检测类型：typeof xxx 输出类型（小写字母打头） xxx instanceof Array 输出布尔值

js没有块级作用域：函数的if、for语句中定义的变量，可以在整个函数作用域中使用。

fun.bind(thisArg[, arg1[, arg2[, ...]]])

[,表示可选，即函数可以接收1~无穷多个参数

**JS基础概念部分**

组成：ES 、DOM（W3C标准，所有浏览器共同遵守）、BOM（各个浏览器根据BOM在各自浏览器上实现）

DOM：HTML和XML的应用程序接口（API）

DOM节点：元素节点（标签如<p>等）、文本节点（内容）、属性节点（元素属性，如a的href属性等）

BOM：处理浏览器窗口和框架（包含浏览器特定的JS扩展如弹出新窗口等）

访问网页的过程：浏览器访问BOM对象（window），通过BOM访问DOM对象。

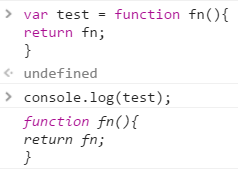
注：window包含document，后者为DOM的根节点；DOM最根本的对象是BOM的window对象的子对象；DOM描述了处理网页内容的方法和接口，BOM描述了与浏览器进行交互的方法和接口。

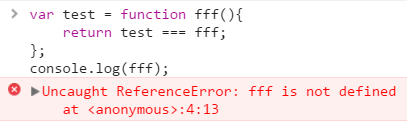
**函数部分：**

**定义：**

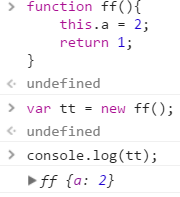
1. 函数声明 function xxx(){} 附，函数name属性为function*关键字*后面的*标识符*xxx。
2. 函数定义表达式 var xxx = function(){}（匿名函数） var xxx = function xx(){}一般不需要名称，匿名函数即可。适合只使用一次的函数。

几段代码：

test\test()\test()()都是输出函数fn，因为总是return fn。

对于该函数，有变量名称test，有函数名称fff，区别是，*函数名只能在函数内部使用*。另，console.log(test())输出true，因为test一直都在。

1. Function构造函数（略）

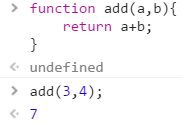
关于new操作符，得到的是对象。return值不是对象时，返回该对象；否则返回返回的对象值。对于构造函数（通常函数名大写），不传参的时候，new ff或者new ff()都可以。

this定义的a，只能通过实例tt访问，不能ff.a（undefined）。

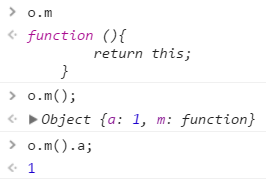
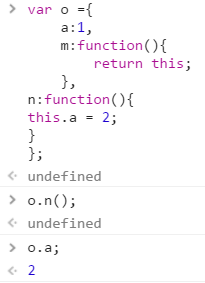
new作用:创建一个空对象（var tt = {}）；tt.constructor = ff()；tt的原型指针指向ff函数的原型；执行ff内的代码。(改变上下文this的指向)

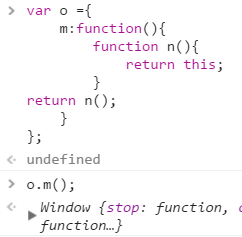
**函数调用**

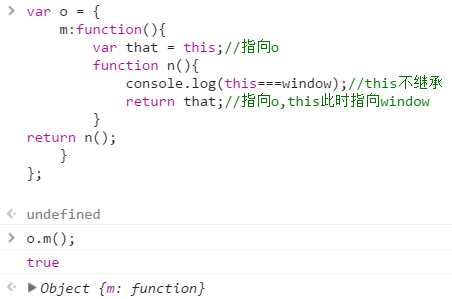
函数调用：函数不是一个对象的属性。正常函数调用如下：

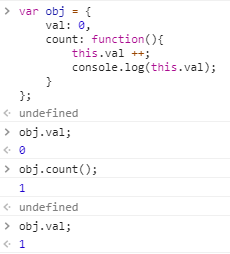
函数调用时，如果函数体内有this，比如console.log(this);则this指向window全局对象。故，函数体内写有this.a = 1时，执行后会改变全局对象a的值。

方法调用：

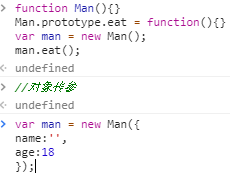
方法调用时，this：谁调用指向谁。此处对象o调用，所以指向o（注意o.m()的结果，此处疑似有误，少了一个n:f，且o.m().a也该为2）。

嵌套函数同理，作为函数调用，this指向window。要想改变这种指向，需定义that，如下：

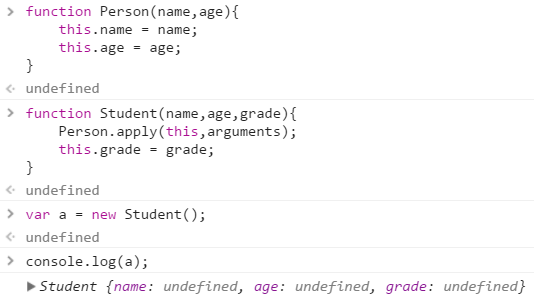


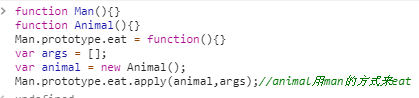
极晚绑定。count的作用就是改变val属性值。

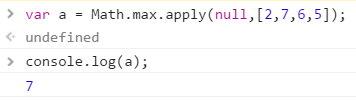
构造函数调用（略）



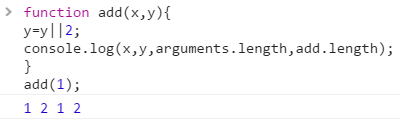
间接调用（apply call）

student中使用Person中定义的属性。

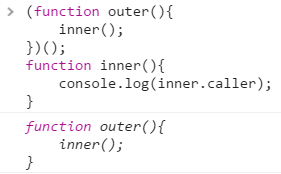
另一个apply调用的例子

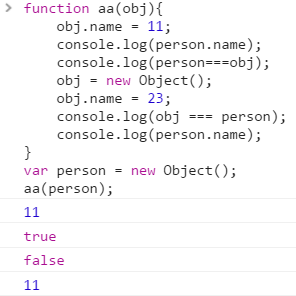
正常Math.max参数不支持数组，故可通过此来解决。

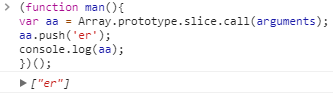
**函数参数**

||意思是给y设置默认值，只有当实参个数不包含y时才会读取||后面的2。注意，arguments只能在函数体内使用。此时x和arguments[0]值相同，但是命名空间不同。

arguments.callee()，指向arguments的函数。（阶乘）（解耦也可以使用具名的函数表达式，即var f = function ff(){}）

*函数的*caller保存着调用当前函数的函数的引用，如左显示，输出调用inner的函数。另，如果outer中输出outer.caller，由于是在全局作用域中调用，所以为null。

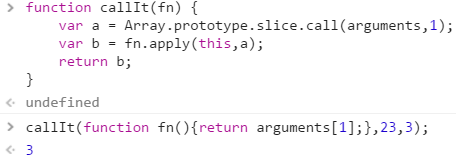
向函数参数传递引用类型person时，按值传递。注意，这里不要和构造函数混为一谈，这里并没有实例化对象。

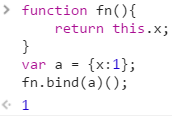
关于arguments转化成数组。注意arguments本身不是数组，所以无法使用push方法。slice 方法可以用来将一个类数组（Array-like）对象/集合转换成一个数组，只需将该方法绑定到这个对象上。可用[].slice.call(arguments)代替。可用bind代替（将call方法绑定到……）。

**函数属性和方法**

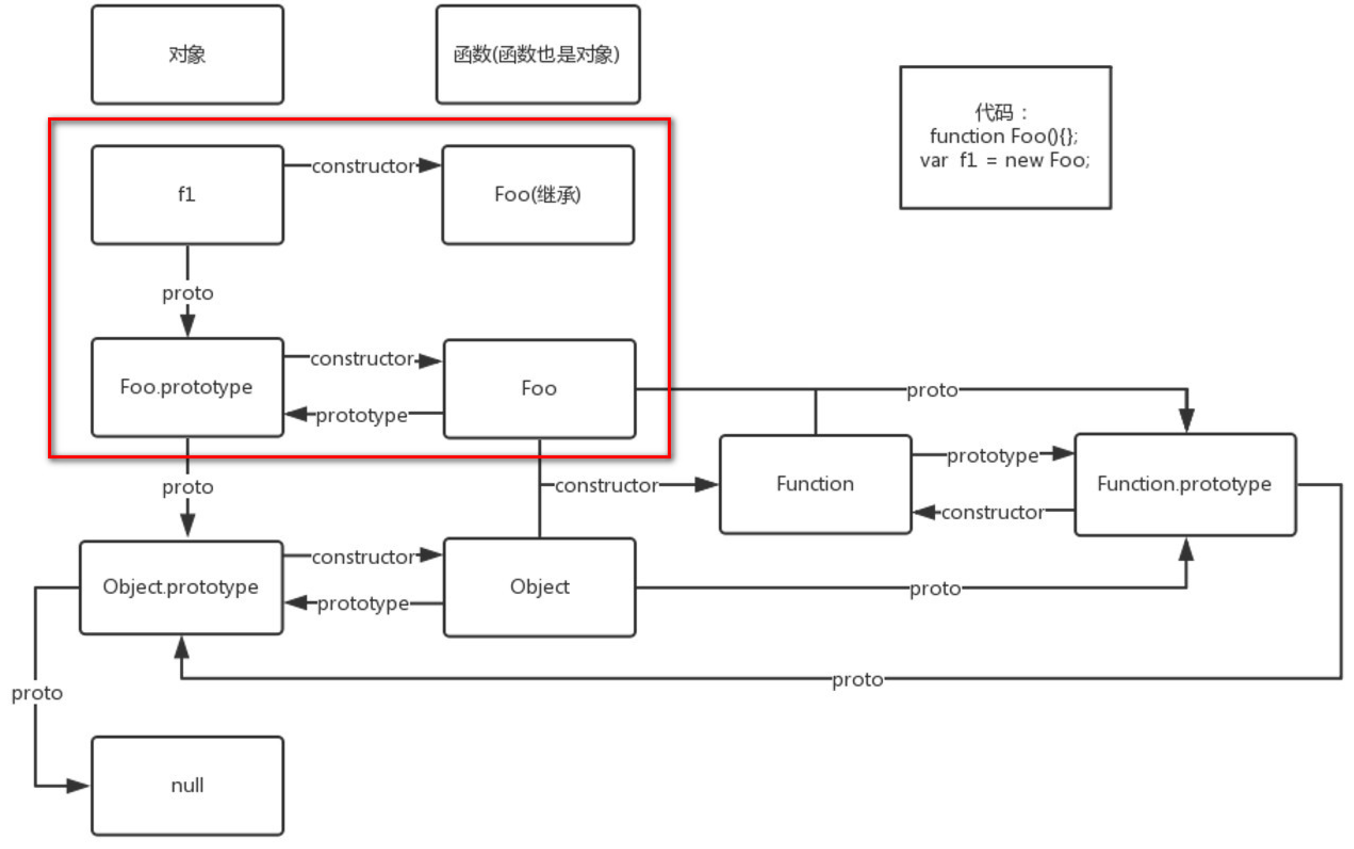
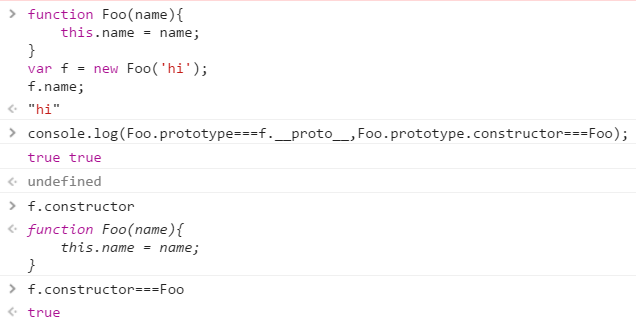
属性：length name prototype

方法：apply/call（函数原型方法Function.prototype.apply）:传入的第一个参数作用是改变函数当前上下文对象。null/undefined会把当前执行环境指向window。

例子：在arguments中执行slice。内部调用函数参数。

bind（把对象a绑定到函数fn上。实际应用，让函数在某个对象内执行，特别是对于涉及作用域的问题）

***关于prototype constructor \_proto\_***



f1 ——Foo.prototype——Object.prototype——null 实例对象的原型链

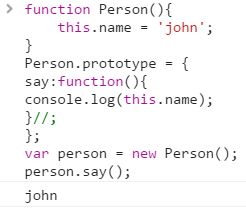
Foo——Function.prototype——Object.prototype——null 构造函数的原型链

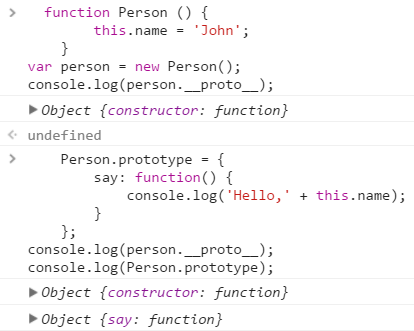
*\_proto\_（隐式原型）和prototype（显式原型）的区别与关系：*

前者每个对象都有，后者只有构造函数有

联系：隐式原型指向创建这个对象的函数(constructor)的prototype。

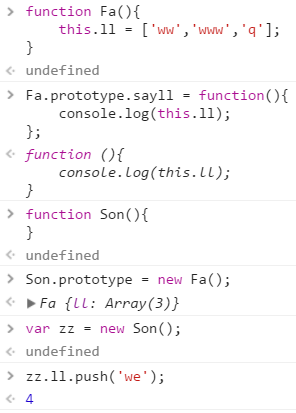
隐式原型构成原型链，对象属性查找会沿着\_proto\_依次查找。（参考：https://www.zhihu.com/question/34183746）

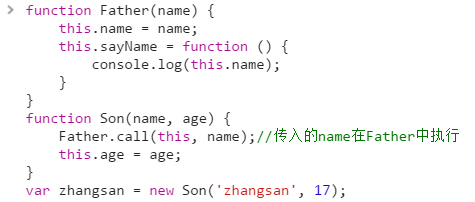
注意顺序。如果先实例化对象，再更改Person的原型，则输出person.say()不是一个方法，报错。当var person = new Person()时，Person.prototype为：Person {} (当然了，内部还有constructor属性),即Person.prototype指向一个空的对象{}。接着重置Person的原型对象，指向另一个对象，此时person的\_proto\_的指向仍未空对象，所以找不到person.say()方法。（基本如下图所示）



现在Triangle.prototype的构造函数已经不是指向Triangle，需要手动设置Triangle.prototype.constructor=Triangle。

当调用Tiger的属性或方法时，首先在其构造函数找，如果没有，则到Tiger构造函数的原型kitty中找。

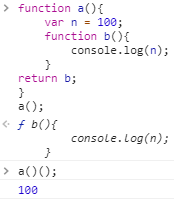
纯原型链继承：把*继承的对象的prototype*等于*继承自构造函数的实例*。缺点：直接修改了构造函数的属性，导致再var xx=new Son();xx.ll;时，xx继承的是修改后的属性。

构造函数继承：每次new实例化对象，都向对象上添加诸多属性和方法。缺点：函数不能复用。

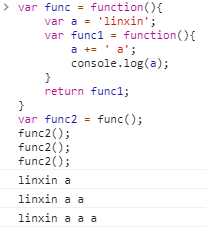
组合继承：不改变构造函数的属性；

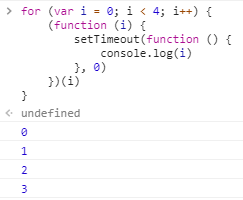
**函数闭包（closure）**

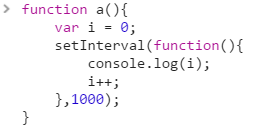
解决问题：

（利用返回值）从外部读取函数内部变量；

备注：*函数对象中总是有一个[[scope]]属性，保存着该函数被定义的时候所能够直接访问的作用域对象*

保存变量不被回收。闭包函数func1调用外部定义的变量a ，a没被回收，生命周期延续；

闭包保存每次循环的i值。如果循环体内直接定时器，由于异步，所以全输出4。

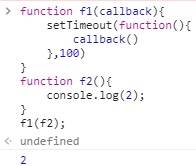
上述代码的另一种实现方式。

备注：*关于闭包内存管理 <https://github.com/lin-xin/blog/issues/8>*

**JS异步实现**

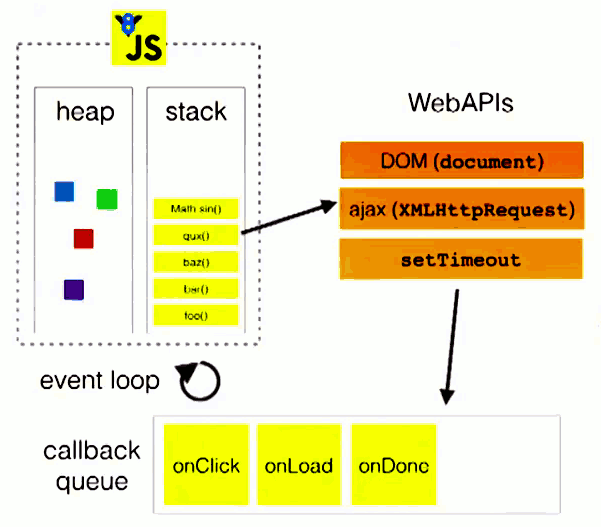
主线程（执行栈）、任务队列。事件循环（EventLoop）：主线程从任务队列中读取事件

回调 （优：易理解；缺点：不易维护，且存在回调地狱）

正常同步为：f1();f2();依次执行，f1会阻塞程序运行。现在为回调函数，耗时的计时器改为异步运行。

事件监听：任务执行取决于事件的发生（运行流程不清晰，略）

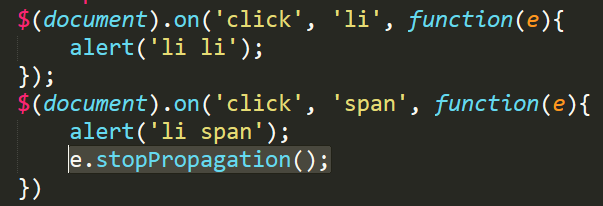
promise：每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数，如f1回调函数为f2，写为 f1().then(f2);

主线程运行的时候，产生堆（heap）和栈（stack），栈中的代码调用各种外部API，它们在"任务队列"中加入各种事件（click，load，done）。只要栈中的代码执行完毕，主线程就会去读取"任务队列"，依次执行那些事件所对应的回调函数。

**事件**

**事件流：**冒泡/捕获 / **事件处理程序**

DOM事件流：事件捕获、目标处理（会被看成冒泡的一部分）、事件冒泡

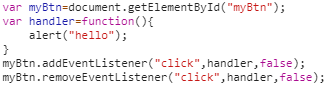


若html如下：<li><span></span></li>，由于冒泡事件流的存在，点击span会触发li。解决办法有两种，一是span事件处理程序采用stopPropagation中止，二是li事件中检测span元素（条件判断e.target.nodeName == 'SPAN'，为真，则stopPropagation，return后继续执行。

备注：*事件作用范围为该事件元素包含区域以及嵌套元素所占区域（典型例子是css处理后视觉空间上嵌套元素不在内部了）*

0级DOM（前者：动作和内容耦合，即修改时既要修改html又要改js，已弃用；后者：添加多个事件会覆盖）：html中的onclick=“” / js中的document.getElementById(‘’).onclick = function(){}

2级DOM：两个方法——addEventListener()、removeEventListener()，()内有三个参数——事件名、函数、布尔值（false为冒泡、true为捕获，默认为false冒泡）。

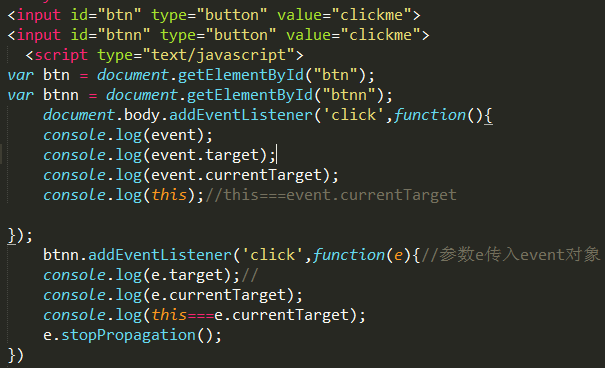
函数表达式声明，addEventListener参数可以匿名函数，但是removeEventListener参数只能使用变量名。

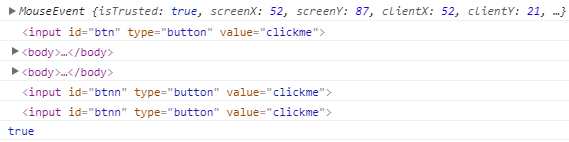
3级DOM：UI（load scroll）/鼠标/焦点（blur focus）/滚轮（mousewheel）/文本（textInput）/键盘（keydown）/合成/变动事件等

**事件对象（event）**

触发DOM上的事件都会产生一个对象。

主要属性和方法如下代码概括;





**事件代理：**

对许多元素添加事件时，可以添加到父节点，即委托父节点来触发处理函数。（冒泡机制予以支持）

**事件句柄：**

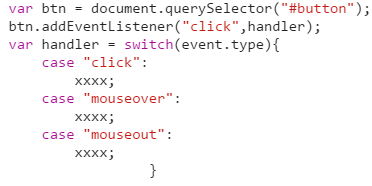
onclick等，用户点击时调用该事件句柄

**DOM事件流属性：**

*target：*返回触发此事件的元素（事件的目标节点（实际目标））——btn是click事件的目标。由于btn按钮没有注册事件处理程序，故冒泡到document.body才进行执行。

*currentTarget：*返回其事件监听器触发该事件的元素——事件写在谁上面就是谁。

注：事件处理程序内部，event.currentTarget===this

*type：*返回当前event对象表示的事件名称——一个函数处理多个事件时比如：xxxx可写为：event.target.style.backgroundColor = “pink”

bubbles：布尔值，检测当前事件是否会向DOM树上层元素冒泡——上述代码输出event.bubbles均为true；

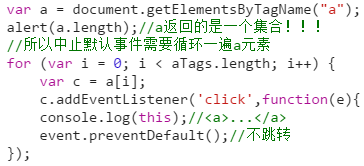
cancelable：布尔值，事件是否拥有可取消的默认动作——比如a标签默认动作是跳转链接；

eventPhase：返回数字，确定当前事件处在事件流的哪个阶段（捕获1 目标对象阶段2 冒泡3）——上述代码body事件监听输出3，btnn的输出2。

timeStamp：时间戳，发生事件的日期和时间（毫秒数）。

**DOM事件流方法：**

preventDefault():阻止浏览器执行与事件关联的默认动作

左侧代码阻止超链接自动跳转。

stopPropagation():不再派发事件——上述代码如若不加该方法，执行顺序为btnn-body，均为冒泡。由于加上该方法，阻止了向body的冒泡。

stopImmediatePropagation()：阻止之后事件处理程序被调用。用于一个事件有多个事件处理程序时。如btn添加了两次click的事件监听，如果在第一次click时执行该方法，则第二次的click事件处理不会被执行。

**IE事件相比于DOM的区别**

*方法属性：*

event.cancelBubble=true 🡸event.stopPropagation() 阻止冒泡

event.returnValue=false 🡸event.preventDefault() 阻止默认事件

event.srcElement 🡸event.target 获取事件目标

element.attachevent(“onclick”,func) 🡸element.addEventListener(“click”,func,true)为元素绑定事件

通用：element.onclick=func。onclick只有执行一个过程，而attachevent和addeventlistener执行的是一个过程列表，也就是多个过程。如：element.attachevent(”onclick”, func1);element.attachevent(”onclick”, func2)这样func1和func2都会被执行。

一种写法：if(event.stopPropagation){event.stopPropagation();}else{event.cancelBubble=true;}根据浏览器不同选择阻止事件冒泡的方式

*事件流：*

IE冒泡型事件模型，DOM先捕捉后冒泡

**鼠标事件**

mousedown mouseup click dbclick（按顺序触发）

mouseover mouseout 会冒泡（同心圆外到内触发两次，第二次为内圆冒泡触发）

mouseenter mouseleave 不冒泡（移到后代元素上不触发）

mousemove 需监控鼠标变化，更耗资源

event.button 表明当前事件由鼠标哪个按键触发（chrome中，0为左键，2为右键）

**自定义事件**

定义一个数组，添加事件就push进事件处理函数。执行时，遍历数组即可。

函数式实现：调用函数。（全局变量过多）

字面量实现：实际是对象方法调用和原型调用的区别，后者耦合性弱。

原型模式实现（以下代码详见<http://www.zhangxinxu.com/study/201203/js-custom-events-prototypal.html>）：



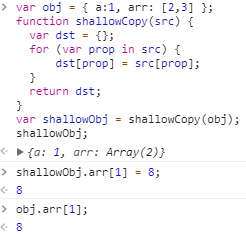
调用：var xx = new a(); //new构造

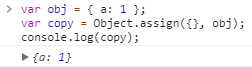
xx.addEvent(“”,function); //添加该方法

document.onclick = function(){ xx.fireEvent(“”); } //执行该方法

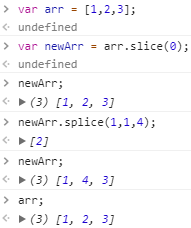
**浅拷贝（拷贝表层，同变）、深拷贝（无关联）——针对复杂对象**

*浅拷贝：*

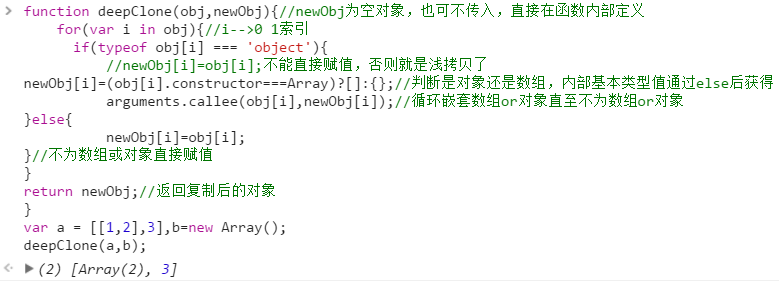
直接复制属性，a为基本类型，不存在浅拷贝；arr为数组，所以会同步改变。实际就是直接复制，等同于var objj = obj;objj;。

Object.assign()方法（Object构造函数创建一个对象包装器 ）。第一个参数为目标对象，该函数返回值为目标对象。不适用于属性值为引用对象的对象，比如var obj = {name:’a’,age:{child:24}}，age的属性值只是拷贝的指针，此时如果修改obj.age.child，复制后的对象内部该值也会改变。

*深拷贝*

slice针对数组。以及concat。（适用于不包含引用对象的一维数组即这两种情况不适合，本质还是浅拷贝）

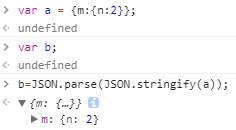
递归实现

递归实现的重点是，避免了直接给对象赋值导致的浅拷贝。从外到内的循环，比如该例子，a[0]到a[0][0]，先出数组空间，再复制值。注意，[]也可以表示对象读取属性，所以才要判断是对象还是数组。for循环的i，在数组中表示索引，在对象中表示属性名。

之所以避免了浅拷贝，是因为这相当于新建一个空数组（内存空间），一个个推入数组元素，而不是单纯的简单复制。（因为数字、字符串的赋值不会相互影响）

该方法缺点是，如有相互引用对象（比如obj.a = obj），会进入死循环。故添加判断从句如下，存在死循环即退出循环：

if(obj[i]===obj){continue;} 即跳出当前循环，继续执行下一个循环。

使用JSON实现。

*插入（关于JSON）*：序列化对象、数组、数值等的语法，针对对象和数组的表现形式为——属性名双引号字符串。

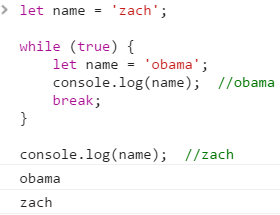
JSON.parse(text[,reviver]))：字符串转换为对象。reviver为可选处理函数。

JSON.stringify(value[,replacer[,space]])：对象转换成字符串。

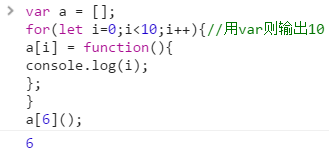
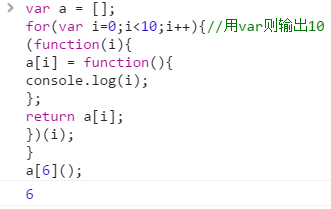
上述方法缺点：转换完后constructor指向Object，原来的构造函数自动被忽略。（能正确处理那些可以被JSON直接表示的数据结构）序列化JS对象时，函数和原型对象会被忽略。即a中有function对象不会被复制，a的原型不会被复制。

**ES6**

**let/const**

let命令在所在的代码块内有效。如果是var，则全部显示obama（均为全局变量，后面定义的覆盖前面的）。

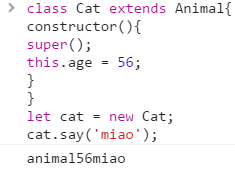
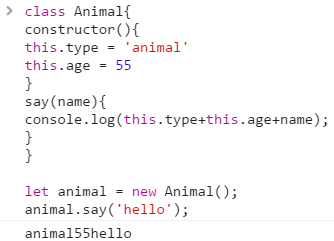
另，var的缺点：计数循环变量泄露为全局变量，导致显示最后一个值。

如果使用var，也可以通过闭包解决。

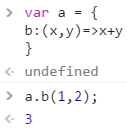
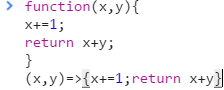
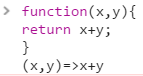
const声明常量，声明后常量不可更改。常用场景：引用第三方库时，const m = require(‘m’);

**class extends super**

class定义类，内部有constructor方法（构造方法）。extends实现继承。super指代父类实例。

本例中，cat继承了animal的属性和方法，但是要修改this，需要加上super()，因为继承的子类没有自己的this对象，只能继承父类的进行再加工。

**箭头函数（更短 不绑定this、arguments、super等）**



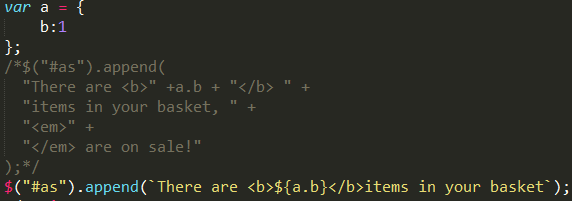
上述三个示例认识箭头函数。缩减的一个是匿名函数的函数名，一个是纵向结构变成横向。注意，函数体多列使用{}，只有return省略return。

解决的ES5问题：寻找this困难。（特别是对于定时器部分，如下示例 ）

注释部分的定时器引用的this是指向全局作用域的。而arrow function所引用的this是定义时的对象，而不是引用的对象。实质是箭头函数没有this，它的this继承自外层代码块。

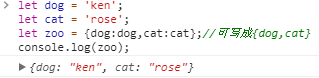
解决定时器问题还有两种方法，一是var that = this，在计时器内部直接使用外部的this（实际为that）；二是

**模板字符串**

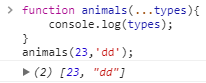
模板字符串要点：

``来表示起始；${}引用变量。空格和缩进保留在输出中。

**解构（destructuring）**

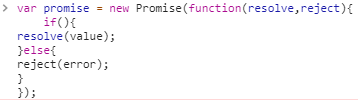


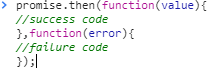
**rest参数**

实际类似arguments。function xx(a,…xxx){}，“…”会把第一个参数之后的实参放入一个数组，并把该数组赋值给xxx变量。

**Promise（解决异步）——构造函数**

传统解决方法是回调、事件等。可以把Promise看成一个容器，内部保留着一个异步操作的结果，实际会先执行同步事件，最后再执行promise中的异步事件，比如图片异步加载等。

如有右，创建了一个Promise的实例。传入一个函数作为形参。该函数有resolve和reject两个参数，它们是两个由js引擎提供的函数。异步操作成功执行前者，否则后者。

promise实例生成后，then指定两种状况的回调函数。注意，reject的对象通常是Error的实例，即new Error()。注意，实际应使用后面的catch语句来代替这里。

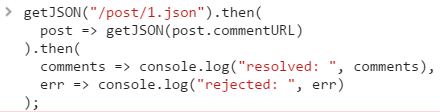
一个关于定时的实际例子。



同时，支持另一个promise实例作为resolve的参数，比如let p1=new Promise();let p2 = new Promise((resolve,reject)=>{setTimeout(()=>resolve(p1),100)});

实例的方法 = 构造函数的原型的方法（原型知识！），所以定义的实例方法均为Promise.prototype.method()：

*Promise.prototype.then()：*()内两个函数参数，第二个rejected状态的可选。注意，then()返回的是一个新的Promise实例，故链式写法，即then().then()。

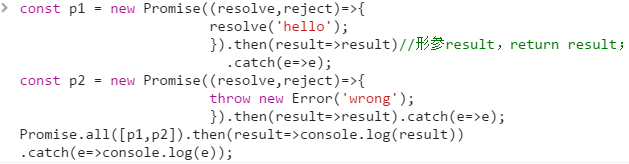


*.catch()*:上述提到rejected状态的回调函数，即Promise函数中的第二个函数参数，应使用此方法替换

catch可以捕获前面then中的错误。若无该函数，Promise会吃掉函数比如直接resolve(x+2)，由于x未定义会出现错误，但是脚本会继续运行。可以.catch().then()，捕捉错误后可以继续执行后面的代码以及，catch内部也可以抛出错误，需要下一个catch来捕获。

*Promise.all() Promise.race()*

var p = Promise.all([p1,p2,p3]);其中，传入的数组为各个Promise实例组成。只有数组内各个实例状态变为fulfilled之后，或其中一个变为rejected，其返回值才会组成一个数组传给p的回调函数。

注意箭头函数的写法及含义。

而race方法的区别在于，只要一个实例改变状态，即把返回值传给p的回调函数（then后面的函数）。