第九课 闭包

学习目录

- **匿名函数**
- **什么是闭包**
- **内存泄漏与垃圾回收机制**

一．匿名函数

匿名函数就是没有名字的函数，匿名函数既可以配合其他语法使用，也可以单独使用。

```
var btn = document.getElementById('btn');
btn.onclick = function(){}
```

匿名函数自执行

匿名函数自执行的调用方式是使用()将匿名函数包括起来，调用的话在后面加上一对圆括号，这个圆括号里也可以传递在匿名函数中要使用的参数。

```
(function(a,b){ console.
    log(a + b);
    console.log('匿名函数自执行');
})(100,200);
```

模仿块级作用域

很多时候我们做逻辑封装的时候，既希望有些变量可以被使用者访问，有些变量不想被

访问只想作为私有变量来在封装逻辑的内部使用，这就可以用到匿名函数自执行的特点来做到，这也是模仿块级作用域的方法。

1. 通过一个变量来接收匿名函数自执行的结果。
2. 访问这个匿名函数返回的允许被外部使用的变量和方法。

```
var addNum =
    (function(num1,num2){ var num3
    = 3;
    function _addFn(){
        return num1 + num2;
    }
    function addMore(){
        return _addFn() * num3;
    }
    return {
        num3:num3,
        getResult:addMore
    };

})(100,200);

console.log(addNum.num3);
console.log(addNum.getResult());
```

二．什么是闭包

闭包的官方定义是指函数和对其周围状态的引用捆绑在一起构成闭包。

实际上闭包最通俗的解释可以理解成一个可以访问另一个函数内部变量的函数。

从开发角度也就是说，闭包可以让你从内部函数访问外部函数作用域，就是在一个函数　内部创建另一个函数，通过另一个函数访问这个函数的局部变量。

函数作为返回值

```
function fn1(){
    var a = 100;
    return function(){
        var b = 200;
        console.log(a + b);//300
    }
}
var f1 = fn1();
f1();
```

函数作为参数

```
function fn1(a){
    var a = 300;
    var b = 200;
    console.log(a + b);//500
```

```
}

(function(fn){
    var a = 100;
    fn(a);
})(fn1);
```

执行 fn1()时，返回的是一个函数，这个返回的函数也可以创建一个独立的局部作用域， 另一方面是，在返回的这个函数内部中，还有变量 a，这个变量 a 是引用自 fn1 作用域下的a，所以为了保证这个变量能够继续被返回的新函数继续引用，因此，这个 a 不会被销毁， 一旦销毁 a 的引用就会报错了。因此，调用 fn1()形成执行环境不会被销毁，还依然存在与执行栈中，也就是说变量 a 会一直在内存中保留。

因此闭包的第一个问题是会增加内存开销，不过闭包的优点也显而易见，因为它允许将 函数与其所操作的外部环境数据关联起来，这种关联关系有利于变量的持久化缓存，可以加 快有些变量的访问速度，同时这种关联性也类似于面向对象编程。在面向对象编程中，对象 允许我们将这个对象的某些属性和方法与一个或者多个方法相关联。

三．内存泄漏与垃圾回收机制

上节课中关于函数作为参数形成闭包的描述这里需要补充一下，也是根据作用域链取值的原理进一步说明闭包在函数作为参数的时候的提现。

```
var a = 300;
function fn1(){
   var b = 200;
   console.log(a + b);
}
```

```
(function(fn){
    var a = 100;
    fn();
})(fn1);
```

这里的fn1内部使用的a也可以不从匿名函数自执行中传递，可以在fn1这个函数定义的作用域下，这里也就是全局环境下定义一个a，当fn1以参数的形式fn在匿名函数自执行里调用的时候，fn1内部的a首先会在自己的作用域中查找a，如果找不到就会向他的上一层作用域中查找，也就是fn1这个函数定义的作用域中查找，这里是从离fn1最近的全局作用域里取值的，不会从fn1调用的匿名函数自执行的作用域里取值。

```
var a = 300;
function fn2(){
    // var a = 400;
    function fn1(){
        var b = 200;
        console.log(a + b);
    }
    return fn1;
}
(function(fn){
    var a = 100;
    fn();
})(fn2());
```

如果在fn1是定义在另一个函数fn2中，这个fn2的作用域中也没有找到a，那么就会按照作用域链的查找逻辑，在找到fn2这个函数定义所在的作用域中进行查找，直到查找到全局作用域下，如果找不到就会报错。

内存泄漏

浏览器在运行过程中，当发现一些内存在使用完毕之后会自动使用一定规则的垃圾回收机制来查找程序中不在占用内存的那些变量清除掉，同时把对应的内存空间返还给操作系统或者内存池。

内存泄露的定义是当有些内存不在被程序使用的时候，由于某种原因，这些内存中依然存放着这个程序中的某些变量，因此这些内存并没有返还给操作系统或者内存池，因此就会造成内存空间的长期占用，如果这种占用是较大增量的、持续的就会造成内存泄露，很有可能会导致浏览器以及程序卡顿、崩溃。

我们之前说过闭包可以让变量进行持久化的缓存，这些变量不会被释放，一直存储在内存中，给我们的功能开发以及页面访问带来方便，也正是如此，闭包会带来一定程度的内存开销的增加。

但是并不是说内存开销的增加就是内存泄露，如果这种内存开销是可控的，可以手动清理的，这就不能算是内存泄漏。

另外在js中闭包中对于外层函数的值类型的引用基本不会造成内存泄露，所以这一点不用担心。能够造成内存泄露的情况，一般情况下是闭包中引用了外层函数的引用类型导致的，同时外层函数的引用类型的变量还必须要是数据量较大、容易快速增量变化的引用类型的变量，比如说对象、数组、定时器实例对象、绘图实例对象、第三方依赖库的实例化对象等。

垃圾回收机制

js本身具有自动垃圾收集机制，这种自动垃圾收集机制就是找出那些不再继续被使用的变量，然后释放其占用的内存。

在函数被调用形成的执行环境中，这个执行环境会负责管理代码执行过程中使用的内存，这个过程是js自动进行的，每个变量所使用和分配的内存空间全部自动管理，同时当他们不在呗使用的时候也会被自动回收。

因此，垃圾收集器会按照一定的时间间隔周期性的执行垃圾回收机制，把不使用的变量清除掉，把他们占用的内存空间释放出来。

在如何确定哪个变量需要被清除释放的问题上，js中有两种确定方式，一种是标记清除，另一种是引用清除。

标记清除

垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记，当变量进入执行环境时，就将这个变量标记为“进入环境”。而当变量离开执行环境时，则将其标记为“离开环境”。

在垃圾回收器检测的时候，它会去掉环境中的变量以及被环境中的变量引用的变量的标记，同时除了之前这两种变量之外，其他的用不到的或者使用过的变量会被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。之后，垃圾收集器就开始内存清除工作，销毁那些带标记的值并回收它们所占用的内存空间。

引用清除

引用计数用的不多，其实就是记录一个变量引用的次数，当这个值的引用次数变成 0 时，则说明没有办法再访问这个值了，因而就可以将其占用的内存空间回收回来。这样，当垃圾收集器下次再运行时，它就会释放那些引用次数为零的值所占用的内存。

因此不管是哪种确定方式，都可以明显的发现他们无法指定被闭包持久化引用的变量，所以在使用闭包的使用要尽量避免不正确的使用方法。

避免使用闭包造成内存泄漏的简单好用的方法如下：

1.避免在闭包中引用外层函数中数据量较大、容易快速增量变化的变量。

2.在编写闭包时尽量避免与定时器相互嵌套使用，如果有在一起使用的情况，一定要对使用过的定时器进行清除。

3.对闭包中引用的外层函数的变量进行手动解除引用，解除引用的真正作用是让值脱离执行环境，以便垃圾收集器下次运行时将其回收。

```
function fn1(){
    var a = {
        x:100,
        y:200
    };
    return function(){
        console.log(a.x + a.y);
    }
    a = null;
}
var f1 = fn1();
f1();
```

谢谢观看！

我是星星课堂老师：周小周