1. **分析：Instanceof运算符**的第一个变量是一个对象，暂时称为A；第二个变量一般是一个函数，暂时称为B。

Instanceof的判断队则是：沿着A的proto这条线来找，同时沿着B的prototype这条线来找，如果两条线能找到同一个引用，即同一个对象，那么就返回true。如果找到终点还未重合，则返回false。

---------------------

简单的数据类型:  
1. Null,

2. undefined

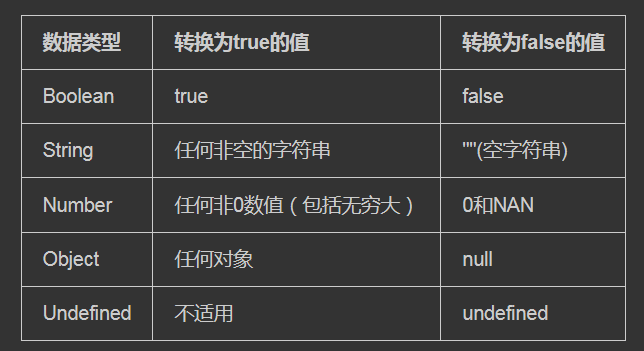
3. number

4. string

5. Boolean

复杂的数据类型 ： Object

将不同类型转换为boolean类型的转换表



Label语句：

主要配合break 和 continue来使用

Var num = 0;

Start:

For(let I = 0; I < 10; I ++){

If(I == 5){

Continue start:

}

Num ++;

}

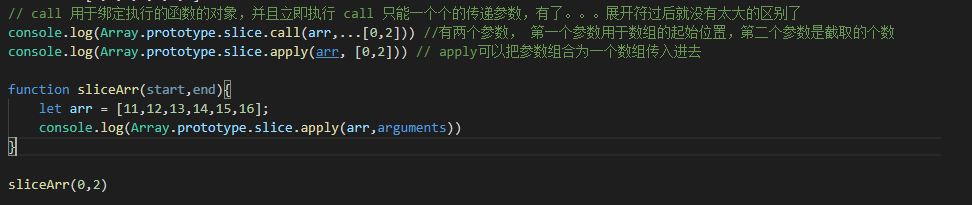
函数的实参对象 arguments，类似于数组的对象，可以arguments[i] 访问当前的参数，arguments.length 访问当前函数的参数个数， 使用 arguments.callee引用当前的函数对象

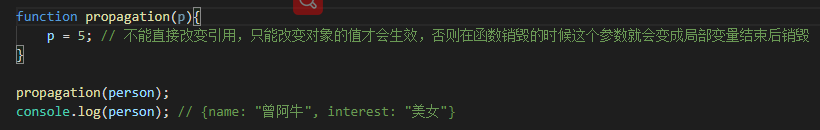


在非严格模式下，可以重写arguments的值并且会同步到出入的参数并修改，在严格模式下是不能使用arguments对参数的值进行修改的

可以通过判断arguments的参数长度来做出不同的操作，完成类似于重载的操作

Call 和apply的区别， 他们都是改变函数执行的对象，并且立即执行，但是，call只能一个个的传入函数需要的参数，但是apply能都将参数整理为一个数组传递进去



ECMAScript的所有函数的参数都是按值传递的，其实和java里面的参数传递差不多

Typeof用于检查是否是基本类型，不能检查应用类型和null

基本类型会返回对应的字符串, number,string,undefined,Boolean

Null 和引用类型都会返回object

每个函数都有自己的执行**环境（就相当于函数中的this）**。当执行流进入一个函数时，函数的环境就会被推入一个环境栈中。  
而在函数执行之后，栈将其环境弹出，把控制权返回给之前的执行环境。 ECMAScript 程序中的执行流  
正是由这个方便的机制控制着

// **说的是函数执行的时候关于使用到的变量，是有一条作用域链的，作用域链的最后一个对象是window， 第一个对象是arguments对象中包含的该变量，如果没有就下上去找直到找到window还没有找到那么，就没有该变量**

代码在一个环境中执行时，会创建变量对象的一个作用域链（scope chain）。作用域链的用途，是  
保证对执行环境有权访问的所有变量和函数的有序访问。作用域链的前端，始终都是当前执行的代码所  
在环境的变量对象。如果这个环境是函数，则将其活动对象（activation object）作为变量对象。活动对  
象在最开始时只包含一个变量，即 arguments 对象（这个对象在全局环境中是不存在的）。作用域链中  
的下一个变量对象来自包含（外部）环境，而再下一个变量对象则来自下一个包含环境。这样，一直延  
续到全局执行环境；全局执行环境的变量对象始终都是作用域链中的最后一个对象。

使用var声明的函数变量，会自动添加到最近的环境中，在函数内部，最接近的环境就是函数的局部环境

****

'use strict' // 严格模式开启的方式

说明:

1.在全局作用域使用的话,那整个js脚本就会开启这种模式

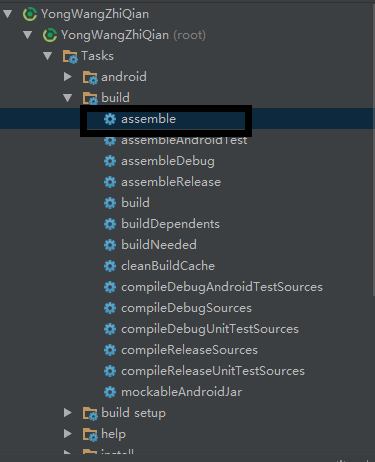
2.如果是只在函数内部使用的话,那么就只是该函数内部开启而已

[?](https://www.jb51.net/article/141419.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | function doSomething(){ |

IE 中，调用 **window.CollectGarbage()**方法会立即执行垃圾收集。在 Opera 7 及更  
高版本中，调用 **window.opera.collect()**也会启动垃圾收集例程。

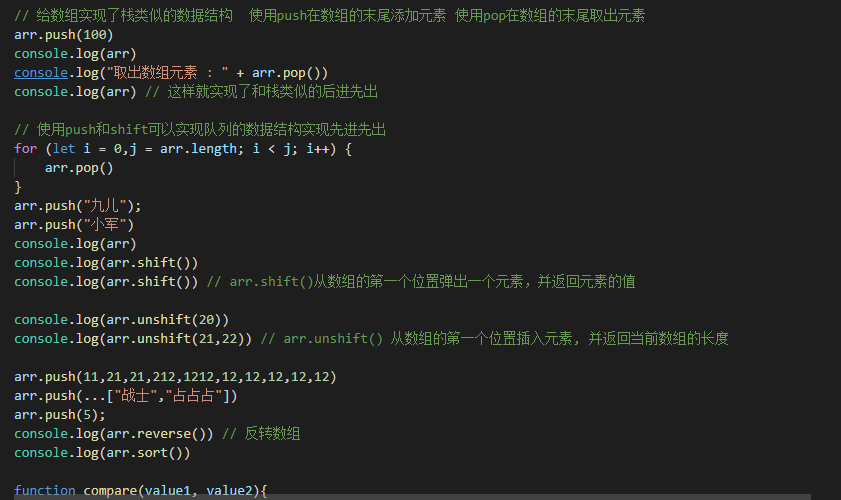
**打包未签名的apk**

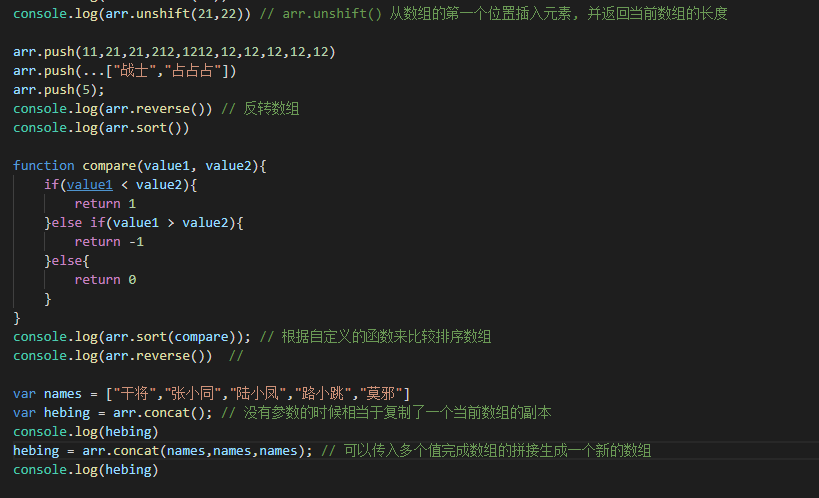
****

**属性中可以包含非字母和非数字的可以包含空格，但是包含空格的属性只能使用【】 来访问 不能使用 . 访问**

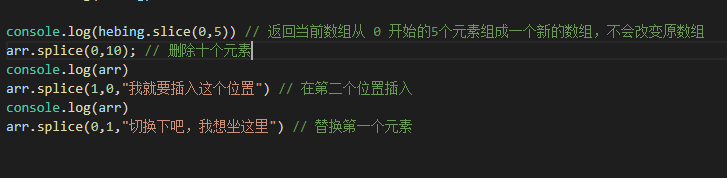
**数组的相关API**

**Js中的数组有相关的api可以实现栈，队列等数据类型**

****

****

**Splice方法的强大， 可以删除任意位置的数组， 可以替换数组中的值，随意位置插入元素**

****

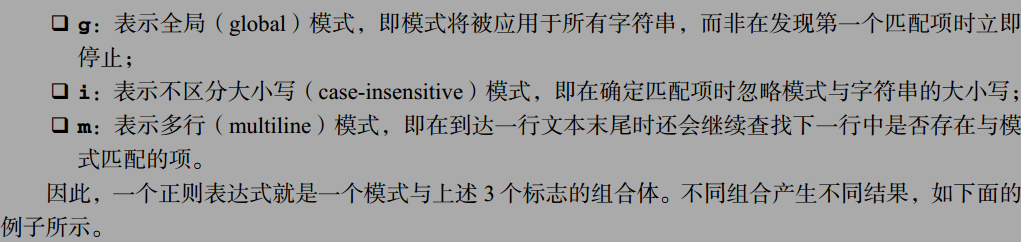
**indexOf(), lastIndexOf()**

****

var num = 10;  
**alert(num.toFixed(2)); //"10.00" 基本的数字类型可以使用toFixed() // 来返回对应的数字小数位的字符串**

****

**正则表达式:**

****

**通过使用forin来实现对象的属性循环获取,**

**Obj.hasOwnProperty(name); 判断name是不是obj的属性，如果是就获取属性的值**

****

## 属性类型

1. 数据属性

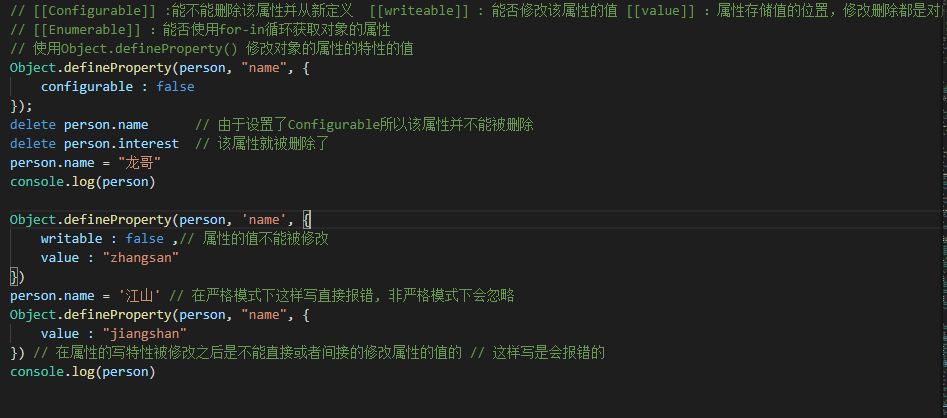
数据属性： 包含一个数据值的位置，在这个位置可以读取和写入值，数据属性有四个描述其行为的特性:

1.[[configurable]] : 表示能否通过delete删除属性从而重新定义属性，能否修改属性的特性或者能否把属性改为访问器属性， 这个特性默认是true configurable

2.[[Enumerable]] : 能否通过forin返回属性， 默认是true

3.[[Writable]] : 能否修改属性的值 默认是true

4.[[Value]] : 包含了这个属性的数据值， 读取属性值的时候从这个地方读取，写入属性值的时候从这个地方写入， 默认值是undefined





## 访问器属性

访问器属性： 不包括属性的值， 是包含一对getter和setter函数（不一定都存在），读取访问器属性的时候会调用getter函数， 在写入访问器属性的时候会调用setter函数并传入新值，这个函数负责决定如何处理数据

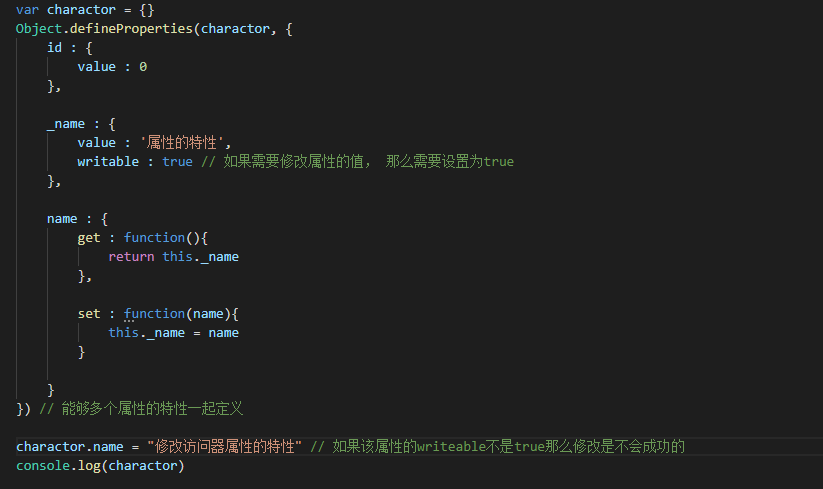
访问器的四个特性：

1. [[Configurable]] : 这个特性负责能否删除访问器属性，能否修改为数据属性，能否修改属性的特性 默认true
2. [[Enumerable]] : 这个特性负责能不能被for-in循环返回属性 默认true
3. [[Get]] : 读取属性时调用该函数， 默认undefined
4. [[Set]] : 修改属性时 调用函数， 默认值 为 undefined

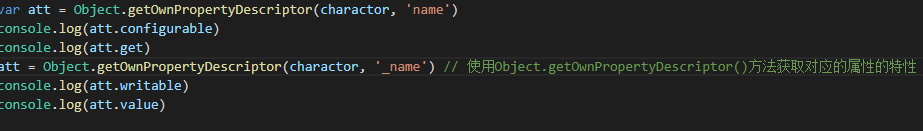
Note : 访问器的属性不能直接定义，必须使用Object.defineProperty()来定义



使用Object.defineProperties() 方法来给对象的多个属性设置对应的数据属性特性，和访问器属性特性



Object.getOwnPropertyDesctiptor() 方法获取属性的特性的值



## Global对象: 全局对象是ECMAScript中最特别的一个对象因为不管你从什么角度看这个对象都是不存在的，不属于其他任何对象的属性和方法都是它的属性和方法诸如 isNaN， parseInt， isFinite， parseFloat等

1. URI 编码方法：
2. Eval方法： ECMAScript中最强大的方法： eval() 像是一个完整的ECMAScript解析器，它只接受一个字符串参数
3. 当解析器发现代码中调用 eval()方法时，它会将传入的参数当作实际的 ECMAScript 语句来解析，  
   然后把执行结果插入到原位置。通过 eval()执行的代码被认为是包含该次调用的执行环境的一部分，  
   因此被执行的代码具有与该执行环境相同的作用域链。
4. 严格模式下，在外部访问不到 eval()中创建的任何变量或函数，因此前面两个例子都会导致错误。  
   同样，在严格模式下，为 eval 赋值也会导致错误（严格模式下少用） // 也要防止代码注入攻击



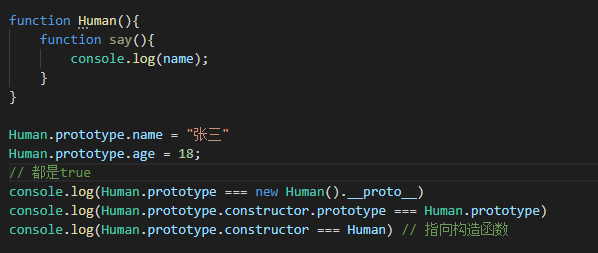


## 创建对象

使用new来创建对象要经历四个步骤

1. 创建一个新的对象
2. 将构造函数的作用域赋值给新的对象（这里的this指向了这个新对象）
3. 执行构造函数的代码（给这个新对象添加属性）
4. 返回新对象

**原型模式 ：每一个我们创建的函数都有一个prototype（原型） 属性，这个属性是一个指针，指向一个对象而这个对象的哟用途是包含可以由特定类型的所有实例共享的方法和属性**

****

**调用构造函数创建一个实例后，那么该实例内部包含一个指针（内部属性） 指向构造函数的原型对象 [[prototype]] 没有标准的访问方式， 在大多的浏览器上使用实例的\_\_proto\_\_ 属性访问构造函数的原型对象**

Person 构造函数、 Person 的原型属性以及 Person 现有的两个实例之间的关系。  
在此，Person.prototype 指向了原型对象，而 Person.prototype.constructor 又指回了 Person。  
原型对象中除了包含 constructor 属性之外，还包括后来添加的其他属性。

测试是否是虽然在所有实现中都无法访问到[[Prototype]]，但可以通过 isPrototypeOf()方法来确定对象之  
间是否存在这种关系。从本质上讲，如果[[Prototype]]指向调用 isPrototypeOf()方法的对象  
（Person.prototype），那么这个方法就返回 true，

**Object.getPrototypeOf()返回的对象实际就是这个对象的原型。**

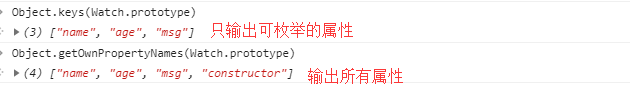
**当代码读取某个对象的某个属性的时候，会执行一次搜索，目标是具有给定名字的属性，在搜索的过程中，首先是从实例的属性上面查找是否存在该名字的属性，如果没有指针就会指向原型对象从原型对象中去查找**

**获取原型对象的属性特征**

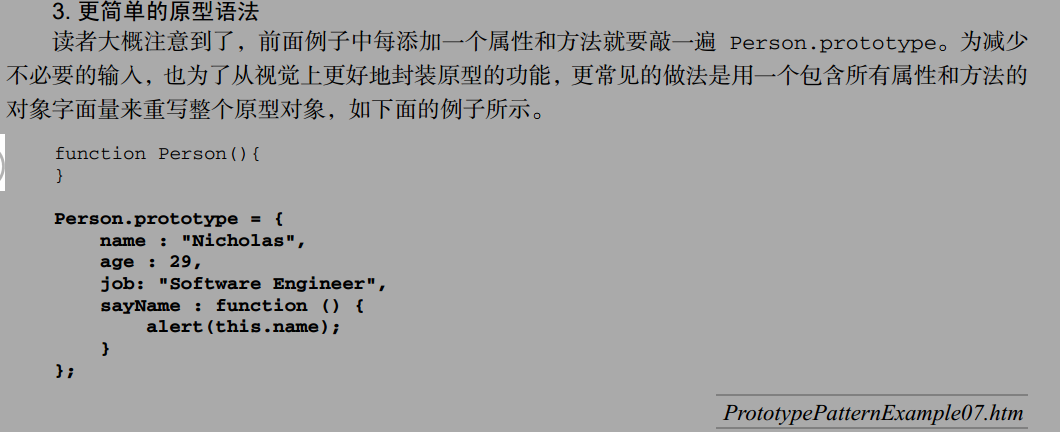
****

由于 in 操作符只要通过对象能够访问到属性就返回 true， hasOwnProperty()只在属性存在于  
实例中时才返回 true，因此只要 in 操作符返回 true 而 hasOwnProperty()返回 false，就可以确  
定属性是原型中的属性。

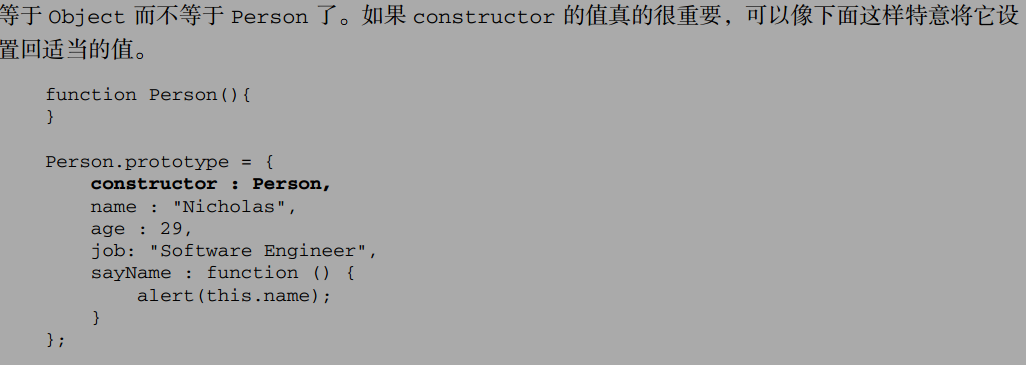
要取得对象上所有可枚举的实例属性，可以使用 ECMAScript 5 的 Object.keys()方法。这个方法

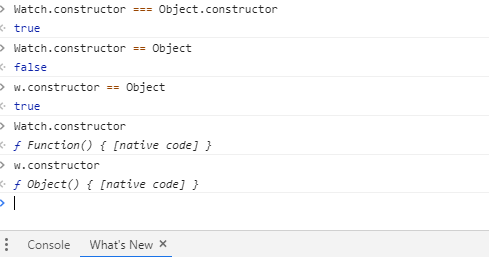
  
接收一个对象作为参数，返回一个包含所有可枚举属性的字符串数组。

如果你想要得到所有实例属性，无论它是否可枚举，都可以使用 Object.getOwnPropertyNames()  
方法。

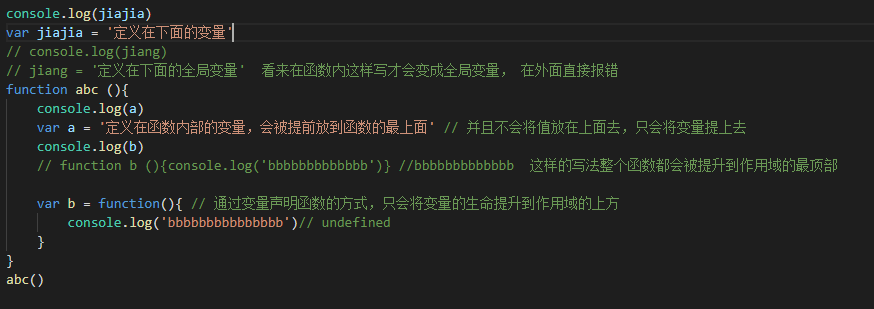
****

在上面的代码中，我们将 Person.prototype 设置为等于一个以对象字面量形式创建的新对象。  
最终结果相同，但有一个例外： constructor 属性不再指向 Person 了。前面曾经介绍过，每创建一  
个函数，**就会同时创建它的 prototype 对象，这个对象也会自动获得 constructor 属性**。而我们在  
这里使用的语法，本质上完全重写了默认的 prototype 对象，因此 constructor 属性也就变成了新  
**对象的 constructor 属性（指向 Object 构造函数），不再指向 Person 函数。**此时，尽管 instanceof  
操作符还能返回正确的结果，但通过 constructor 已经无法确定对象的类

****

****

**关于声明变量的js编译器，会提前， 根据不同的作用域提前不同的方式， 在全局变量和函数作用域的变量中声明的变量会被提前，但是赋值部分还是在代码书写的位置，函数的声明时如果没有使用变量式的声明那么整个函数都会提升到作用域的最顶部**

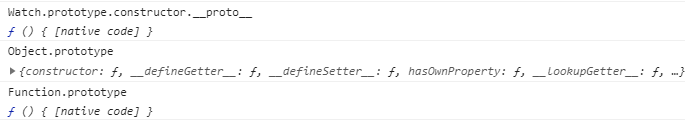
****

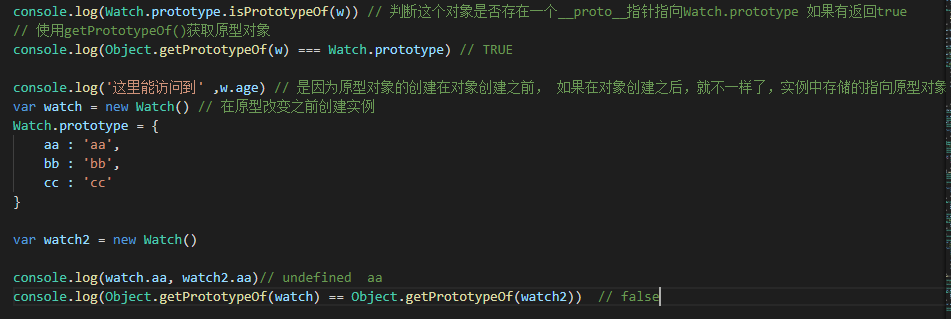
**创建对象的方式**

1. **工厂模式**
2. **构造函数**
3. **原型模式**

**从实例属性中查找的时候 ：在属性名字相同的情况下， 实例属性会屏蔽掉原型属性的值， 因为在访问属性的时候就是最先从实例属性中去查找，找到了就返回，找不到才会继续在原型中查找**

**从原型属性中查找的时候： 使用Watch.prototype只会从原型中查找，如果没有就返回undefined**

****

****

1. **组合使用（构造函数 + 原型模式） 完成实例的独立副本，共享变量**
2. **寄生构造函数**

**Function Person(){**

**Var o = new Object();**

**o.name = arguments[0];**

**o.age = arguments[1];**

**return o ; 这种方式可以创建特殊的对象，并且创建的对象不能使用instanceof来判断对象的类型**

**}**

1. **稳妥的构造函数**

**Function Person(name,age,address){**

**Var o = new Object();**

**o.getName(){**

**return name;**

**}**

**return o;**

**}**

**这样防止了直接点name就能访问属性的值**

**实现继承的方式 ：**

1. **原型链:**

在创建子类型的实例时，不能向超类型的构造函数中传递参数。实际上，  
应该说是没有办法在不影响所有对象实例的情况下，给超类型的构造函数传递参数。

所以原型链完成的继承在修改父类的属性值的时候会在所有子类的实例中反应出来

1. **借用构造函数的方式**

**Function Person(){}**

**Function Zhangsan(){**

**Person.call(this) // 使用现有的对象调用，Person函数完成继承**

**}**

如果仅仅是借用构造函数，那么也将无法避免构造函数模式存在的问题——方法都在构造函数中定  
义，因此函数复用就无从谈起了。而且，在超类型的原型中定义的方法，对子类型而言也是不可见的，结  
果所有类型都只能使用构造函数模式。考虑到这些问题，借用构造函数的技术也是很少单独使用的。

1. **组合式继承（原型链 + 借用构造函数）**

组合继承避免了原型链和借用构造函数的缺陷，融合了它们的优点，成为 JavaScript 中最常用的继  
承模式。而且， instanceof 和 isPrototypeOf()也能够用于识别基于组合继承创建的对象。

****

1. **原型式继承**

**Function obj(o){**

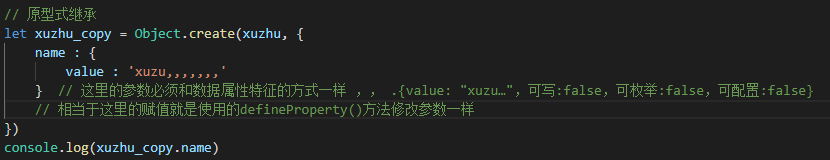
**Var F = {}**

**F.prototype = o**

**Return o**

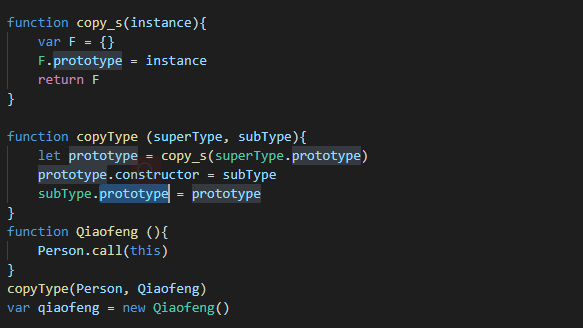
**} 实现传入对对象的浅复制 和 Object.create方法有相似性**

**Object.create() 两个参数， 第一个参数是新对象原型的对象， 第二个参数是需要增加或者修改的属性使用 {}封装所有的属性即可**

****

1. **寄生组合式继承**

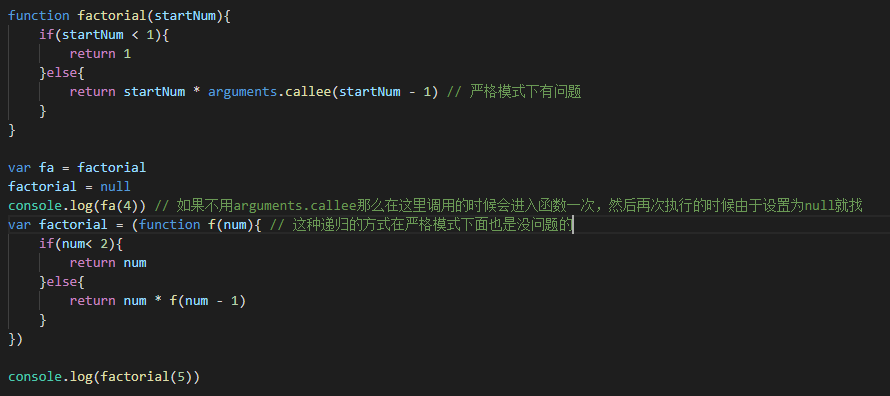
**但是不能直接访问到父类的原型方法**

****

## 函数

关于函数声明，它的一个重要特征就是**函数声明提升（function declaration hoisting）**，意思是在执行  
代码之前会先读取函数声明。这就意味着可以把函数声明放在调用它的语句后面。

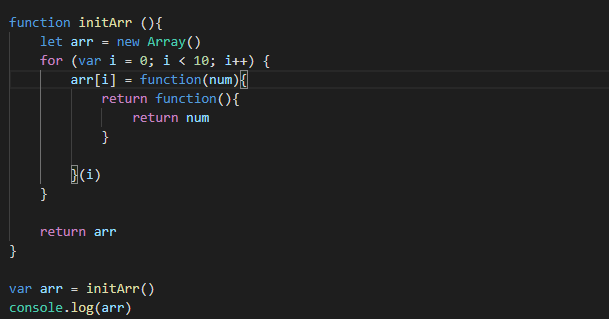
1. 递归

****

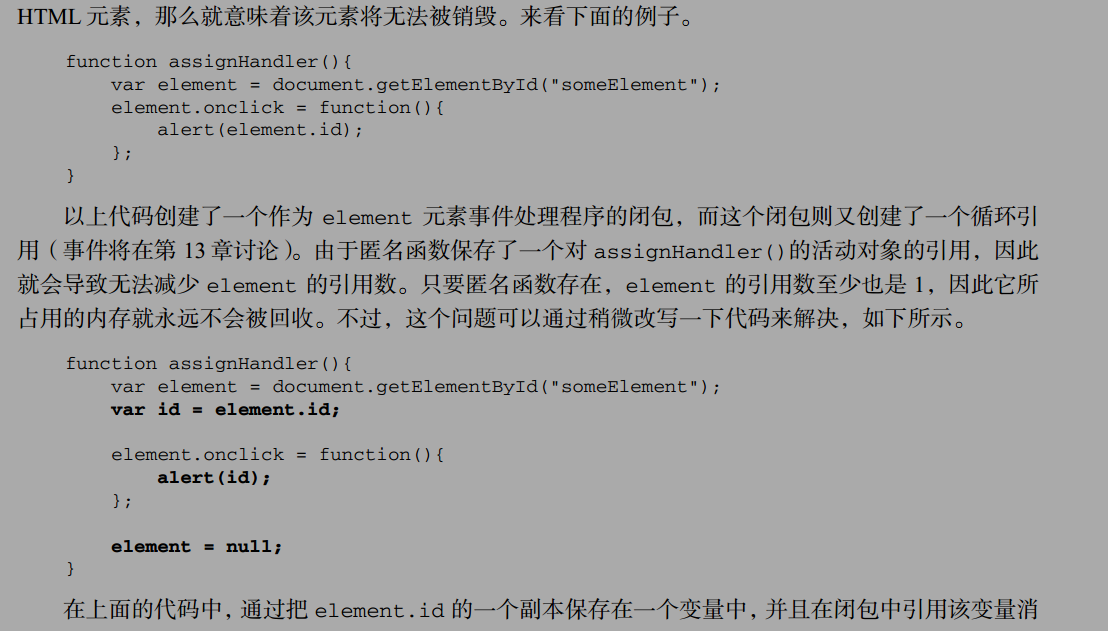
1. **闭包 ：**闭包是指有权访问另一个  
   函数作用域中的变量的函数。

无论什么时候在函数中访问一个变量时，就会从作用域链中搜索具有相应名字的变量。一般来讲，  
当函数执行完毕后，局部活动对象就会被销毁，内存中仅保存全局作用域（全局执行环境的变量对象）。  
但是，闭包的情况又有所不同。 **（闭包中存在外部函数的活动对象，当外部函数执行完毕后，它的执行环境的作用域链会被销毁，但是它的活动对象仍然会保存在内存中知道匿名函数（内部函数）被销毁，活动对象才会被销毁）**

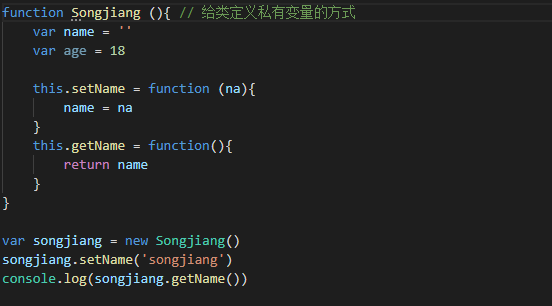
在调用每个匿名函数时，我  
们传入了变量 i。由于函数参数是按值传递的，所以就会将变量 i 的当前值复制给参数 num。而在这个  
匿名函数内部，又创建并返回了一个访问 num 的闭包。这样一来， result 数组中的每个函数都有自己  
num 变量的一个副本，因此就可以返回各自不同的数值了。

****

**. 每个函数在被调用时都会自动取得两个特殊变量： this 和 arguments。内部函  
数在搜索这两个变量时，只会搜索到其活动对象为止，因此永远不可能直接访问外部函数中的这两个变  
量（这一点通过图 7-2 可以看得更清楚）。不过，把外部作用域中的 this 对象保存在一个闭包能够访问  
到的变量里，就可以让闭包访问该对象了，**



1. **私有变量**

****

### Js window对象

1. Window.open(arg0, arg1,arg2,arg3); // 打开一个窗口

Arg0 : 打开的窗口的地址

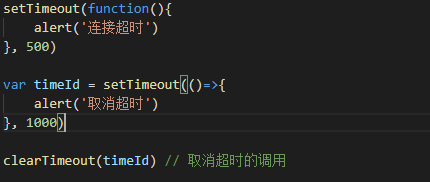
Arg1 ： 会在有该名字的框架或者window对象上打开对应的窗口地址

Arg2 : 一个特性的字符串

Arg3： 是否打开新窗口

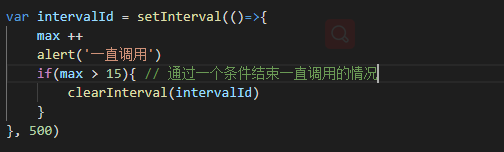
如果是浏览器扩展或其他程序阻止的弹出窗口，那么 window.open()通常会抛出一个错误。因此，  
要想准确地检测出弹出窗口是否被屏蔽，必须在检测返回值的同时，将对 window.open()的调用封装  
在一个 try-catch 块中，如下所示。  
**var blocked = false;  
try {**var wroxWin = window.open("http://www.wrox.com", "\_blank");  
if (wroxWin == null){  
**blocked = true;**}  
**} catch (ex){  
blocked = true;  
}  
if (blocked){  
alert("The popup was blocked!");  
}***PopupBlockerExample01.*

1. setTimeout()

虽然这两种调用方式都没有问题，但由于传递字符串可能导致性能损失，因此不建议以字符串作为  
第一个参数。  
第二个参数是一个表示等待多长时间的毫秒数，但经过该时间后指定的代码不一定会执行。  
JavaScript 是一个单线程序的解释器，因此一定时间内只能执行一段代码。为了控制要执行的代码，就  
有一个 JavaScript 任务队列。这些任务会按照将它们添加到队列的顺序执行。 setTimeout()的第二个  
参数告诉 JavaScript 再过多长时间把当前任务添加到队列中。如果队列是空的，那么添加的代码会立即  
执行；如果队列不是空的，那么它就要等前面的代码执行完了以后再执行。  
调用 setTimeout()之后，该方法会返回一个数值 ID，表示超时调用。这个超时调用 ID 是计划执  
行代码的唯一标识符，可以通过它来取消超时调用。要取消尚未执行的超时调用计划，可以调用  
clearTimeout()方法并将相应的超时调用 ID 作为参数传递给它，如下所示。

Note : 超时调用的代码都是在全局作用域中执行的，因此函数中 this 的值在非严格模  
式下指向 window 对象，在严格模式下是 undefined。

1. **setInterval :**

****

**如果没有执行clearInterval那么就会一直执行到页面结束**

**Location 对象**

****

**查看地址信息**

**Navigator查看插件信息，以及浏览器的相关硬件信息**

**History 历史记录 不能得到用户访问过的url 但是可以通过 go**

**Back方法来前进后退页面， 判断当前页面是不是第一个页面**

**Top对象表示的是浏览器最外层的框架对象 也就是浏览器的整个窗口**

**Parent代表当前框架的window对象**

# 事件

1. 事件捕获阶段 （Netscape Communicator 团队提出） ， 到达目标阶段， 冒泡阶段(ie提出)

事件捕获阶段 是从document 开始向目标进发的过程

冒泡阶段是 从目标向document进发的过程

1. Html事件处理 （由于可能处理的js代码还没有加载成功， html事件就被触发了， 所以大多数的程序员是摒弃了html事件处理的） ，关于这个动态创建的函数，另一个有意思的地方是它扩展作用域的方式。在这个函数内部，可以像  
   访问局部变量一样访问 document 及该元素本身的成员。这个函数使用 with 像下面这样扩展作用域：  
   function(){  
   with(document){  
   with(this){  
   //元素属性值  
   }  
   }  
   }  
   如此一来，事件处理程序要访问自己的属性就简单多了。
2. DOM0级事件处理：每个元素（包括 window 和 document）都有自己的事件处理程序属性，这些属性通常全部小写使用 DOM0 级方法指定的事件处理程序被认为是元素的方法。因此，这时候的事件处理程序是在  
   元素的作用域中运行；换句话说，程序中的 this 引用当前元素。 【在事件的冒泡阶段被处理】
3. DOM2级事件处理程序： 定义了两个方法 用于添加和删除事件addEventListener()和 removeEventListener() 所有的dom节点都包含了这两个方法它们都接受 3 个参数：要处理的事件名、作为事件处理程序的函数和一个布尔值。最后这个布尔值参数如果是 true，表示在捕获阶段调用事件处理程序；如果是 false，表示在冒泡阶段调用事件处理程序。
4. 事件处理上ie有自己的添加删除事件的事件处理程序， 在跨浏览器使用的时候可以根据具体的情况判断使用， 默认使用DOM0级事件来处理事件

var EventUtil = {  
addHandler: function(element, type, handler){  
if (element.addEventListener){  
element.addEventListener(type, handler, false);  
} else if (element.attachEvent){  
element.attachEvent("on" + type, handler);  
} else {  
element["on" + type] = handler;  
}  
},  
removeHandler: function(element, type, handler){  
if (element.removeEventListener){  
element.removeEventListener(type, handler, false);  
} else if (element.detachEvent){  
element.detachEvent("on" + type, handler);  
} else {  
element["on" + type] = null;  
}  
}  
};  
*EventUtil.j*

# 函数柯里化

Function add(a,b){

Return a + b

}

Function curry(fn){

Var args = Array.prototype.slice.call(arguments, 1)

Retrun function(){

Var arr = Array.prototype.slice.call(arguments).concat(args)

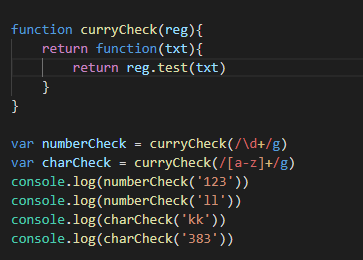
Return fn.apply(null, arr)

}

}

Console.log(curry(add,2,3)(10)) 函数柯里化 将一个参数的函数变成能够完成两个参数以上的操作， 传入函数和参数 返回一个函数来处理多余出来的参数

函数柯里化的作用l;

1. 实现函数的复用， 可以将返回的函数进行复用， 在函数使用对象不同的情况下，完成复用

# Js模块化

**Js模块的规划 ： 主要分为 commomJS 和 AMD**

1. **AMD是异步加载模块的定义， 主要用于浏览器模块化开发， CommonJS是用于服务端开发**
2. **AMD 主要使用define定义模块 使用 require 导入模块**
3. **CommonJs 是Node.js主要使用的模块化定义的规范， 使用module.exports定义一个模块 ， 使用require导入模块**
4. **关于模块化 还可以使用ES6 的export ， export default 定义模块， 使用 import来导入模块**

**CMD 通用模块的定义 ， 使用define 就近使用加载的模块**require是可以把其他模块导入进来的一个参数，而export是可以把模块内的一些属性和方法导出的。  
CMD规范地址：[https://github.com/seajs/seaj...](https://github.com/seajs/seajs/issues/242" \t "_blank)

AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。

CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。

对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。

AMD:提前执行（异步加载：依赖先执行）+延迟执行

CMD:延迟执行（运行到需加载，根据顺序执行）

CMD 推崇依赖就近，AMD 推崇依赖前置。看如下代码：

// CMD

define(function(require, exports, module) {

var a = require('./a')

a.doSomething()

// 此处略去 100 行

var b = require('./b') // 依赖可以就近书写

b.doSomething()

// ...

})

// AMD 默认推荐的是

define(['./a', './b'], function(a, b) { // 依赖必须一开始就写好

a.doSomething()

// 此处略去 100 行

b.doSomething()

...

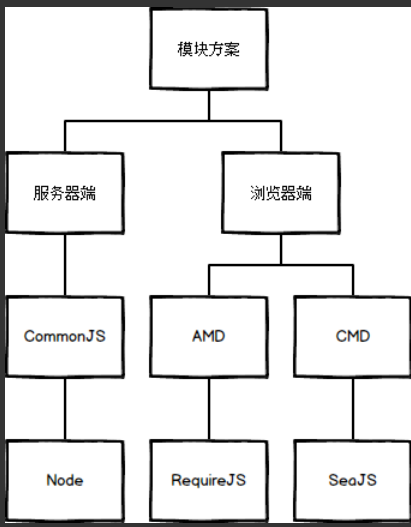
})

另外一个区别是：

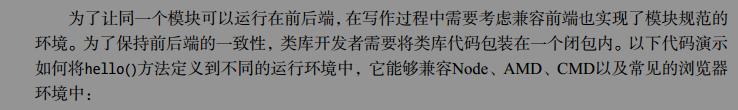
AMD:API根据使用范围有区别，但使用同一个api接口

CMD:每个API的职责单一

AMD的优点是：异步并行加载，在AMD的规范下，同时异步加载是不会产生错误的。  
CMD的机制则不同，这种加载方式会产生错误，如果能规范化模块内容形式，也可以

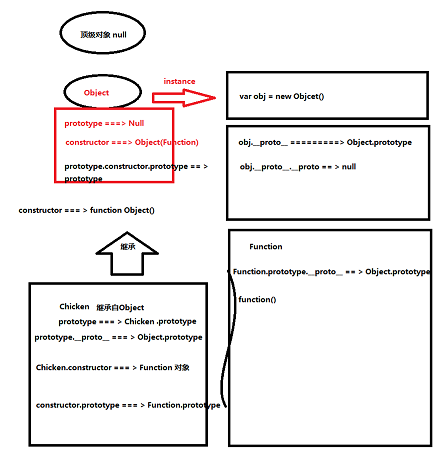
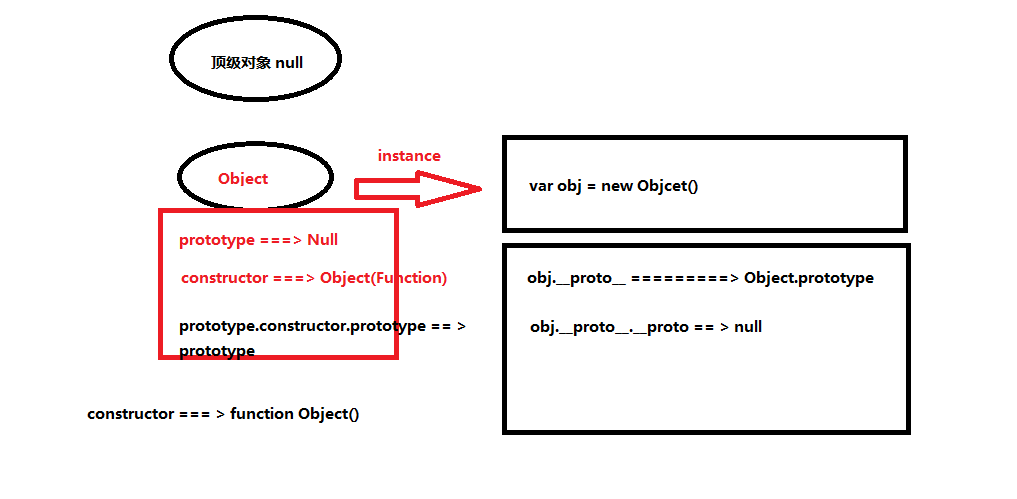
****

**关于模块兼容的问题**

;(function (name, definition) {  
// 检测上ူ文环境是否为AMDई CMD  
var hasDefine = typeof define === 'function',  
// 检查上ူ文环境是否为Node  
hasExports = typeof module !== 'undefined' && module.exports;  
if (hasDefine) {  
// AMD环境ई CMD环境  
define(definition);  
} else if (hasExports) {  
// 定义为೵通Node模块  
module.exports = definition();  
} else {  
// 将模块的执行结ࠬࡕ在windowՎ量中ǈ在៓બ器中thisኸၠ window对象  
this[name] = definition();  
}  
})('hello', function () {  
var hello = function () {};  
return hello;  
});

**RequestAnimFrame 动画绘制的回调函数 优化并行动画 ， 将其放在一个渲染周期内**

* 浏览器可以优化并行的动画动作，更合理的重新排列动作序列，并把能够合并的动作放在一个渲染周期内完成，从而呈现出更流畅的动画效果
* 一旦页面不处于浏览器的当前标签，就会自动停止刷新。这就节省了CPU、GPU和电力
* 解决浏览器不知道JavaScript动画在什么时候开始 动画之间的时间间隔的问题， 它是跟着浏览器的绘制走的， 浏览器的绘制时间间隔 就是他的时间间隔

****

## 闭包

概念: 闭包能够读取其他函数的局部变量的函数， 函数没有被释放整条作用域链上的变量都将被保存， 由于JavaScript的特性， 只有子函数具有这种特性， 所以闭包就是函数内部的一个函数

**节流**