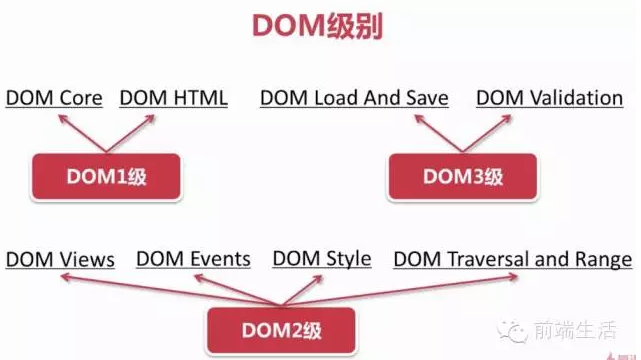
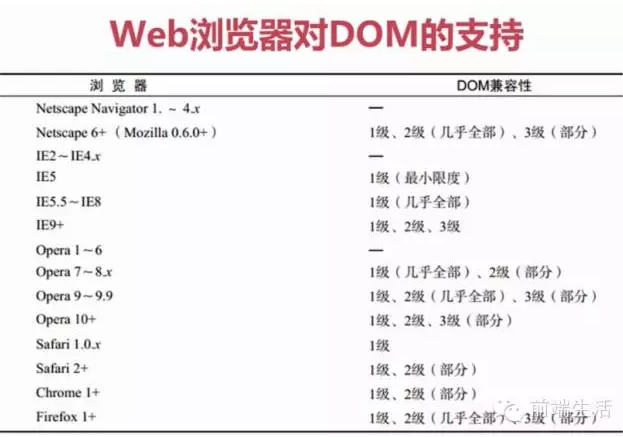
今天继续整理笔记。权当是复习一下原生js的一些方法。

首先，我们看一下DOM级别和兼容性：

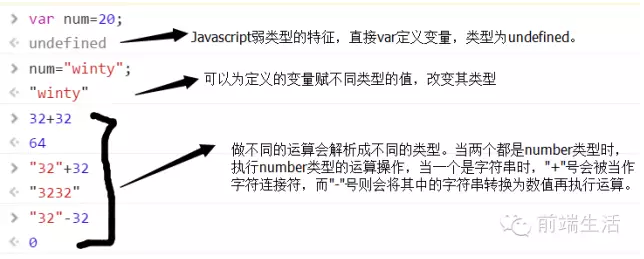


之前好像在某本上看到说DOM有0级，实际上，DOM0级标准是不存在的。DOM有1、2、3三个级别。DOM1级由两个模块组成（DOM Core和DOM HTML），其中，DOM核心规定的是如何映射基于XML的文档结构，以便简化对文档中任意部分的访问和操作。DOM HTML模块则在DOM核心的基础上加以扩展，添加了针对HTML的对象和方法。DOM2级在原来DOM的基础上又扩充了视图和用户界面事件、范围、遍历（迭代DOM文档的方法）等细分模块，而且通过对象接口增加了对CSS（Cascading Style Sheets，层叠样式表）的支持。DOM3级则进一步扩展了DOM，引入了以统一方式加载和保存文档的方法–在DOM加载和保存（DOM Load and Save）模块中定义；新增了验证文档的方法–在DOM验证（DOM Validation）模块中定义。



一、六种数据类型

Javascript是一种弱数据类型，这种弱数据类型有什么特征呢？看看下面的例子：



JavaScript一共有6种数据类型，其中五种原始类型（基本类型），一种对象类型（引用类型）。

引用类型：

object，包括Object类型、Array类型、Date类型、RegExp类型、Function类型。

基本类型：

number、string、boolean、null、undefined。

类型的转换和检测：

（1）隐式转换：巧用+、-、==规则转换类型

num-0 这样可以将字符串转化为数字

 例如：“17”-7     //10

num+ " "  这样可以将数字转换为字符串

例如：“17”+7      //177

类型相同，用===检测是否相等，类型不同，尝试类型转换和比较,部分规则为：

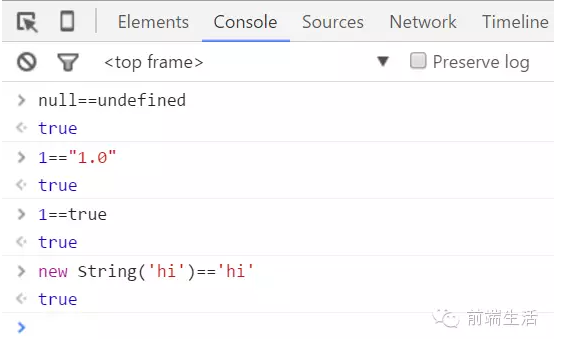
null==undefined   相等

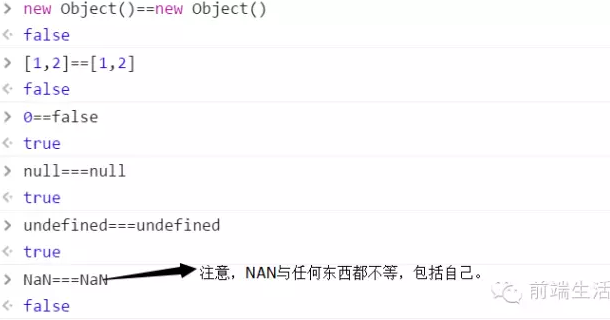
number==string 转number

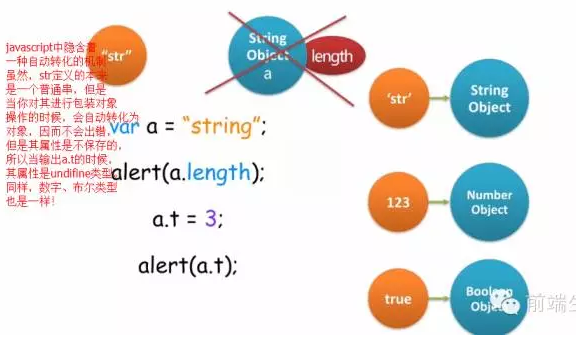
boolean==？转number

object==number|string 尝试对象转为基本类型

其他：false

例如：  
注意：===是绝对等的意思，只有类型和数值都相等才会成立。



更多：

（2）类型检测

检测方法：typeof、instanceof、Object.prototype.toString、constructor、duck type

typeof和instanceof比较常用，其中typeof比较适合函数对象和基本类型的判断，instanceof常用于判断对象类型，其判断专责是基于原型链判断的。例如：

typeof 100   //"number"

typeof ture   //"boolean"

typeof(undefined)  //"undefined"

typeof new Object()  //"object"

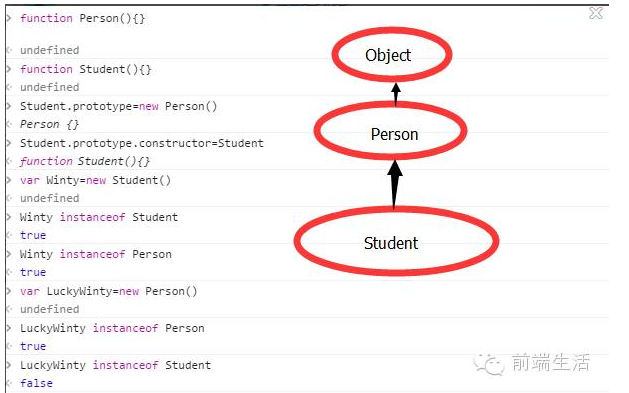
typeof [1,2]  //"object"

typeof NAN   //"undefined"

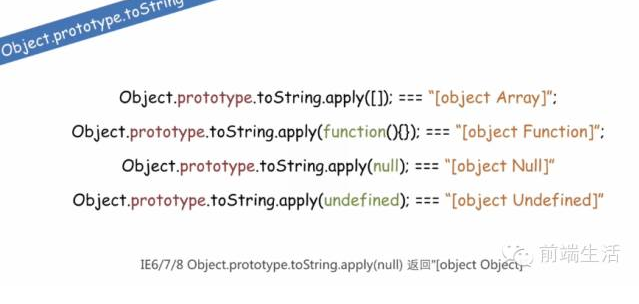
typeof function  //“function”

typeof null //"object"

备注：历史上曾经把null类型===''null''，但是造成许多网站没有办法正常访问，后来就决定null返回的是Object对象。所以typeof null==="object"  //true

  
注意：不同window或iframe间的对象类型检测不能使用instanceof！

更多：





二、表达式和运算符

表达式类型：原始表达式、初始化表达式、函数表达式、属性访问表达式、调用表达式、对象创建表达式。

初始化表达式：new Array(1,2)、[1,2]等

属性访问表达式：var o={x:1}; o.x;o['x]

调用表达式：func()

对象创建表达式：new Object;

运算符：逗号“，”、delet、in、new、this

逗号用法：a,b

如：var val=(1,2,3)

delet用法:  delet obj.x

如：var obj={x:1};

       obj.x;   //1

       delet obj.x

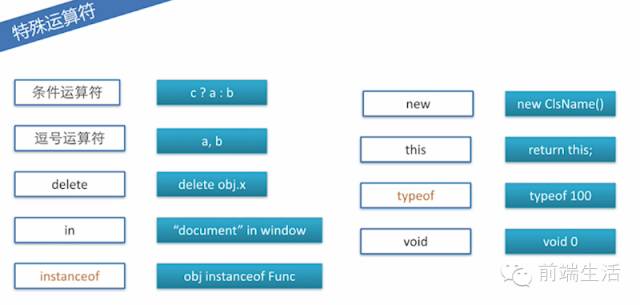
       obj.x;   //undefined

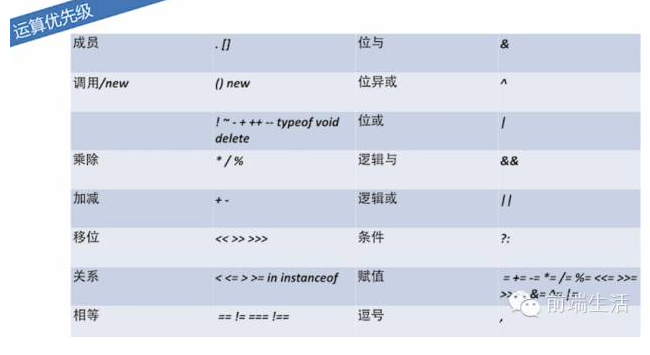
注意：从IE9开始，新增的configurable标签，当这个的值为true的元素才可以被delet。创建对象时，属性默认的configurable的值是true。

in的用法：window.x=1;

                 'x' in window;     //true

更多：

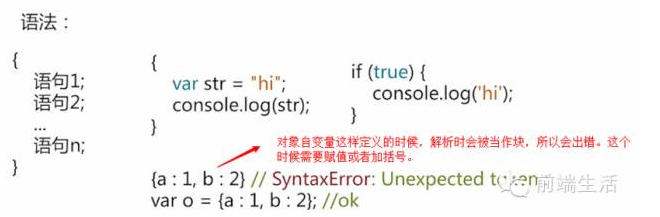




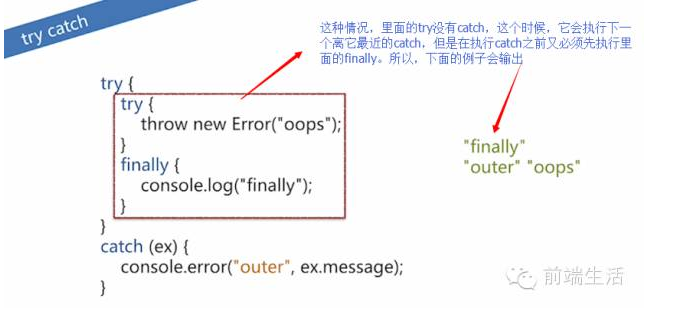
三、语句

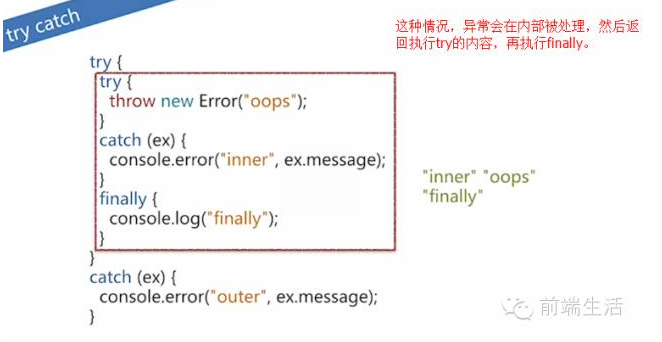
语句分别有:block、var、try catch、function、for...in、with

block:

  
注意：js中没有块级作用域。所以在for循环中定义的变量的作用域会被扩大。即在for循环之后仍然可以访问for循环里面定义的变量。

try catch:





function:

函数声明：

fd();  //true

function fd(){...}

函数表达式：

var fe=function(){...}

注意：函数声明会在执行之前生效，所以在函数之前调用该函数也可以。但是函数表达式不会在执行之前生效，所以不能提前。

for...in:

var p;

var obj={x:1,y:2}

for(p in obj){....}

注意：

for...in的遍历顺序是不确定的，只是会把该遍历的每一个元素都遍历输出，但是不一定按照定义时候的顺序。而且，当enumerable为false时，不能用for...in遍历。另外，for...in遍历对象属性时受原型链影响。

严格模式：

严格模式是一种特殊的运行模式，它修复了部分语言上的不足，提供更安强的错误检查，并增强安全性。严格模式是向上兼容的。

严格模式和普通模式的区别：

①不允许使用with，若使用会报错（语法错误）

②不允许未声明的变量被赋值

③arguments变为参数的静态副本

④delete参数、函数名报错  
⑤delete不可配置的属性也会报错

⑥对象字面重复属性名报错

⑦禁止八进制字面量

⑧eval、arguments变为关键字，不能作为变量、函数名

四、对象

对象中包含一系列属性，这些属性是无序的。每一个属性都有一个字符串key和对应的value。

数据属性：configurable、enumerable、writable、value

访问器属性：configurable、enumerable、get、set，访问器属性不能直接定义，必须使用Object。definedProperty()来定义。



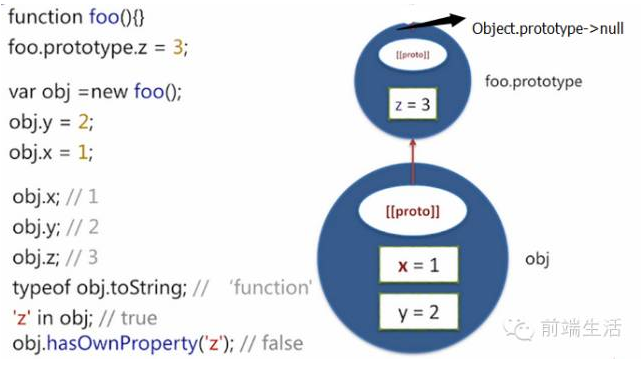
**创建对象**

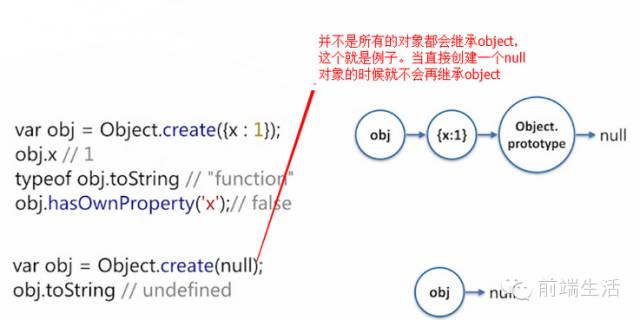
**第一种是使用new操作符后跟Object构造函数**，

如下所示：

var person = new Object(); person.name = "Nicholas"; person.age = 29;

这种方法要注意原型链的问题：

  
特别注意，并不是所有的对象都会继承object：



**另一种方式是使用对象字面量表示法。**

var person = { name : "Nicholas", age : 29 };

另外，使用对象字面量语法时，如果留空其花括号，则可以定义只包含默认属性和方法的对象，如：

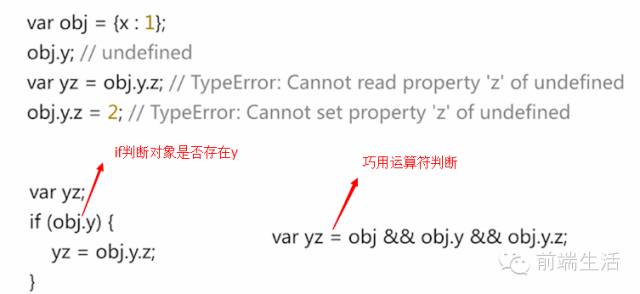
 var person = {}; //与 new Object()相同

person.name = "Nicholas"; person.age = 29;

**属性操作**

属性操作包括：读写对象属性、属性异常、删除属性、检测属性、枚举属性

属性异常：

  
删除属性：



检测属性：

用hasOwnProperty()方法来判断某个对象是否有某个属性，用propertyIsEnumerable()方法判断某个属性是否可以枚举。

**extensible标签：**



**设计模式**

①工厂模式

function createPerson(name, age, job){

     var o = new Object();

     o.name = name;

     o.age = age;

     o.job = job;

     o.sayName = function(){

              alert(this.name);

      };

     return o;

 }

var person1 = createPerson("winty", 29, "Software Engineer");

var person2 = createPerson("LuckyWinty", 27, "Student");

②构造函数模式

function Person(name, age, job){

        this.name = name;

        this.age = age;

        this.job = job;

        this.sayName = function(){

                    alert(this.name);

         };

 }

var person1 = new Person("Winty", 29, "Software Engineer");

var person2 = new Person("LuckyWinty", 27, "Student");

③原型模式

function Person(){ }

Person.prototype.name = "Winty";

Person.prototype.age = 29;

Person.prototype.job = "Software Engineer";

Person.prototype.sayName = function(){

        alert(this.name);

        };

var person1 = new Person();

person1.sayName(); //"Winty"

var person2 = new Person();

person2.sayName(); //"Winty"

alert(person1.sayName == person2.sayName); //true

④组合使用构造函数模式和原型模式

function Person(name, age, job){

              this.name = name;

              this.age = age;

              this.job = job;

              this.friends = ["Shelby", "Court"];

}

Person.prototype = {

            constructor : Person,

            sayName : function(){

                              alert(this.name);

            }

}

var person1 = new Person("winty", 29, "Software Engineer");

var person2 = new Person("LuckyWinty", 27, "Student");

person1.friends.push("Van");

alert(person1.friends); //"Shelby,Count,Van"

alert(person2.friends); //"Shelby,Count"

alert(person1.friends === person2.friends); //false

alert(person1.sayName === person2.sayName); //true

⑤动态原型模式

function Person(name, age, job){

              this.name = name;

              this.age = age;

              this.job = job;

              if (typeof this.sayName != "function"){

               Person.prototype.sayName = function(){

                            alert(this.name);

               };

             }

}

var friend = new Person("Winty", 29, "Software Engineer");

friend.sayName();

⑥：寄生构造函数模式

function Person(name, age, job){

         var o = new Object();

         o.name = name;

         o.age = age;

         o.job = job;

         o.sayName = function(){

                  alert(this.name);

         };

     return o;

}

var friend = new Person("Winty", 29, "Software Engineer");

friend.sayName(); //"Winty"

⑦稳妥构造函数模式

所谓稳妥对象，指的是没有公共属性，而且其方法也不引用this的对象。稳妥对象最适合在一些安全的环境中（这些环境中会禁止使用this和new），或者在防止数据被其他应用程序（如Mashup程序）改动时使用。稳妥构造函数遵循与寄生构造函数类似的模式，但有两点不同：一是新创建对象的实例方法不引用this；二是不使用new操作符调用构造函数。按照稳妥构造函数的要求，可以将前面的Person构造函数重写如下。

function Person(name, age, job){

 var o = new Object();

 o.sayName = function(){

    alert(name);

 };

 return o;

}

五、数组

  数组是数据的有序列表，每个元素在数组中都有数字位置编号，也就是索引。JS中的数组是弱类型，每一项都可以保存任何类型的数据。

**创建数组**

①使用Array构造函数

var arr=new Array();

var array=new Array("red","green","blue");   //传入数组应该包含的项

var a=new Array(100);    //直接指定数组长度

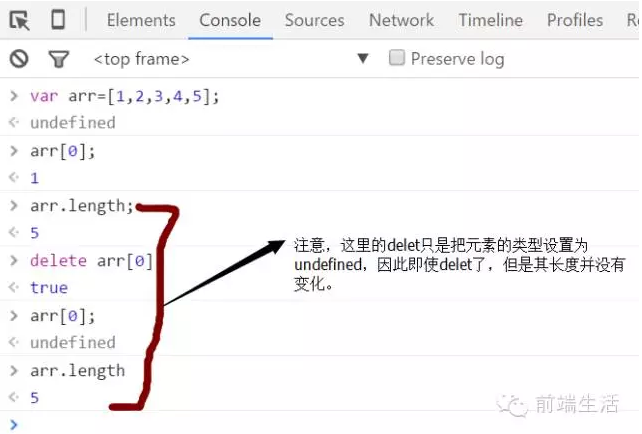
注意：数组的长度最大为4294967295

  
②使用数组字面量表示法

var Person=['winty','lzh','LuckyWinty']   //创建一个3个字符串的数组

var test=[1,2,]                        //不要这样，这样会创建一个包含2或3项的数组

**数组元素读写**

  
**数组元素操作**

①栈方法

push()方法可以接收任意数量的参数，把它们逐个添加到数组末尾，并返回修改后数组的长度。而pop()方法则从数组末尾移除最后一项，减少数组的length值，然后返回移除的项。这样结合使用可以实现类似栈的行为，请看下面的例子：

var colors = new Array();

var count = colors.push("red", "green");

alert(count); //2

count = colors.push("black");

 alert(count); //3

var item = colors.pop();

alert(item); //"black"

alert(colors.length); //2

②队列方法

由于push()是向数组末端添加项的方法，因此要模拟队列只需一个从数组前端取得项的方法。实现这一操作的数组方法就是shift()，它能够移除数组中的第一个项并返回该项，同时将数组长度减1。结合使用shift()和push()方法，可以像使用队列一样使用数组。

var colors = new Array(); //创建一个数组

var count = colors.push("red", "green"); //推入两项

alert(count); //2

count = colors.push("black");

 alert(count); //3

var item = colors.shift(); //取得第一项 ú

alert(item); //"red"

alert(colors.length); //2

ECMAScript还为数组提供了一个unshift()方法。顾名思义，unshift()与shift()的用途相反：它能在数组前端添加任意个项并返回新数组的长度。因此，同时使用 unshift()和 pop()方法，可以从相反的方向来模拟队列，即在数组的前端添加项，从数组末端移除项，如下面的例子所示。

var colors = new Array();

var count = colors.unshift("red", "green");

alert(count); //2

count = colors.unshift("black");

alert(count); //3

var item = colors.pop();

alert(item); //"green"

alert(colors.length); //2

**重排序方法**

数组中已经存在两个可以直接用来重排序的方法：reverse()和 sort()。

var values = [1, 2, 3, 4, 5];

values.reverse();

alert(values); //5,4,3,2,1

但是sort()方法是比较字符串以确定如何排序的，即使数组中的每一项都是数值，如：

var values = [0, 1, 5, 10, 15];

values.sort();

alert(values); //0,1,10,15,5

不用说，这种排序方式在很多情况下都不是最佳方案。因此sort()方法可以接收一个比较函数作为参数，以便我们指定哪个值位于哪个值的前面。 比较函数接收两个参数，如果第一个参数应该位于第二个之前则返回一个负数，如果两个参数相等则返回0，如果第一个参数应该位于第二个之后则返回一个正数。以下就是一个简单的比较函数：

function compare(value1, value2) {

             if (value1 < value2) {

                   return -1;

             } else if (value1 > value2) {

                   return 1;

                      } else {

                             return 0;

                         }

 }

这个比较函数可以适用于大多数数据类型，只要将其作为参数传递给sort()方法即可，如下面这个例子所示。

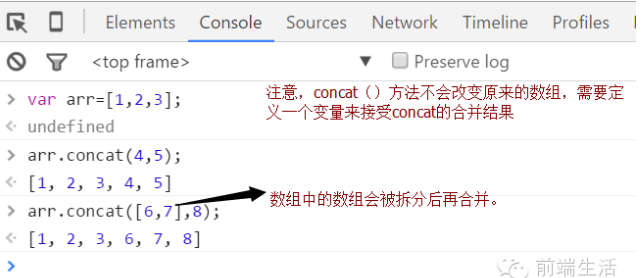
var values = [0, 1, 5, 10, 15];

values.sort(compare);

alert(values); //0,1,5,10,15

**操作方法**

合并数组：concat()



**slice()方法，有三种用法。**

删除：可以删除任意数量的项，只需指定 2个参数：要删除的第一项的位置和要删除的项数。

插入：可以向指定位置插入任意数量的项，只需提供3个参数：起始位置、0（要删除的项数）和要插入的项。如果要插入多个项，可以再传入第四、第五，以至任意多个项。

替换：可以向指定位置插入任意数量的项，且同时删除任意数量的项，只需指定 3个参数：起始位置、要删除的项数和要插入的任意数量的项。插入的项数不必与删除的项数相等。

例如：

var colors = ["red", "green", "blue"];

var removed = colors.splice(0,1); // 删除第一项

alert(colors); // green,blue

alert(removed); // red，返回的数组中只包含一项

removed = colors.splice(1, 0, "yellow", "orange"); // 从位置1 开始插入两项

alert(colors); // green,yellow,orange,blue

alert(removed); // 返回的是一个空数组

removed = colors.splice(1, 1, "red", "purple"); // 插入两项，删除一项

alert(colors); // green,red,purple,orange,blue

alert(removed); // yellow，返回的数组中只包含一项

**join()方法将数组转为字符串：**

var arr=[1,2,3]

arr.join("-");    //  "1\_2\_3"

**位置方法**



**迭代方法**

every()：对数组中的每一项运行给定函数，如果该函数对每一项都返回true，则返回true。

 filter()：对数组中的每一项运行给定函数，返回该函数会返回true的项组成的数组。

forEach()：对数组中的每一项运行给定函数。这个方法没有返回值。

map()：对数组中的每一项运行给定函数，返回每次函数调用的结果组成的数组。

some()：对数组中的每一项运行给定函数，如果该函数对任一项返回true，则返回true。

以上方法都不会修改数组中的包含的值。

例子：

var arr=[1,2,3,4,5];

arr.forEach(function(x,index,a){

                          console.log(x+"---"+index+"----"+(a===arr));

});

//1---0----true

//2---1----true

//3---2----true

//4---3----true

//5---4----true

arr.map(function(x){

  return x+10;

});    //[11,12,13,14,15]

arr;   //[1,2,3,4,5]     原数组不会被修改

**归并方法**

reduce()和 reduceRight()。这两个方法都会迭代数组的所有项，然后构建一个最终返回的值。其中，reduce()方法从数组的第一项开始，逐个遍历到最后。而reduceRight()则从数组的最后一项开始，向前遍历到第一项。

给reduce()和 reduceRight()的函数接收 4个参数：前一个值、当前值、项的索引和数组对象。

这个函数返回的任何值都会作为第一个参数自动传给下一项。第一次迭代发生在数组的第二项上，因此第一个参数是数组的第一项，第二个参数就是数组的第二项。 使用reduce()方法可以执行求数组中所有值之和的操作，比如：

var values = [1,2,3,4,5];

var sum = values.reduce(function(prev, cur, index, array){

                  return prev + cur;

               });

alert(sum); //15

var sum1=arr.reduce(function(x,y){

                   return x+y

},100);

alert(sum1)// 115,有参数传入，所以会把参数也加上。

**数组和一般对象的比较：**

相同：两者都可以继承，数组是对象，对象不一定是数组，都可以当作对象添加删除属性。

不同：数组自动更新length，按索引访问数组常常比访问一般对象属性明显迅速。数组对象继承Array.prototype上的大量数组操作方法。

六、函数

函数是一块JavaScript代码，被定义一次，但可执行和调用多次。JS中的函数也是对象。所以JS函数可以像其它对象那样操作和传递。所以我们也常叫JS中的函数为函数对象。

调用方式

直接调用：foo();

对象方法：o.method();

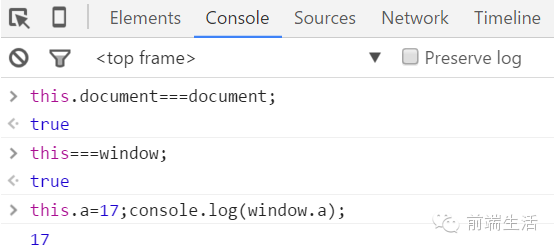
构造器：new Foo();

call/apply/bian:func.call(o);  
函数声明、表达式、构造器比较：

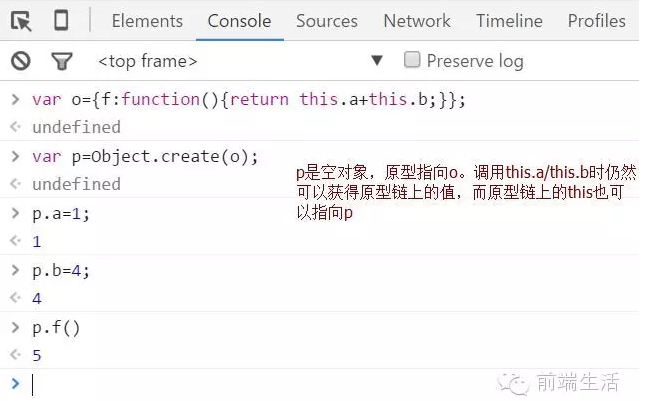


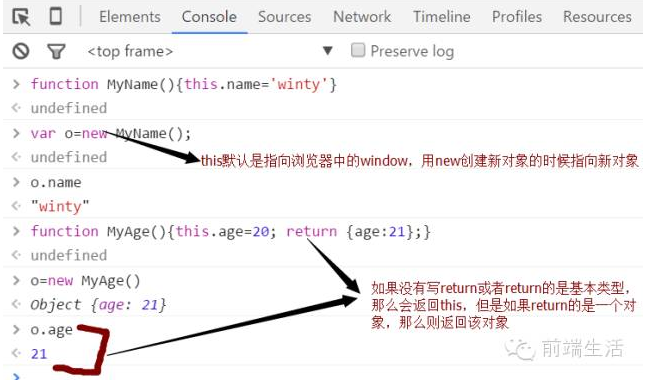
this

①全局作用域上的this指向的就是this

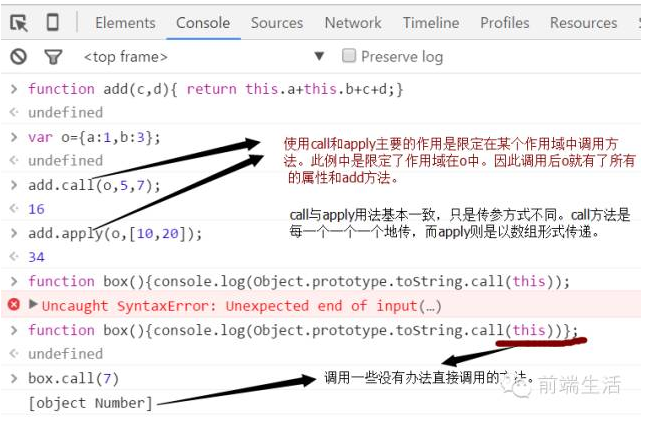
  
②一般函数的this

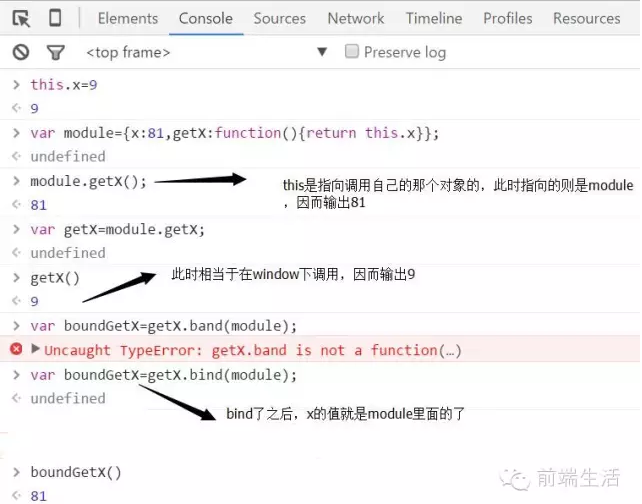
  
③对象原型链上的this

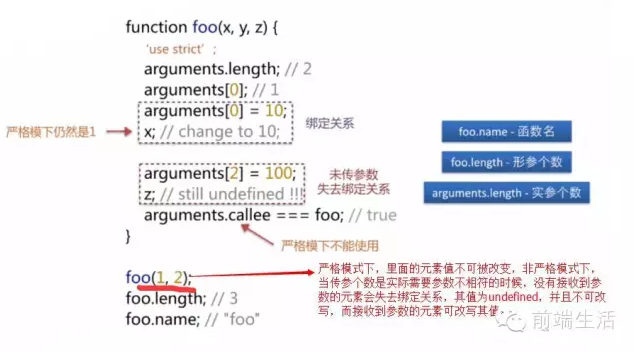
  
④构造器上的this



⑤call/apply方法与this

  
⑥bind方法与this(IE9+才会支持该方法)

  
函数属性&arguments



函数闭包

闭包是指有权访问另一个函数作用域中的变量的函数。创建闭包的常见方式，就是在一个函数内部创建另一个函数。

**闭包与变量**

作用域链的这种配置机制引出了一个值得注意的副作用，即闭包只能取得包含函数中任何变量的最后一个值。别忘了闭包所保存的是整个变量对象，而不是某个特殊的变量。

下面这个例子可以清晰地说明这个问题。

function createFunctions(){

     var result = new Array();

     for (var i=0; i < 10; i++){

           result[i] = function(){

                      return i;

           };

      }

    return result;

}

这个函数会返回一个函数数组。表面上看，似乎每个函数都应该返自己的索引值，即位置0的函数返回0，位置1的函数返回1，以此类推。但实际上，每个函数都返回10。因为每个函数的作用域链中都保存着 createFunctions()函数的活动对象，所以它们引用的都是同一个变量 i。当createFunctions()函数返回后，变量i 的值是10，此时每个函数都引用着保存变量i的同一个变量对象，所以在每个函数内部i的值都是10。

但是，我们可以通过创建另一个匿名函数强制让闭包的行为符合预期，如下所示。

function createFunctions(){

        var result = new Array();

        for (var i=0; i < 10; i++){

              result[i] = function(num){

                              return function(){

                                         return num;

                              };

              }(i);

         }

      return result;

 }

在这个版本中，我们没有直接把闭包赋值给数组，而是定义了一个匿名函数，并将立即执行该匿名函数的结果赋给数组。这里的匿名函数有一个参数num，也就是最终的函数要返回的值。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。

**可以利用闭包的这个特性，封装函数，模仿块级作用域。**

function(){

//这里是块级作用域

})（）；

无论在什么地方，只要临时需要一些变量，就可以使用私有作用域，例如：

function outputNumbers(count){

     (function () {

        for (var i=0; i < count; i++){

             alert(i);

         }

})();

     alert(i);   //导致一个错误！

 }

在这个重写后的outputNumbers()函数中，我们在for循环外部插入了一个私有作用域。在匿名函数中定义的任何变量，都会在执行结束时被销毁。因此，变量i只能在循环中使用，使用后即被销毁。而在私有作用域中能够访问变量count，是因为这个匿名函数是一个闭包，它能够访问包含作用域中的所有变量。 这种技术经常在全局作用域中被用在函数外部，从而限制向全局作用域中添加过多的变量和函数。一般来说，我们都应该尽量少向全局作用域中添加变量和函数。在一个由很多开发人员共同参与的大型应用程序中，过多的全局变量和函数很容易导致命名冲突。而通过创建私有作用域，每个开发人员既可以使用自己的变量，又不必担心搞乱全局作用域。例如：

 (function(){

     var now = new Date();

     if (now.getMonth() == 0 && now.getDate() == 1){

        alert("Happy new year!");

     }

 })();

把上面这段代码放在全局作用域中，可以用来确定哪一天是1月1日；如果到了这一天，就会向用户显示一条祝贺新年的消息。其中的变量now现在是匿名函数中的局部变量，而我们不必在全局作用域中创建它。

七、oop

JS解释器按照如下顺序找到我们定义的函数和变量：

1.函数参数（若未传入，初始化该参数为undefined）  
2.函数声明（若发生命名冲突，会覆盖）——函数声明提升的原因  
3.变量声明（初始化变量值undefined,若发生命名冲突，会忽略）  
在全局作用域下，函数声明和变量声明会被前置到全局执行上下文（执行环境）中。  
在浏览器环境下，当this表示全局对象时，this就指window对象  
匿名函数，加上括号就变成了函数表达式，再加个括号，就变成了立即执行函数，再在前面加个！号可以将函数声明变为函数表达式，防止函数被前置执行，留下最后那个括号...  
同一个函数，被调用多次的话，每次调用函数时都会有独立的执行上下文，每个执行上下文环境都会记录各自的变量参数等信息。

继承

ECMAScript只支持实现继承，而且其实现继承主要是依靠原型链来实现的。

 实现原型链有一种基本模式，其代码大致如下。

function SuperType(){

        this.property = true;

}

SuperType.prototype.getSuperValue = function(){

          return this.property;

};

function SubType(){

        this.subproperty = false;

}

//继承了SuperType

SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function (){

               return this.subproperty;

 };

var instance = new SubType();

alert(instance.getSuperValue()); //true

借用构造函数

在解决原型中包含引用类型值所带来问题的过程中，开发人员开始使用一种叫做借用构造函数（constructor stealing）的技术（有时候也叫做伪造对象或经典继承）。这种技术的基本思想相当简单，即在子类型构造函数的内部调用超类型构造函数。如下所示：

function SuperType(){

              this.colors = ["red", "blue", "green"];

}

function SubType(){

 //继承了 SuperType

 SuperType.call(this);

}

var instance1 = new SubType();

instance1.colors.push("black");

alert(instance1.colors); //"red,blue,green,black"

var instance2 = new SubType();

alert(instance2.colors); //"red,blue,green"

组合继承

既通过在原型上定义方法实现了函数复用，又能够保证每个实例都有它自己的属性。下面来看一个例子。

function SuperType(name){

              this.name = name;

              this.colors = ["red", "blue", "green"];

}

SuperType.prototype.sayName = function(){

        alert(this.name);

};

function SubType(name, age){

 //继承属性

 SuperType.call(this, name);

 this.age = age;

}

 //继承方法

SubType.prototype = new SuperType();

SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){

               alert(this.age);

};

var instance1 = new SubType("Nicholas", 29);

instance1.colors.push("black");

alert(instance1.colors); //"red,blue,green,black"

instance1.sayName(); //"Nicholas";

instance1.sayAge(); //29

var instance2 = new SubType("Greg", 27);

alert(instance2.colors); //"red,blue,green"

instance2.sayName(); //"Greg";

instance2.sayAge(); //27

原型式继承

var person = { name: "Nicholas", friends: ["Shelby", "Court", "Van"] };

var anotherPerson = object(person);

anotherPerson.name = "Greg";

anotherPerson.friends.push("Rob");

var yetAnotherPerson = object(person);

yetAnotherPerson.name = "Linda";

yetAnotherPerson.friends.push("Barbie");

alert(person.friends); //"Shelby,Court,Van,Rob,Barbie"

克罗克福德主张的这种原型式继承，要求你必须有一个对象可以作为另一个对象的基础。如果有这么一个对象的话，可以把它传递给object()函数，然后再根据具体需求对得到的对象加以修改即可。在这个例子中，可以作为另一个对象基础的是person对象，于是我们把它传入到object()函数中，然后该函数就会返回一个新对象。这个新对象将person作为原型，所以它的原型中就包含一个基本类型值属性和一个引用类型值属性。这意味着person.friends不仅属于person所有，而且也会被anotherPerson以及yetAnotherPerson 共享。实际上，这就相当于又创建了person对象的两个副本。

ECMAScript 5 通过新增Object.create()方法规范化了原型式继承。这个方法接收两个参数：一个用作新对象原型的对象和（可选的）一个为新对象定义额外属性的对象。在传入一个参数的情况下，Object.create()与object()方法的行为相同。

var person = { name: "Nicholas", friends: ["Shelby", "Court", "Van"] };

var anotherPerson = Object.create(person);

anotherPerson.name = "Greg";

anotherPerson.friends.push("Rob");

var yetAnotherPerson = Object.create(person);

yetAnotherPerson.name = "Linda";

yetAnotherPerson.friends.push("Barbie");

alert(person.friends); //"Shelby,Court,Van,Rob,Barbie"

Object.create()方法的第二个参数与Object.defineProperties()方法的第二个参数格式相同：每个属性都是通过自己的描述符定义的。以这种方式指定的任何属性都会覆盖原型对象上的同名属性。

寄生式继承

即创建一个仅用于封装继承过程的函数，该函数在内部以某种方式来增强对象，最后再像真地是它做了所有工作一样返回对象。以下代码示范了寄生式继承模式。

function createAnother(original){

              var clone = object(original); //通过调用函数创建一个新对象

              clone.sayHi = function(){ //以某种方式来增强这个对象

                        alert("hi");

              };

     return clone;

 } / /返回这个对象

在这个例子中，createAnother()函数接收了一个参数，也就是将要作为新对象基础的对象。然后，把这个对象（original）传递给object()函数，将返回的结果赋值给clone。再为clone对象添加一个新方法sayHi()，最后返回clone对象。

可以像下面这样来使用createAnother()函数：

var person = { name: "Nicholas", friends: ["Shelby", "Court", "Van"] };

var anotherPerson = createAnother(person);

anotherPerson.sayHi(); //"hi"

这个例子中的代码基于person返回了一个新对象——anotherPerson。新对象不仅具有person的所有属性和方法，而且还有自己的sayHi()方法。

寄生组合式继承

寄生组合式继承的基本模式如下所示。

function inheritPrototype(subType, superType){

              var prototype = object(superType.prototype); //创建对象

              prototype.constructor = subType; //增强对象

              subType.prototype = prototype; //指定对象

 }

这个示例中的 inheritPrototype()函数实现了寄生组合式继承的最简单形式。这个函数接收两个参数：子类型构造函数和超类型构造函数。在函数内部，第一步是创建超类型原型的一个副本。第二步是为创建的副本添加constructor属性，从而弥补因重写原型而失去的默认的constructor属性。最后一步，将新创建的对象（即副本）赋值给子类型的原型。这样，我们就可以用调用 inherit- Prototype()函数的语句，去替换前面例子中为子类型原型赋值的语句了。

function SuperType(name){

              this.name = name;

              this.colors = ["red", "blue", "green"];

}

SuperType.prototype.sayName = function(){

                  alert(this.name);

};

function SubType(name, age){

              SuperType.call(this, name);

              this.age = age;

}

inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function(){

               alert(this.age);

};