLis = document.getElementsByTagName("li");
        var lastOne = 0;
        for (var i = 0; i < allLis.length; i++) {
            (function (index) {
                allLis[index].onmouseover = function () {
                    // 清除
                    allLis[lastOne].className = "";
                    // 设置
                    this.className = "current";
                    // 赋值
                    lastOne = index;
                }
            })(i);
        }
    </script>
```

---

215. 闭包场景4-参数传递(了解)



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <style>
        body{
            text-align: center;
        }
    </style>
</head>
```

```
<body>
<img id="l" src="images/left.png" alt="">
<br>
<img id="r" src="images/right.png" alt="">
<script>
    /*function test1(x) {
        return function (y) {
            console.log(x);
            console.log(y);
        }
    }

    // x = 10, y = 20
    test1(10)(20);*/

    var left = 0;
    var right = 0;
    var speed = 60;
    var lImg = document.getElementById("l");
    var rImg = document.getElementById("r");

    /*lImg.onmousedown = function () {
        left -= speed;
        this.style.marginLeft = left + "px";
    };

    rImg.onmousedown = function () {
        right += speed;
        this.style.marginLeft = right + "px";
    };*/

    function move(speed) {
        var num = 0;
        return function () {
            num += speed;
            this.style.marginLeft = num + "px";
        };
    }

    lImg.onmousedown = move(-50);
    rImg.onmousedown = move(50);

</script>
</body>
</html>
```

216. 闭包场景5-函数节流(了解)

`window.onscroll` 和 `window.onresize`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<script>
    /*var timer = null;
    window.onresize = function () {
        clearTimeout(timer);
        timer = setTimeout(function () {
            console.log(document.documentElement.clientWidth);
        }, 400);
    }*/

    function throttle(fn, delay) {
        var timer = null;
        return function () {
            clearTimeout(timer);
            timer = setTimeout(fn, delay);
        };
    }

    window.onresize = throttle(function () {
        console.log(document.documentElement.clientWidth);
    }, 400);
</script>
</body>
</html>
```

> **个人理解**
> 调用 `throttle` 得到了一个闭包函数，这个闭包函数引用了外部自由变量 `timer`。
> 该闭包函数在发生 `window` 大小改变的时候会被调用。
> 每次调用 `throttle`，都会新生成一个自由变量，每次调用间互不影响。

以下是自己写的 `demo` 程序，它说明了每次调用之间闭包引用的自由变量都是独立的，闭包自成一个命名空间。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
```

```
</head>
<body>
<script>
    function add(a, b) {
        var count = 0;
        return function () {
            count += 1;
            console.log("第" + count + "次" + (a + b));
        };
    }

    add(1, 2)();  // 第1次3
    add(3, 4)();  // 第1次7
    var func = add(2, 8);
    func();  // 第1次10
    func();  // 第2次10
</script>
</body>
</html>
```

完成于 `2019.3.8`