20170727

性能优化：堆栈内存的销毁问题

堆内存释放（销毁）

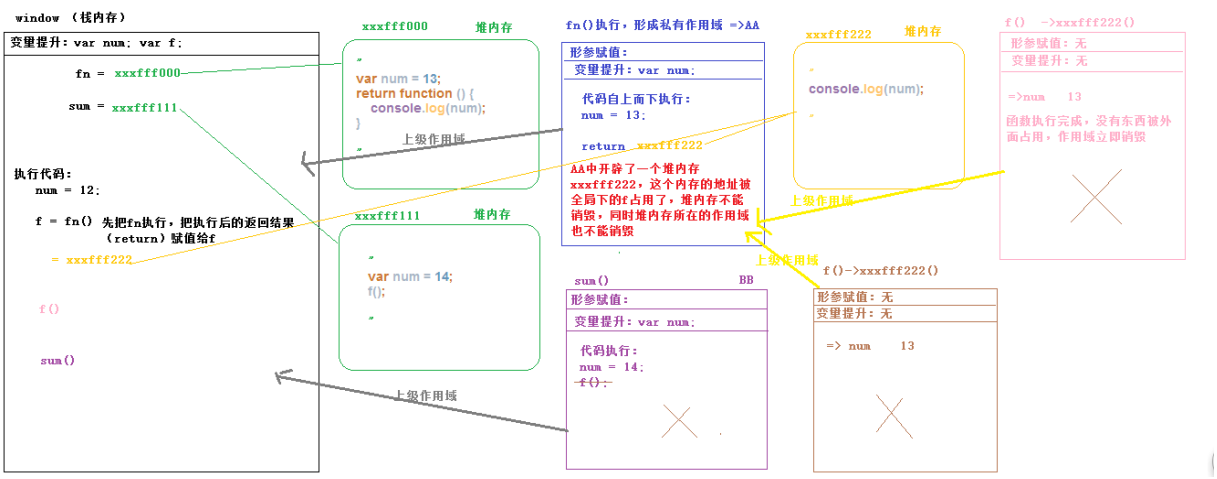
> 如果有变量（函数名）占用了堆内存的地址，那么当前的堆内存则不能释放
>
> 如果当前堆内存的地址，没有被任何的东西所引用，当前的堆内存就没用了，浏览器会在空闲的时候清理掉这些没用的堆内存（谷歌浏览器）
>
> IE下的堆内存释放采用的是计数器机制，被一个变量占用，计数器就累加1，如果之前的某个占用被移除，计数器减1；但是很多时候IE的计数器计数的时候出现问题，导致“内存泄漏” => 【IE浏览器的内存泄漏问题？】

```
var obj = { name:"Junior" }; //->obj=xxxfff000此时的obj把堆内存占用了

obj = null; //->obj不占用堆内存了，浏览器在空闲的时候会销毁这个无用的堆内存

/*null：空对象指针，不指向任何的堆内存*/
```

栈内存（作用域）的释放

> **全局作用域：**浏览器加载页面的时候形成全局作用域，在浏览器把当前页面关闭的时候，全局作用域销毁 【刷新一次页面先销毁再创建】
>
> **私有作用域：**函数执行的时候会形成私有作用域，一般情况下，函数执行完成，形成的这个私有作用域立即释放销毁
> **特殊情况：**当私有作用域中的某一个东西（一般指的都是私有作用域中开辟的那个堆内存）被作用域以外的变量给占用了，当前的私有作用域（栈内存）就不能销毁了；这个私有作用域不能销毁，代表着它里面存储的私有变量也不会销毁了；

```
console.log(num);
console.log(fn());
var num = 12;
function fn() {
    var num = 13;
    return function () {
        console.log(num);
    }
}
var f = fn();
f();//13
function sum() {
    var num = 14;
    f();
}
sum();//13
```



i++ 和 ++i 的区别

都是在自身基础上累加1，不同地方在于和其它值运算的时候，累加1在前还是在后

> i++：先运算，再累加
> ++i：先累加，再运算

```
//i++
var i=10;
console.log(5+i++); // ->先计算5+i，计算完成后再把i累加1   =>15   i=11
console.log(5+(i++)); // ->加上括号也是先运算再累加        =>15   i=11

//++i
var i=10;
console.log(5+(++i)); // ->先让i累加1，把累加后的结果和5进行运算   i=11
=>16
```

思考题：

```
var i=4;
var res=5+(++i)+(i++)+(i++)+(++i);
console.log(res,i);
```

闭包的作用

> 你了解过闭包么？（易车面试题）

目前外界普遍认为“形成一个不销毁的私有作用域”才是闭包

```
var fn = (function(){
    var n = 12;
    return function(){
        console.log(++n);
    }
})();
```

闭包的作用：

- 保护里面的私有变量不受外界干扰（里面的变量和全局变量没关系，防止全局变量污染）

```
// ->例如：我们封装类库或者组件插件的时候，为了防止和全局变量冲突，我们都使用闭包把代码包裹起来（jQuery就是这样处理的）
;(function(){
    var fn = null;
    var jQuery = function(){
        …
    }// ->jQuery=xxxfff000:

    window.jQuery = window.$=jQuery;// ->给全局增加了一个jQuery的属性和$的属性，这样在外面就可以使用jQuery或者$了【或者return…】
})();//把全局下的window传入这里

var fn = undefined; // ->和闭包中的fn没有关系，防止了冲突和污染
$();// ->里面私有jQuery执行了 $=xxxfff000
```

- 可以形成一个不销毁的私有作用域，来存储一些值

```
//->选项卡部分代码节选

for (var i = 0; i < oList.length; i++) {
    oList[i].onclick = (function (i) {
        return function () {
            tabChange(i);
        }
    })(i);
}

//或者这样写

for (var i = 0; i < oList.length; i++) {
    ~function (i) {//闭包方式 - 耗性能
     oList[i].onclick = function () {//事件绑定是js异步编程
     //tabChange(this.dataIndex);
     tabChange(i);// ->每一次循环把全局作用域下的i变量存储的值，当作实参传递给形参i（形参是私有变量）
     }
     }(i);
}
```

```
// ->思考题:
```

this

> this - 这个，是执行这个方法的主体

```
// ->以下所有规律都是在非严格模式下生效

// ->1、自执行函数执行，函数中的 [this] 是 window

~function(){
    console.log(this); //->window
}();

// ->2、给元素的事件绑定方法，当事件触发方法执行的时候，方法中的[this]是当前操作的元素

oDiv.onclick = funciton(){
    console.log(this); // ->oDiv
}

// ->3、方法执行，看方法名前面是否有“点”，有“点”，“点”前面是谁，this 就是谁，没有“点”，this 就是 window

var n = 200;
function fn(){
    console.log(this);
}
var obj = {
    n:100,
    aa:fn
};
fn(); // ->this：window
obj.aa();// ->this：obj
```

综合题