JavaScript进阶

---JS作用域及执行上下文
---JS中的IIFE模式
---JS闭包

河北师范大学软件学院
Software College of Hebei Normal University

JavaScript进阶

---JS作用域及执行上下文

河北师范大学软件学院
Software College of Hebei Normal University

内容提纲

- **JS作用域及其特点**
- **JS执行上下文与调用栈（Call Stack）**
- **作用域链与执行上下文**

JS作用域及其特点

- 什么是作用域
  - 作用域就是变量与函数的可访问范围（变量生效的区域范围，即在何处可以被访问到）
  - 作用域控制着变量与函数的可见性和生命周期，它也是根据名称查找变量的一套规则

```
var a = 10,                    全局作用域
    b = 20;
function fn(){                 fn作用域
    var a = 100,
        c = 200;
    function bar(){            bar作用域
        var a = 500,
            d = 600;
    }
}
```

左侧实例（嵌套作用域）中：
变量d只能在bar作用域中被访问到，
变量c只能在fn和bar作用域中被访问到

在bar中访问a时为500（覆盖性）
在bar中访问c时为200（链式关系）

参见实例demo01 作用域实例

JS作用域及其特点

- JS作用域特点（词法作用域）
  - JS采用的是词法作用域（静态性），这种静态结构决定了一个变量的作用域
  - 词法作用域不会被函数从哪里调用等因素影响，与调用形式无关（体现了静态性）

```
> var name = 'Jack';
  function echo() {
      console.log(name);
  }
  echo();//输出Jack

  Jack
```

```
> var name = 'Jack';
  function echo() {
      console.log(name);
  }
  function env() {
      var name = "Bill";
      echo();
  }
  env();//Bill or Jack
```


参见实例demo02 静态词法作用域实例

JS作用域及其特点

- JS作用域特点（静态词法作用域补充部分）
  - 通过new Function创建的函数对象不一定遵从静态词法作用域
  - 对比下边两个例子（通过不同形式定义的函数对象，访问到的scope的区别）

```
var scope = "global";
function checkScope() {
    var scope = "local";
    return function(){
        return scope;
    };
}
console.log(checkScope()());
```

```
local
```

```
var scope = "global";
function checkScope() {
    var scope = "local";
    return new Function("return scope;");
}
console.log(checkScope()());
```

```
global
```

JS作用域及其特点（关于块作用域）

- 大多数语言都有块级作用域
  - 变量“存活”在最近的代码块中，比如Java中

```
public static void main(String[] args){
    {//block start
        int foo = 4;
    }//block end
    System.out.println(foo);//报错，无法访问到foo
}
```

- JS（ES5）采用的是函数级作用域，没有块作用域

```
{//block start
    var foo = 4;
}//block end
console.log(foo);//正常输出4
```

```
4
```

JS作用域特点（关于块作用域）

- 无块作用域的问题（变量污染、变量共享问题）

```
//变量污染问题,尤其是异步执行的情况下
var userId = 123;
document.onclick = function () {
    console.log(userId);
};
//...
```

```
//...
//一长串代码后，忘记了上边定义的userID
var a=2,b=3;
if(a<b){
    var userId = 234;//重复定义，变量污染
}
```

- 解决方案IIFE（更多内容参见IIFE部分）

```
//通过IIFE引入一个新的作用域来限制变量的作用域
(function(){
    var a=2,b=3;
    if(a<b){
        var userId = 234;
    }
}());
```


参见实例demo03 变量污染及解决办法

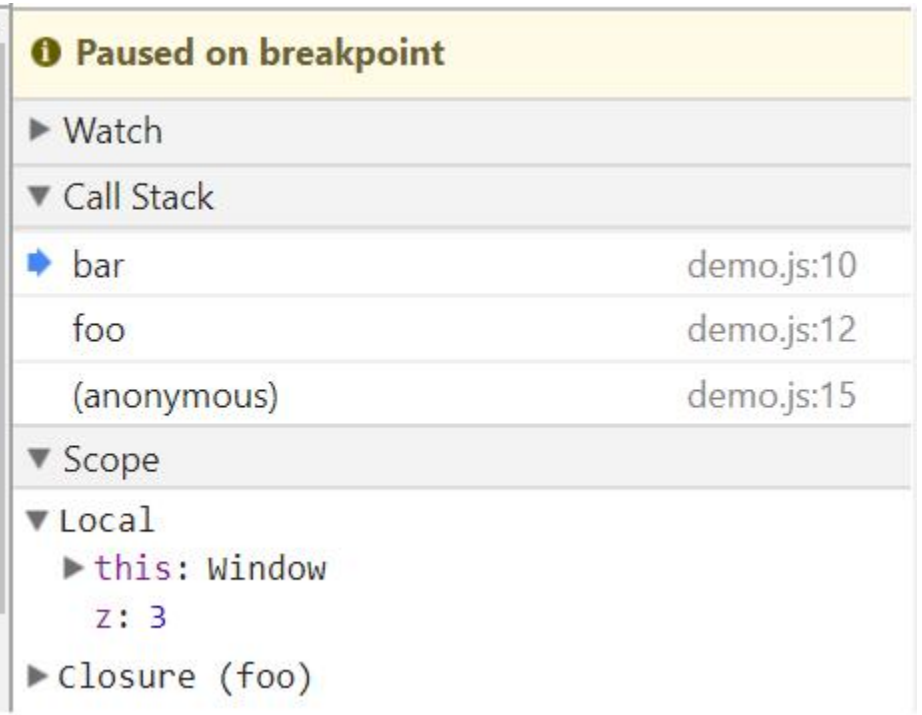
内容提纲

- JS作用域及其特点
- JS执行上下文与调用栈（Call Stack）
- 作用域链与执行上下文

JS执行上下文和调用栈

- 执行上下文
  - 执行上下文指代码执行时的上下文环境（包括局部变量、相关的函数、相关自由变量等）
  - JS运行时会产生多个执行上下文，处于活动状态的执行上下文环境只有一个

```
1  console.log("全局上下文-start");
2  var x = 1;
3  function foo(){
4      console.log("foo上下文-start");
5      var y = 2;
6      function bar(){
7          console.log("bar上下文");
8          var z = 3;   z = 3
9          console.log(x+y+z);
10         console.log("bar上下文");
11     }
12     bar();
13     console.log("foo上下文-end");
14 }
15 foo();
16 console.log("全局上下文-end");
```



断点：实例中执行到断点时，就有一个当前断点所对应的执行上下文（对应当前执行环境）

JS执行上下文和调用栈

- 理解执行上下文（通俗的例子）
  - 小明回家
  - 在家-做作业中 1 ...
  - 在家-做作业中 2 ... 发现笔没油了
  - 去文具店
  - 在文具店-买文具中 ...
  - 在文具店-买文具中 ... 发现没带钱
  - 去银行
  - 在银行-取钱 ... 返回文具店
  - 在文具店-买好文具 ... 返回家
  - 在家-继续做作业...

```
console.log("小明回家");
var xx = "小明家中（书桌-书包-铅笔盒-...）";
console.log("在家-做作业中 1 ...");
function goToStore(){
    var yy = "文具商店中（文具店老板-出售的文具...）";
    console.log("在文具店-买文具中 ...");
    console.log("在文具店-买文具中 ... 发现没带钱");
    goToBank();
    console.log("在文具店-买好文具 ... 返回家");
}
function goToBank(){
    var zz = "银行中（银行职员-柜员机...）";
    console.log("在银行-取钱 ... 返回文具店");
}
console.log("在家-做作业中 2 ... 发现笔没油了");
goToStore();//笔没油了，去商店买笔
console.log("在家-继续做作业...");
```


参见实例demo04_1 理解执行上下文的通俗例子

JS执行上下文和调用栈

- 调用栈（Call Stack）
  - 代码执行时JS引擎会以栈的方式来处理和追踪函数调用（函数调用栈 Call Stack）
  - 栈底对应的是全局上下文环境，而栈顶对应的是当前正在执行的上下文环境

```
1  console.log("全局上下文-start");
2  var x = 1;
3  function foo(){
4      console.log("foo上下文-start");
5      var y = 2;
6      function bar(){
7          console.log("bar上下文");
8          var z = 3;   z = 3
9          console.log(x+y+z);
10         console.log("bar上下文");
```

Paused on breakpoint

▶ Watch

▼ Call Stack

bar	demo.js:10
foo	demo.js:12
(anonymous)	demo.js:15

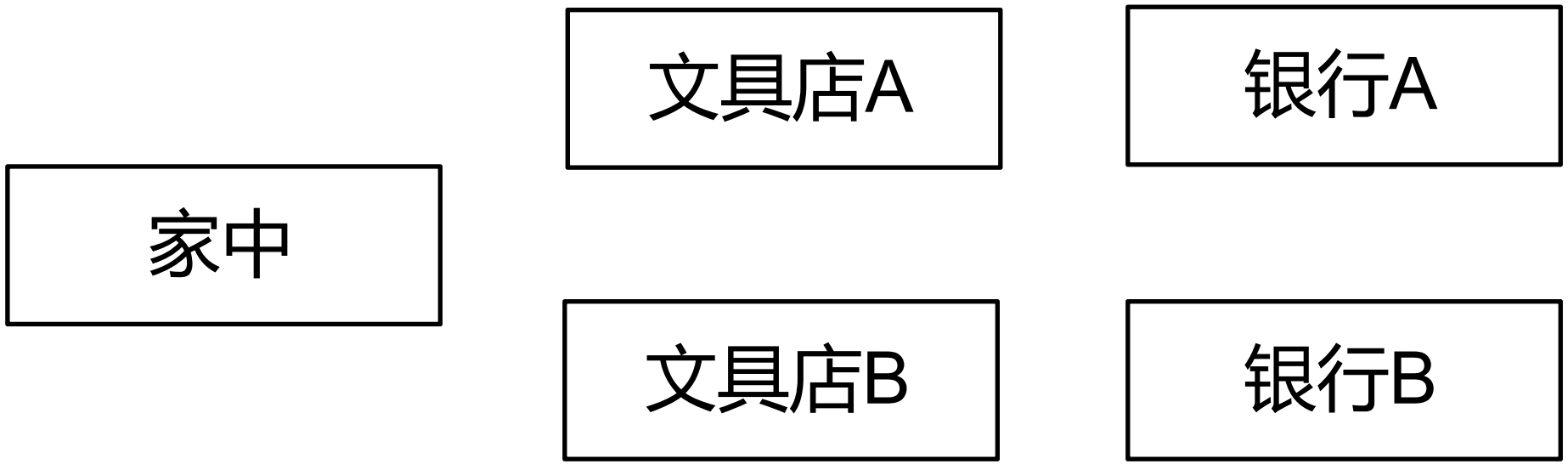
参见实例demo04_2和index04_2.html 调用栈实例

内容提纲

- JS作用域及其特点
- JS执行上下文与调用栈（Call Stack）
- **作用域链与执行上下文**

JS执行上下文和调用栈

- 理解代码执行时形成的作用域链（继续小明的例子）
  - 如果有多个文具店和多个银行，那么执行就有多种可能，形成不同的链式关系
  - 依然要遵从静态词法作用域（在A文具店，应该有A店老板，而不应有B店老板）

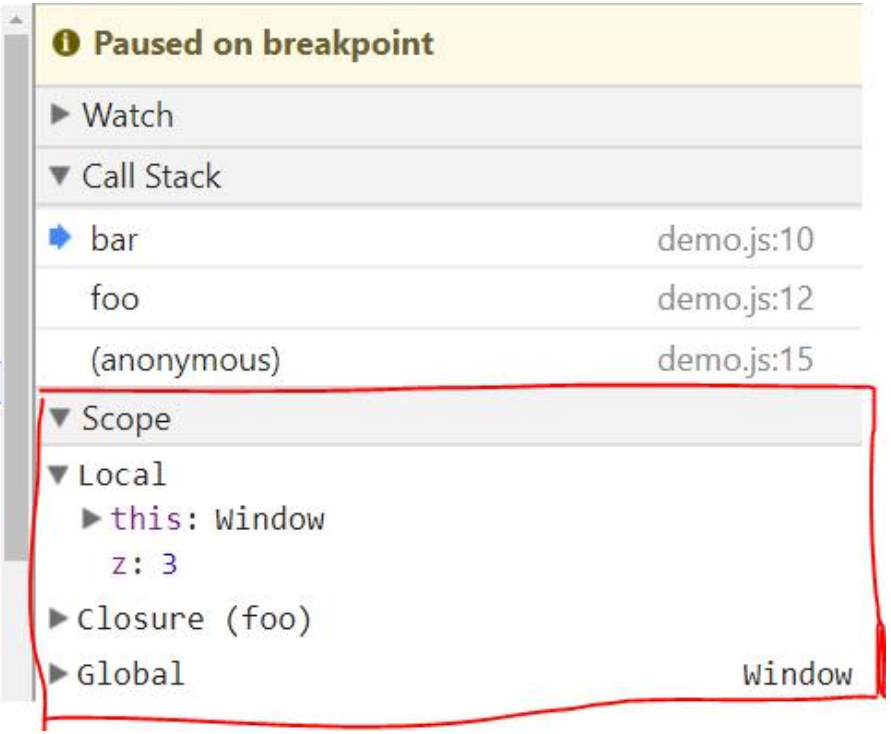


作用域链与执行上下文

•作用域链与执行上下文

- 执行时，当前执行上下文，对应一个作用域链环境来管理和解析变量和函数（动态性）
- 变量查找按照由内到外的顺序（遵循词法作用域），直到完成查找，若未查询到则报错
- 当函数执行结束，运行期上下文被销毁，此作用域链环境也随之被释放

```
1  console.log("全局上下文-start");
2  var x = 1;
3  function foo(){
4      console.log("foo上下文-start");
5      var y = 2;
6      function bar(){
7          console.log("bar上下文");
8          var z = 3;   z = 3
9          console.log(x+y+z);
10         console.log("bar上下文");
11     }
12     bar();
13     console.log("foo上下文-end");
14 }
15 foo();
16 console.log("全局上下文-end");
```



参见实例demo05 index05.html 作用域链

总结

- JS作用域及其特点
- JS执行上下文与调用栈（call stack）
- 作用域链与执行上下文

Have a Break!

河北师范大学软件学院
Software College of Hebei Normal University

补充：

- 环境：变量的管理
  - 当程序运行到变量所在的作用域时，变量被创建，此时需要一个存储的空间
  - JS中提供存储空间的数据结构被称为环境，每个函数都有自己的执行环境
  - 每个执行环境都有一个与之关联的变量对象，环境中所有变量和函数都保存在此对象中
  - Web浏览器中，全局执行环境为window对象
- 作用域链（在 ECMA262 中的解释，涉及到内部属性）
  - 任何执行上下文时刻的作用域，都是由作用域链 (scope chain) 来实现。 在一个函数被定义的时候，会将它定义时候的 scope chain 链接到这个函数对象的[[scope]]属性。 在一个函数对象被调用的时候，会创建一个活动对象 (也就是一个对象，然后对于每一个函数的形参，都命名为该活动对象的命名属性，然后将这个活动对象做为此时的作用域链 (scope chain) 最前端， 并将这个函数对象的 [[scope]] 加入到 scope chain 中

JavaScript进阶

---JS中的IIFE模式

河北师范大学软件学院
Software College of Hebei Normal University

内容提纲

- 什么是IIFE以及其使用方式
- 通过IIFE来解决的问题（JS缺陷）
- IIFE实际应用案例

什么是IIFE（发音：iffy）

- IIFE英文全称：Immediately-Invoked Function Expression即立即执行的函数表达式

```
function max(x,y){
    return x>y?x:y;
}
max(2,3);
3
```

```
(function max(x,y){
    return x>y?x:y;
}(2,3));
3
```

- IIFE的作用（建立函数作用域，解决ES5作用域缺陷所带来的问题，如：变量污染、变量共享等问题）

IIFE的写法

- 使用小括号的写法（最常见的两种）

```
(function foo( x,y){ ... }(2,3));  //2,3为传递的参数
(function foo(x,y){ ... })(2,3);
```

- 与运算符结合的写法（先执行函数，再进行运算）

```
var i = function( ){ return 10; }( ); //i为10
true && function( ){  ... }( );
~function(arg1,arg2){ ... }(x,y); //x,y为传递参数 位运算非操作符
!function( ){ ...  }( );//思考 !function(){return 2; }( ); 与 !function(){return 0; }( );
```

```
> !function(){
      return 2;
  }();
< false
```

```
> !function(){
      return 0;
  }();
< true
```


参见实例demo06 掌握不同形式的IIFE

内容提纲

- 什么是IIFE以及其使用方式
- 通过IIFE来解决的问题（JS缺陷）
- IIFE实际应用案例

通过IIFE来解决JS缺陷

- 通过IIFE对作用域的改变（限制变量生命周期）
  - JS（ES5）中没有块作用域，容易造成js文件内或文件间的同名变量互相污染
  - 我们往往会通过IIFE引入一个新的作用域来限制变量的作用域，来避免变量污染

```
var userId = 12;
document.onclick = function () {
    console.log("userId = ",userId);
};

//一长串代码后，忘记了上边定义的userID
var a=2,b=3;
if(a<b){
    var userId = 34;//重复定义，变量污染
}
```

参见实例：
demo07_1
前半部分

同一文件内
的变量污染

通过IIFE来解决JS缺陷

- 通过IIFE对作用域的改变（限制变量生命周期）
  - JS（ES5）中没有块作用域，js文件内和文件间的同名变量容易互相污染
  - 我们往往会通过IIFE引入一个新的作用域来限制变量的生命周期

```
//(function () {  // IIFE开始
    var x = 10;
    document.onclick = function () {
        console.log("x = ",x);
    };
//})();           // IIFE结束
```

文件1中的代码

通过立即执行表达式来避免变量污染

思考：如果不用立即执行表达式，而是直接写函数，然后再调用，是否可以实现同等效果

```
//(function () {  // IIFE开始
    var x = 20;
//})();           // IIFE结束
```

文件2中的代码，污染了文件1中的x

参见实例：
demo07_1后半部分
demo07_2
index07_1_2.html
不同文件之间的变量污染

非期望的变量共享问题及解决办法

- 通过IIFE对变量存储的改变（避免变量共享错误）
  - 当程序运行到变量所在作用域时，变量被创建，JS（ES5）没有块作用域，变量可能会共享
  - 如下例：在函数作用域中创建的变量 i 只有一个，出现了变量 i 共享问题，可通过IIFE解决

```
function f(){
    var getNumFuncs = [];//函数数组
    for(var i=0;i<10;i++){
        (function (j) {
            getNumFuncs[j] = function(){return j;};
        })(i);
    }
    return getNumFuncs;
}
var tmp = f();
tmp[3]();//输出为3，tmp[0]()...tmp[9]()都为是期望的结果
```

参见实例demo08和index08
变量共享及解决

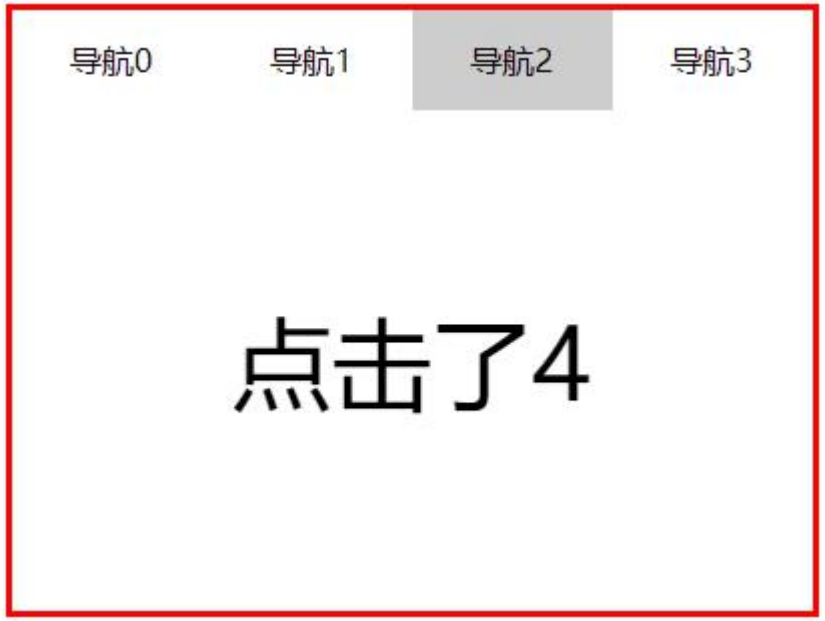
查看Scope窗体中getNumFuncs中每一个函数的内部属性[[Scopes]]中的闭包中的变量，看是否存在共享问题

内容提纲

- 什么是IIFE以及其使用方式
- 通过IIFE来解决的问题（JS缺陷）
- **IIFE实际应用案例**

IIFE实际引用案例（页面导航问题）

- 避免闭包中非期望的变量共享问题

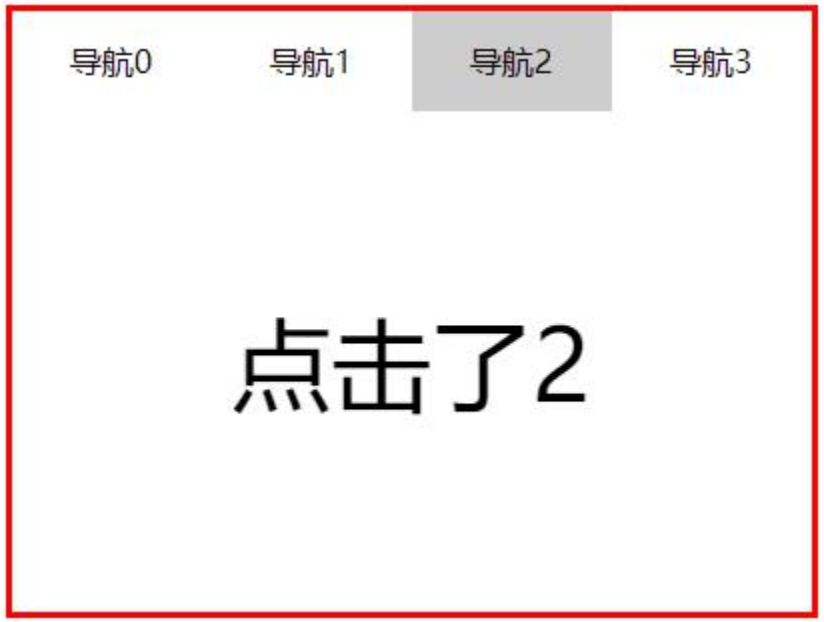



```
for(var i=0;i<tabs.length;i++){
    //!function(i){     //IIFE start
        tabs[i].onclick=function(){
            for (var j = 0; j < tabs.length; j++) {
                tabs[j].className = '';
            }
            this.className = 'active';
            contents.innerHTML = "点击了"+i;
        }
    //}(i);          //IIFE end
}
```

tab的length为4，由于变量共享在同一个作用域下，所以变量 i 只有一个，并最终i为4，所以点击任何标签，都输出“点击了4”

IIFE实际引用案例（页面导航问题）

- 避免闭包中非期望的变量共享问题，解决方式 IIFE




```
for(var i=0;i<tabs.length;i++){
    !function(i){   //IIFE start
        tabs[i].onclick=function(){
            for (var j = 0; j < tabs.length; j++) {
                tabs[j].className = '';
            }
            this.className = 'active';
            contents.innerHTML = "点击了"+i;
        }
    }(i);           //IIFE end
}
```

tab的length为4，立即执行了4次函数，有4个函数作用域，所以变量 i 生成了4次，所以点击时能正常输出1到4

参见实例demo09 和 index09.html

IIFE实际引用案例（定时器案例）

- 避免闭包中非期望的变量共享问题，解决方式 IIFE

```
for (var i = 0; i < 3; i++) {
    (function(j) {  // j = i
        setTimeout(function() {
            console.log(new Date, j);
        }, 1000*i);
    })(i);
}
undefined
Mon Sep 04 2017 16:51:26 GMT+0800 (中国标准时间) 0
Mon Sep 04 2017 16:51:27 GMT+0800 (中国标准时间) 1
Mon Sep 04 2017 16:51:28 GMT+0800 (中国标准时间) 2
```


参见实例demo10 定时器案例

总结

- 什么是IIFE以及其使用方式
- 通过IIFE来解决的问题（JS缺陷）
- IIFE实际应用案例

Have a Break!

河北师范大学软件学院
Software College of Hebei Normal University

JavaScript进阶

---JS闭包（closure）

河北师范大学软件学院
Software College of Hebei Normal University

内容提纲

- **闭包的概念**
- **闭包的常见形式**
- **闭包的作用及常用场景**

闭包（closure）的概念

- 闭包是由函数和与其相关的引用环境组合而成的实体
- 闭包是词法作用域中的函数和其相关变量的包裹体

```
function foo() {
    var i = 0;
    function bar() {
        console.log(++i);
    }
    return bar;
}
var a = foo();
var b = foo();
a();//1
a();//2
b();//1
```

函数bar和其相关词法上下文中的自由变量i，构成了一个闭包

返回的函数bar，依然能够访问到变量i（藕断丝连）

思考：foo和它相关作用域的变量是否形成闭包?

更详细描述：参见
深入理解JS 16.10章节

参见实例demo11 理解闭包的概念

内容提纲

- 闭包的概念
- 闭包的常见形式
- 闭包的作用及常用场景

闭包的常见形式（作为函数返回值返回）

```
var tmp = 100;//注意：词法作用域
function foo(x) {
    var tmp = 3;//注意：词法作用域
    return function (y) {
        console.log(x + y + (++tmp));
    }
}
var fee = foo(2); // fee 形成了一个闭包
fee(10);//
fee(10);//
fee(10);//
```

思考：此实例中fee函数对象相关作用域的变量都有哪些？foo中的tmp是否调用后就释放？

使用断点调试查看代码的运行状况

参见实例demo12 Part1

闭包的常见形式（作为函数返回值返回）

```
function foo(x) {
    var tmp = 3;
    return function (y) {
        x.count = x.count ? x.count + 1 : 1;
        console.log(x + y + tmp,x.count);
    }
}
var age = new Number(2);
var bar = foo(age); //和相关作用域形成了一个闭包
bar(10); //输出什么?
bar(10); //输出什么?
bar(10); //输出什么?
```

思考：此实例中bar函数对象相关作用域的变量都有哪些？foo中的tmp是否调用后就释放?

使用断点调试查看代码的运行状况

参见实例demo12 Part2

闭包的常见形式（作为对象的方法返回）

```
function counter() {
    var n = 0;
    return {
        count:function () {return ++n;},
        reset:function () {n = 0;return n;}
    }
}
var c = counter(),d = counter();
console.log(c.count());
console.log(d.count());
console.log(c.reset());
console.log(c.count());
console.log(d.count());
```

思考：此实例中总共有几个闭包？

使用断点调试查看代码的运行状况

参见实例demo13 Part1

闭包的常见形式（函数作为参数）

- 综合实例（闭包、高阶函数、静态词法作用域、IIFE）

```
var max = 10;
var fn = function (x) {
    if(x > max){
        console.log(x);
    }
};
(function (f) {
    var max = 100;
    f(15);
})(fn);
```

左侧实例输出什么？
使用断点调试查看代码的运行状况

参见实例demo13 Part2

内容提纲

- 闭包的概念
- 闭包的常见形式
- **闭包的作用及常用场景**

闭包的作用

- 可通过闭包来访问隐藏在函数作用域内的局部变量
- 使得函数中的变量都被保存在内存中不被释放

```
function f1(){
    var n = 999;
    function f2(){
        console.log(++n);
    }
    return f2;
}
var f = f1();
f();//输出多少?
f();//输出多少?
```

左侧实例中，无法在f1函数外直接得到变量n的值，可以通过闭包间接的在f1函数外访问和修改n

注意：由于闭包的存在n在f1调用后并不直接释放

参见实例demo14 参见相关实例

闭包的实际应用案例

```
function fn() {
    var a;
    return function() {
        return a || (a = document.body.appendChild(document.createElement('div')));
    }
};
var f = fn();
f();

function closureExample(objID, text, timedelay) {
    setTimeout(function() {
        //document.getElementById(objID).innerHTML = text;
        console.log(objID,text);
    }, timedelay);
}
closureExample("myDiv","Closure is Create", 1000);
```

单例模式实例：因为闭包，所以a常驻内存，始终存在

定时修改DOM节点案例，1秒后执行，仍能访问到objID

实例demo15 闭包应用案例
实例demo16 index16.html 函数相关操作综合实例

闭包的注意事项

- 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包
- 使用闭包时要注意不经意的变量共享问题，可以通过立即执行表达式来解决

Thank You!

作业

- codefordream网站上JavaScript基础-初级训练营
- codefordream网站上JavaScript中级