正式课第一周

预解释、作用域链

1、i++ 和 ++i

如果 i++/++i 自己出现在一行没有区别.都是自身累加 1

如果 i++和++i 和其他值做运算的时候就有区别，++i 是累加后运算 而 i++先运算后累加，累加是把自己累加，运算是和别人运算

```
var k = 2; //3 4 5  6
console.log(3 + k++ /5/ + ++k /9/ + ++k /14/+ k++ /19/);
//?? 19
console.log(k);
```

2、this 的用法

this 用法： 在全局作用域中的 this 是 window

1 在事件绑定的函数中的 this 就是事件发生的时候触发事件的元素

2 函数在执行的时候，函数中的 this 就是调用函数的主体，其实就是看函数执行的时候前面有没有"."，如果有"."那么点前面是谁 this 就是谁。如果没有那么就是 window

ps: this 是谁根本就不用看在哪里定义的，就看在哪里执行的

ps: 只要 this 被函数包括了那么 this 就变了

3 自运行函数中的 this 永远是 window

4 定时器中的 this 也是 window

5 构造函数中的 this 是当前实例

6 call 和 apply 可以强制改变 this

3、内存释放机制

堆内存和栈内存:

堆内存就是用来存储值的，一般存储都是引用数据类型值(函数，对象)

栈内存：在代码运行的时候就会产生一个供代码运行的环境就是栈内存,而这个栈内存我们也可以叫做作用域。只要打开一个页面，浏览器会首先提供全局的运行环境，其实就是全局作用域用 window 表示

```
//对于内存(堆，栈)如果没有被占用那么浏览器就会找个合适的时间点去释放,否则被占用的内存是不可以释放的
var obj1 = {name:'tianxi',age:30}; //obj1 占用着这个对象的堆内存
var obj2 = obj1; //obj2 和obj1 同时占用这个堆内存地址
obj1 = null; //把obj1 重新赋值,原来占用着对象的堆内存，重新赋值之后就不占用了
var obj3 = {
    xx:obj2 //obj3 的xx 属性也占用着这个堆内存地址,堆内存的占用不一定必须是变量也可以是一个对象的属性
};
obj2 = null; //现在这个对象的内存地址再也没有人占用了,那么
```

浏览器就会主动去释放这个内存

//把 obj2=null 这就是一个主动释放的过程

//栈内存(作用域)释放问题：基本数据类型都存在栈内存

栈内存不释放：

1 函数执行的时候，如果函数中的某一部分(引用数据类型)如果被函数外的变量或者对象的属性占用，那么函数执行时产生的私有作用域(栈)不会被释放，从而这个作用域内的私有变量和函数也不会被释放,那么永远都是保存的最后的被修改的那个值

2 再给 dom 元素绑定事件的时候，把一个自运行函数的返回值赋值给元素的事件属性，同样符合被对象占用的内存不释放条件

3 暂时不释放 ： 函数运行的返回值仍然是一个函数立刻执行，属于暂时不释放的条件，执行结束之后就会被释放掉

```
//1 内存不释放
var num = 1200;
function fn(){
    var num = 120; //这个num=120 没有被释放我就可以理解为被存在这个私有的作用域中了
    return function (){
        console.log(num)
    }
}
var f = fn(); //f 占用着fn()运行的返回值
```

```
f();

//2 内存不释放
var div1 = document.getElementById('div1');

(function (){ //闭包的方式处理事件
    var x = 100;
    div1.onclick = function (){
        console.log(++x);
    }
})();
for(var i=0; i<3/lis.length/; i++){
    (function (i){ // 0 1 2
       / var i=0;
        var i=1;
        var i=2;/
        /lis[i].onclick = function (){
            lis[i].className = 'cur';
        }/
    })(i); //0 1 2
    console.log(i); //0 1 2
}
```

```
div1.onclick = (function (){
    var x = 100; //101 102 103 104
    return function (){
        console.log(x++);
    }
})()
//3 暂时不释放
function fn(num){
    //var num = 10; 11 12 13
    return function (n){
        console.log(n + ++num);
    }
}
var f = fn(10);
f(3); //14
fn(9)(5);
f(6); //18 18
f = null;
f = fn(10);
f(10); //? 21
f(11);
//////////////////////////////////////////////////
```

```
var num1 = 8;
console.log(xx);
function num(){
    console.log(window.num1);
    function xx(){}
}
num();
console.log(num1);
var xxx;
(function (){
    ~function (xxx){
       xxx = 100;
    }()
})()
console.log(xxx);//undefined,如果上面没有形参就是 100 了

var y = function fn(){}
if("num" in window){ //
    var num = 1000;
}
(function xxx(){})()
```

```
function fn(f){
    //var f = 10;
    var f = 10000;
    return (function y(){
        //return 9;
    })()
    (function z(){})()
}
var res = fn(10); // 1 2 3

function fn(){console.log(1)}
var fn = 100;
fn();//
function fn(){ console.log(2)}

////////////////////////////////////////
var ary = [100];
function fn(ary){
    ary[ary.length] = 200;
    ary = [];
    ary[ary.length] = 100;
    console.log(ary);
```

```
}
fn(ary);
////////////////////////////////////////////////
var num = 12; //全局 window.num = 12 ==> window.num+=3 ==> 15 ==> 第一次执行fn 的时候，这个15 变成了14  window.num -= 1
var obj = { //obj 存储就是这个对象的地址
    num: 3, //obj.num = 3,第二次执行obj.fn 的时候， obj.num -= 1 ==> 2
    fn: (function (num) {
        // var num = 12;
        num += 2; // num = 14
        this.num += 3; //自运行函数中的this 永远是window==> window.num += 3
        var num = 5; // num = 5;  这个作用域内的私有num 是5 ，第一次fn 执行的时候把num 从5 -= 1 变成了4   num = 4 ，第二次执行obj.fn 的时候，这个4 又被改了一次  num-=1   ==> 3
        return function () {
            num -= 1; //第一次fn 执行把上一级作用域的num 修改了
            this.num -= 1; // fn()执行this 是fn 前面没有点，this 是window ==> window.num -= 1 ,第二次执行的时候obj.fn()
```

```
的时候，this 是obj ==> obj.num -=1
            console.log(num); //私有没有，上一级获取到刚刚被修改过的4 .打印4
        }//obj.fn 就是这个匿名函数，被对象的obj.fn 属性占用，导致这个自运行函数运行产生的作用域没有释放，从而这个作用域内私有的num=5 被保存下来了
    })(num) //12
};
var fn = obj.fn; //fn 和obj.fn 共用自运行函数返回的那个匿名函数的内存地址
fn(); //4
obj.fn(); //3
console.log(num,obj.num); //14 2
```

4、预解释

解释(变量提声)：当浏览器在解析 js 代码之前，把所有带 var 和 function 都会做提前声明。但是带 var 的变量仅预仅是提前声明了，但是没有被赋值，浏览器会默认赋值一个 undefined。带 function 在提前声明阶段不仅仅是声明了，还把函数名赋值了.赋值就是把这个函数的引用地址赋值。当浏览器解析到函数代码的时候由于在预解释阶段已经完成了赋值动作，所以直接跳过

var 和 function 的区别：function 函数在预解释阶段就已经把值赋值完了，但是带 var 的在预解释阶段没有赋值，只是默认赋值了

一个 undefined，当代码运行到赋值那一行的时候才会把值赋值给变量

函数的赋值过程：

1 先开辟一个堆内存空间地址

2 把函数体内的代码当作字符串存入到这个空间中

3 把空间地址赋值给函数名

ps：函数名就是一个地址

作用域链：

如果当前作用域内使用的变量不存在那么到上一级作用域去查找，如果没有继续到上一级作用域查找，一直查找到全局作用域 window，如果还没有报错！！！，我们把这种查找机制就可以叫做作用域链

ps：全局变量任何时候都可以被访问，所以不安全。

闭包。。。。

全局带 var 和不带 var 的区别。。。

全局变量带 var 和不带 var 的区别就是是否会被预解释

* 而在变量的赋值过程中，如果有私有变量那么直接赋值给私有变量，如果没有那么通过作用域链去上一级作用域查找，如果查找到 window 顶级作用域还没有那么就直接赋值给 window 当作属性

* ps：上一级作用域是谁，就看函数被哪个 function 包着，上一级作用域是谁就看函数在哪里定义的，和在哪里执行没有关系

* 闭包：函数执行的时候会形成一个新的私有作用域，我们把这种

私有作用域保护私有变量不受外界干扰的机制就叫做闭包

*　形成闭包要有函数执行才可以，否则别谈闭包

全局变量和私有变量：

*　　当浏览器打开一个网页的时候就会提供一个全局作用域 window，而我们把直接定义在全局作用域内的变量叫做全局变量. ps: 直接暴露在 script 标签下的变量就可以理解为全局变量,没有 function 包着的一般都是全局变量

*　　私有变量：我们把定义在函数中的变量叫做私有变量. 由于在函数外访问不到所以称之为私有

*　函数执行的过程：　实名函数　匿名函数　自运行函数　事件发生的时候

*　　1 会形成一个新的私有的作用域(栈内存)

*　　2 形参赋值(行参相当于在当前函数体内声明的私有变量)

*　　3 预解释函数体内带 var 和 function 的

*　　4 代码逐行执行

*　　ps：每次执行形成的私有作用域是互不干预的

*　　ps：如果形参和私有变量重名，那么在预解释阶段私有变量就不用提前声明了，因为在形参赋值的过程就已经声明过了。但是当代码执行到赋值过程的时候还是会重新赋值的

*　　ps：预解释只发生在当前作用域

//

```
var total = 0; //全局变量 total=0 由于执行 sum 函数这个全局的 total 被修改成了 30
function sum(num1,num2){ //10,20
    console.log(total); //0 说明去上一级的 window 全局把 total 的值拿来打印
    total = num1 + num2; //30
    console.log(total); //刚刚被修改成为 30 就打印出来了
}
sum(10,20);
console.log(total); //由于执行 sum 的时候 total 给修改了，所以打印 30
```

全局变量：函数在执行的过程中，如果当前作用域内有私有变量(形参和 var)那么优先使用私有变量，如果没有那么到上一级作用域去查找，如果有直接拿来使用，如果没有继续向上查找，一直到全局作用域.如果还没有报错

```
//////////////////////////////////////////
var ary = [100, 200]; //全局的 ary 是一个引用地址
function fn(ary) {
    ary[ary.length] = 300;
    var ary = ary.slice(); // 把一个新数组复制 ary 变量 ==>把 ary 这个私有变量重新赋值，ary 已经不再代表以前的数组了，是
```

```
一个新数组的引用地址了
    ary[0] = [100];
    ary.length--;
    console.log(ary);
}
fn(ary); //函数的运行: 1 2 3 4
console.log(ary);
//[[100],200] [100,200,300]
////////////////////////////////////////
var ary = [100, 200]; //xxxfff000  [100,200,300]
function fn(ary) {
    //var ary = xxxfff000
    //私有的 ary 和全局的 ary 公用一个数组的堆内存地址
    ary[ary.length] = 300; //在数组的末尾添加一项
    var ary = ary.slice(); //由于 slice 是返回一个新数组,
所以是把私有的 ary 变量的值赋值为一个新的数组的地址
[100,200,300]
    ary[0] = [100]; //[[100],200,300]
    ary.length--; //[[100],200]
    console.log(ary); //[[100],200]
}
fn(ary);
```

```
console.log(ary); //[100,200,300]
////////////////////////////////////////
var ary = [10,20,30]; //[10,20]
function fn(ary){
    ary.length--;
    ary = []; //从这行代码开始私有的 ary 再也不和全局的 ary 公用一个地址了，，我自己使用这个空数组的引用地址
    ary[ary.length] = 0; //都是向空数组中添加一项
    console.log(ary); //[0]
}
fn(ary);
console.log(ary); //[10,20]
```

5、预解释是一种无节操的机制

```
*       1 预解释只看变量不看值，即使值一个函数函数名字也不会被预解释
 *       2 预解释不理会条件  if(false){  var num }  掌握 in 判断属性是否属于一个对象
 *       3 自运行函数即使有函数名字也不会被预解释
 *       4 return 后面即使是一个函数也不会被预解释，把函数当作一个整体返回。但是 return 下面的代码虽然不会执行但是却会被预解释,如果 return 后面是一个自运行函数，那么要等这个自运行函
```

```
数先去执行完，然后把自运行函数的运行结果留给 return
 *       5 如果函数名字和变量名字重名，在预解释阶段以最后一个函数为准。在代码执行阶段，从变量赋值开始就开始代表这个变量的值了  var fn = 100
 //1 不看值
 //console.log(f); // f is not defined
 //console.log(fn); //undefined
/* var fn = function f(){
    console.log(1);
 }*/
 // 2 预解释不理会条件
 // in 运算符是判断一个属性是否属于一个对象,属性名字是一个字符串
 //window.setTimeout  对于这种全局对象 window 的属性，window 可以省略不写。
 // 在全局作用域下的全局变量都可以理解为是 window 的一个属性
 console.log(num); // 没有报错，说明在执行之前就被声明过
 if('num' in window){ //在预解释阶段这个条件就已经成立了
var num = 100; //只要这个 num 被预解释过，那么这个 num 就是 window 的一个属性
 }
 console.log(num); //100
```

```
//3  自运行函数即使有函数名字也不会被预解释
//console.log(fnn);
(function fnn(){
    //console.log(1);
})()
//4  return 相关的 return 后面即使是一个函数也不会被预解释，把函数当作一个整体返回。但是 return 下面的代码虽然不会执行但是却会被预解释,如果 return 后面是一个自运行函数，那么要等这个自运行函数先去执行完，然后把自运行函数的运行结果留给 return
function foo(){
    var num1 = 12;
    //console.log(fxx); //
    return  (function fyy(){
        return 123123;
    })()
    function fxx(){}
}
var res = foo(); // 1 新作用域 2 形参赋值 3 预解释  4 执行代码
console.log(res); //8
//5 如果函数名字和变量名字重名，在预解释阶段以最后一个函数为
```

准。在代码执行阶段，从变量赋值开始就开始代表这个变量的值了

```
var fn = 100
fn();  // 2
function fn(){  console.log(1); }
fn();  // 2
var fn = "100"; //从这行代码开始 fn 不是一个函数了
fn(); // 100()  fn is not a function
function fn(){  console.log(2); }
fn(); //
```

6、类的封装，单例，工厂，构造函数

->单例模式的作用:把描述同一个事物的属性和方法放在相同的命名空间下,这样的话本命名空间下定义的方法就会和其他的命名空间下的方法不冲突(单例模式是项目中实现模块化开发的一个最简单最基本也是最常用的方式)

```
    var obj1 = {
        fn: function () {}
    };
    var obj2 = {
        fn: function () {}
    };
```

->工厂模式:把实现一个功能的代码封装到一个函数中,以后在想实现这个功能,不需要重新的编写这些代码了,只需要把函数执行一遍即可

('破函数') ->低耦合高内聚:减少页面中的冗余代码地提高代码的重复利用率

->面向对象编程思想需要我们掌握的就是类和实例之间的这点关系,如果想把关系屡明白,需要我们研究一下有关于类的继承封装和多态

->函数的封装(类的封装)

```
    function sum() {
        var total = null;
        for (var i = 0; i < arguments.length; i++) {
            var cur = Number(arguments[i]);
            if (!isNaN(cur)) {
                total += cur;
            }
        }
        return total;
    }
    sum(1, 2);
    sum(2, 3, 4);
    sum(100, 200);
```

->多态:多种形态-->JS 重载和 JS 重写

```
function fn(num1, num2) {
    return;
```

}

->后台语言中的重载:方法名相同，参数的类型或者个数不一样,这样的话我们可以把他们理解为不同的方法,以后执行的时候根据传递参数的不同,会找自己对应的方法执行 ->这一点非常能体现函数的多种形态

```
public void sum(int num1,int num2){}
public void sum(int num1,int num2,int sum3){}
sum(1,2)
sum(1,2,3)
```

->JS 重载:相对于后台语言,从严格意义上来说 JS 中不存在重载,因为只要方法名相同,后面的就会把前面的给覆盖掉,最后只能保留一个,以后不管传什么参数,永远都执行的是最后一个

->非严谨的角度上来讲,JS 中的重载指的是相同一个方法,我们通过传递的参数的不一样,实现不同的功能

```
function sum(num1, num2) {
    console.log(1);
}
function sum(num1, num2, num3) {
    console.log(2);
}
sum(1, 2);
sum(1, 2, 3);
```

```
function sum(num1, num2) {
    if (typeof num2 === "undefined") {->定义了形参没有传递任何的值
        console.log("参数错误！");
        return;
    }
    console.log(num1 + num2);
}
sum(1);
sum(1, 2);
```

-> 构造函数模式： 构造函数中的 this 是当前实例

* 实例，类，对象

* 1 定义一个构造函数(类)和定义一个普通函数相同，但是一般情况我们把类名字的首字母大写 Array String

* 2 区分构造函数还是普通函数只有在函数执行的时候是否使用 new

* 3 当作构造函数执行的时候，会默认返回一个实例

* 4 构造函数中的 this 是当前实例

* 5 在构造函数中声明的私有变量对实例的返回没有影响

* 6 在构造函数中如果 return 一个引用数据类型会破坏实例返回

```
*       7 任何一个实例都是对象数据类型的
*
*   ps: 任何一个对象数据类型(包括函数)都是 Object 这个类的实例,Object 是所有对象数据类型的基类
*
*   ps: instanceof 判断实例是否属于一个类 true/false
//var ary1 = new Array(); //内置类 Number String ... Object
//console.log(ary1); //[]
//Human; // tianxi geng
// Tab;
// Banner;
// Animate;
// Array,String,Number,Date,RegExp,Boolean,Function

function Tab(name,age){
    //var num = 1000;
    this.name = name;
    this.age = age;
    //return ['我是来捣乱的'];
    //return 5;
}
//Tab(); // window.name = 'xx'  window.age = 30
```

```
var tab1 = new Tab('tianxi',30); //把 Tab 当作一个类去执行
console.dir(typeof tab1);

var tab2 = new Tab('geng',40);
console.dir(tab2);
console.log(tab2 instanceof Tab);
```

->原型模式

* 任何一个函数都天生自带一个属性叫做 prototype(原型),而属性的值仍然是一个对象数据类型的，并且也天生自带两个属性，一个是 constructor，另外一个是__proto__。其中 constructor 的值是这个类(构造函数)本身

*　　Human.prototype.constructor = Human　　true

* 任何一个实例也都天生自带一个属性叫做__proto__,而这个属性的值是这个实例自己的所属类的原型,这个原型相对于每个实例来讲都是一个公有的，在这个原型上属性对于每个实例来说也都是公有属性

* ps: 任何一个引用数据类型都会天生自带__proto__属性

原型链:

*　　实例通过__proto__属性查找到自己所属类的原型，然后原

型也可以通过__proto__继续向上查找。我们把这种查找方式就叫原型链

类都是函数数据类型的,它是一个普通的函数还是一个类取决于执行的时候是否使用了 NEW 关键词,Fn()这就是一个普通的函数,new Fn()这就是一个类,我们相当于创建了类的一个实例

f1 是 Fn 这个类的一个实例 -> f1 instanceof Fn ->true

```
    function Fn() {
        this->当前类的实例 f1  this.xxx=xxx ->给当前的实例增加私有的属性
        this.name = '珠峰';->f1.name='珠峰'  给当前的实例增加一个叫做 name 的属性,属性值是'珠峰'
    }
    var f1 = new Fn();
 * new Fn / new Fn()
 * ->只有使用 NEW 创建类实例的时候,如果不需要传递参数的话,那么小括号可以加可以不加
 * ->当 new Fn 的时候,此时的 Fn 就不仅仅是普通的函数了,它还是一个类,f1 是这个类的一个实例(所有的类都是函数数据类型的,所有的实例都是对象数据类型的)
 * ->当 new Fn 的时候,首先 Fn 会向普通函数一样执行形成一个私有的作用域,然后给形参赋值,再然后进行私有作用域中的预解释；相对
```

于传统的函数，会在函数体中代码执行之前“浏览器默认创建一个对象数据类型的值，而默认创建的这个对象就是我们的实例，而且此时函数体中出现的 THIS 就是默认创建的这个实例，而出现的 this.xxx=xxx 都是在给当前的实例增加私有的属性，最后浏览器会默认的把创建的对象实例返回到外面，所以在外面定义一个 var f=... 中的 f 就是当前类的一个实例”

```
console.log(f1.name);->'珠峰'
console.log(f1.hasOwnProperty('name'));->true
console.log(f1 instanceof Fn);->true  instanceof 用来检测 f 是否属于 Fn 这个类的一个实例(它的本身意思就是用来检测某一个实例是否属于这个类)

function Fn() {
    var n = 10;->n 是当前作为一个普通函数执行,形成的私有的作用域中的一个私有的变量,和当前类的实例是没有必然的联系的,只有 this.xxx=xxx 才相当于在给当前的实例扩展私有的属性(因为构造函数模式中的 this 就是当前类的一个实例)
    this.index = Math.pow(n, 2);
}
var f = new Fn;
    console.log(f instanceof Fn);->true
    console.log(f instanceof
```

```
Object);->true ?console.log(f.hasOwnProperty('index'))
;->true
        console.log('index' in f);->true
console.log(f.index);->100
    console.log(f.n);->undefined 说明 n 不是当前实例的一个
属性 console.log('n' in f);->false
    function Fn() {
        var n = 10;
        this.index = Math.pow(n, 2);
        ->在构造函数模式中,浏览器会默认把创建的实例返回,所以
外面的 f 是当前类的实例
        ->如果我们自己在加上 RETURN
        返回的是一个基本数据类型：不会对最后的结果产生的任何
的影响
        返回的是一个引用数据类型：自己返回的结果会把默认返回
的实例覆盖掉,此时外面的接收的 f 就不在是 Fn 的实例了
        return {
            name: "珠峰"
        };
    }
    var f = new Fn;
```

```
    console.log(f);->{name:'珠峰'}
    console.log(f instanceof Fn);->false
```

8、在内置类的原型上扩展方法

->Array 数组的内置类

ary -> Array.prototype -> Object.prototype

Array.prototype:push、pop、shift、unshift、splice、slice、concat、join、toString、sort、reverse、indexOf、lastIndexOf、forEach、map、filter...

->重写内置的 SLICE 方法:我们可以在内置类的原型上扩展和重写内置方法,重写完成后以后在调取使用以重写后的为主

```
    Array.prototype.slice = function () {
        console.log("ok");
    };
    var ary = [12, 23, 34];
    ary.slice();->'ok'
```

->为了保证不修改内置的方法,我们在定义方法的时候最好把方法名前面追加特殊的前缀

```
    Array.prototype.myUnique=function(){};
    ~function (pro) {
        function myUnique() {
            ->this 是当前需要去重的那个数组
            var obj = {};
```

```
            for (var i = 0; i < this.length; i++) {
                var cur = this[i];
                if (obj[cur] == cur) {
                    ->已经重复了我们需要把当前这一项删除掉
                    this[i] = this[this.length - 1];
                    this.length--;
                    i--;
                    continue;
                }
                obj[cur] = cur;
            }
            obj = null;
            return this;
        }
        pro.myUnique = myUnique;
    }(Array.prototype);
->在数组原型上扩展的方法,那么数组的每一个实例都可以调取这个方法使用了 ary.myUnique...
Array.prototype.myUnique = function () {
    ->THIS 是当前要去重操作的那个数组
    var obj = {};
    for (var i = 0; i < this.length; i++) {
```

```
            var cur = this[i];
            if (obj[cur] == cur) {
                this[i] = this[this.length - 1];
                this.length--;
                i--;
                continue;
            }
            obj[cur] = cur;
        }
        obj = null;
        return this;->为了实现链式写法
};
var ary = [1, 2, 3, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2,1];
    ary.myUnique();->this 是 ary
    console.log(ary);
var ary2 = [2, 3, 4, 2, 3, 1, 2, 3, 4];
ary2.myUnique().sort(function (a, b) {
    return b - a;
});
console.log(ary2);
var ary = [1, 2, 3, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2,1];
    ary.sort();
```

ary.reverse();

ary.sort().reverse();->链式写法　ary 可以使用 sort 的原因是：sort 是 Array.prototype 上的一个方法,ary 是 Array 的一个实例,所以就可以调取使用了；ary.sort 执行完成的返回结果是排序后的数组,也依然是Array的实例,所以可以紧接着调取reverse这个方法了;

ary.sort().reverse().push(100).pop();->Uncaught TypeError: ary.sort(...).reverse(...).push(...).pop is not a function 执行完成 push 后返回结果是一个数字,新数组的长度,不在是 Array 的实例了,所以不能调取 pop 这个方法...

console.log(ary);