## 页面加载及优化

* + - * 1. 页面加载流程：
  1. 用户访问网页，发送一个HTTP请求到网络服务器
  2. 网络服务器（应用服务器）解析请求，发送请求给数据库服务器
  3. 数据服务器返回数据给网络服务器，网络服务器解析数据，并生成HTML文件内容放入HTTP response中，返回给浏览器
  4. 浏览器解析HTTP response
  5. 浏览器创建DOM树
  6. 浏览器下载css，并应用在DOM树上，进行渲染
  7. 浏览器下载js并解析执行js

1. 缺陷：
   1. 服务端：
      1. 网络服务器：无法获取到资源文件（404），或者由于并发的原因暂时无法处理你的请求（最常见的500错误），浏览器会长时间处于空白状态，直到服务器返回状态，或者进行超时处理。
      2. 数据层：如果服务器停止，或者忙于处理大数据，长时间无法返回数据给网络服务器，网络服务器一直处于等待状态中，如果请求量达到最大值，那么后面的请求都被堵塞，从而无法返回内容给浏览器
   2. 客户端：
      1. JavaScript：如果js写在body中的div里，并且这个js执行复杂逻辑，那么整个页面处于等待状态中。不论js代码是内联还是包含在一个不相干的外部文件中，页面下载和解析过程肯定会停下，等待脚本执行完，然后才能继续。

加载过程遇到外部css文件或者图片资源，浏览器会另外发出一个请求获取资源，这是异步请求，不影响文档加载。但是当文档加载过程遇到js文件，HTML文档会挂起渲染（加载解析渲染同步）的线程，等到js文件加载解析执行完毕，才能恢复HTML的渲染线程

原因：js可能会修改DOM，最为经典的document.write，这意味着，在js执行完成前，后续所有资源的下载是没有必要的。

办法：将外部引用的js文件放在</body>前。

* + 1. CSS：可以同时下载多个CSS文件。

把CSS文件放在页面底部，虽然可以使页面内容更快的加载（因为将加载的css文件的时间放在最后，从而使页面内容先显示），但是内容没有样式，CSS文件加载进来后，浏览器再对DOM使用样式，会出现“无样式之闪烁”。

上下都放置CSS样式，浏览器会先按照上面的渲染，再按照下面的额样式回流和重绘，用户体验差。

repaint(重绘)是在一个元素的外观被改变，但没有改变布局的情况下发生。——如果只是改变某个元素的背景色、文字颜色、边框颜色等等不影响它周围或内部布局的属性，将只会引起浏览器repaint。

reflow（回流）：浏览器发现某个部分发生了点变化影响了布局，需要倒回去重新渲染，这个回退的过程就叫回流。

1. 解决方案
   1. 减少http请求，合理设置HTTP缓存
   2. 使用浏览器缓存
   3. 压缩：服务器端压缩文件，浏览器端解压缩
   4. 合并css图片
   5. 懒加载：不一定能减少HTTP请求书，能在某些条件下或页面刚加载时减少HTTP请求数。对图片而言，在页面刚加载的时候只加载第一屏，用户继续往后滚屏的时候才加载后续的图片
   6. CSS放在页面上部（<head>标签内），JavaScript放在</body>前
   7. 异步请求（将一些行为样式提取出来，控制加载的时机）
   8. 减少cookie传输
   9. JavaScript代码优化：
      1. HTML Collection：document.images、document.forms、getElementsByTagName()返回的都是HTMLCollection类型的集合，每次访问该集合时都会重新执行查询。遍历HTML Collection的时候，尽量将它转为数组后再访问，以提高性能。
      2. 慎用with。增加了作用域链长度。
      3. 避免使用eval和Function。每次 eval 或Function 构造函数作用于字符串表示的源代码时，脚本引擎都需要将源代码转换成可执行代码。这是很消耗资源的操作
      4. 减少作用域链查找。可将全局变量赋值给局部变量；减少闭包使用
      5. 字符串拼接。使用join而非“+”
   10. CSS选择符优化：浏览器对CSS选择符解析是从右往左进行的
   11. CDN加速：本质是缓存，将数据缓存在离用户最近的地方。

## 原型链

原型组成的链，对象的[[Prototype]]指针指向对象的原型，原型也是一个对象，也有[[Prototype]]指针，指向的是原型的原型，就这样可以一直向上找，这就是原型链。当向上找到Object原型的时候，原型链就到头了。

[[Prototype]]指针在脚本中没有标准的方式访问，但FireFox，Safari和Chrome在每个对象上都支持\_proto\_属性。

Instanceof运算符用于判断指定构造函数.prototype对象是否在某个对象的原型链上。

例如：var cat=new Cat(); cat instanceof Animal; 判断Animal.prototype是否在cat的原型链上，以此判断cat是否是Animal的实例。

JavaScript通过原型链来实现继承！

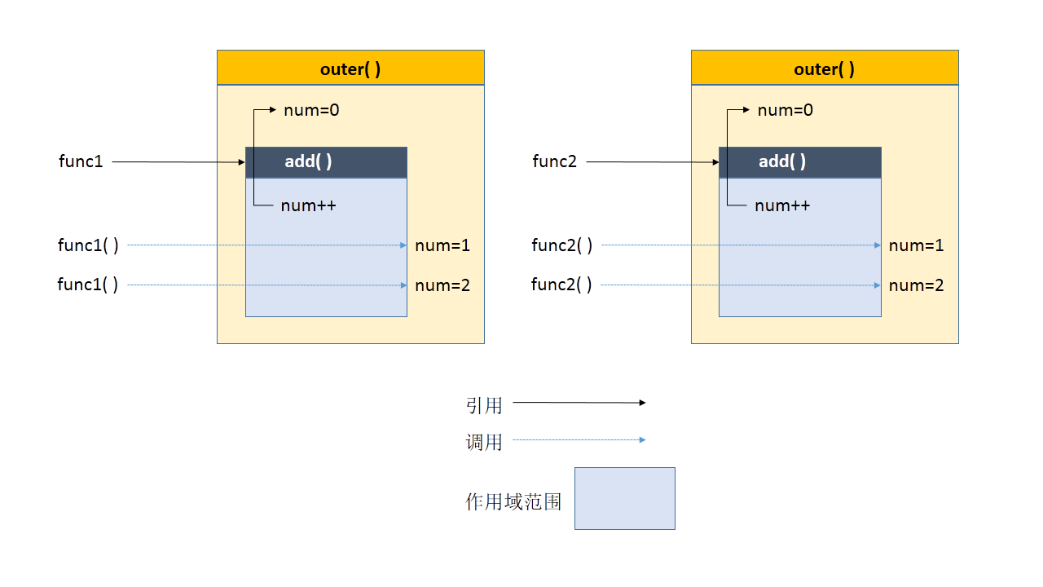
原型链通过将子类构造函数的原型赋值为父类的实例

## 闭包的概念

闭包：有权访问另一个函数作用域中变量的函数，创建闭包的常见方式是在一个函数内创建另一个函数。

本质：实现一个暴露内部变量，而且外部可以访问修改的函数。





闭包的作用：

1. 使用闭包可以访问函数中的变量
2. 可以使变量长期保存在内存中，生命周期比较长

闭包的缺点：

闭包不能滥用，否则会导致内存泄露，影响网页的性能。闭包使用完了后，要立即释放资源，将引用变量指向null

## 快排的时间复杂度和空间复杂度

快排的基本思想：通过一趟排序将待排记录分成两部分，一部分记录的关键字均比另一部分小，则可分别对两部分记录继续排序，以达到整个序列有序。

**基准+分区**

算法描述：

1. 定基准。从数列中挑选出一个元素，作为“基准”
2. 分区。依次遍历数列的剩余元素，比基准值小的放在基准前，大的放在基准后（相同的数放在任一边），分区退出后，基准就位于数列的中间位置。
3. 递归的把小于基准值元素的子序列和大于基准值元素的子数列排序。



1. 时间复杂度和空间复杂度

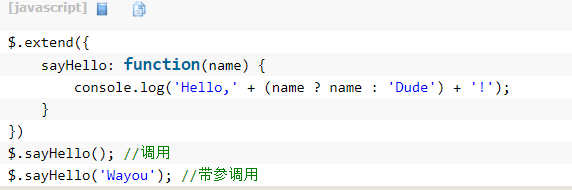
最佳情况：T(n) = O(nlogn)   最差情况：T(n) = O(n2)   平均情况：T(n) = O(nlogn)

空间复杂度：O(logn)

## 手写一个jQuery插件

jQuery插件开发方式主要有三种：

1. 通过$.extend()来扩展。在jQuery命名空间或者理解成jQuery身上添加了一个静态方法。直接通过$符号调用（$.myfunction()），而不需要选中DOM元素($('#example').myfunction())

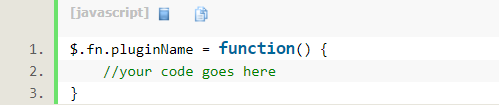


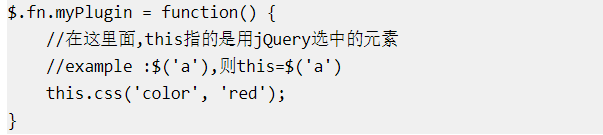
缺点：无法利用jQuery强大的选择器带来的便利，要处理DOM元素以及将插件更好的运用于所选择的元素身上，需要使用$.fn。

1. 通过$.fn向jQuery添加新的方法。 使用最多。
2. 通过$.widget()应用向jQuery UI部件工厂方式创建。用来开发高级jQuery部件，该模式开发出来的部件带有很多jQuery内建的特性，例如插件的状态信息自动保存，各种关于插件的常用方法等

**插件开发：**

（1）基本方法



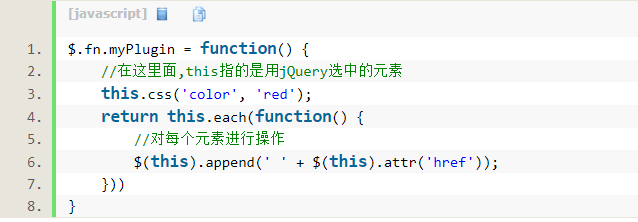


this指向jQuery选择器返回的集合，例如上图中$(‘a’)返回的是页面上所有的a标签的集合，且该集合已经是jQuery包装类型了，在对其进行操作的时候可直接调用jQuery的其他方法而不需要再用$符号来包装了。通过调用jQuery得到each()方法可以处理集合中的每个元素。需要注意，在each方法内部，this指代的是普通DOM元素，如果需要调用jQuery的方法那就需要用 $ 重新包装一下。如下图中$(this).



1. 支持链式调用

jQuery支持链式调用，要使插件不打破链式调用，可以将调用对象返回即可。

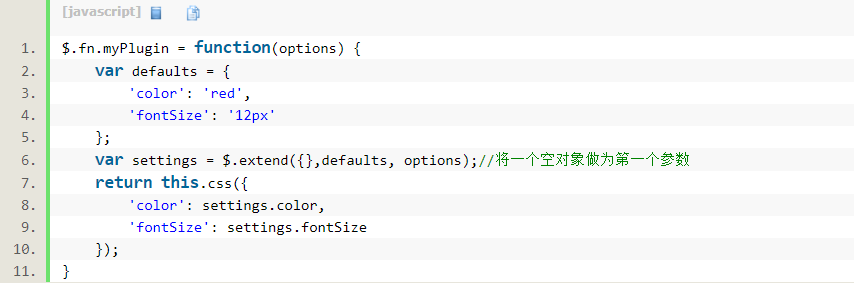


1. 让插件接收参数并保护好默认参数

jQuery的extend方法：

1. 给extend方法传递单个对象的情况下，这个对象会合并到jQuery身上。
2. 一个以上参数：将所有参数对象合并到第一个里，如果对象中有同名属性，合并的时候后面的会覆盖前面的

可以在插件里定义一个保存插件参数默认值的对象，将一个新的空对象做为$.extend的第一个参数，defaults和用户传递的参数对象紧随其后，这样做的好处是所有值被合并到这个空对象上，保护了插件里面的默认值。最后就实现了用户指定了值的参数使用指定的值，未指定的参数使用插件默认值。



1. 面向对象的插件开发

将需要的重要变量定义到对象的属性上，函数变成对象的方法，通过对象来获取。一是方便管理，二是不影响外部命名空间，因为这些变量名和方法名都是在对象内部。



1. 命名空间

自调用匿名函数包裹自己的代码，避免与他人的代码冲突



1. 参考地址：<http://blog.csdn.net/unionz/article/details/54914716>

在jquery方法和原型上面添加方法的区别和实现（$.extend,$.fn.extend），以及jquery对象的实现（return new jQuery.fn.init）

## 手写一个递归函数（考察arguments.callee，以及arguments的解释）

函数内部有两个特殊的对象：arguments和this。arguments是一个类数组对象，包含着传入函数中的所有参数，可以使用方括号访问它的每一个元素（第一个元素是arguments[0]，以此类推），用length属性获取参数个数。具有callee属性。

callee属性是一个指针，指向拥有这个arguments对象的函数。

*//阶乘***function** *factorial*(num) {  
 **if**(num<=1){  
 **return** 1;  
 }**else**{  
 **return** num\*arguments.**callee**(num-1);  
 }  
}

Caller指向函数的调用者

## JS 在循环的时候，HTML会不会卡住？

会。测试情况：将页面样式设置放在10000次的循环后，样式会过一瞬间再显示

## 作用域及作用域链

作用域是变量与函数的可访问范围。

JavaScript中，变量的作用域有全局作用域和局部作用域。

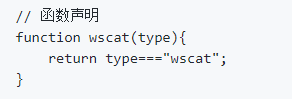
函数也是一个变量，它的作用域和变量的作用域一样，也是声明其定义时所在的作用域，与运行时无关，函数的作用域在函数体内的变量声明之前就可用了。（JavaScript中的函数运行在他们被定义的作用域里，而不是他们被执行的作用域里。——JavaScript权威指南）

* 1. 代码中任何地方都能访问到的对象拥有全局作用域
  2. 局部作用域只在固定的代码片段内可访问到，最常见的是函数内部

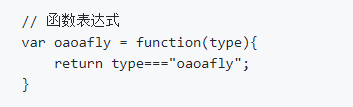
当查找变量的时候，会先从当前上下文的变量对象中查找，如果没有找到，就会从父级(词法层面上的父级)执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。这样由多个执行上下文的变量对象构成的链表就叫做作用域链。

## 函数定义方式及区别

1、函数声明



1. 函数表达式



1. Function()构造函数（不常用）
2. 函数声明和函数表达式的区别
   1. 函数声明在JS解析时进行函数提升，提升到作用域的最前面，不管函数声明在哪里定义，该函数都可以被调用
   2. 函数表达式在JS运行时确定，并且在表达式赋值完成后，该函数才能被调用

## call()，.apply()，bind()

每个函数都包含两个非继承来的方法apply()和call()。这两个函数的用途是在特定的作域中调用函数，实际上是设置函数体内的this对象的值，给函数指定作用域（就是将函数放在哪个里面执行）。

* 1. apply()方法接收两个参数：一个是在其中运行函数的作用域；另一个是参数数组（Array实例或arguments对象）
  2. call()方法与apply()方法的作用相同，仅仅是接收参数的方式不同。call()方法第一个参数是this，其余参数逐个列举出来传递。
  3. bind()方法：创建一个函数的实例，this的值绑定到传递给bind函数的对象。

## 对象继承，对象的某些特性

对象的概念：是无序的属性的集合，属性可以包含基本值、对象或是函数。

继承有原型链继承、构造函数继承、组合继承（原型链继承和经典继承双剑合璧）、原型式继承、寄生式继承、寄生组合式继承

## In和hasOwnProperty的区别

1. hasOwnProperty（）方法可以检测一个属性是存在于实例中还是原型中
2. in操作符在通过对象能访问给定属性时返回true，无论该属性是在实例中还是原型中。
3. 通过hasOwnProperty和in，能够确定属性时存在于实例中还是原型中。当in为true，hasOwnProperty为false时可以确认属性存在于原型中。



## ECMAScript中的属性

[[特性]]描述了属性。

1. 数据属性：包含一个数据值的位置，在该位置可以读取和写入值
   * 1. [[Configurable]]：能否通过delete删除属性从而重新定义属性。默认false
     2. [[Enumerable]]：能否通过for-in循环返回属性。默认false
     3. [[Writable]]：能够修改属性的值。默认false
     4. [[Value]]：包含这个属性的数据值。
     5. 直接在对象上定义的属性，特性默认值是true，value是undefined
2. 访问器属性：不包含数据值，包含一对getter和setter函数（这两函数不是必需的）。读取访问器属性时，会调用getter函数，getter函数返回有效值；写入访问器属性时，调用setter函数并传入新值，setter函数负责决定如何处理数据
3. [[Configurable]]
4. [[Enumerable]]
5. [[Get]]
6. [[Set]]
7. 修改属性默认的特性，Object.defineProperty(属性所在对象，属性名，描述符对象)。描述符对象必须是：configurable、enumerable、writable和value。使用Object.defineProperty()创建一个新属性时，如果不指定，configurable、enumerable、writable默认是false；如果调用Object.defineProperty()修改已定义的属性的值，则无此限制。

例1：定义数据属性  
**var *person***={};  
Object.defineProperty(***person***,**'name'**,{  
 **writable**:**true**,  
 **enumerable**:**true**, //属性设置为可枚举  
 **value**:**'zhangsan'**});  
Object.defineProperty(***person***,**'age'**,{  
 **writable**:**true**,  
 **enumerable**:**false**,//属性设置为不可枚举  
 **value**:**'10'**});  
**for** (**var property in *person***){  
 **console**.log(**property**)  
}

结果：name。

for-in语句用来枚举对象的属性。

例2：定义访问器属性

**var *book***={  
 **\_year**:2010,  
 **edition**:1  
};  
Object.defineProperty(***book***,**'year'**,{  
 get:**function** () {  
 **return this**.**\_year**;  
 },  
 set:**function** (value) {  
 **if**(value>2010){  
 **this**.**\_year**=value  
 **this**.**edition**=2;  
 }  
 }  
});  
**console**.log(***book***.**year**);//2010  
***book***.**year**=2018;  
**console**.log(***book***.**edition**);//2

1. 定义多个属性：Object.defineProperties(对象，{属性名1：描述符对象1}，{属性名2：描述符对象2},…)。
2. Object.getOwnPropertyDescriptor()可以取得给定属性的描述符，例如Object.getOwnPropertyDescriptor(book,"version")

## 字符串逆序输出

1. 借助数组的reverse方法。首先将字符串转化成数组，通过调用数组原生的reverse()方法逆序，再通过join方法得到逆序字符串。
2. 利用charAt（）方法。从尾部开始遍历字符串，再逐个拼接字符。
3. 构造栈结构，利用栈的后进先出原则
4. 递归
5. Call方法，Array.prototype.slice.call(str)得到一个数组
6. 将字符串使用split(‘’)转换成数组，使用Array类型的pop方法

## 事件委托

利用冒泡的原理，将事件监听器添加到较高层次元素上面，监听器根据子元素冒泡上来的事件进行分析和处理。

**优点**

* + - * 1. 大量节省内存占用，减少事件注册
        2. 方便动态的修改和添加元素，不需要因为元素的改动而修改事件绑定
        3. JavaScript和DOM节点之间的关联变少了，减少了因循环引用而带来的内存泄漏发生的概率

**缺点**

1. 不是所有的元素都能冒泡。Blur、focus、load、unload不能冒泡，事实上blur和focus可以用事件捕获获得（除IE之外）
2. 如果层级过多，冒泡过程可能受阻

JQuery中的事件委托：

* $.on: 基本用法: $('.parent').on('click', 'a', function () { console.log('click event on tag a'); })，它是 ‘.parent’ 元素之下的 a 元素的事件代理到 $('.parent') 之上，只要在这个元素上有点击事件，就会自动寻找到 .parent 元素下的 a 元素，然后响应事件；
* $.delegate: 基本用法: $('.parent').delegate('a', 'click', function () { console.log('click event on tag a'); })，同上，并且还有相对应的 $.delegate 来删除代理的事件；
* $.live: 基本使用方法: $('a', $('.parent')).live('click', function () { console.log('click event on tag a'); })，同上，然而如果没有传入父层元素 $(.parent)，那事件会默认委托到 $(document) 上；(已废除)

## DOM事件

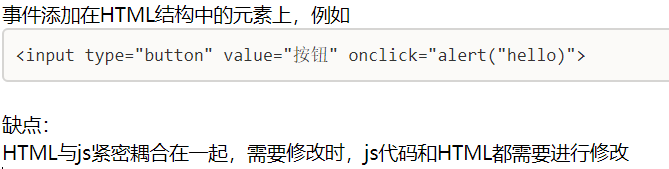
事件流：从页面中接收事件的顺序。分为事件冒泡流和事件捕获流

事件冒泡：从内部元素传播到外部元素

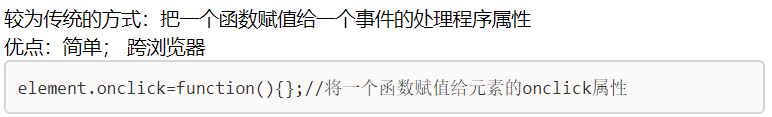
事件捕获：从最不具体的元素（document）传播到最具体的元素

### 事件处理程序

* 1. HTML事件处理程序



* 1. DOM0级事件处理程序



* 1. DOM2级事件处理程序



获取事件对象：event.target

* 1. IE事件处理程序

支持IE事件处理程序的浏览器：IE和Opera

attachEvent()：添加事件

detachEvent()：删除事件

参数：事件处理程序的名称（含有on，例如‘onclick’），事件处理程序的函数

获取事件对象：event.srcElement

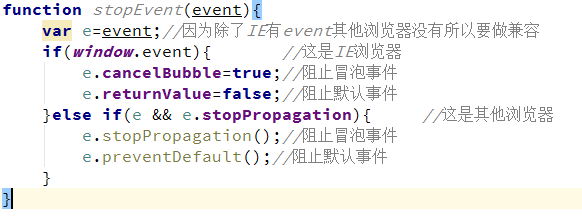
### 事件对象

1. DOM中的事件对象属性

* type属性用于获取事件类型
* target属性用于获取事件目标。例如click事件中，target指向点击的元素
* stopPropagation()方法：用于阻止事件传播
* preventDefault() 方法：阻止事件的默认行为。例如阻止a标签的默认属性跳转

1. IE中的事件对象（window.event）

* type属性用于获取事件类型
* srcElement属性用于获取事件目标
* cancelBubble属性 用于阻止事件冒泡
  + ture阻止 false不阻止冒泡
* returnValue属性 阻止事件默认行为
  + false表示阻止事件的默认行为



## 为什么很多网站的静态资源使用独立的域名？

1. 浏览器限制了同域请求并发数（包括二级域名），资源太多会阻塞，加载慢。比如有10个静态资源在同一个域名下，一次只会加载6个，加载完才会加载剩余4个。
2. 避免域名污染。当浏览器向服务器请求一个静态资源时，会先发送同域名的cookie。而请求静态资源是不需要这些cookie，这些cookie不会做任何处理，毫无意义的消耗带宽。
3. 独立的域名方便在代理服务层做动静分离，动静分离有利于CDN
4. 分布式部署可以减轻服务器的负载压力

## 安全：用户执行一个提交登录密码的操作，怎么开发相关的安全措施。

1. 采用HTTPS加持
2. 后端不需要解密前端传过来的密码，这种情况下用md5加密
3. 后端需要解密前端传过来的密码，可以用AES(令牌+密码)

## CSS或JS实现GIF动图的停止与播放

* + 1. CSS3 animation控制。使用CSS3 animation的steps()方法将动画过渡分割成段，而不是从一种状态持续到另一种状态的过渡。本质上是控制background-position属性，再通过animation-play-state:paused控制动画的状态。
    2. JS通过canvas标签修改image的src属性，若存在跨域，则将image标签的透明度修改为0，直接添加一个canvas元素，

## 柯里化

创建固定一个或多个参数的函数。

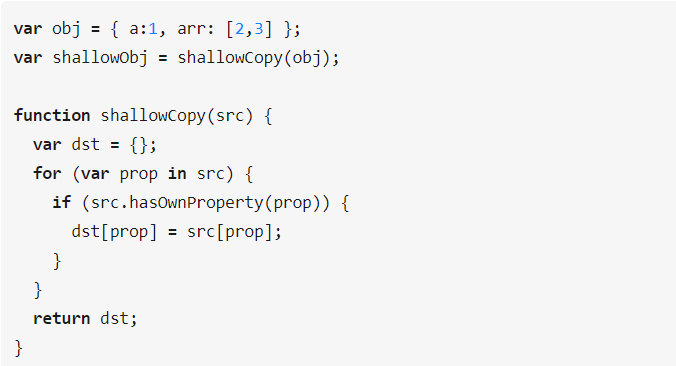
ECMAScript5的bind方法也实现了柯里化

handler.handlerClick.bind(handler,”my-btn”)//handler是绑定的对象，“my-btn”是传入的固定的参数。

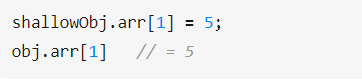


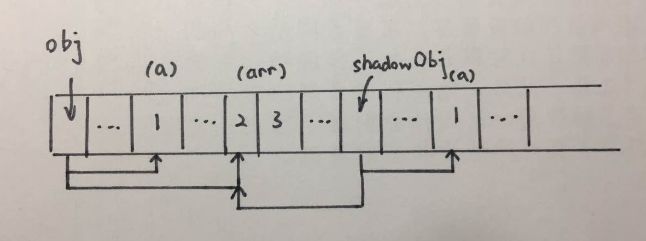
## 深拷贝和浅拷贝

（1）浅拷贝：只复制一层对象的属性。



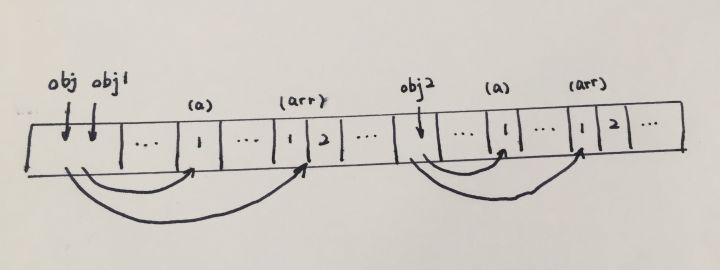
结果如下：





因为浅复制只会将对象的各个属性依次复制，并不会递归复制，JavaScript存储对象都是存地址的，所以浅复制会导致obj.arr和shallowObj.arr指向同一块内存地址。

（2）深拷贝：递归复制所有层级。不仅将原对象的各个属性逐个复制出去，并将原对象各个属性所包含的对象依次采用深复制的方法递归复制到新对象上，就不会存在上面obj和shallowObj的arr属性指向同一个对象的问题。



如果对象较大，层级较多，深复制会带来性能上的问题。实际的应用场景中，浅复制更为常用。



## 垃圾回收

JavaScript具有自动垃圾回收机制。原理：找到那些不再继续使用的变量，释放其占用的内存，按照固定的时间间隔（或代码执行中预定的收集时间）周期性的执行这一操作。

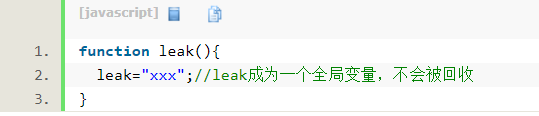
1. **标记清除**：进入环境标记，离开环境，标记为离开
2. **引用计数**：每一个对象负责维护对象所有引用的计数值。当一个新的引用指向对象时，引用计数器递增，去掉一个引用时，引用计数递减。当引用数为0时，该对象释放占有的资源。存在**“循环引用”**问题

## 内存泄漏

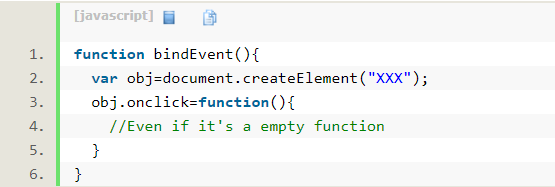
内存泄漏：不再用到的内存，没有及时被释放

内存泄漏的原因：

* + - * 1. 全局变量引起的内存泄漏



* + - * 1. 闭包引起



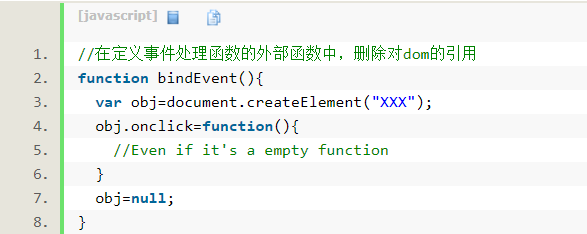
闭包可以维持函数内局部变量，使其得不到释放。 上例定义事件回调时，由于是函数内定义函数，该匿名函数保持着对包含函数的引用，导致obj不能释放，并且内部函数（事件回调）的引用外暴了，形成了闭包。

解决方法：

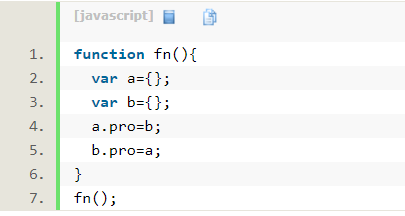
（1）将事件处理函数定义在外部



（2）删除对dom的引用，设置的事件处理仍然可以正常响应

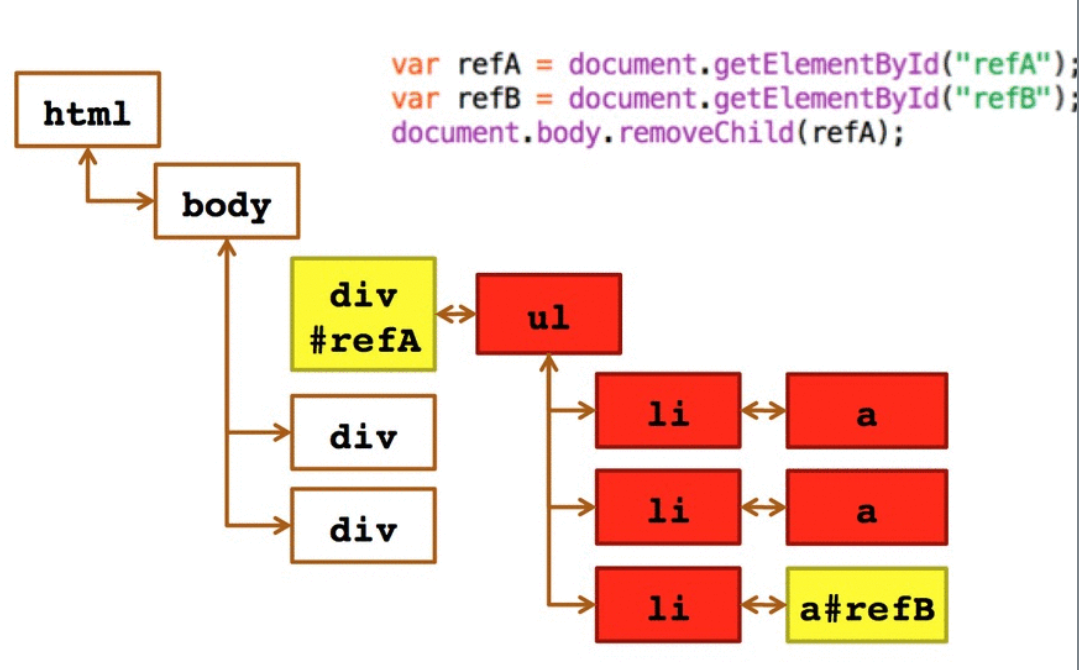


* + - * 1. setTimeout的第一个参数使用字符串而非函数时
        2. IE7/8引用计数策略下使用循环引用



fn()执行完毕后，两个对象都已经离开环境，在标记清除方式下是没有问题的，但是在引用计数策略下，因为a和b的引用次数不为0，所以不会被垃圾回收器回收内存，如果fn函数被大量调用，就会造成内存泄漏。在IE7与IE8上，内存直线上升。

* + - * 1. dom清空或删除时，事件未清除
        2. 子元素存在引用



黄色是指直接被js变量所引用；

红色是指间接被js变量所引用；

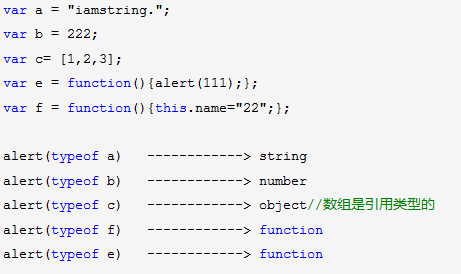
refA被refB间接引用，导致及时refA变量被清空，也是不会回收的；

refB的父元素（上图红色）由于被refB间接引用，除非refB=null，否则也不会回收

ES6解决内存泄漏的方式：WeakSet和WeakMap

## 判断类型的方法

1. Typeof



var test=undefined;

typeof test ===“undefined”；//true

test=null;

typeof test ===“object”；//true

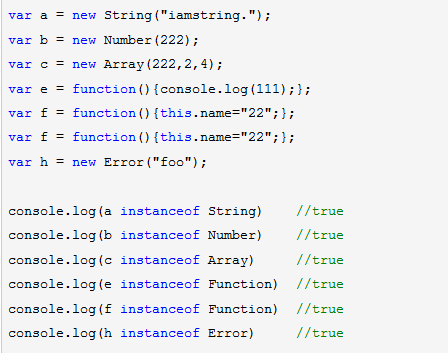
test=true;

typeof test === ‘boolean’;//true

1. 实例 instanceof 对象类型

Instanceof 用来判断一个构造函数的prototype属性所指向的对象是否存在于另外一个要检测对象的原型链上，null和undefined没有构造函数，无法使用instanceof检测

obj instanceof Object 检测Object.prototype是否存在于参数obj的原型链上。



1. 实例.constructor===对象类型



同null和undefined没有构造函数，所以无法使用constructor判断出这两者

1. toString:[object XXX]



var test=null;

Object.prototype.toString.call(test);// "[object Null]"

test=undefined;

Object.prototype.toString.call(test);// "[object Undefined]"

可用于所有数据类型的判断

## 正则和字符串

###### 定义正则

var re = new RegExp(正则表达式，修饰符); //RegExp对象。

var re = /正则表达式/,修饰符); // 简写方法 推荐使用 性能更好

修饰符：a）i：区分大小写；b）g：全局搜索；c）m：多行匹配

()：小括号，分组符。正则中的每一个带小括号的项，叫做这个正则的子项。

var str = '2013-6-7';

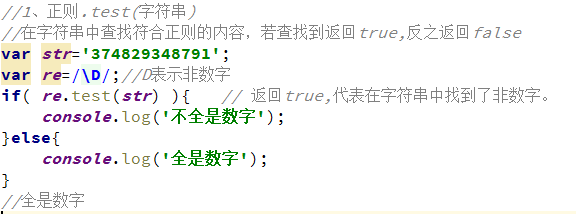
var re1 = /\d-+/g; // 全局匹配数字，横杠，横杠数量至少为1，匹配结果为： 3- 6-

var re1 = /(\d-）+/g; // 全局匹配数字，横杠，数字和横杠整体数量至少为1 3-6-

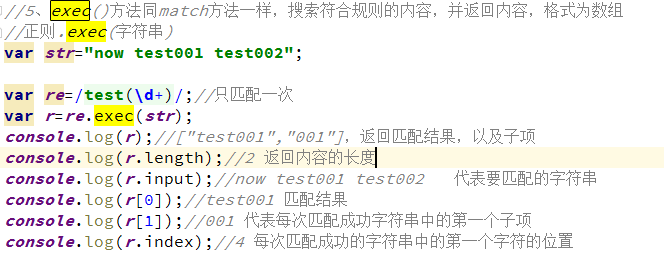
var re2 = /(\d+)(-)/g; // 全局匹配至少一个数字，匹配一个横杠 匹配结果：2013- 6-

###### RegExp的方法

* 1. test()：reg.test(string)。在字符串中查找符合正则的内容，若查找到返回true，否则返回false



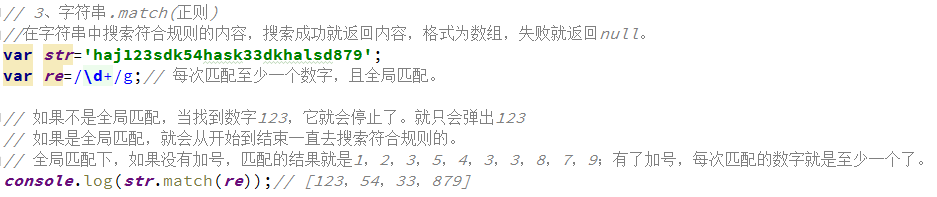
* 1. exec()：reg.exec(string)。搜索符合规则的内容，并返回内容，格式为数组。数组第一项为匹配结果，其余项为子项



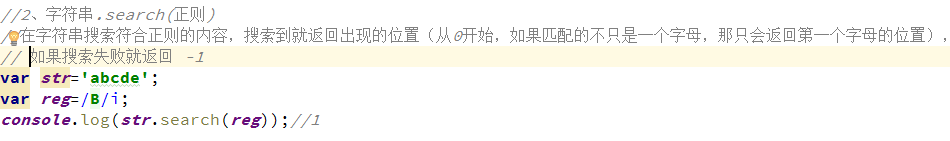
* 1. compile()：改变正则表达式。Reg.compile(newReg,新修饰符)

###### String使用正则

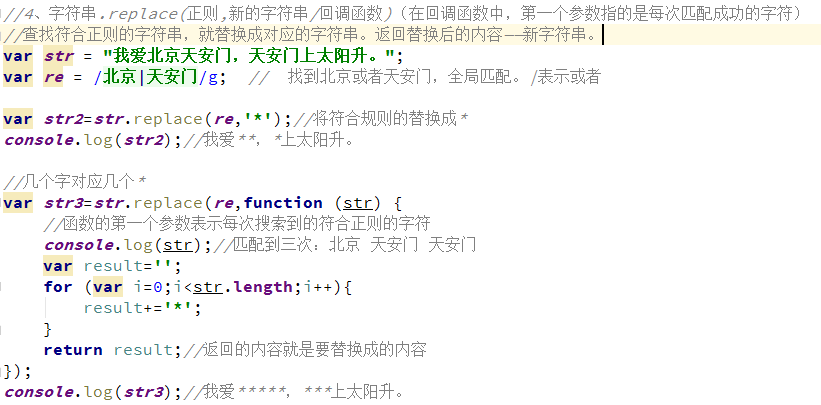
1. match()：返回数组，无论是否有g，数组第一项是匹配结果，其余为子项。如果有g，可以循环执行match，找出所有匹配的结果。

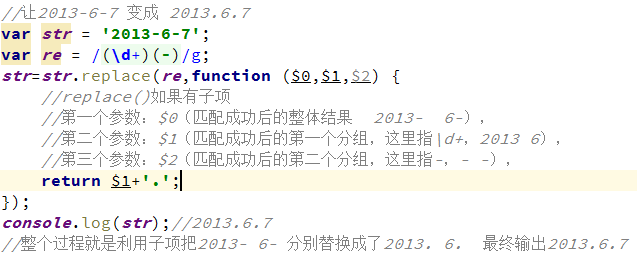


1. search()：str.search(reg)。搜索到符合正则的内容，找到就返回出现的第一个位置，失败返回-1

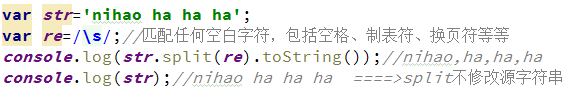


1. replace()：str.replace(reg)。查找符合正则的字符串，将符合的替换成对应的字符串，返回替换后的内容。str.replace(reg,function($0,$1,$2){})。$0：匹配成功的整体结果，$1：匹配成功的第一个分组





1. split()：string.split(正则)



## 隐式类型转换

**隐式转换**：没有指定变量类型，在进行相关运算时进行类型转换

js是弱类型语言。弱类型就是不必在变量声明时指定它的类型，可以赋给它任一类型的值。

#### 1、js数据类型

js中有七种数据类型，分为两类：原始类型、对象类型

**基础类型（原始值）**：String、Number、Boolean、Null、Undefined、Symbol（es6新出）；

**复杂类型（对象值）**：object。array和function都是object的特例

#### 2、三种隐式转换类型

涉及隐式转换最多的两个运算符+和==。

+运算符即可数字相加，也可字符串相加。==不同于===，也需要隐式转换。-\*/运算符只会针对number类型，转换的时候只能转换成number类型

隐式转换中主要涉及到三种转换：

##### 2.1、通过ToPrimitive将值转换为原始值

ToPrimitive（input，PreferredType）

（1）Input是要转换的值，

（2）PreferredType是可选参数，可以使Number或String类型。它只是一个转换标志，转化后的结果不一定是这个参数所属的类型，但是一定是原始值或报错。

**2.1.1、PreferredType被标记为Number**

1、如果输入的值是原始值，直接返回它

2、如果输入的是一个对象，调用该对象的valueOf()方法。

A）如果valueOf()返回一个原始值，返回这个原始值；

B）否则，调用这个对象的toString()方法，如果toString（）返回原始值，则返回这个原始值；否则，抛出TypeError异常

**2.1.2、PreferredType被标记为String**

1、如果输入的是原始值，直接返回它

2、否则，如果输入的是一个对象，调用该对象的toString（）方法

A）如果toString()返回一个原始值，返回这个原始值；

B）否则，如果调用这个对象的valueOf()方法，如果valueOf()返回一个原始值，返回这个原始值；否则，抛出TypeError异常

**2.1.3、如果不设置PreferredType**

按照下面的规则自动设置：

* 1. 该对象是Date类型，Preferred被设置为String
  2. 否则，Preferred被设置为Number

**2.1.4、valueOf()和toString()解析**

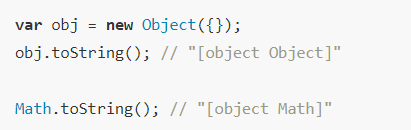
**valueOf()**会将String、Number、Boolean基础类型的对象转换成基础类型，Date类型转换成毫秒数，其他的返回对象自身

**toString()**会将所有对象转换成字符串。

Number、String、Boolean、Array、Function、Date类型都有自己的toString实现



除了上述的对象及其实例化对象外，其他对象返回的都是该对象的类型。



null和undefined没有toString()方法。null和undefined无法获取constructor，自然无法获取Object.prototype.toString()。

##### 2.2、通过ToNumber()将值转换为数字



##### 2.3、通过ToString将值转换为字符串



#### 3、==运算符隐式转换

比较x==y

* + 1. 类型相同时：没有类型转换，只需要比较值是否相同，注意NaN不与任何值相等，NaN！=NaN
    2. 类型不同时

（1）x，y为null、undefined中的一个，返回true

（2）x，y为Number和String类型时，转换为Number类型比较

（3）有Boolean类型时，Boolean转化为Number类型比较

（4）一个Object类型，一个String或Number类型时，将Object类型进行原始类型转换后，按上面的流程进行原始值比较

## ==vs===

==：相等运算符，允许类型转换

===：严格相等运算符，不允许类型转换

1. ==运算规则
   1. 操作数为同一类型：按照严格相等比较
   2. 类型不同，进行类型转换
      1. null==undefined;//=>true
      2. true=>1,false=>0后再比较
      3. 一个值是对象，另一个是数字或字符串，会先使用toString()和valueOf()将对象转换成原始值，再比较
      4. 操作数有一个是NaN，直接返回false。NaN不等于NaN
      5. 操作数都是对象，判断是否指向同一对象，同一个返回true
2. ===运算符
   1. null===null，undefined===undefined，null！==undefined

[0]==0; //true

Null==undefined; //true

0.1+0.2==0.3; //false，浮点数值计算会产生舍入误差

## Prototype，construtor

Parent.prototype指向了原型对象，Parent.prototype.constructor指向Parent；Parent的每一个实例都包含一个[[Prototype]]指针（内部属性），在Chrome、Firefox、Safari在每个对象上都支持\_proto\_属性，该属性指向Parent.prototype。实例不包含属性和方法，却可以通过查找原型链获得。

## Array.prototype.sort()的实现方式

各浏览器的实现方式不同：

Google Chrome：插入排序+快速排序

Mozilla FireFox：归并排序

Safari：归并排序和桶排序

IE（9+）和Microsoft Edge：快速排序

## Babel是什么，运行机制

ES6存在兼容性问题，很多浏览器尚不支持ES6语法。可以使用语法解析转换工具，将ES6转换成ES5。Babel是一个转码器，将es6代码转成es5代码。类似的工具还有jsx、traceur、es6-shim。

## Number类型

1、浮点数：浮点数值计算会产生舍入误差，这是使用基于IEEE754的通病。

2、数值范围：超出js数值范围，数值会被自动转换成Infinity（-Infinity）。判断正负无穷isFinity()

3、NaN：表示一个本来要返回数值的操作数未返回数值的情况。isNaN()判断是否“不是数值”

（1）任何涉及NaN的操作都返回NaN

（2）NaN与任何值都不相等，包括NaN自身。

## 千位分隔符

var test=’12345.678’; test.toLocalString();//12,345.678

思路：

1、先分离出小数部分；

2、对整数部分逆序为数组；

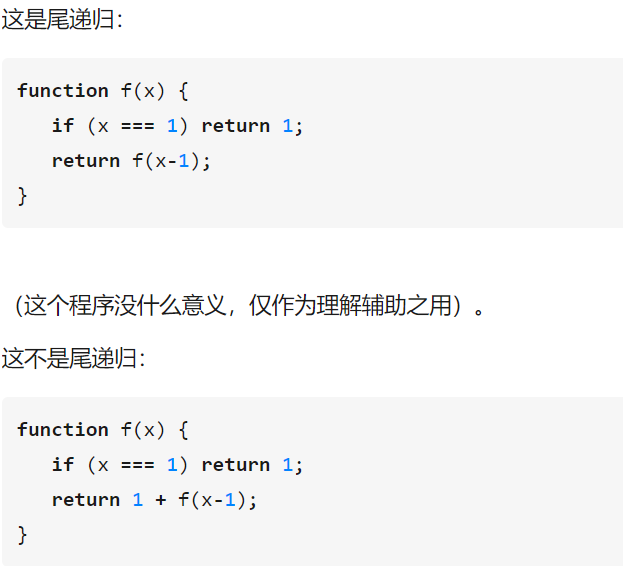
3、每三个数字插入一个逗号，如果是3的倍数位则最后一个不插入；

4、再逆序回来，拼接小数部分（如果有的话）。



## 尾递归

尾递归：操作最后一步是调用自身的递归。



**尾递归的判断标准是函数运行最后一步是否调用自身，而不是是否在函数的最后一行调用自身。**

好处：进入函数的最后一步后不再需要外层函数（调用者）的信息，因此没必要保存外层函数的stack，递归需要用的stack只有当前要执行的函数的，避免了栈溢出的风险。

Python、Java中没有尾递归，可以使用iteration写尾递归

## ES6：暂时性死区

ES6明确规定，如果区块内存在let和const命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域，凡是在声明之前就使用这些变量，就会报错。语法上，称为“暂时性死区”（temporal dead zone，TDZ）

if (true) {

// TDZ开始

tmp = 'abc'; // ReferenceError

console.log(tmp); // ReferenceError

let tmp; // TDZ结束

console.log(tmp); // undefined

tmp = 123;

console.log(tmp); // 123

}

## this指向总结

1、定时器：如果定时器调用执行了某个函数，函数中有this，则这个this指向window

2、普通函数中this：谁调用这个函数，this指向谁，如果是直接在全局中调用，说明是window调用的，this==window

3、事件函数中：this指向触发该事件的标签

4、构造函数：this指向新创建出的对象

5、call、apply修改this的指向（call是列举传参，apply是数组传参）

6、箭头函数：this指向函数被定义时所在的对象

## 创建对象时new的工作流程

1、创建一个空对象

2、将构造函数的作用域赋给该空对象（this指向该空对象）

3、执行构造函数中的方法（给空对象添加属性）

4、返回新对象

以var p1=new Person()

## Object.create()用法以及实现

Object.create(obj,propertiesObject) ;将obj的属性和方法继承到对象的\_\_proto\_\_上

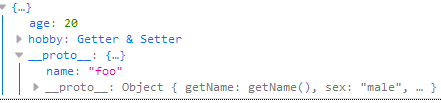
参数：

（1）obj:一个对象，应该是新创建的对象的原型。

（2）propertiesObject：可选。该参数对象是一组属性与值，该对象的属性名称将是新创建的对象的属性名称，值是属性描述符（这些属性描述符的结构与Object.defineProperties()的第二个参数一样）。

例：

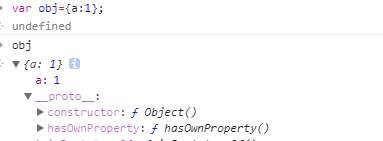




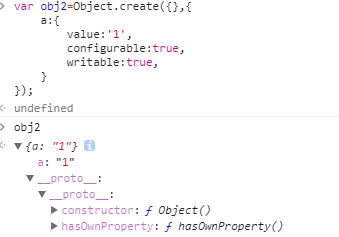
test.\_\_proto\_\_继承到f1上的实例属性；可以通过原型链访问到f1的原型上的属性和方法。

**Object.create()和{…}的区别**

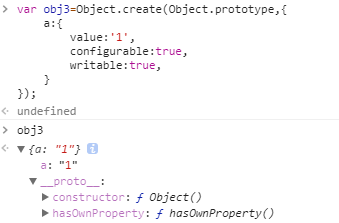
直接使用{}声明新对象



若使用{}为原型创建对象，从结果可以看出，出现\_\_proto\_\_嵌套情况



直接使用Object.prototype为原型创建，同字面量方式创建的结果基本一致



Object.create(obj,{a:{value:1,configurable:true,writable:true}})底层实现原理：

function create(obj){

var tar={};

if(obj) tar.\_\_proto\_\_=obj;

}

## 执行上下文、作用域、作用域链

**执行上下文（环境）：**定义了变量或函数有权访问的其他数据，决定了它们各自的行为。每个执行环境都有一个与之关联的**变量对象**和一个**作用域链**，环境中定义的所有变量和函数都保存在其变量对象中。执行环境分为：

1. 全局执行环境

最外围的一个执行环境，其变量对象就是全局活动对象（window activation object），全局执行环境直到应用程序退出——例如关闭网页或浏览器——时才会被销毁。

1. 函数执行环境

每个函数都有自己的执行环境。当执行流进入一个函数时，函数环境就会被推入一个环境栈中。当函数执行完之后，栈将其环境弹出，把控制权返回给之前的执行环境。函数执行环境的变量对象是该函数的活动对象（activation object）。

**执行上下文的建立过程：**

1. 建立阶段（调用函数，尚未执行函数体内的代码）
   1. 建立变量对象。
      1. 建立arguments对象。检查当前上下文中的参数，建立arguments对象下的属性以及属性值。
      2. 检查当前上下文中的函数声明。每检查到一个函数声明，就在变量对象下面用函数名建立一个属性，属性值是该函数在内存中的一个引用地址。如果该函数名已经存在，将其覆盖。
      3. 检查当前上下文中的变量声明。每检查到一个变量声明，就在变量对象中用该变量名建立一个属性，属性值为undefined，如果该变量名已经存在，直接跳过
   2. 建立作用域链

函数定义时，就获取了[[scope]]指针（不可见），该指针指向函数定义时的作用域链（例如funcA在funB中定义，[[scope]]指向funcB的作用域）。将函数的[[scope]]指针指向的作用域与函数的变量对象一起构成作用域链。

* 1. 确定this的值

1. 执行阶段
   1. 变量赋值、函数引用、执行其他代码

**作用域**：有权访问到的变量和函数的集合。其实就是作用域链上的所有的变量对象（相对于当前对象而言）。

**作用域链：**一个链表，每个指针指向一个变量对象。每个执行环境都有一个与之关联的作用域链，每个执行环境的作用域链的前端，始终是该执行环境的变量对象。对于全局执行环境就相当于window对象，对于函数执行环境就相当于该函数的活动对象。

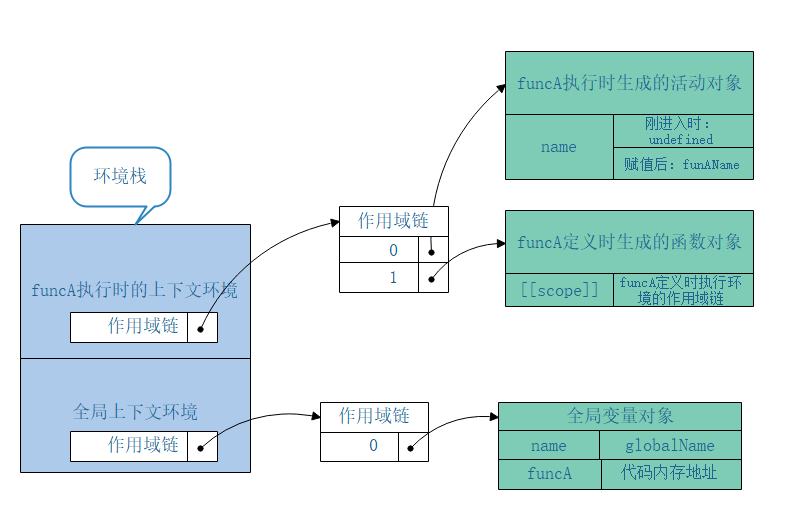
**延长作用域链**：（1）try-catch（2）with

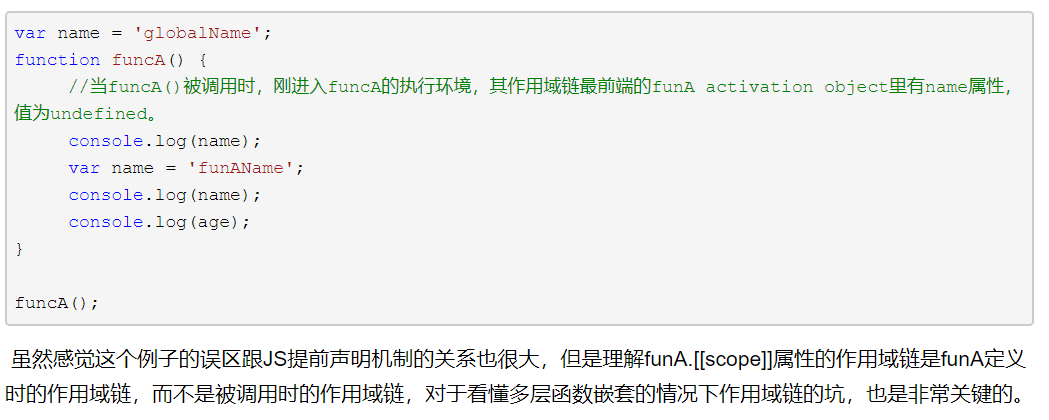
**执行上下文和作用域的区别**：执行上下文在运行时确定，随时可能改变；作用域在定义时确定，永远不会改变。

**函数对象**：在一个函数定义的时候，会创建一个这个函数对象的[[scope]]属性（内部属性,只有JS引擎可以访问, 但FireFox的几个引擎（SpiderMonkey和Rhino）提供了私有属性\_\_parent\_\_来访问它)，并将这个[[scope]]属性指向定义它的作用域链上。例如函数funcA在funcB中定义，则funcA的[[scope]]指向funcB的作用域链。

**活动对象：**在一个函数对象被调用的时候，会创建一个活动对象，首先将该函数的每个形参和实参，都添加为该活动对象的属性和值；将该函数体内显示声明的变量和函数，也添加为该活动的的属性（在刚进入该函数执行环境时，未赋值，所以值为undefined，这个是JS的提前声明机制）。

然后将这个活动对象做为该函数执行环境的作用域链的最前端，并将这个函数对象的[[scope]]属性里作用域链接入到该函数执行环境作用域链的后端。





## Promise.all和Promise.race的区别？

Promise.all 把多个promise实例当成一个promise实例,当这些实例的状态都发生改变时才会返回一个新的promise实例，才会执行then方法。

Promise.race 只要该数组中的 Promise 对象的状态发生变化（无论是resolve还是reject）该方法都会返回。

## HTML DOM Location对象

Location 对象包含有关当前 URL 的信息。

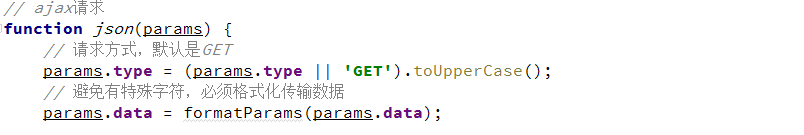
#### Location 对象属性

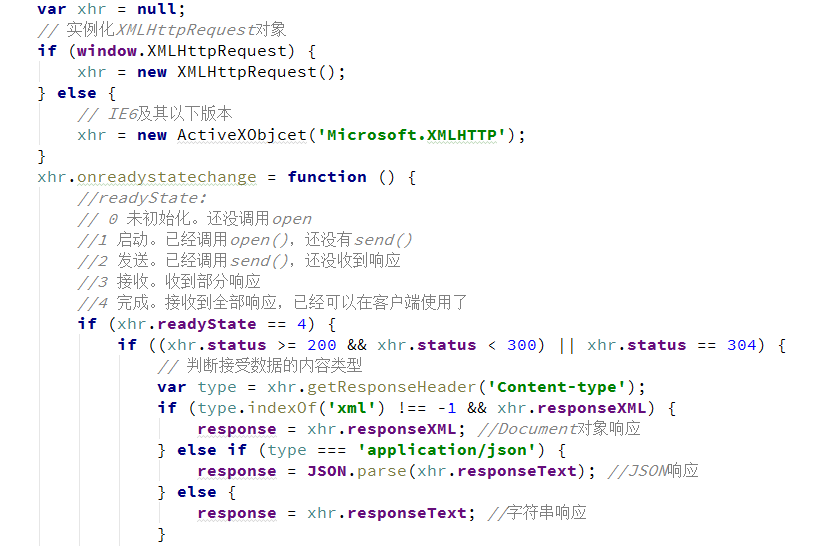
|  |  |
| --- | --- |
| **属性** | **描述** |
| [hash](http://www.w3school.com.cn/jsref/prop_loc_hash.asp) | 设置或返回从井号 (#) 开始的 URL（锚）。 |
| [host](http://www.w3school.com.cn/jsref/prop_loc_host.asp) | 设置或返回主机名和当前 URL 的端口号。 |
| [hostname](http://www.w3school.com.cn/jsref/prop_loc_hostname.asp) | 设置或返回当前 URL 的主机名。 |
| [href](http://www.w3school.com.cn/jsref/prop_loc_href.asp) | 设置或返回完整的 URL。 |
| [pathname](http://www.w3school.com.cn/jsref/prop_loc_pathname.asp) | 设置或返回当前 URL 的路径部分。 |
| [port](http://www.w3school.com.cn/jsref/prop_loc_port.asp) | 设置或返回当前 URL 的端口号。 |
| [protocol](http://www.w3school.com.cn/jsref/prop_loc_protocol.asp) | 设置或返回当前 URL 的协议。 |
| [search](http://www.w3school.com.cn/jsref/prop_loc_search.asp) | 设置或返回从问号 (?) 开始的 URL（查询部分）。 |

#### Location 对象方法

|  |  |
| --- | --- |
| **属性** | **描述** |
| [assign()](http://www.w3school.com.cn/jsref/met_loc_assign.asp) | 加载新的文档。 |
| [reload()](http://www.w3school.com.cn/jsref/met_loc_reload.asp) | 重新加载当前文档。 |
| [replace()](http://www.w3school.com.cn/jsref/met_loc_replace.asp) | 用新的文档替换当前文档。 |

## 原生AJAX实现







## 强类型、弱类型、静态语言与动态语言

强类型：偏向于不容忍隐式类型转换。譬如说haskell的int就不能变成double

弱类型：偏向于容忍隐式类型转换。譬如说C语言的int可以变成double

静态类型：编译的时候就知道每一个变量的类型，因为类型错误而不能做的事情是语法错误

动态类型：编译的时候不知道每一个变量的类型，因为类型错误而不能做的事情是运行时错误。譬如说你不能对一个数字a写a[10]当数组用。

## 参数传递—传值！

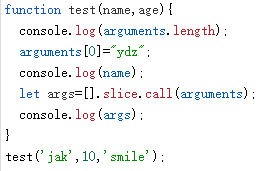
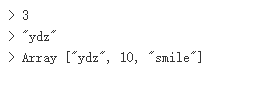
1. 基本类型：直接将基本类型的值复制给参数
2. 引用类型：将引用指针的值复制给参数

## Arguments对象

arguments 是一个对应于传递给函数的参数的类数组对象。

Arguments对象是所有非箭头函数中可用的局部变量。

Arguments对象中的每一个数据和传入的参数一一对应，可修改arguments对象，其对应的形参的值也会变化。

代码解析：

1. arguments对象表示实参数据，可用于传递可变数量的参数，使用arguments.length确定传递给函数的参数的实际数量；
2. 使用索引方式取arguments中的数据，arguments[0]
3. 可以设置参数，arguments[0]="ydz"; 相应位置形参对应的值会变化

## Array对象

1. Array.from（arrayLike[, mapFn[, thisArg]]）从一个类数组或可迭代对象中创建一个新的数组实例
   1. Arraylike：想要转换成数组的伪数组对象或可迭代对象。
   2. mapFn：如果指定了该参数，新数组中的每个元素会执行该回调函数
   3. thisArg：执行回调函数mapFn时this对象