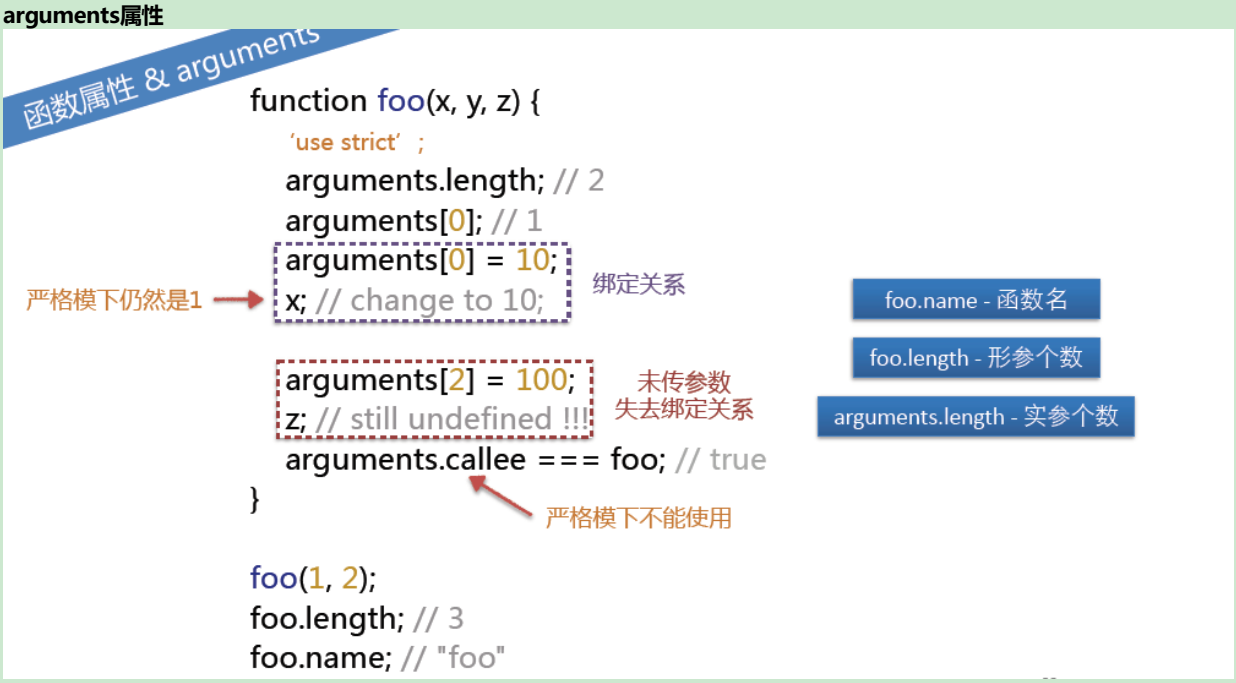
JS深入浅出3

arguments属性



arguments是什么？

答:1:arguments是收到的实参副本
在词法分析中，首先按形参形成AO的属性,值为undefined
当实参传来时，再修改AO的相应属性.

2:并把所有收到实参收集起来,放到一个arguments对象里
t(a,b,c){},
调用时：t(1,2,3,4,5) 5个参数
此时 ，AO属性只有a,bc,3个属性，arguments里有1,2,3,4,5，所有的值

对于超出形参个数之外的实参，可以通过arguments来获得
3:arguments 的索引 从 0，1,2,....递增,与实参逐个对应
4:arguments.length 属性代表实参的个数
5:arguments一定不是数组，是长的比较像数组的一个对象,虽然也有length属性
6:arguments每个函数都会有,因此,arguemnts只会在内部找自身的arguments,
无法引用到外层的arguments

bind与new

new会忽略bind()方法

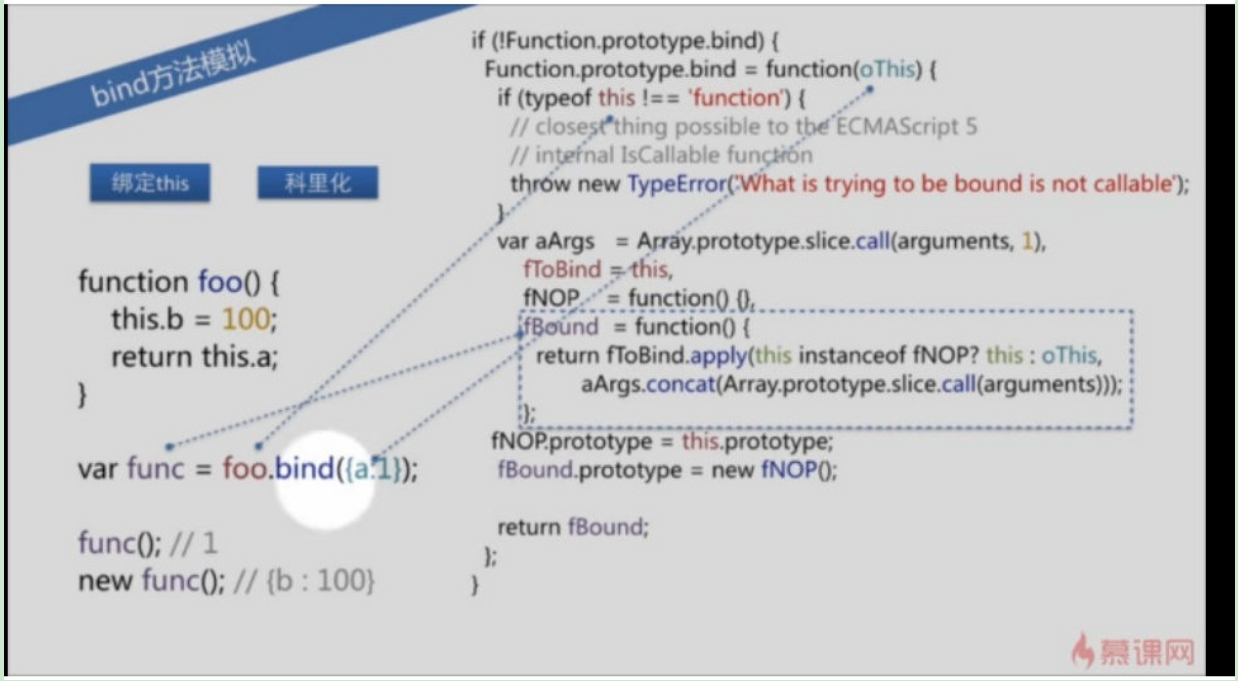
bind与curr ying

```
function add(a, b, c) {
return a + b + c;
}
var func = add.bind(undefined, 100);
func(1, 2); // 103
var func2 = func.bind(undefined, 200);
func2(10); // 310

function getConfig(colors, size, otherOptions) {
console.log(colors, size, otherOptions);
}
var defaultConfig = getConfig.bind(null, "#CC0000", "1024 * 768");
defaultConfig("123"); // #CC0000 1024 * 768 123
defaultConfig("456"); // #CC0000 1024 * 768 456
```

bind方法模拟

绑定this 颗粒化




```
if (!Function.prototype.bind) {
Function.prototype.bind = function(oThis) {
if (typeof this !== 'function') {
// closest thing possible to the ECMAScript 5
// internal IsCallable function
throw new TypeError('What is trying to be bound is not callable');
}
var aArgs = Array.prototype.slice.call(arguments, 1),
fToBind = this,
fNOP = function() {},
fBound = function() {
return fToBind.apply(this instanceof fNOP? this : oThis,
aArgs.concat(Array.prototype.slice.call(arguments)));
};
fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();
return fBound;
};
}

function foo() {
this.b = 100;
return this.a;
}
var func = foo.bind({a:1});
func(); // 1
new func(); // {b : 100}
```

闭包

```
function outer() {
var localVal = 30;
return function() {
return localVal;
}
}
var func = outer();
func(); // 30
```

闭包-常见错误之循环闭包

```
document.body.innerHTML = "<div id=div1>aaa</div>"
+ "<div id=div2>bbb</div><div id=div3>ccc</div>";
for (var i = 1; i < 4; i++) {
document.getElementById('div' + i).
addEventListener('click', function() {
alert(i); // all are 4!
```

```
});
}
document.body.innerHTML = "<div id=div1>aaa</div>"
+ "<div id=div2>bbb</div><div id=div3>ccc</div>";
for (var i = 1; i < 4; i++) {
!function(i) {
document.getElementById('div' + i).
addEventListener('click', function() {
alert(i); // 1, 2, 3
});
}(i);
}
```

for循环中的闭包只有等循环结束后才会执行，因此，如果要for循环内部的闭包与其同时执行，则需要用到立即执行方式：

闭包-封装

在计算机科学中，闭包（也称词法闭包或函数闭包）是指一个函数或函数的引用，与一个引用环境绑定在一起。这个引用环境是一个存储该函数每个非局部变量（也叫自由变量）的表。

闭包，不同于一般的函数，它允许一个函数在立即词法作用域外调用时，仍可访问非本地变量。

灵活方便	空间浪费 性能消耗
封装	内存泄露

作用域

全局 /函数 /eval

Javascript是没有块级作用域的

作用域链

```
function outer2() {
var local2 = 1;
function outer1() {
var local1 = 1;
// visit local1, local2 or global3
}
outer1();
}
var global3 = 1;
outer2();
function outer() {
var i = 1;
var func = new Function("console.log(typeof i);");
func(); // undefined}
outer();
}
```

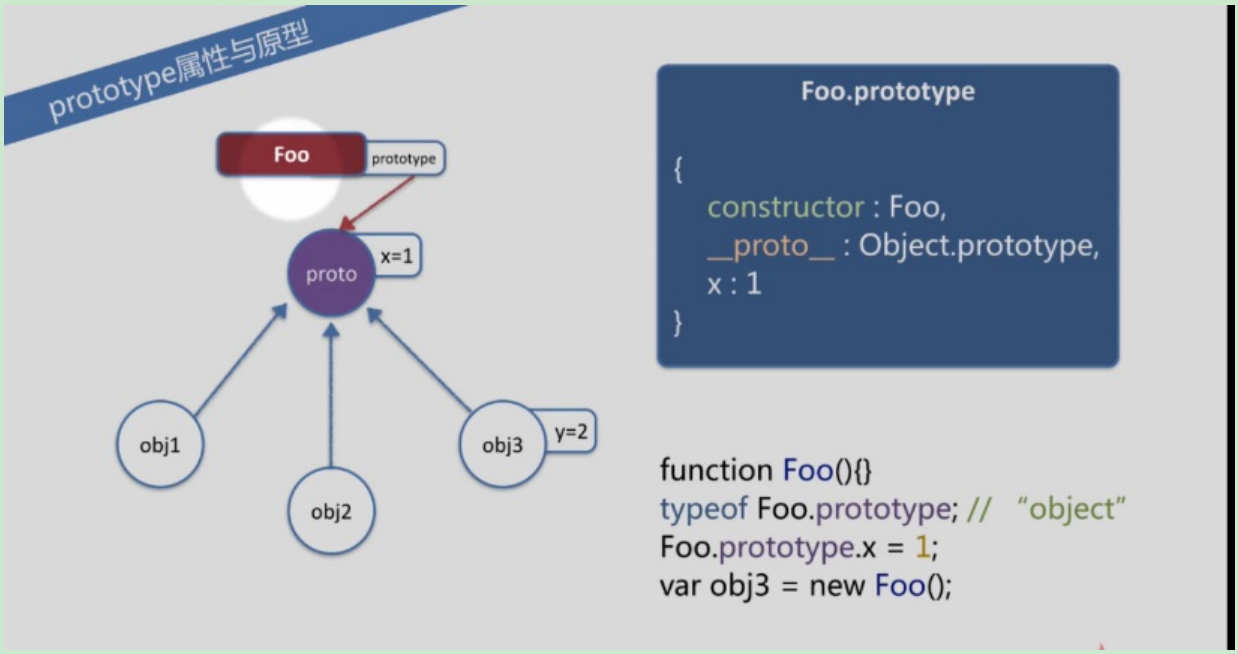
执行上下文


```
console.log('EC0');
function funcEC1() {
console.log('EC1');
var funcEC2 = function() {
console.log('EC2');
var funcEC3 = function() {
console.log('EC3');
};
funcEC3();
}
funcEC2();
}
```

funcEC1();
// EC0 EC1 EC2 EC3

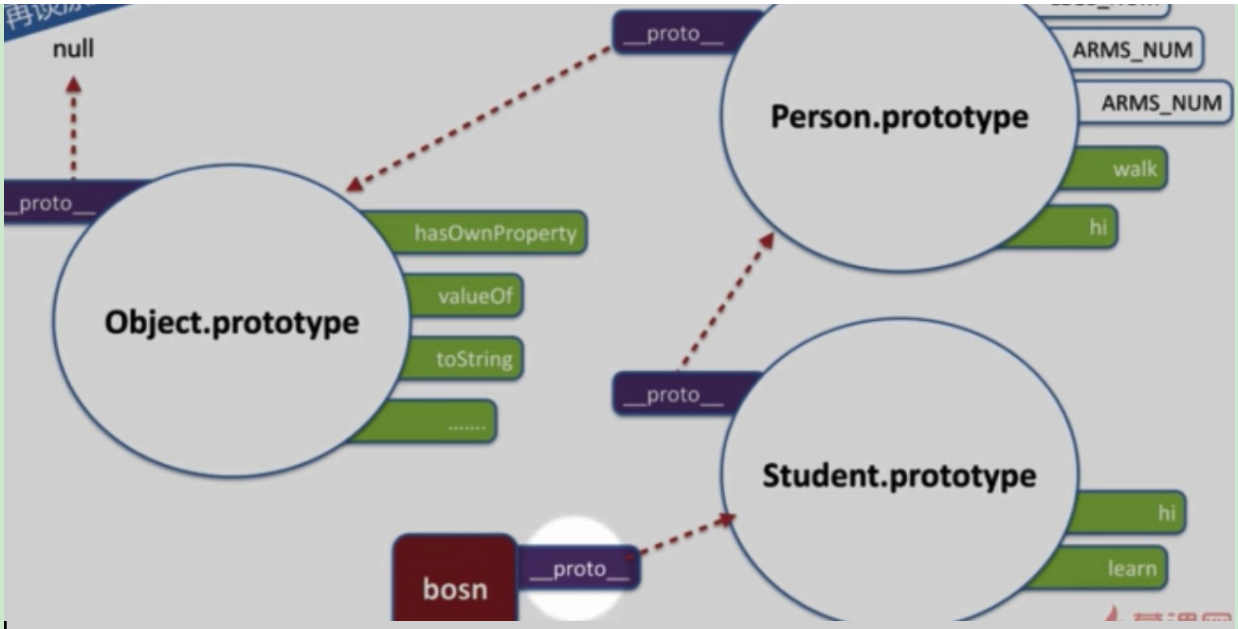
变量对象(Variable Object, 缩写为VO)是一个抽象
概念中的“对象”，它用于存储执行上下文中的：
1. 变量
2. 函数声明
3. 函数参数
VO按照如下顺序填充:
VO(globalContext) ===[[global]];
1. 1.
函数参数
(若未传入，初始化该参数值为undefined)
2. 2.
函数声明
(若发生命名冲突，会覆盖)
3. 3.
变量声明
(初始化变量值为undefined，若发生命名冲突，会忽略。)
函数表达式不会影响VO
oop
面向对象程序设计是一种 程序设计范型，同时也是一种程序开发的方法。对象指的是类的实例，它将对象作为程序的设计单元，将程序和数据包装封装其中，以提高软件的重用性 ，灵活性与扩展性
继承 封装 多态 抽象

原型



constructor 会将原型指向其本身

再谈原型链




```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.hi = function() {
    console.log('Hi, my name is ' + this.name + ", I'm " +
    this.age + " years old now.");
};

Person.prototype.LEGS_NUM = 2;
Person.prototype.ARMS_NUM = 2;
Person.prototype.walk = function() {
    console.log(this.name + " is walking...");
};

function Student(name, age, className) {
    Person.call(this, name, age);
    this.className = className;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

Student.prototype.hi = function() {
    console.log('Hi, my name is ' + this.name + ", I'm " +
        this.age + " years old now, and from " +
        this.className + ".");
};

Student.prototype.learn = function(subject) {
    console.log(this.name + ' is learning ' + subject +
        ' at ' + this.className + '.');
};

// test
var bosn = new Student('Bosn', 27, 'Class 3, Grade 2');
bosn.hi(); // Hi, my name is Bosn, I'm 27 years old now, and from Class 3, Grade
bosn.LEGS_NUM; // 2
bosn.walk(); // Bosn is  walking...
bosn.learn('math'); // Bosn is learning math at Class 3, Grade 2.
```

JavaScript的prototype原型不像java中的class是一旦写好了容易改变
但是JavaScript中的原型也是普通的对象，也可以动态的删除添加属性

动态的修改prototype的属性的时候
动态修改prototype属性会影响已创建和新创建的实例的。
修改整个prototype赋值给新的对象对已经创建的实例没有影响，会影响后续创建的实例。
第一个再实例创建后，给原型添加属性 会直接反应到实例里，是因为这个实例对应到一个student.prototype对象了已经 ， 再做student.prototype={y:2}只是改变student的prototype属性，不会影响到刚才那个已经存在的对象
修改prototype后，对已经创建的对象没有影响，但是影响后续创建的对象的值

```
Student.prototype.x=101;
bosn.x;//101

Student.prototype={y:2};
bosn.y;//undefined
bosn.x;//101

var nunnly =new Student('',3,'Class LOL kengB');
nunnly.x;//undefined
nunnly.y;//2
```

如果只想用对象上的属性，而不向上查找

用
obj.hasOwnProperty();//这个方法会判断是不是这个对象上的

实现继承的方式

1.Student.protptype = Person.prototype; // 禁止使用，修改子类时会一并修改父类
2.Student.prototype = new Person(); // 不推荐使用，使用Person的构造器创建会带回Person的参数
3.Student.prototype = Object.create(Person.prototype); // 理想的继承方式
如果不支持ES5的话

```
if(!Object.create){
Object.create=function(proto){
function F(){}
F.prototype= proto;
return     new F;
}
}
```

所有构造器/函数的__proto__都指向Function.prototype，它是一个空函数（Empty function）

二、所有对象的__proto__都指向其构造器的prototype

模拟重载

调用子类方法

```
function  Person(name) {this.name=name;}
function Student(name,classname){
this.className=className;
Person.call(this,name);
}
var bosn=new Student('Bosn','Network064');
bosn;//Student(calssName:"Newwork064",name:"Bosn")
Person.prototype.init=function(){};
Student.prototype.init=function(){
Person.prototype.init.apply(this,arguments);
}
```

链式调用

```
function classManager.prototype.addClass=function(str){
console.log('Class'+str+'added');
return this;
}
var manager=new classmanage();
manager.addClass('classA').addClass('classb')
.addClass('classC');
//Class:classA and added;
//Class:classB and added;
//Class:classC and added;
```

抽象类

```
function DetectorBase(){
throw new Error('Abstract class can not invoked directly!');
}
DetectorBase.detect =function(){console.log('Detection starting.....');};
DetectorBase.stop=function(){console.log('Detection stoping');};
DetectorBase.init=function(){throw new Error('Error');};

function LinkDetector(){}
LinkDetector.prototype =Object.create(DetectorBase.prototype);
LinkDetector.prototype.constructor=LinkDetector;
//...add methods to LinkDetector...
```

模块化

```
var  moduleA;
   moduleA=function(){
var prop=1;
```

```
function func() {}
return {
func: func,
prop:prop
}
}();

var  moduleA;
moduleA=new function(){
var prop =1;
function func(){}
this.func=func;
this.prop=prop;
};
```