预解析、作用域、this 指向、闭包

预解析

定义：

在当前作用域下,js 运行之前，会把带有 var 和 function 关键字的事先声明，并在内存中安排好。然后再从上到下执行 js 语句。

预解析只会发生在通过 var 定义的变量和 function 上。

var：

通过 var 关键字定义的变量进行预解析的时候：都会声明，不管它有没有赋值，都会赋值 undefined

只要是通过 var 定义的，不管是变量，还是函数，都是先赋值 undefined，如果是变量，也不管变量有没有赋值，在预解析阶段，都是会被赋值为 undefined。

function：

function 进行预解析的时候，不仅会声明而且还会定义，但是它存储的数据的是代码是字符串，没有任何意义。

```
console.log(a);
function a(){
    console.log(“预解析 function")
}
立即执行
(function a(){
    console.log(“预解析 function")
}())

//闭包

(function a(){
```

```
    console.log(“预解析 function")
})()
```

预解析需要注意的情况

如果在当前作用域下的一个变量没有预解析，就会向它的上一级去找，直到找到 window，如果 window 下也没有定义，就会报错。所以，在函数内通过 var 定义的变量是局部变量，没有能过 var 定义的变量是全局变量。

函数表达式与函数声明

函数的好处有哪些？

1）代码复用
2）封装（隐藏细节）
3）控制代码的执行时机

函数声明方式

1 声明式

```
function test(){
    //声明会被提前，可以在任何地方调用
}
```

2 函数表达式

```
var test = function(){
    //只能在该函数以下位置，调用
     //建议使用这种方式，会使程序更具逻辑性、可读性
}
```

在编译的过程中，**所有的声明语句，都会被提前。**

```
b(); //当执行函数 b 的时候，由于声明被提前，可以执行
a(); //当执行函数 a 的时候，由于 a 仍未定义，不可执行
function b(){
    document.write("aa");
}
var a=function(){
    document.write("123");
}
以上代码相当于：
var a;
function b(){
document.write("aa");
}
b();
a();
a=function(){
        document.write("123");
}
```

函数与事件的结合？

onclick 单击
ondblclick 双击
onload 页面加载完成
onkeydown 键盘被按下
onkeyup 键盘被松开
onkeypress 键盘按下并松开
onsubmit 表单提交事件
onmouseenter 鼠标划入
onmouseleave 鼠标划出（区别于 onmouseout）

匿名函数如何运行？？

```
(function(){
    //函数在定义的同时，立即执行。 也叫作函数的自运行。
})();
或
(function(){
    //函数在定义的同时，立即执行。 也叫作函数的自运行。
}());
```

arguments 是什么？

该对象包含了函数执行时，传入的所有参数

```
function add(){
        //console.log(arguments instanceof Array); //确认是不是数组类型
        //console.log(arguments);
        var sum = 0;
        for(var i=0; i<arguments.length; i++){
                sum += arguments[i];
        }
        return sum;
}
console.log( add(10,222,32,46,5,6, 11,10,35) );

//arguments 本身不是数组，而是经过封装的一个对象，它包含一个数组，即我们的参数列表
//由于它可以当成数组来使用 for 循环遍历，因此，我们管这类对象，叫做伪数组
```

arguments.callee 是什么?

代表了函数本身

```
(function(num){
        if(num == 0) return 1;
        return num * arguments.callee(num-1);
})(5);
```

this 指向

- 作为对象的方法调用。
- 作为普通函数调用。
- 构造器调用。
- Function.prototype.call 或 Function.prototype.apply 调用

1.作为对象的方法调用

作为对象的方法调用 当函数作为对象的方法被调用时，this 指向该对象:

```
var obj = {
    a: 1,
    getA: function(){
        console.log( this === obj ); // 输出:true alert ( this.a ); // 输出: 1
        }
};
obj.getA();
```

2.作为普通函数调用

当函数不作为对象的属性被调用时，也就是我们常说的普通函数方式，此时的 this 总是指 向全局对象。在浏览器的 JavaScript 里，这个全局对象是 window 对象 。

```
window.name = 'globalName';
var getName = function(){
        return this.name;
};
console.log( getName() );
```

或者

```
window.name = 'globalName';
var myObject = {
        name: 'sven',
```

```
        getName: function(){ return this.name;  }
} };
var getName = myObject.getName;
```

getName 是一个函数， 这个时候函数里面的 this 是全局对象

3. 构造器调用

```
var MyClass = function(){
      this.name = 'sven';
};
var obj = new MyClass();
alert ( obj.name ); // 输出:sven
```

4. Function.prototype.call 或 Function.prototype.apply 调用

跟普通的函数调用相比，用 Function.prototype.call 或 Function.prototype.apply 可以动态地 改变传入函数的 this:

call 和 apply 用法

JavaScript 中的每一个 Function 对象都有一个 apply()方法和一个 call()方法，它们的语法分别为：

```
/*apply()方法*/
function.apply(thisObj[, argArray])

/*call()方法*/
function.call(thisObj[, arg1[, arg2[, [,...argN]]]]);
```

举例：

```
var school = '江汉大学';
function showinfo(name,weight) {
   console.log(this.school);
   console.log(name,weight);
}
showinfo.apply({school :'长江大学'},['laney','80'])
showinfo.call({school :'长江大学'},'laney','80')
```

call 和 apply 的用途

1. 改变 this 指向

call 和 apply 最常见的用途是改变函数内部的 this 指向，我们来看个例子:

```
var obj1 = {
        name: 'sven',
        getName: function(){ return this.name; }
};
var obj2 = { name: 'anne' };
console.log( obj1.getName( ) );
console.log( obj1.getName.call( obj2 ) );
```

2. Function.prototype.bind

```
var obj = {
    name: 'sven'
};
var func = function(){
    console.log ( this.name );
}.bind( obj);
func();
```

3. 借用其他对象的方法

```
var A = function( name ){
    this.name = name;
};
var B = function(){
   A.apply( this, arguments );
//站在 B 对象的立场， B 对象借用了 A 对象的方法，
//站在 A 对象的立场： A 对象的作用被改成了 B
}
```

```
B. prototype.getName = function(){
        return this.name;
};

var b = new B(‘laney’);
console.log( b.getName() );
```

apply，call 调用一个对象的一个方法，用另一个对象替换当前对象。

例如：B.apply(A, arguments);即 A 对象借用 B 对象的方法，达到继承的目的。

例如：B.call(A, args1,args2);即 A 对象调用 B 对象的方法,达到继承的目的。

共同之处：

都“可以用来代替另一个对象调用一个方法，将一个函数的对象上下文从初始的上下文改变为由 A 指定的新对象”。

```
function Person(){
   this.name = '木木';
   this.age = '21';
   console.log(this.name);
   console.log(this.age);
}
var obj = {
   sex:'男'
}
Person.apply(obj);
console.log(obj);

/*定义一个人类*/
function Person(name,age)
{
   this.name=name;
   this.age=age;
}
/*定义一 个学生类*/
function Student(name,age,grade)
{
   Person.apply(this,arguments);
   this.grade=grade;
}
//创建一个学生类
var student=new Student("zhangsan",21,"一年级");
```

```
//测试
console.log(student)
```

Array.prototype.push 可以实现两个数组的合并

同样 push 方法没有提供 push 一个数组，但是它提供了 push(param1,param2...paramN)，同样也可以用 apply 来转换一下这个数组，即：

```
var arr1=[1,2,3];
var arr2=[4,5,6];
Array.prototype.push.apply(arr1,arr2);  //但是会改变原数组 arr1
//得到合并后数组的长度，因为 push 就是返回一个数组的长度

var newSr = arr1.concat(arr2); //原数组 arr1 不变
```

Function 扩展

(1) Function.prototype.bind(obj) :
(2) 作用: 将函数内的 this 绑定为 obj, 并将函数返回
(3) 面试题: 区别 bind()与 call()和 apply()?
(4) 都能指定函数中的 this
(5) call()/apply()是立即调用函数
(6) bind()是将函数返回

```
var obj = {username: 'kobe'};

function foo(){
   console.log(this); //指向全局对象
}

// foo();
// foo.call(obj);
// foo.apply(obj);


//传入参数的形式
function foo(data){
   console.log(this, data);
}
```

```
// foo.call(obj, 33);//直接从第二个参数开始，依次传入
// foo.apply(obj, [33]);//第二参数必须是数组，传入的参数放在数组里

var bar = foo.bind(obj);
console.log(bar);
bind 的特点：绑定完 this 不会立即调用当前的函数，而是将函数返回
bar();
bind 传参的方式同 call 一样 foo.bind(obj, 33)();

setTimeout(function(){
    console.log(this);
}.bind(obj), 1000);

setTimeout(function(name){
    console.log(this,name);
}.bind(obj,'laney'), 1000);
```

call 和 apply 方法能很好地体现 JavaScript 的函数式语言特性，在 JavaScript 中，几乎每一次 编写函数式语言风格的代码，都离不开 call 和 apply。在 JavaScript 诸多版本的设计模式中，也 用到了 call 和 apply。

闭包

1. 闭包的定义及其优缺点

闭包是指有权访问另一个函数作用域中的变量的函数，创建闭包的最常见的方式就是在一个函数内创建另一个函数，通过另一个函数访问这个函数的局部变量

闭包的缺点就是常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。

闭包是 javascript 语言的一大特点，主要应用闭包场合主要是为了：设计私有的方法和变量。

一般函数执行完毕后，局部活动对象就被销毁，内存中仅仅保存全局作用域。但闭包的情况不同！

2. 闭包的特点：

1.函数嵌套函数

2.函数内部可以引用外部的参数和变量

3.参数和变量不会被垃圾回收机制回收

```
function A(x) {
   var count = 0;
   function B(y) {
     console.log(‘x’,x);
     console.log(‘y’,y);
     count ++;
     console.log(‘count’,count);
   }
   return B;
}
var C = A(2);
C(3);// 1
C(4);// 2
C(5);// 3
```

但是执行完毕之后， count 变量以及参数 x 就已经被释放回收了吗？并没有，因为返回值里面还等待使用这些变量，所以此时，A 虽然执行了，但是 A 的变量并没有被释放，return 在等待继续使用这些变量了，**这个时候 C 就是一个闭包。**
因此，函数 A 中的 count 变量会一直保存在内存中。

4. javascript 的垃圾回收原理

代码回收规则如下：
1.全局变量不会被回收。

2.局部变量会被回收，也就是函数一旦运行完以后，函数内部的东西都会被销毁。
3.只要被另外一个作用域所引用就不会被回收
4. 如果一个对象不再被引用，那么这个对象就会被 GC 回收
闭包是很多语言都具备的特性,在 js 中,闭包主要涉及到 js 的几个其他的特性:作用域链,垃圾(内存)回收机制,函数嵌套,等等.

准确来说，闭包是基于正常的垃圾回收处理机制下的。也就是说，一般情况一个函数（函数作用域）执行完毕，里面声明的变量会全部释放，被垃圾回收器回收。但闭包利用一个技巧，让作用域里面的变量，在函数执行完之后依旧保存没有被垃圾回收处理掉。

5. 使用闭包的好处

那么使用闭包有什么好处呢？使用闭包的好处是：
1.希望一个变量长期驻扎在内存中
2.避免全局变量的污染
3.私有成员的存在

5.1、全局变量的累加

```
var a = 1;
function abc(){
  a++;
}
abc();  //2
abc(); //3
//污染全局变量，不好，如何解决， =》 用闭包
```

5.2、局部变量

```
function abc(){
        var a = 1;
        a++;
        alert(a);
}
```

```
abc(); //2
abc();//2
```

那么怎么才能做到变量 a 既是局部变量又可以累加呢?

5.3、局部变量的累加

```
function outer(){
    var x=10;
    return function(){//函数嵌套函数
        x++;
        console.log(x);
    }
}
var y = outer();    //外部函数赋给变量 y;
y();              //y 函数调用一次，结果为 11，相当于 outer()();
y();              //y 函数调用第二次，结果为 12，实现了累加
```

5.4、模块化代码，减少全局变量的污染

```
var abc = (function(){     //abc 为外部匿名函数的返回值
    var a = 1;
    return function(){
        a++;
        alert(a);
    }
})();
abc();   //2 ；调用一次 abc 函数，其实是调用里面内部函数的返回值
abc();   //3
```

5.5、私有成员的存在

```
var mainFun= (function(){
    var a = 1;
    function bbb(){  a++;  }
    return {
        b:bbb,
    }
})();
mainFun.b();    //2
```

5.6.使用匿名函数实现累加

```
//使用匿名函数实现局部变量驻留内存中，从而实现累加
 function box(){
    var age = 100;
    return function(){  //匿名函数
        age++;
        return age;
    };
 }
var b = box();
console.log(b());
console.log(b());

b = null；   //解除引用，等待垃圾回收
```

过度使用闭包会导致性能的下降。函数里放匿名函数，则产生了闭包

5.7、在循环中直接找到对应元素的索引

```
function box(){
   var arr = [];
   for(var i=0;i<5;i++){
      arr[i] = i;
   }
   return arr;
}
box()

function box(){
   var arr = [];
   for(var i=0;i<5;i++){
      arr[i] = function(){
            return i;
         //由于这个闭包的关系，他是循环完毕之后才返回，最终结果是 5
      }
  //这个匿名函数里面根本没有 i 这个变量，所以匿名函数会从父级函数中去找 i，
   }
  //当找到这个 i 的时候，for 循环已经循环完毕了，所以最终会返回 5
```

```
    return arr;
}
```

5.8 使用闭包改写代码

实际的例子:

```
    var aLi = document.getElementsByTagName('li');
      for (var i=0;i<aLi.length;i++){
          aLi[i].onclick = function(){
           //当点击时 for 循环已经结束
            console.log(i);
        };
      }


      <ul>
            <li>123</li>
            <li>456</li>
            <li>789</li>
            <li>010</li>
       </ul>

    var aLi = document.getElementsByTagName('li');
       for (var i=0;i<aLi.length;i++){
          (function(i){
            aLi[i].onclick = function(){
               alert(i);
            };
        })(i);
      }
```

递归：

函数递归：当一个函数调用了函数自身，就会形成递归

1 阶乘 100!

F(1) = 1;
F(2) = 2*1 = 2*F(1)
F(3) = 3*2*1 = 3 * F(2)
F(4) = 4*3*2*1 = 4 * F(3)
.....
F(n) = n * f(n-1)

得出结论
f(n) = n * f(n-1)

```
function f(x) {
    return x * f(x-1);
}
```

最后代码如下：

```
function fn(num){
        if(num <=1){
            return 1;
        }else {
            return num *fn(num-1)
        }
    }
 fn(3);
```

2 求和 1+2+3+4+5.....+n

比如：10+9+8+7+6+....
F(1) = 1
F(2) = 1+2 =F(1) +2
F(3) = 1+2+3 =F(2) +3
F(4) = 1+2+3+4 =F(3) +4

结论：

F(n) = F(n-1)+n

```
function sum(num){
  if(num<=1){
    return 1;
  }else{
    return num+arguments.callee(num-1);
  }
}
```

这种方式很好的解决了函数名指向变更时导致递归调用时找不到自身的问题。但是这种方式也不是很完美，因为在严格模式下是禁止使用 arguments.callee 的。

3.阿里巴巴 2015 年前端面试题

请实现一个 fibNum 函数，要求其参数和返回值如下所示：

```
/**
 *@desc: fibonacci
 *@param: count {Number}
 *@return: result {Number} 第 count 个 fibonacci 值，计数从 0 开始
 斐波那契数列数列为：[1, 1, 2, 3, 5, 8, 13, 21, 34 …]
  则 getNthFibonacci(0)返回值为 1
  则 getNthFibonacci(4)返回值为 5*/
function getfib(count) { }
```

标准答案

```
function fib(n){
  if(n<=2) {
    return 1;
  }
  return fib(n-2)+fib(n-1);
}
```

4.求 n^m 次方

```
var f=function(a,b){
  if(b==0){
    return 1;
  }
  return f(a,b-1)*a;
}
```