网易微专业之《前端开发工程师》

学习笔记

开始时间：2015.12.28

《JavaScript 程序设计》

进阶篇

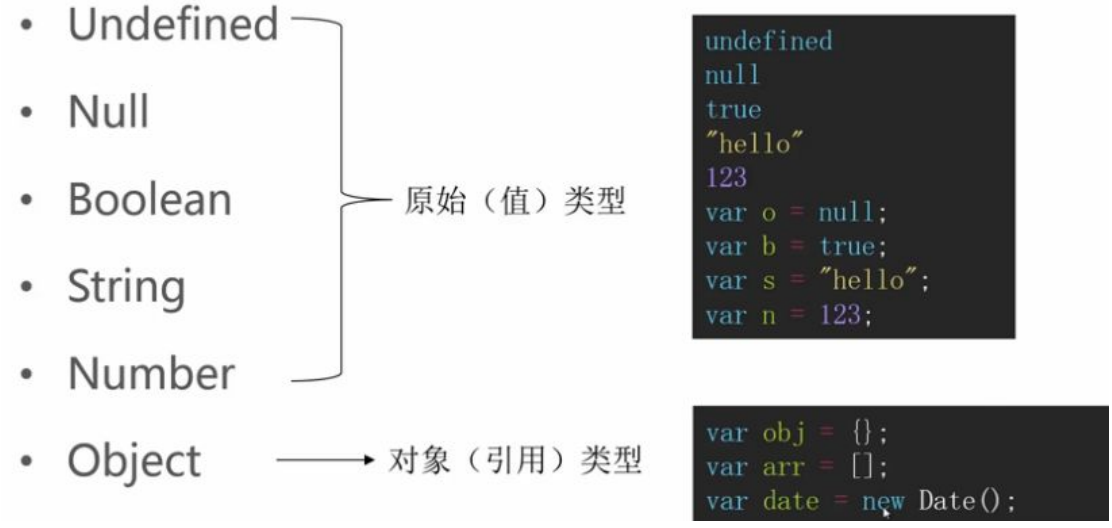
一、类型进阶

JavaScript 基本数据类型包括：

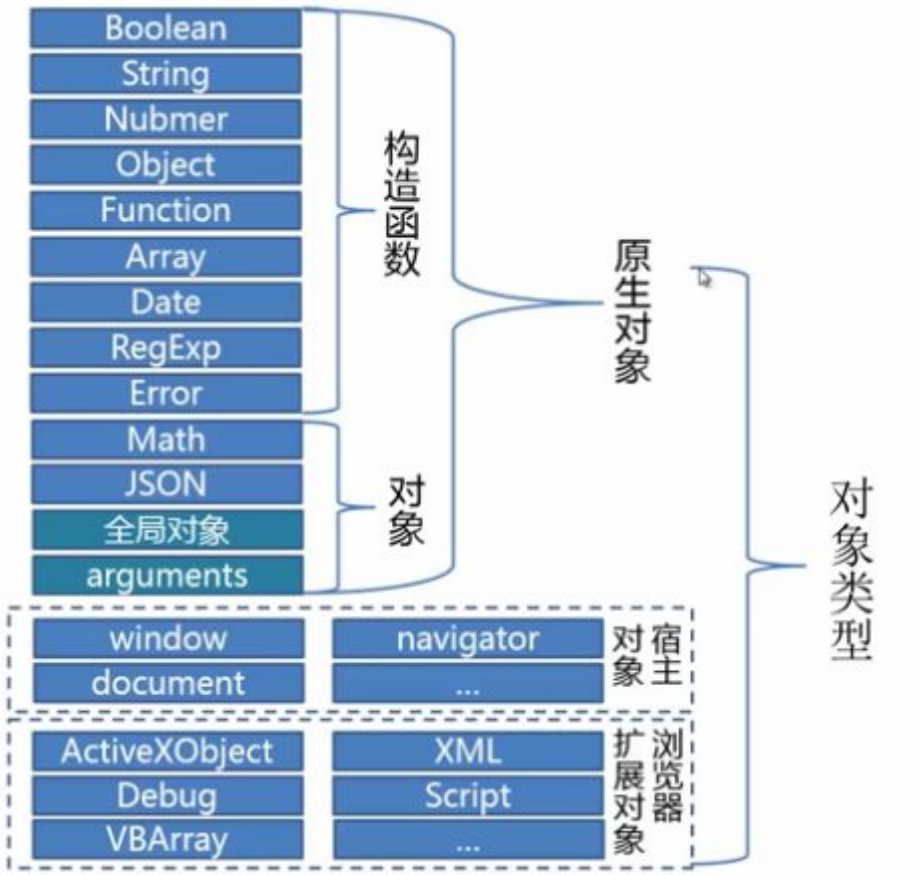
1. **Number(数字型)**
2. **String**（字符串行）
3. **Boolean**（布尔型）
4. **Object** （引用数据类型/对象型）
5. **Null**（空值）
6. **Undefined**（未定义值）

javascript 是以 Object 为基础的语言，除基本数据类型外，其他所有的引用数据类型，本质上都是 Object。

原始类型和引用类型

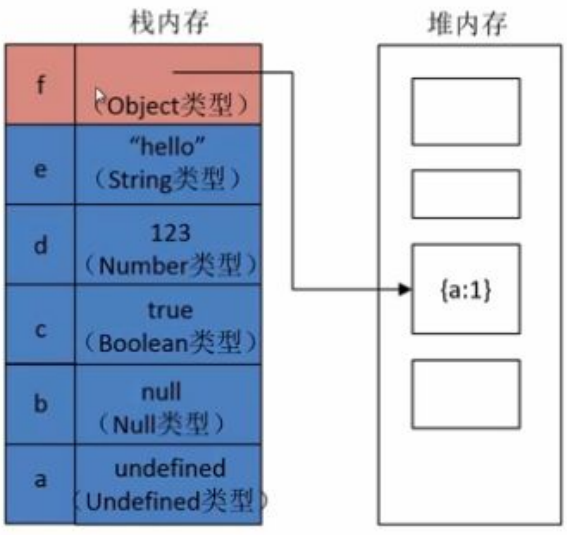


JavaScript 对象类型

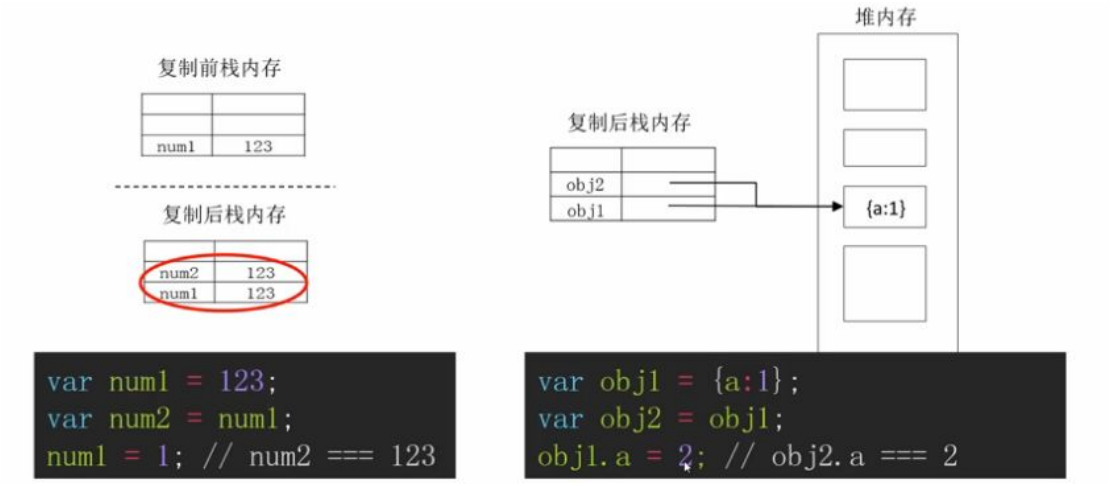


原始类型和对象类型的区别：

```
var a;
var b = null;
var c = true;
var d = 123;
var e = "hello";
var f = {a:1};
```



- 原始类型的值保存在栈内存里；
- 对象类型在栈内存里只是保存了一个地址，其值保存在堆内存里。



隐式类型转换

在 JavaScript 中声明变量不需指定类型， 对变量赋值也没有类型检查，同时 JavaScript 允许隐式类型转换。这些特征说明 JavaScript 属于弱类型的语言。

- 数字运算符
- if 语句
- .
- ==

A: 数字运算符

1. 字符串加数字,数字就会转成字符串。

2. 数字减字符串，字符串转成数字。如果字符串不是纯数字就会转成 NaN。字符串减数字也一样。两个字符串相减也先转成数字。

3. 乘，除，大于，小于跟减的转换也是一样。

```
//隐式转换 + - * == /
// +
10 + '20' //'1020'
// -
10 - '20' //-10
10 - 'one' //NaN
10 - '100a' //NaN
// *
10*'20' //200
'10'*'20' //200
// /
20/'10' //2
'20'/'10' //2
'20'/'one' //NaN
```

4.加法操作顺序是敏感的

类似这样的混合表达式有时令人困惑，因为 JavaScript 对操作顺序是敏感的。例如，表达式：**1+2+"3"; //"33"**

由于加法运算是自左结合的（即左结合律），因此，它等同于下面的表达式：

```
(1+2)+"3"; //"33"
```

与此相反，表达式：`1+"2"+3; //"123"`的计算结果为字符串“123”。左结合律相当于是将表达式左侧的加法运算包裹在括号中。

B. if 语句

判断语句中的判断条件需要是`Boolean`类型，所以条件表达式会被隐式转换为`Boolean`。其转换规则同`Boolean`的构造函数。比如：

```
var obj = {};if(obj){

   while(obj);}
```

C. ==

1).**undefined 等于 null**

2).字符串和数字比较时，字符串转数字

3).数字与布尔比较时，布尔转数字

4).字符串和布尔比较时，两者转数字

```
// ==
undefined == null; //true
undefined === null; //false
'0' == 0;       //true, 字符串转数字
0 == false; //true, 布尔转数字
'0' == false;       //true, 两者转数字
null == false || null==true;    //false
undefined == false || undefined==true;   //false
NaN==NaN;       //false
NaN===NaN;   //false
NaN!=NaN   //true
```

7 个 false 值：**false，0，-0，“”，NaN，null 以及 undefined**，所有其他值都是 truth.

D. .

●操作，JS 进行隐式类型转换。

所有的直接量（数字型/字符串型），用●号去调用某个方法时，JS 运行环境会将这个直接量隐式的转换成对应的对象类型，然后去调用对象类型的方法来实现各种功能。

```
(3.1415).toFixed(2)    "hello world".split(" ")
"3.14"                 ["hello", "world"]
```

隐式类型转换结果：

	原值	Boolean	Number	String	Object
Undefined	undefined	false	NaN	"undefined"	
Null	null	false	0	"null"	
Boolean	true		1	"true"	new Boolean(true)
	false		0	"false"	new Boolean(false)
String	""	false	0		new String("")
	"123"	true	123		new String("123")
	"1a"	true	NaN		new String("1a")
Number	0	false		"0"	new Number(0)
	1	true		"1"	new Number(1)
	Infinity	true		"Infinity"	new Number(Infinity)
	NaN	false		"NaN"	new Number(NaN)
Object	{}	true	NaN	"[object Object]"	

显示类型转换方法

显式转换比较简单，可以直接用类当作方法直接转换。

- Number(),String(),Boolean()
- parseInt(),parseFloat()
- !,!!

案例：

```
Number([]); //0
String([]); //"
Boolean([]); //true
[] == ![]; //true
0 == []; // true
0==![]  //true
NaN!=NaN; //true
```

类型识别：

- typeof
- instanceof
- Object.prototype.toString.call
- Constructor
- getConstructorName

typeof

typeof 是一个一元运算，放在一个运算数之前，运算数可以是任意类型。

它返回值是一个字符串，该字符串说明运算数的类型。typeof 一般只能返回如下几个结果：

number, boolean, string, function, object, undefined。

我们可以使用 typeof 来获取一个变量是否存在，如 if(typeof a!="undefined"){alert("ok")}，而不要去使用 if(a) 因为如果 a 不存在（未声明）则会出错，对于 Array,Null 等特殊对象使用 typeof 一律返回 object，这正是 typeof 的局限性。

```
typeof "jerry"; // "string"
typeof 12;  // "number"
typeof true;    // "boolean"
typeof undefined;   // "undefined"
typeof null;    // "object"
typeof {name: "jerry"}; // "object"
typeof function(){};    // "function"
typeof []; // "object"
typeof new Date; // "object"
typeof /\d/; // "object"
function Person(){};
typeof new Person;  // "object"
```

typeof总结：

- 可以识别标准类型（Null除外）
- 不能识别具体的对象类型（Function除外）

instanceof

instanceof 用于判断一个变量是否某个对象的实例(一个实例是否属于某一类型)，用来识别对象类型。

```
var a=new Array();
alert(a instanceof Array); //true
alert(a instanceof Object);//true.(因为 Array 是 object 的子类)
function test(){};
var a=new test();
alert(a instanceof test); //true
```

```
//能够判别内置对象类型
[] instanceof Array;    // true
/\d/ instanceof RegExp; // true
//不能判别原始类型
1 instanceof Number;    // false
"jerry" instanceof String;   // false
//能够判别自定义对象类型及父子类型
function Point(x, y){
    this.x = x;
    this.y = y;
}
function Circle(x, y, r) {
    Point.call(this, x, y);
    this.radius = r;
}
Circle.prototype = new Point();
Circle.prototype.constructor = Circle;
var c = new Circle(1, 1, 2);
c instanceof Circle // true
c instanceof Point  // true
```

Instanceof总结：

- 判别内置对象类型
- 不能判别原始类型
- 判别自定义对象类型

案例：如何实现四种输入方式返回 Date 的函数

```
/*
* 输入格式：
* '2015-08-05'
* 1438744815232
* {y:2015,m:8,d:5}
* [2015,8,5]
* 返回格式：Date
*/
function toDate(param){
  if (typeof(param) == 'string' ||
      typeof(param) == 'number' ){
    return new Date(param);
  }
  if (param instanceof Array){
    var date = new Date(0);
    date.setYear(param[0]);
    date.setMonth(param[1]-1);
    date.setDate(param[2]);
    return date;
  }
  if (typeof(param) == 'object') {
    var date = new Date(0);
    date.setYear(param.y);
    date.setMonth(param.m-1);
    date.setDate(param.d);
    return date;
  }
  return -1;
```

```
}
```

Object.prototype.toString.call(xxx)或{}.prototype.toString.call(xxx)

实际在使用时 Object.prototype.toString.call(xxx)会返回类似[object String]的字符串给我们，用他我们就可以很好的判断变量的类型了。

示例：

```
> Object.prototype.toString.call(123)
< "[object Number]"
> Object.prototype.toString.call("123")
< "[object String]"
```

将方法封装为一个函数：

```
function type(obj){
    return Object.prototype.toString.call(obj).slice(8, -1);
}
```

```
function type(obj){
    return Object.prototype.toString.call(obj).slice(8, -1);
}
type(1) // "Number"
type("abc") // "String"
type(true) // "Boolean"
type(undefined) // "Undefined"
type(null) // "Null"
type({}) // "Object"
type([]) // "Array"
type(new Date) // "Date"
type(/\d/) // "RegExp"
type(function(){}) // "Function"
function Point(x,y){
    this.x = x;
    this.y = y;
}
type(new Point(1,2)); // "Object"
```

Object.prototype.toString.call总结：

- 可以识别标准类型以及内置(build-in)对象类型
- 不能识别自定义对象类型

Constructor

生成这个对象的构造函数的构造函数本身。

封装构造函数名称的方法：

```
//获取对象构造函数名称
function getConstructorName(obj){
    return obj && obj.constructor && obj.constructor.toString().match(/function\s*([^(]*)/)[1];
}
```

封装函数解析：

- obj //保证是 null/undefined 时，直接返回他们本身，而不进行&&后面的操作。。
- obj.constructor //保证这个对象存在构造函数，保证&&后面的语句正常执行。
- obj.constructor.toString(); //”function Number(){[native code]}”,构造函数的名称，这里假设为 Number()对象。
- .match(/function\s*([^(]*)/)[1]; //”Number” ，通过.match 方法获取字符串中的子字符串

参考资源：

1. **跟我学习 javascript 的隐式强制转换**

---

http://www.jb51.net/article/74894.htm

2. JavaScript 显式类型转换与隐式类型转换

http://blog.csdn.net/yangjvn/article/details/48284163

3.javascript 数据类型隐式转换

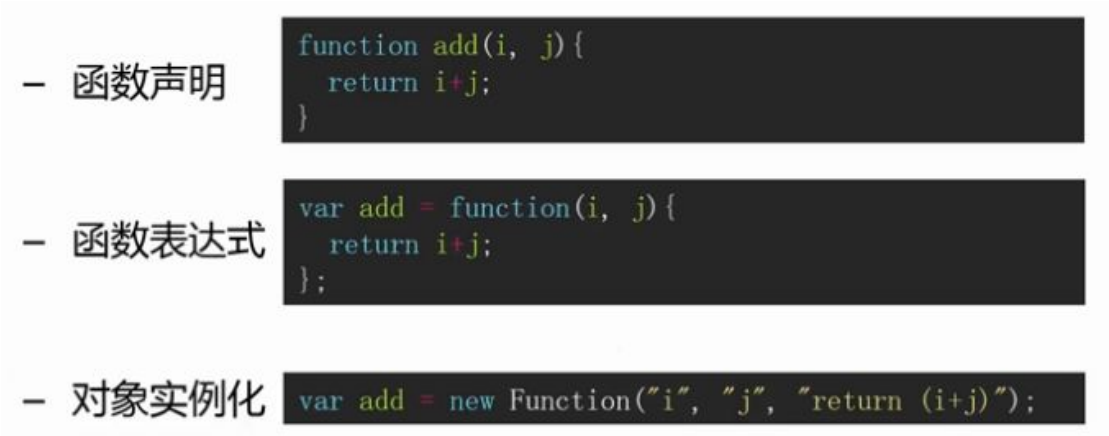
http://blog.csdn.net/zuoanren/article/details/8248818

4. **JavaScript 中的隐式类型转换**

http://www.cnblogs.com/snandy/archive/2011/03/18/1987940.html

二、函数进阶

函数定义（3 种）



1. 函数声明语句通常出现在 JS 代码的最顶层，也可以嵌套在其他函数体内。但在嵌套时，函数声明只能出现在所嵌套函数的顶部。
2. 函数定义表达式不能出现在 if 语句、while 循环或其他任何语句中，正是由于函数声明位置的这种限制，ES 标准规范并没有将函数声明归类为真正的语句。
3. 尽管函数声明语句和函数定义表达式包含相同的函数名，但二者仍然不同。两种方式都创建了新的函数对象，但函数声明语句中的函数名是一个变量名，变量指向函数对象。和通过 var 声明变量一样，函数定义语句中的函数被显式地“提前”到了脚本或函数的顶部。因此它们在整个脚本和函数内都是可见的。使用 var 的话，只有变量声明提前了--变量的初始化代码仍然在原来的位置。然而使用函数声明语句的话，函数名称和函数体均提前；脚本中的所有函数和函数中所有嵌套的函数都会在当前上下文中其他代码之前声明。也就是说，可以在声明一个 js 函数之前调用它。

函数 3 种定义区别：

函数声明与函数表达式、对象实例化的区别

- 用函数声明定义的函数，可以在函数声明之前调用；
- 用函数表达式定义的函数，必须在函数表达式之后调用；
- 对象实例化定义的函数，也只能在对象实例化之后才能调用。

对象实例化与函数声明、函数表达式的区别

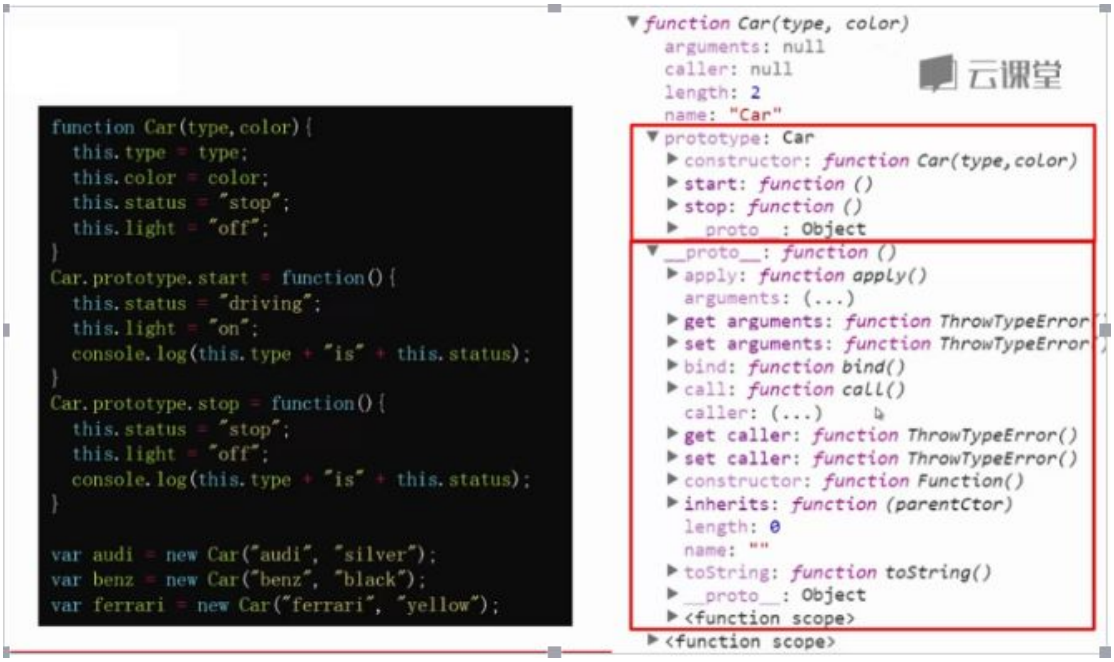
通过对象实例化方法定义的所有函数，都会定义在全局作用域上，因此无法访问到它的父函数上面的所有变量。

- prototype:原型对象属性
- _proto_:原型链属性，是隐藏属性，但原型链属性的方法可以被创建的对象直接调用到；

_proto_原型链属性来自于 function 构造函数属性。

构造函数

以下是一个构造函数的例子，this 用于指定所创建对象（Car）的属性值.Car.prototype 是原型对象属性。



构造函数与普通函数的区别：

- 本质上没有区别，普通函数也可以被当做构造函数来使用。

- 构造函数通常会有 this 指定实例属性，原型对象上通常有一些公共方法。
- 构造函数命名通常首字母大写。

构造函数代表一系列对象的属性及行为的封装，代表一系列对象的类型，通常用首字母大写的方式告诉这是一个类型。

函数调用

构成函数主体的 JS 代码在定义之时并不会执行，只有调用该函数时，它们才会执行。
函数调用时，会自动在函数内部调用两个参数：this 和 arguments 参数。根据函数 this 参数，可以将函数调用分为 4 类：

- 构造函数调用模式
- 方法调用模式
- 函数调用模式
- apply(call)调用模式

构造函数调用模式

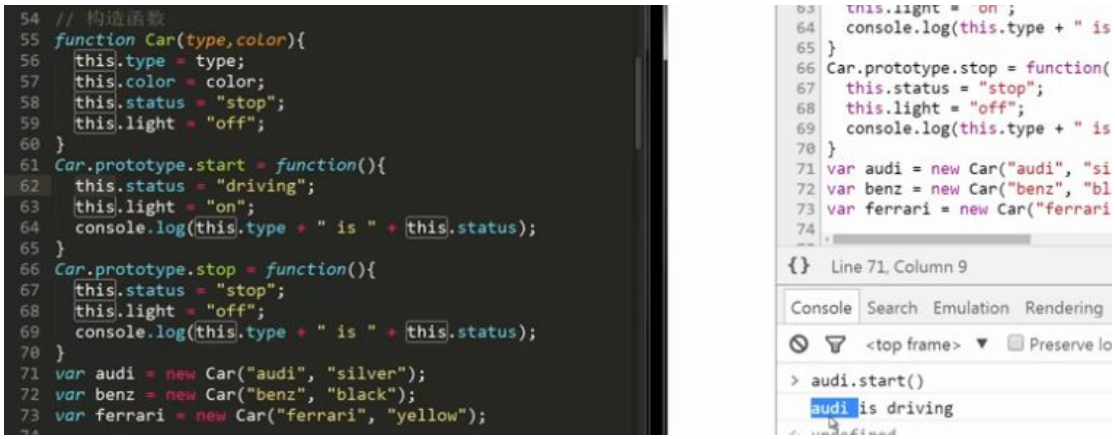
new +构造函数来调用，函数内部的 this 指向被创建的对象

如：

```
// // 构造函数
// function Car(type,color){
//   this.type = type;
//   this.color = color;
//   this.status = "stop";
//   this.light = "off";
// }
// Car.prototype.start = function(){
//   this.status = "driving";
//   this.light = "on";
//   console.log(this.type + " is " + this.status);
// }
// Car.prototype.stop = function(){
//   this.status = "stop";
//   this.light = "off";
//   console.log(this.type + " is " + this.status);
// }
// var audi = new Car("audi", "silver");
// var benz = new Car("benz", "black");
// var ferrari = new Car("ferrari", "yellow");
```

方法调用模式

作为对象属性的函数，称为方法。如下图，start()为 audi 的方法。



函数调用模式

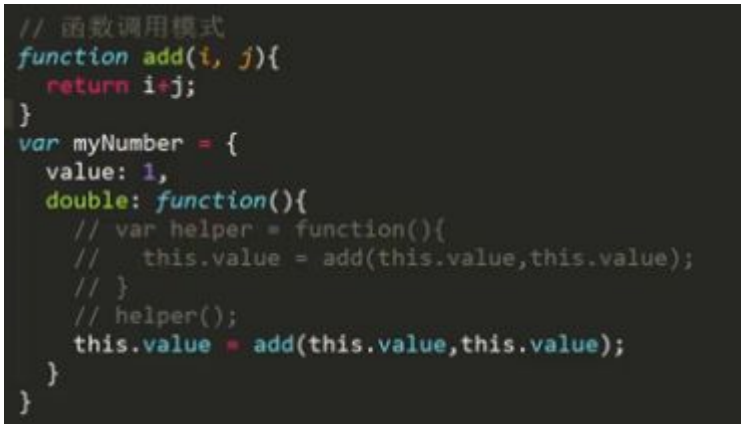
函数调用模式当中，在函数内部定义的子函数被调用时，函数内部的 this 是全局对象，其值仍然指向 window，而不是它的上一级对象。

如：

```
// // 函数调用模式
// function add(i, j){
//    return i+j;
// }
// var myNumber = {
//    value: 1,
//    double: function(){
//      var helper = function(){
//        this.value = add(this.value,this.value);
//      }
//      helper();
//    }
// }
```

helper 子函数中的 this 值仍指向 window,此时 myNumber 的值为 1，而不是 2.

如何解决，下面是两种方法，此时 myNumber 值为 2.



```
// 函数调用模式
function add(i, j){
  return i+j;
}
var myNumber = {
  value: 1,
  double: function(){
    var that = this;
    var helper = function(){
      that.value = add(that.value,that.value);
    }
    helper();
  }
}
```

apply(call)调用模式

任何函数可以作为任何对象的方法来调用，哪怕这个函数不是那个对象的方法。我们可以将 call()和 apply()看作是某个对象的方法，通过调用方法的形式来间接调用函数。call()和 apply()的第一个实参是要调用函数的母对象，它是调用上下文，在函数体内通过 this 来获得它的引用。apply 是 Function 原型构造函数上的一个方法,所有的函数都可以调用 apply 方法

Function.prototype.apply

```
Object.prototype.toString.apply("123"); // "[object String]"
```

```
function Point(x, y){
    this.x = x;
    this.y = y;
}
Point.prototype.move = function(x, y) {
    this.x += x;
    this.y += y;
}
var p = new Point(0,0);
p.move(2,2);
var circle = {x:1, y:1, r:1};
p.move.apply(circle, [2, 1]); // {x: 3, y: 2, r: 1}
```

Function.prototype.bind 方法

```
// // bind 的使用
// function Point(x, y){
//     this.x = x;
//     this.y = y;
// }
// Point.prototype.move = function(x, y) {
//     this.x += x;
```

```
//    this.y += y;
// }
// var p = new Point(0,0);
// var circle = {x:1, y:1, r:1};
// var circleMove = p.move.bind(circle, 2, 1);
// circleMove();
```

函数调用（arguments）

1. Array-like (看上去像数组，但不是数组，不能使用数组的一些方法)

-arguments[index]

-arguments.length

```
// // arguments 转数组
// function add(i, j) {
//   var args = Array.prototype.slice.apply(arguments);
//   args.forEach(function(item){
//     console.log(item);
//   })
// }
// add(1,2,3);
```

2. arguments.callee -指向函数本身，适用场景：匿名函数实现递归

```
// // arguments.callee 使用
// console.log(
//   (function(i){
//     if (i==0) {
//       return 1;
//     }
//     return i*arguments.callee(i-1);
//   })(5)
// );
```

上述代码实现 5 的阶乘，结果为 120

递归

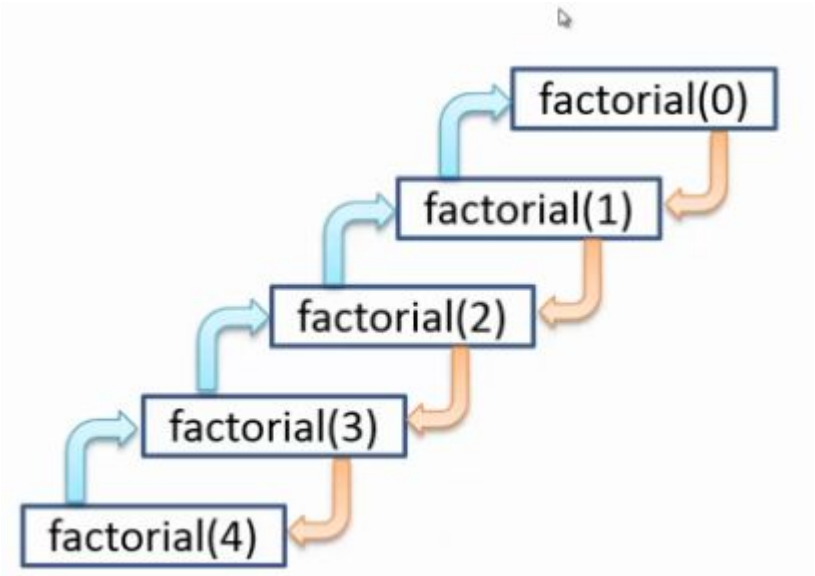
递归是在函数内部调用自身函数的一种编程方法。

- 递归函数必须有一个明确的退出条件；
- 在函数内部，会调用函数本身。

案例：阶乘函数

```
// // 递归
// function factorial(i){
//   if (i==0) {
//     return 1;
//   }
//   return i*factorial(i-1);
// }
```

递归调用栈示意图：（4 的阶乘）



闭包

在函数内部定义一个函数，这个函数调用到它的父函数上的相关联系变量，这些联系变量会被放入闭包作用域里面。

闭包作用：

- 属性隐藏、函数封装
- 记忆函数：减少函数计算量

```
// // 普通递归函数跟记忆函数调用次数对比
// var factorial = (function(){
//   var count = 0;
//   var fac = function(i){
//     count++;
```

```
//     if (i==0) {
//       console.log('调用次数: ' + count);
//       return 1;
//     }
//     return i*factorial(i-1);
//   }
//   return fac;
// })();
// for(var i=0;i<=10;i++){
//   console.log(factorial(i));
// }

// // 记忆函数
// var factorial = (function(){
//   var memo = [1];
//   var count = 0;
//   var fac = function(i){
//     count++;
//     var result = memo[i];
//     if(typeof result === 'number'){
//       console.log('调用次数: ' + count);
//       return result;
//     }
//     result = i*fac(i-1);
//     memo[i] = result;
//     return result;
//   }
//   return fac;
// })();
// for(var i=0;i<=10;i++){
//   console.log(factorial(i));
// }
```

普通递归函数实现 10 的阶乘，累计 66 次计算；通过记忆函数，累计 21 次，可以很明显减少函数的计算量。factorial 是一个闭包，包含了函数和函数的引用环境。当执行 factorial(i)时，一些中间计算结果会被保存到其引用环境中的数组 memo。比如，计算 factorial(10), 则 factorial(1), factorial(2), ..., factorial(10)的值都会被保存到数组 memo 中。下一次计算 factorial(),就可以直接使用存在数组 memo 里的值。

First-class function

JS 当中的函数，可以被当做变量保存，然后被当做参数传递，和被当做返回值返回。

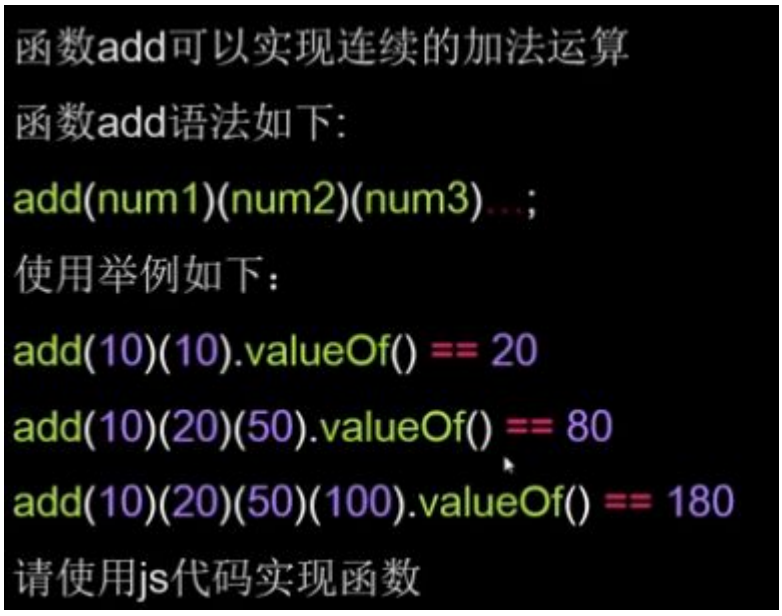
示例：

```
[1,2,3].forEach(function(item){
  console.log(item);
})
```

应用场景：

1.curry:函数柯里化

将接受多个参数的函数，转换为接受单一参数函数并返回一个接受其他后续参数的函数的转换过程。简单来说就是：将复杂问题片段化，使之进行简化。



实现代码：

```
// // curry 函数柯里化
// function add(value){
//     var helper = function(next){
//         value = typeof(value)==="undefined"?next:value+next;
//         return helper;
//     }
//     helper.valueOf = function(){
//       return value;
//     }
//     return helper
// }
```

2.回调

```
function ajax_get(url, callback){
  var createXMLHTTPRequest = function () {
    var xmlHttpRequest;
    if (window.XMLHttpRequest) {
      xmlHttpRequest = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
      xmlHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
    }
    return xmlHttpRequest;
  }
  var xhr = createXMLHTTPRequest();
  if(xhr){
    xhr.open("GET", url, true);
    xhr.onreadystatechange = function(){
      if(xhr.readyState == 4){
        if(xhr.status == 200){
            callback.success(xhr);
        }else{
            callback.fail(xhr);
        }
      }
    }
    xhr.send(null);
  }
}
```

三、JS 原型

原型

原型 VS 类

- 类构造对象，是从抽象到具体；
- 原型构造对象，是从具体到具体。（即使用一个现有对象去构造一个新对象，或者用一个现成对象为原型去构造一个新对象）

设置对象的原型

从原型对象构造新对象的方法：

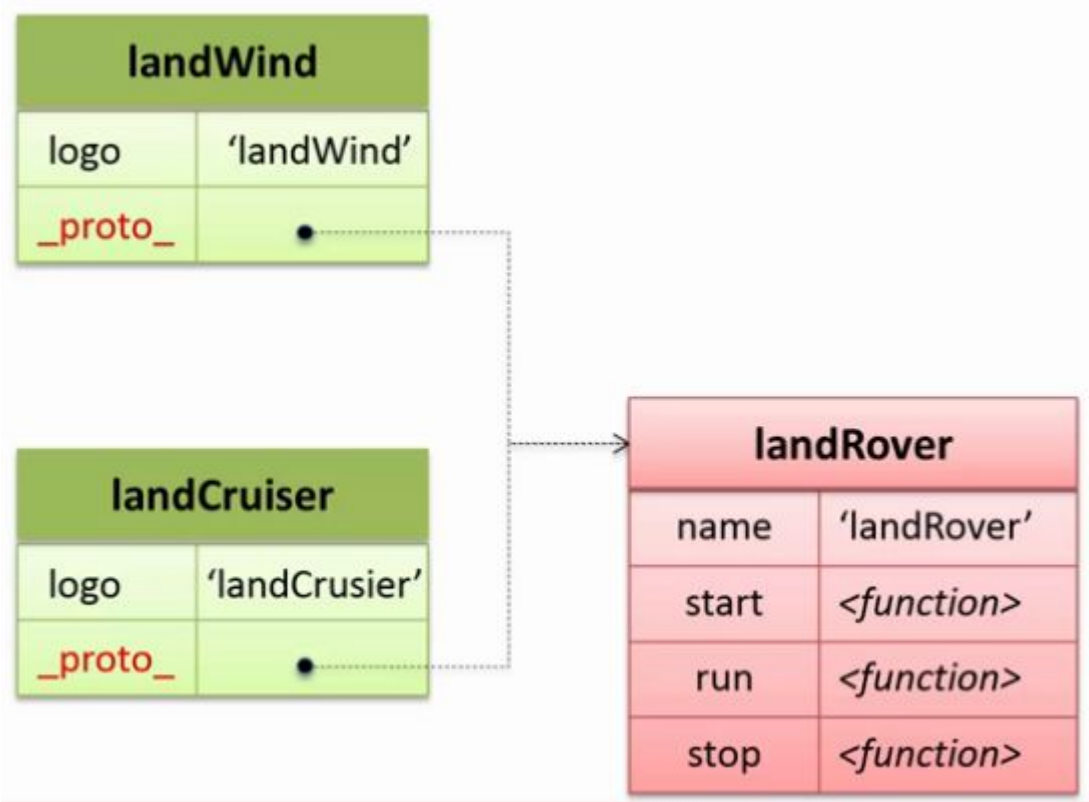
方法:1：Object.creat(proto[,propertiesObject])：

传入一个原型对象 proto,函数执行完会返回一个新的对象

- proto:一个对象，作为新创建对象的原型
- propertiesObject:对象的属性定义

案例：

```
//定义原型对象
var landRover = {
    name: 'landRover',
    start: function() {
        console.log('%s start', this.logo);
    },
    run: function() {
        console.log('%s running', this.logo);
    },
    stop: function() {
        console.log('%s stop', this.logo);
    }
}
//使用原型创建新的对象
var landWind = Object.create(landRover);
landWind.logo = 'landWind';

var landCruiser = Object.create(landRover);
landCruiser.logo = 'landCruiser';
```



_proto_属性：指向对象的原型，是个隐藏属性，不允许被修改。

方法 2：构造函数

- 使用 prototype 属性设置原型
- 使用 new 创建对象：

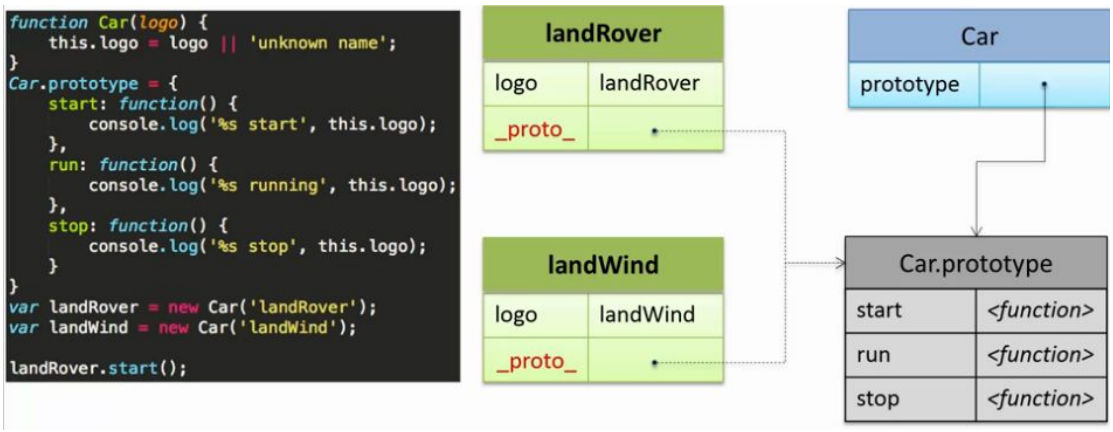
构造函数都有一个指针指向原型，Object.prototype 是所有原型对象的顶层。

prototype 是 javascript 实现与管理继承的一种机制，也是面向对象的设计思想. 构造函数的原型存储着引用对象的一个指针，该指针指向与一个原型对象，对象内部存储着函数的原始属性和方法；我们可以借助 prototype 属性，可以访问原型内部的属性和方法。

当构造函数被实列化后，所有的实例对象都可以访问构造函数的原型成员，如果在原型中声明一个成员，所有的实列方法都可以共享它。

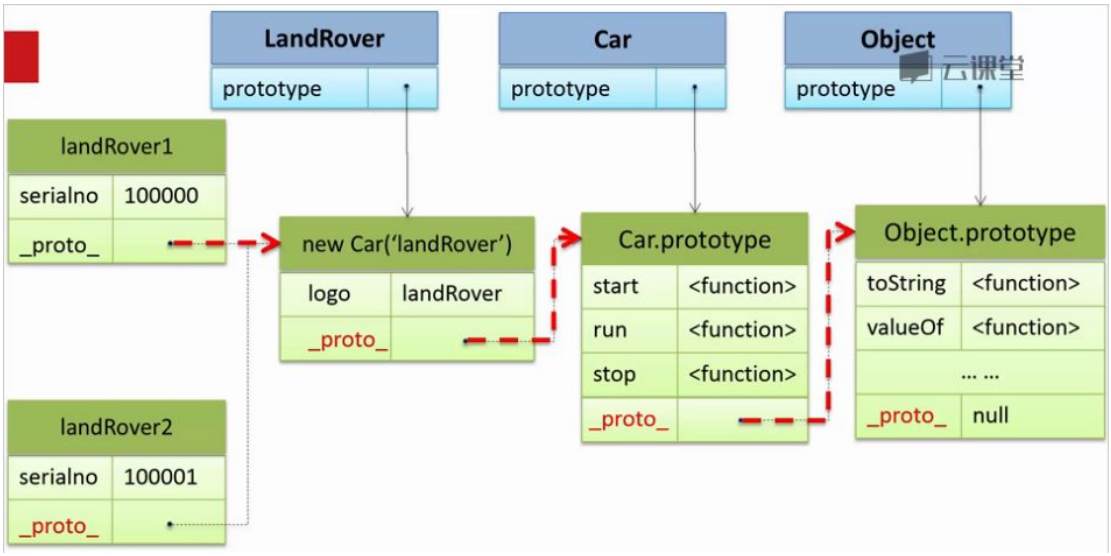
案例：

```
//Car构造函数
function Car(logo) {
    this.logo = logo || 'unknown name';
}
//设置Car的prototype属性
Car.prototype = {
    start: function() {
        console.log('%s start', this.logo);
    },
    run: function() {
        console.log('%s running', this.logo);
    },
    stop: function() {
        console.log('%s stop', this.logo);
    }
}
//创建对象
var landRover = new Car('landRover');
var landWind = new Car('landWind');
//调用方法
landRover.start();
```

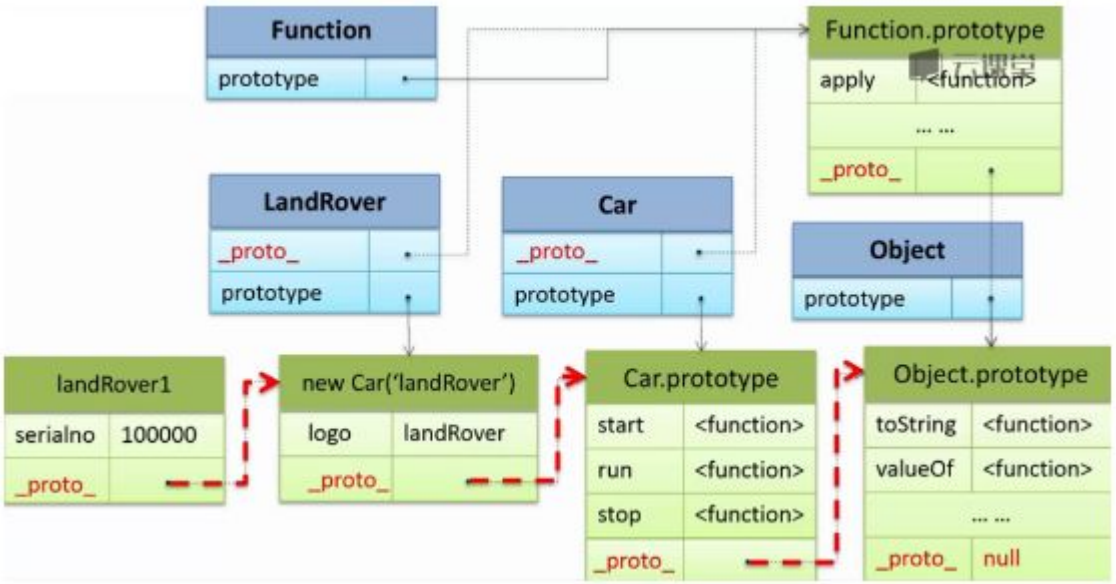


原型链

在原型 prototype 上增加成员属性或者方法的话，它被所有的实列化对象所共享属性和方法，但是如果实列化对象有和原型相同的成员成员名字的话，那么它取到的成员是本实列化对象，如果本实列对象中没有的话，那么它会到原型中去查找该成员，如果原型找到就返回，否则的会返回 undefined



在 javascript 中，一切都是对象，Function 和 Object 都是函数的实列;构造函数的父原型指向于 Function 原型 ,Function.prototype 的父原型指向与 Object 的原型，Object 的父原型也指向与 Function 原型，Object.prototype 是所有原型的顶层;



hasOwnProperty

理解原型查找原理：对象查找先在该构造函数内查找对应的属性，如果该对象没有该属性的话，那么 javascript 会试着从该原型上去查找，如果原型对象中也没有该属性的话，那么它们会从原型中的原型去查找，直到查找的 Object.prototype 也没有该属性的话，那么就会返回 undefined。

因此我们想要仅在该对象内查找的话，为了提高性能，我们可以使用 hasOwnProperty()

来判断该对象内有没有该属性，如果有的话，就执行代码(使用 for-in 循环查找)：如下：

```
var obj = {
    "name":'tugenhua',
    "age":'28'
};// 使用 for-in 循环 for(var i in obj) {
    if(obj.hasOwnProperty(i)) {
        console.log(obj[i]); //tugenhua 28    }
}
```

如上使用 for-in 循环查找对象里面的属性，但是我们需要明白的是：for-in 循环查找对象的属性，它是不保证顺序的，for-in 循环和 for 循环；最本质的区别是：for 循环是有顺序的，for-in 循环遍历对象是无序的，因此我们如果需要对象保证顺序的话，可以把对象转换为数组来，然后再使用 for 循环遍历即可；

四、JS 变量作用域

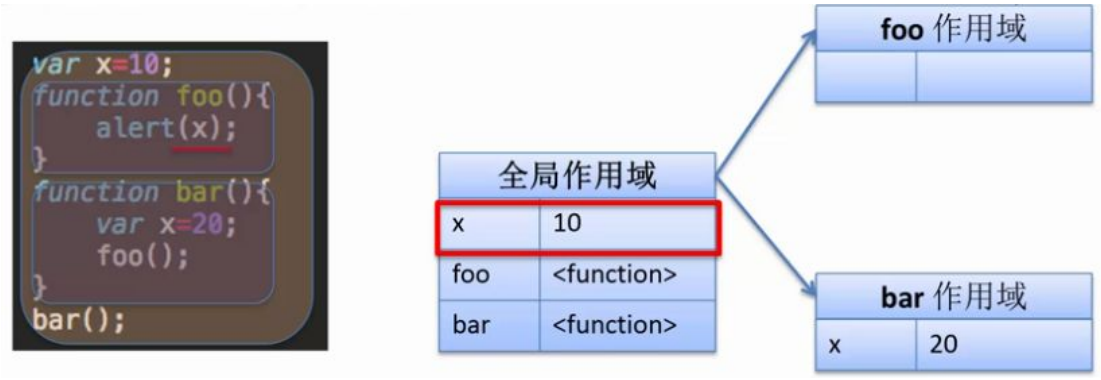
变量作用域解析方式：

1.静态作用域

2.动态作用域

静态作用域

- 又称为词法作用域
- 由程序定义的位置决定



动态作用域

JS 变量作用域

- JS 使用静态作用域
- JS 没有块级作用域，JS 作用域分两种：全局作用域、函数作用域等。
- ES5 中使用词法环境管理静态作用域

词法环境

简单来说，词法环境是描述环境的一个对象。词法环境包括两部分：

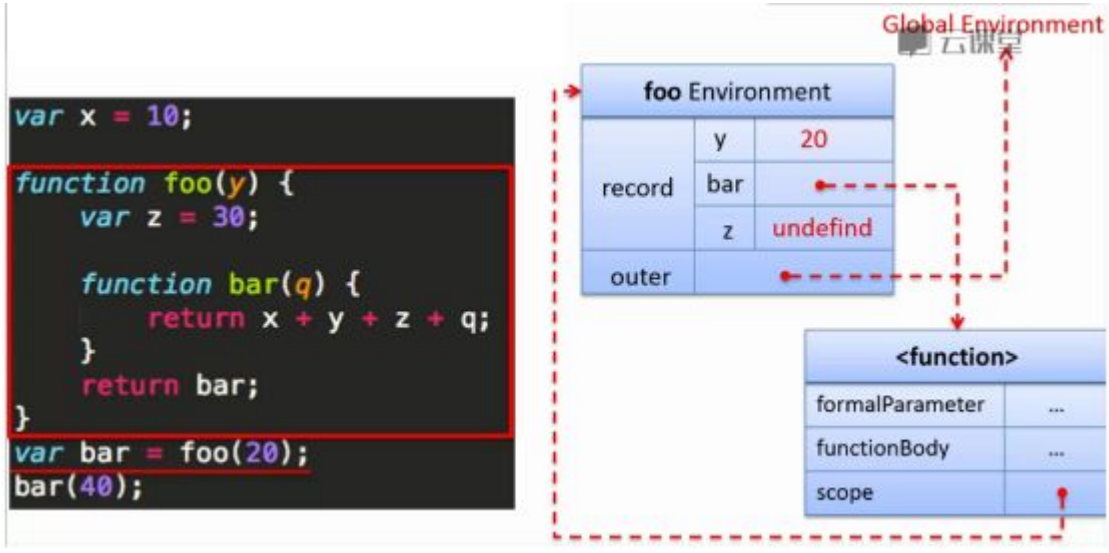
1. 环境记录

- 形参
- 函数声明
- 变量

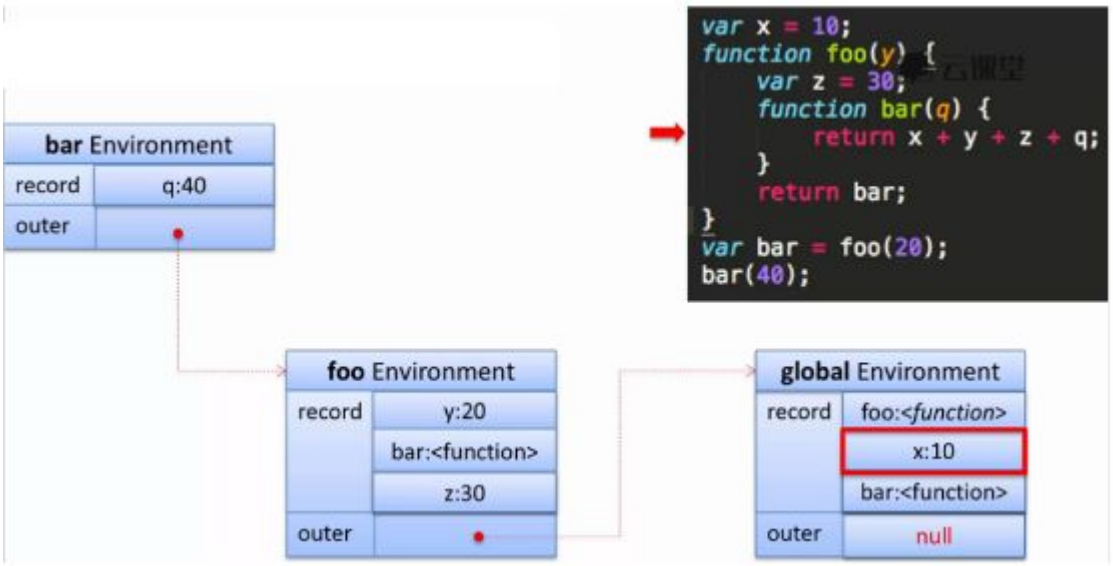
2. 对外部词法环境的引用(outer)

环境记录初始化--声明提前

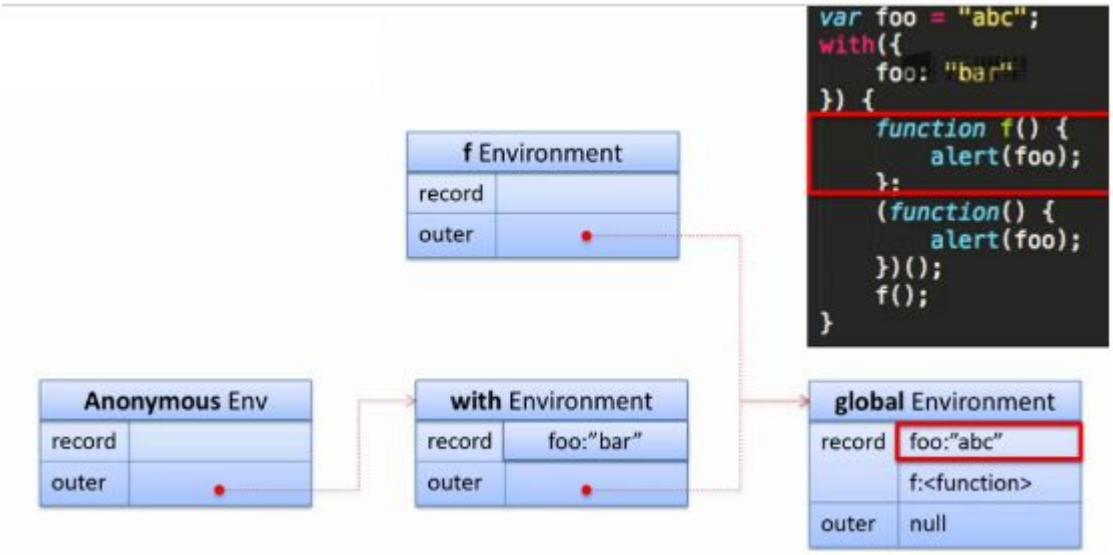
声明提前：指当全局代码或函数代码执行前，它要先扫描函数里面的内容，将函数里面的形参、函数声明、变量先定义到环境记录里面。



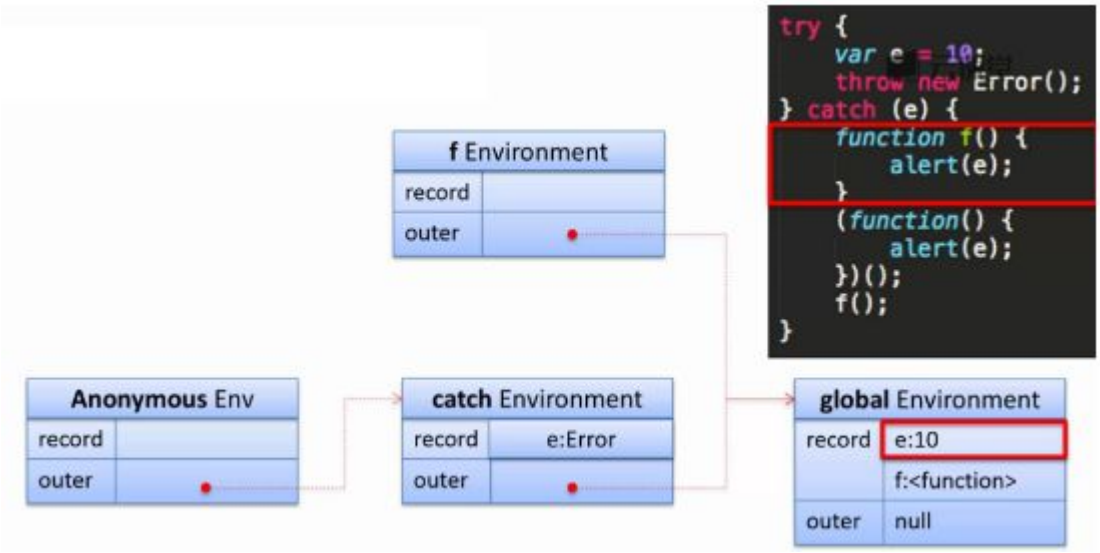
词法环境例子：



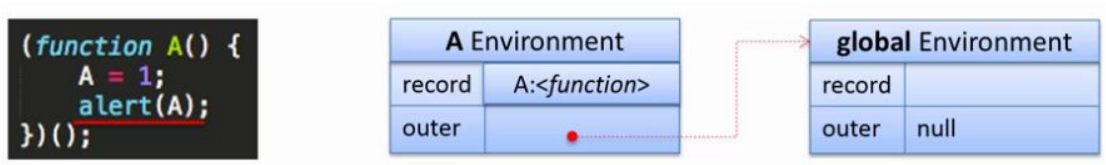
词法环境-with



词法环境-try…catch



JS 作用域-带名称的函数表达式



在变量作用域方面，函数声明和函数表达式有什么区别？

主要区别在于是否提前初始化

函数声明：函数被执行前就已被初始化，和调用顺序没有关系。

函数表达式：只有执行到该定义是，函数才被初始化，和调用顺序有关， 在 with 语句或者 try catch 语句内，作用域可能被改变。

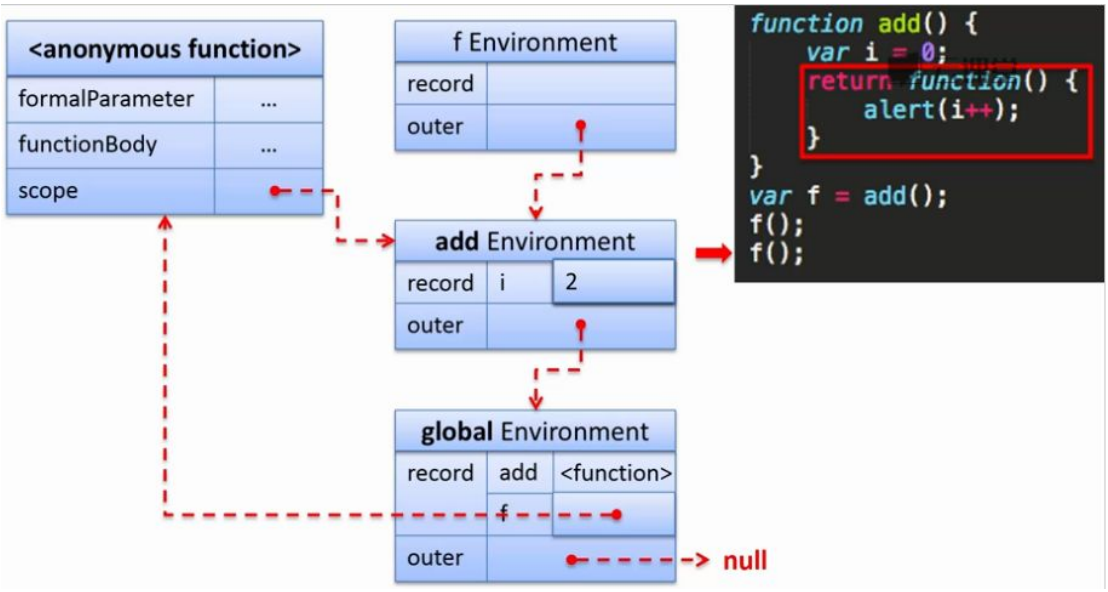
通过函数声明方式创建的函数具有以下特点：

- hoisting 特性，即：函数声明本身会被在其 outer 词法环境中声明提前，也就是说在 outer 词法环境中可以先调用，后声明；
- 在 js 引擎“预编译”时，就将创建，其变量作用域只和函数声明的位置有关，而和运行时的状态无关，也就是说在 with/catch 结构中，其作用域也是外层作用域，而不是 with/catch 临时创建的作用域

通过函数表达式创建的函数具有一下特点：

- 函数表达式在预编译阶段并不会被创建，只有在运行到这个函数表达式所在语句时才会被创建；
- 作用域和运行时状态有关，比如 with, try/catch 的结构中定义的函数表达式，其运行时由于 with/catch 中临时生成了一个词法环境，所以函数表达式产生的函数的词法作用域的 outer 就是指向了 with/catch

五、JS 闭包



- 闭包由函数和与其相关的引用环境的组合而成
- 闭包允许函数访问其引用环境中的变量（又称自由变量）
- 广义上来说，所有 JS 的函数都可以称为闭包，因为 JS 函数在创建时保存了当前的词法环境。

闭包的应用：

1.保存现场

下面这段代码本来需要实现点击 nodes[i]的时候，弹出数字 i，但是最终的效果却是点击每个 nodes[i]都会弹出数字（nodes.length-1）,原因是在为每个 nodes[i]注册点击事件的时候，并没有为相应的函数传入变量 i，导致的结果就是每个 nodes 的点击事件的函数共用了一个 i，而这个 i 就是 for 循环执行完毕时候的那个 i。

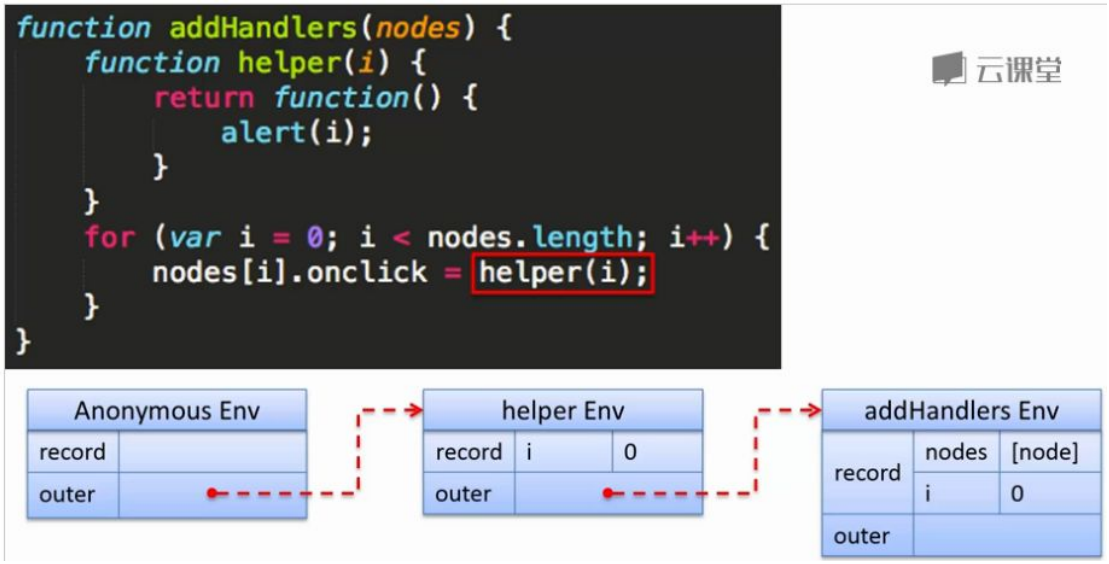
```
function addHandlers(nodes){
    for(var i = 0; i<nodes.length; i++){
        nodes[i].onclick = function(){
            alert(i);
        }
    }
}
```

将上面代码修改为下面代码，应用闭包，在为每个 nodes[i]注册点击事件的时候都会运行函数 helper(i)，这样就会为每个点击事件的函数传入特定的 i，而在 helper(i)函数中则保存了 i，其内部闭包函数运行时需要的 i 就可以从 helper(i)的形参中获取，这样就实现了保存 i 的作用。

```
function addHandlers(nodes){
    function helper(i){
        return function(){
            alert(i);
        }
```

```
    }
    for(var i = 0; i<nodes.length; i++){
        nodes[i].onclick = helper(i);
    }
}
```

上面代码的语法环境如下图所示：



2.模块封装

使用闭包能够将对象的一些属性值隐藏起来，不让外部能够直接访问这些属性，但是也可以给外部确定的接口以供外部访问或者修改，这样能够限制外部的动作。

比如下面的例子，用匿名的立即执行函数创建了一个对象，并将这个对象赋值给 observer，observer 对象虽然只有四个方法，但是这个对象其实是隐藏含有一个数组 observerList 的，这个数组并不能直接用 observer.observerList 的形式访问，因为 observerList 是在匿名函数作用域里面的，外部无法访问，但是 observer 这个对象的 add,empty,getCount,get 四个方法我们却是可以访问的，而这四个函数是定义在匿名函数里面的，形成了四个闭包，这四个闭包可以访问匿名函数里的变量 observerList，所以可以借助 add,empty,getCount,get 四个函数访问或者改变 observerList 的值，这样便限制了外部对 observerList 的访问，让你外部只能读取 observerList 的值和长度，添加数组元素和清空数组。如果外部想要在数组中间插入一个元素，这是办不到的，因为只规定了 4 种功能。所以这可能就是封装的强大之处。

```
var observer = (function(){
    var observerList = [];
    return{
        add: function(obj){
            observerList.push(obj);
        },
        empty: function(){
            observerList = [];
        },
        getCount: function(){
            return observerList.length;
        },
```

```
        get: function(){
            return observerList;
        }
    }
})();
```

六、JS 面向对象编程

全局变量

全局变量就是在任何函数外面声明的或是未声明直接简单使用的。

全局变量的三种定义方法：

1. 使用 var（关键字）+变量名(标识符)的方式在 function 外部声明，即为全局变量，否则在 function 声明的是局部变量。

如：var test = 'some value' //test 为全局变量。

2. 没有使用 var，直接给标识符 test 赋值，这样会隐式的声明了全局变量 test。即使该语句是在一个 function 内，当该 function 被执行后 test 变成了全局变量。

```
（function(){
    var a;
    test = 'some value'; //test  仍为全局变量
}）()
```

3. 使用 window 全局对象来声明，全局对象的属性对应也是全局变量。如下代码：

```
window.test = 'some value';
```

以下代码的 test 也是全局变量：

```
function todo(){
    var a = test = 'some value';
}
```

```
function todo(){
    var a = 'abc',
        b = 'abc';
    test = 'some value';
}
```

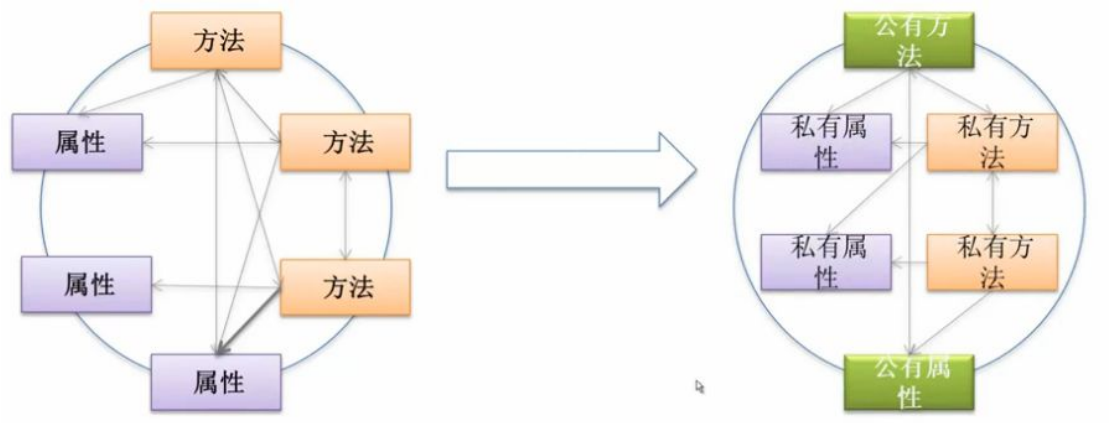
全局变量作用：

在程序的任何地方都可以改写，如下代码的 test:

```
var test = 'aaa';

...

function f1(){
    test = 'bbb';
}
f1();

...

function f2(){
    test = 'ccc';
}
f2();
```

信息隐藏

对于信息隐藏而言，最好的例子是面向对象的封装。



封装问题--信息隐藏

Java 中的封装实例：

```
public class A{
    private int a;
    protected int b;

    private void step1(){
        // TODO
    }

    protected void step2(){
        // TODO
    }

    public void api(){
        // TODO
    }
}
```

- 对象类型—class
- 关键字
  - private
  - protected
  - public

JS 中的封装：

- 使用函数的形式，在函数中使用 this 定义函数的属性及方法
- 使用面向对象的原型方法。

```
function A(){
    this.a = null;
    this.b = null;

    this.step1 = function(){
        // TODO
    }

    this.step2 = function(){
        // TODO
    }

    this.api = function(){
        // TODO
    }
}
```

```
function A() {
    this.a = null;
    this.b = null;
}

var pro = A.prototype;
pro.step1 = function() {
    // TODO
}
pro.step2 = function() {
    // TODO
}
pro.api = function() {
    // TODO
}
```

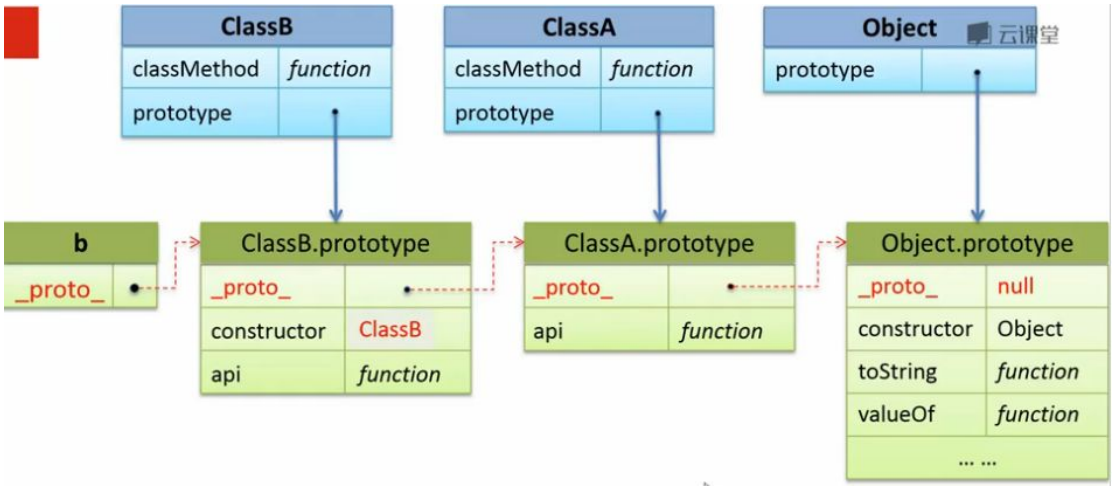
JS 封装形式：

```
function A() {
    var _config = ['A', 'B', 'C'];
    this.getConfig = function() {
        return _config;
    }
}

var pro = A.prototype;
pro._step1 = function() {
    // TODO
}
pro._step2 = function() {
    // TODO
}
pro.api = function() {
    // TODO
}
```

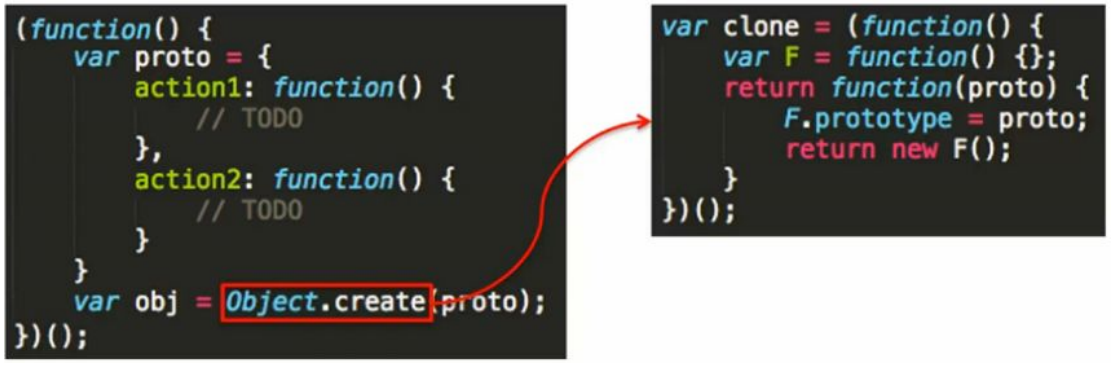
继承

类继承

```
(function() {
    function ClassA() {}
    ClassA.classMethod = function(){}
    ClassA.prototype.api = function(){}

    function ClassB() {
        ClassA.apply(this, arguments);
    }
    ClassB.prototype = new ClassA();
    ClassB.prototype.constructor = ClassB;
    ClassB.prototype.api = function() {
        ClassA.prototype.api.apply(this, arguments);
    }

    var b = new ClassB();
    b.api();
})();
```



原型继承



Object.create 是 ES5 中定义的规范，低版本的浏览器可能不支持，可以用 prototype 属性来模拟 Object.create 方法的功能，如右图。

类继承 VS 原型继承

类继承：是基于类的继承。所有类的实例必须通过类创建， 而类是一个抽象的描述没有具体的实体。 类就像盖房子的图纸，而每个房子都会按照图纸进行搭建。
JS 中的**类式继承（构造函数）**是在函数对象内调用父类的构造函数，使得自身获得父类的方法和属性。call 和 apply 方法为类式继承提供了支持。通过改变 this 的作用环境，使得子类本身具有父类的各种属性。
原型继承，是基于原型的继承。原型本事就是一个实体，是一个可以使用的对象。以原型为模板再去创建实例。就好像盖房子前先建一个房间，其他的房间就按照这个房间结构去创建。原型继承不在对象本身，而在对象的原型上，是将父对像的方法给子类的原型。子类的构造函数中不拥有这些方法和属性。

两者对比：
构造函数继承的方法都会存在父对象之中，每一次实例，都会将 funciton 保存在内存中，这样的做法毫无疑问会带来性能上的问题。
类式继承是不可变的。在运行时，无法修改或者添加新的方法，这种方式是一种固步自封的死方法。而原型继承是可以通过改变原型链接而对子类进行修改的。
类式继承不支持多重继承，而对于原型继承来说，你只需要写好 extend 对对象进行扩展即

可。
JS 通过原型链的方式继承原型上的属性和方法。每个对象都有__proto__属性指向上一级的对象。
PS:
构造函数是一个特殊的对象：在他上面有一个属于函数自身的__proto__指向 Function 和在 prototype 下的__proto__指向 Object。
构造函数创建出来的实例的__proto__指向构造函数的 prototype。 这种链式指向关系就建立起了原型继承。

参考资源：

1. 深入理解 Javascript 面向对象编程

http://www.cnblogs.com/tugenhua0707/p/5068449.html

2. JS 原型继承和类式继承

http://www.cnblogs.com/constantince/p/4754992.html

3.JS 之函数调用：

http://my.oschina.net/u/866703/blog/220461

4.JS 中的全局变量与局部变量

http://blog.csdn.net/zyz511919766/article/details/7276089