# 一、绪论

1. JavaScript核心是ECAMScript，文档对象模型DOM，浏览器对象模型BOM
2. DOM把整个页面映射为一个多层节点结构
3. BOM控制页面以外的浏览器行为

# 二、JavaScript基础

1. 在HTML中使用JavaScript有两种方式：使用<script>嵌入代码；使用<script></script>引入外部JavaScript文件(HTML和JavaScript分离)
2. 解释器是按照HTML和JavaScript代码的顺序加载执行的
3. 在使用<script>嵌入代码时，一定不要在代码任何地方出现</script>，否则会加载错误，使用<\/script>解决
4. 使用<script src=”example.js”></script>加载外部JavaScript代码时，只有在XHTML文档中可以使用<script />
5. 在解析外部JavaScript文件或<script></script>标签内部JavaScript代码时，页面的处理也会暂停
6. 一般把<script>元素发到页面最后，</body>的前边或者后边，但是要想对某些显示就生效的内容，要放在这些内容之后
7. 使用var定义的变量称为定义该变量的作用域内的局部变量；不使用var定义的变量是全局变量(不推荐如此定义全局变量)
8. ECMAScript有5种基本类型：Undefined、Null、Boolean、Number、String，和一种复杂类型Object

typeof(操作符、非函数): 检测给定变量的数据类型，检测返回类型有：”undefined”、”boolean”、”string”、”number”、”object”和”function”

Undefined类型：未初始化和未声明的变量，执行typeof都会返回undefined

Null类型：null == undefined 会返回true，意在保存对象的变量未保存时，应该初始化为null

Boolean类型及转换规则，在条件语句中自行执行转换

|  |  |  |
| --- | --- | --- |
| 数据类型 | 转换成true的值 | 转换成false的值 |
| Boolean | true | false |
| String | 任何非空字符串 | “” 空字符串 |
| Number | 任何非零数字(含无穷大) | 0和NaN |
| Object | 任何对象 | null |
| Undefined | 不适用 | undefined |

Number类型：isFinite()函数判断是否位于最大值和最小值之间；isNaN()可以判断参数是否是 数值；parseInt(“0xAF”,16)将字符串类型转换成整数，建议指定基数，避免因浏览器实现不同而出现错误解析；parseFloat()只解析十进制字符串。

String类型：同java相同，一旦创建不可改变；对数字类型调用toString(para)可以返回以para为基数的字符串；在不知道转换值是否为null或者undefined的情况下，可使用String(value)方法

1. Object类型

constructor： 保存用于创建当前对象的函数，即构造函数

hasOwnProperty(propertyName) : 检查给定的属性在当前对象实例中(不是在原型中)是否存在

isPrototypeof(object): 检查传入的对象是否是当前对象的原型

valueOf(): 返回对象的字符串、数值或Boolean表示

1. 操作符
   1. 递增和递减操作符借鉴了C的
   2. 逻辑与（&&）

如果第一个数是对象，则返回第二个数

如果两个操作数都是对象，则只有在第一个操作数的求值结果为true的情况才会返回该对象

如果两个操作数都是对象，则返回第二个数

如果有一个操作数是null/NaN/undefined，则返回null/NaN/undefined

* 1. 逻辑或（||）

如果第一个操作数是对象，则返回第一个操作数

如果第一个操作数求值的结果为false，则返回第二个操作数

如果两个操作数都是对象，则返回第一个操作数

如果两个操作数都是null/NaN/undefined,则返回null/NaN/undefined

* 1. 特别注意条件条件表达式使用相等操作符而非赋值操作符
  2. ECMAScript中没有块级作用域，使用var声明的变量会被自动添加到最接近的环境中；若不适用var声明，则变量会被添加到全局环境中
  3. with语句是将代码的作用域设置到一个特定的对象中，大量使用with，会导致代码性能下降(链式查找)，也会对调试代码造成困难
  4. switch语句使用的全等操作，不会发生类型转换

1. 向函数中传递参数，使用的是arguments对象(与数组类似，可以使用下标访问)，与定义函数有几个参数无关，也不关心参数的类型，可以使用arguments.length获取参数的长度；因此JavaScript中没有函数的重载，定义两个名字相同的函数，只有后定义的函数有效

var *doAdd* = function (*num1*,*num2*){  
 return "doAdd(num1,num2)";  
}  
console.log(*doAdd*()); // doAdd(num1,num2)  
function *doAdd*(){  
 return "doAdd()";  
}  
console.log(*doAdd*()); // doAdd(num1,num2)

1. 使用typeof检测基本类型的数据，使用instanceof检测引用类型的数据
2. 在web浏览器中，全局执行环境被认为是window对象，因此所有的全局变量和函数都是作为window对象的属性和方法创建的
3. 当代码在一个环境中执行时，会创建一个作用域链，保证对执行环境有权访问的所有变量和函数的有序访问。嵌套的层次越多，效率越差，引擎优化不错，或许可以忽略不计。with和try-catch语句中的catch可以延长作用域链
4. 数据不再有用时，最好通过将其值设置为null来释放其引用

# 三、引用类型

## 1．下章着重介绍对象的实例化

以下这两种定义是等价的。可以使用字面量作为参数传递到函数中，对必须值使用命名参数，对可选参数使用字面量封装

var *person*={  
 name:"Nicholas",  
 age: 29,  
 getName: function(){return name;}  
};  
  
var *person1*= new Object();  
*person1*.name-"Nicholas";  
*person1*.age-29;  
*person1*.getName = function(){return *person1*.name;}

## 2. Array类型

// Array 类型定义  
var *colors* = new Array();  
var *colors1* = Array(3);  
var *colors2* =['red','blue','green'];

① 数组的length属性是可读写的，通过改变length的值，可以改变数组的大小

// 利用length属性在数组末尾添加值  
*colors2*[*colors2*.length] = 'black';

② 检测是否为数组类型

// 数组类型检测  
function *isArray*(*value*) {  
 return Object.prototype.toString.call(*value*) == "[object Array]";

}

③ 转换方法

console.log(*colors2*.toString()); // red,blue,green  
console.log(*colors*.valueOf()); console.log(*colors*);// red,blue,green  
console.log(*colors*.join("||"));// red||blue||green

④ 栈方法

*colors*.push("red","green"); // 返回修改后数组的长度  
*colors*.pop(); // 返回移除的项

⑤ 队列方法

colors.push("red","green"); // 返回修改后数组的长度  
colors.shift(); // 移除第一项，并返回  
// 使用unshift和pop也可以模拟队列  
colors.unshift("black"); // 推入第一项  
colors.pop();

⑥ 重排序方法

var *values*=[0, 1, 5,10, 15];  
*values*.sort(); // 默认情况下，是转换成字符串比较  
*values*.sort(function(*value1*, *value2*){  
 if(*value1* < *value2*) return -1; // 第一个参数应该位于第二个参数之前  
 else if(*value1* > *value2*) return 1;  
 else return 0;  
});

⑦ 操作方法

*colors2*=*colors*.concat("yellow",["black", "brown"]);  
// slice(begin,end): 不会影响原有数组  
*colors2*=*colors*.slice(1);  
*colors2*=*colors*.slice(1,4);  
// splice 方法，主要用途是向数组中插入项  
// 删除，可以删除任意项， 需要指定两个参数  
*colors*.splice(0,2);  
// 插入/删除：可以向指定位置插入任意数量的项，三个参数：起始位置、要删除项的数目、要插入的项  
var *removed* =*colors*.splice(2,0,"red","green"); // 插入  
*removed* = *colors*.splice(2,1,"red","green"); // 替换

⑧ 位置方法： indexOf() lastIndexOf()

⑨ 迭代方法：every() some() filter() map() forEach()

// 迭代方法  
var *numbers*=[1,2,3,4,5,4,3,2,1];  
// every: 有一项为false，则停止执行，返回false， 与&&操作等价  
var *everyResult* = *numbers*.every(function(*item*,*index*,*array*){  
 return *item*>2;  
});  
// some: 只要有一项返回true， 就返回true  
var *somResult*=*numbers*.some(function(*item*,*index*,*array*){  
 return *item*>2;  
});  
// filter: 对数组中的每一项运行给定函数，并返回函数返回true的项组成的数组  
var *filterResult* = *numbers*.filter(function (*item*, *index*, *array*) {  
 return *item*>2;  
});  
// map: 返回一个数组，数组中每一项都是对原始数组相应项计算之后的值  
var *mapResult* = *numbers*.map(function (*item*, *index*, *array*) {  
 return *item*\*2;  
});  
// 没有返回值  
*numbers*.forEach(function(*item*, *index*, *array*){  
 // do something  
});

10 归并方法 reduce()、reduceRight(): 这两个方法都会迭代数组中所有的项，构建一个最终返回值

*numbers*.reduce(function (*prev*, *cur*, *index*, *array*) {  
 return *prev* + *cur*;  
});

## 3. Date类型

* 1. Date类型的创建

// Date类型 创建  
var *now* = new Date(); // 返回当前时间, 日期表示  
var *someDate* = new Date("2016-08-03T08:27:00"); // 如果字符串不能转换成日期，返回NaN  
var *allFives* = new Date(2005, 4, 5, 17, 55, 55);//年月日时分秒，月是从0开始的  
var *start* = Date.now(); // 当前时间的毫秒数

* 1. 其他方法

toDateString() toTimeString() toLocaleDateString() toLocaleTimeString() toUTCString()

## 4. RegExp类型

* 1. 字面量定义正则

var *expressinog*= /pattern/flags

②正则表达式中使用的元字符都必须转义： ([{\^$|}?.\*+])

* 1. 使用RegExp构造函数，因为其参数是字符串，所以要对字符进行双重转义
  2. RegExp对象主要方法是exec()，设置了全局模式下，一次也只返回一个匹配项，需要多次调用； test()方法经常被用在if语句中
  3. RegExp的属性

|  |  |  |
| --- | --- | --- |
| 长属性名 | 短属性名使用下标[]访问 | 说明 |
| input | $\_ | 最近一次要匹配的字符串 |
| lastMatch | $& | 最近一次的匹配项 |
| lastParen | $+ | 最近一次的捕获组 |
| leftContext | $` | input字符串中lastMatch之前的文本 |
| Multiline | $\* | 是否使用多行模式 |
| rightContext | $’ | input字符串中lastMatch之后的文本 |

## 5. 函数Function类型

* 1. 解析器会率先读取函数声明，并使其在执行任何代码前可用，函数表达式必须等到解析器执行到它所在的代码，才会真正被解释执行

var *doAdd* = function (*num1*,*num2*){  
 return "doAdd(num1,num2)";  
};  
console.log(*doAdd*()); // doAdd(num1,num2)  
function *doAdd*(){ // 首先会被解析  
 return "doAdd()";  
}  
console.log(*doAdd*()); // doAdd(num1,num2)

* 1. arguments的callee属性，使用该属性可以达到函数名解耦

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

* 1. 十分注意this属性：该属性是执行函数的环境setTimeout()应该是全局环境
  2. arguments.callee.caller 该环境中保存着调用当前函数的函数的引用，返回的是源码
  3. length属性表示希望接收的参数个数；prototype保存它们所有实例方法的真正所在
  4. apply(this, arguments) 和 call(this,argument1,argument2,...)，这两个函数可以扩充函数运行的作用域

window.color = "red";  
var *o* = {color:"blue"};  
function *sayColor*(){console.log(this.color);}  
*sayColor*.call(this); // red  
*sayColor*.call(window); // red  
*sayColor*.call(*o*); // blue

## 6．String类型

① 字符方法

var *str* = "hello world";  
console.log(*str*.charAt(1)); // 字符  
console.log(*str*.charCodeAt(1)); // 字符编码

② 字符串操作方法

// slice() substr() substring() 接受两个参数，第一个指定子字符串的开始位置，第二个参数表示在哪里结束  
// slice和substring第二个参数是子字符串最后一个字符后面的位置；substr指定的是子字符串的长度  
// 如果传递的是负值，slice会将负值与字符串的长度相加，substr会将第一个参数加上字符串的长度，将负的第二个参数转换成0； substring会将所有的负值参数转换成0  
console.log(*result*.slice(-3)); // "rld!"  
console.log(*result*.substring(-3)); // "hello world!"  
console.log(*result*.substr(-3)); // "rld!"  
console.log(*result*.slice(3, -4)); // "lo w"  
console.log(*result*.substring(3,-4)); // "hel" 同 result.substring(0,3)  
console.log(*result*.substr(3, -4)); // "" 空字符串

③字符串位置方法

// 如果没有找到该字符串，则返回-1  
console.log(*result*.indexOf("o", 6)); // 从前向后搜索,第二个参数为搜索的起始位置  
console.log(*result*.lastIndexOf("o")); // 从后向前搜

④字符串匹配方法 match、replace

// 字符串匹配方法  
var *text* = "cat, bat, sat, fat";  
var *matches* = *text*.match(/.at/);  
console.log(*matches*.index); //0  
console.log(*matches*[0]); // "cat"  
console.log(*text*.search(/.at/)); // 0  
  
console.log(*text*.replace("at","ond")); // "cond, bat, sat, fat"  
console.log(*text*.replace(/at/g, "ond")); // "cond, bond, sond, fond"  
// 通过特殊的字符序列，可以使用最近一次匹配结果的内容  
console.log(*result*.replace(/(at)/g, "word ($1)")); // word (cat), word (bat), ...  
/\*replace 的第二个参数也可以是函数\*/  
function *htmlEsacpe*(*text*) {  
 return *text*.replace(/[<>"&]/g, function(*match*/\*模式的匹配项\*/, *pos*/\*位置\*/, *originalText*){  
 switch (*match*){  
 case "<": return "&lt;";  
 case ">": return "&gt;";  
 case "&": return "&amp;";  
 case "\"": return "&quot;";  
 }  
 });  
}

* 1. 浮点数舍入方法

Math.ceil() 执行向上舍入

Math.floor() 执行向下舍入

Math.round() 执行标准舍入，即四舍五入

* 1. random()方法：返回大于等于0小于1的随机数，

值 = Math.floor(Math.random() \* 可能值的总数 + 第一个可能的值)

# 四、面向对象的程序设计

## 1. 使用数据属性和访问器属性

可以定义私有变量 Object.defineProperty()，查看数据或者访问器的属性：Object.getOwnPropertyDescriptor() p140

## 创建对象

①工厂方法

function *createPerson*(*name*, *age*, *job*){  
 var o=new Object();  
 o.name = *name*;  
 o.age=*age*;  
 o.job=*job*;  
 o.sayName = function(){console.log(this.name);};  
 return o;  
}

## 构造函数

必须使用new操作符来新建对象，可以使用instanceof和constructor来检测对象的类型；

缺点：每个实例之间，方法都要重新创建一遍，即每个Person实例都包含不同的Function实例

// 这种方式创建对象，要使用new操作符  
// 过程：①创建一个新对象；②将构造函数的作用域赋给新对象（this指向这个新对象）；③执行构造函数中的代码；④返回新对象  
var person1 = new *Person*("Greg", 27,"Doctor");  
var *person2* = new *Person*("Greg", 27,"Doctor");  
console.log(person1.constructor == *Person*); // true  
console.log(person1 instanceof *Person*); // true  
console.log(person1 instanceof Object); //true  
*Person*("Greg", 27, "Doctor"); // 作为普通函数调用，会将属性和方法添加到window上  
var *o* = Object();  
*Person*.call(*o*, "Kristen", 25, "Nurse"); // 还可以使用o的作用域，属性和方法都会添加到o上  
console.log(person1.sayName == *person2*.sayName); // false

## 原型对象：可以让所有对象实例共享它所包含的属性和方法

可以通过对象实例访问保存在原型中的值，但不能通过对象实例重写原型中的值，若在实例中添加一个属性，就会屏蔽原型中的那个属性

* 1. 定义

function *PersonPro*(){}  
*PersonPro*.prototype.name = "Nicholas";  
*PersonPro*.prototype.age=29;  
*PersonPro*.job="Software Engineer";  
*PersonPro*.prototype.sayName=function(){console.log(this.name);};  
var *person3* = new *Person*();  
var *person4* = new *Person*();  
console.log(*person3*.sayName == *person4*.sayName); // true  
*person3*.sayName(); // 会执行两次搜索 person3没有sayName属性，person3.prototye含有sayName属性  
*person3*.name = "Greg"; // 会屏蔽原型中的name, 不会修改原来的属性  
cnosole.log(*person3*.name); // Greg  
console.log(*person4*.name); // Nicholas  
//hasOwnProperty(): 可以检测一个属性是原型中的，还是实例中的  
*person3*.hasOwnProperty("name"); // true  
*person4*.hasOwnProperty("name"); // false

delete *person3*.name; // delete操作符可以完全删除实例属性，可从新访问原型中的属性

* 1. 原型与in操作符

// 单独使用  
console.log("name" in *person3*); // true, 通过对象能够访问即返回true，不论该属性是在实例中还是在原型中  
// 判断属性是否在原型中  
function *hasPrototypeProperty*(*obj*, *name*) { return !*obj*.hasOwnProperty(*name*) && (*name* in *obj*);}  
// for-in 循环时，会返回所有能够通过对象访问的属性，一般为自定义的属性

③更简单的原型语法

function *Person*(){}  
*Person*.prototype={  
 constructor: *Person*, // 如果不设置，constructor将指向Object  
 name:"Nicholas",  
 age:29,  
 job: "Software Engineer",  
 sayName : function(){console.log(this.name);}  
};

* + 1. 原型的动态性

实例中的指针指向的是原型而不是构造函数，下面的例子中将原型修改为另外一个对象就等于切断了构造函数与最初原型之间的联系，它引用的仍然是最初的原型 p157

function *Person*(){}  
var *friend* = new *Person*();  
*Person*.prototype = {  
 constructor: *Person*, // 如果不设置，constructor将指向Object  
 name:"Nicholas",  
 age:29,  
 job: "Software Engineer",  
 sayName : function(){console.log(this.name);}  
};  
*friend*.sayName(); // error

* + 1. 不推荐在程序中修改原生对象的原型

String.prototype.startsWith = function(*text*){  
 return this.indexOf(*text*) == 0;  
};

* + 1. 原型对象最大的缺点就是：所有属性共享

## 5. 组合使用构造函数和原型模式

构造函数用于定义实例属性，原型模式用于定义方法和共享属性，集合了两种模式之长

function *Person*(*name*, *age*, *job*){  
 this.name = *name*;  
 this.age = *age*;  
 this.job = *job*;  
}  
*Person*.prototype={  
 constructor: *Person*,  
 sayName: function(){console.log(this.name);}  
};

## 动态原型模式

通过必要的时候在构造函数中初始化原型，来把所有信息都封装到构造函数中，更像一个类

function *Person*(*name*, *age*, *job*){  
 this.name = *name*;  
 this.age = *age*;  
 this.job = *job*;  
 if(typeof this.sayName != "function"){  
 // 不能使用字面量重写原型，否则会断开现有实例与新原型之间的联系  
 *Person*.prototype.sayName = function(){console.log(this.name);}  
 }  
}

## 寄生构造函数模式

适用于将原生对象属性添加到定义的对象中 (jQuery)

function *SpecialArray*(){  
 var values = [];  
 values.push.apply(values, arguments);  
 values.toPipedString = function(){  
 return this.join("|");  
 };  
 return values;  
}  
var *colors* = new *SpecialArray*("red","blue","green");  
console.log(*colors*.toPipedString());

## 稳妥构造函数模式

所谓稳妥构造函数模式是指没有公共属性，其方法也不引用this对象

function *Person*(*name*, *age*, *job*){  
 var o=new Object();  
 // 不是使用this定义属性  
 sex = 'boy';  
 o.sayName = function () {  
 console.log(*name*);  
 }  
}

## 继承--原型链

原型链是实现继承的主要方法，通过重写原型对象，将其指向父类的原型对象实例。

下面例子中，调用instance.getSuperValue经历了三个步骤：搜索实例；搜索SubType.prototype；搜索SuperType.prototype

function *SuperType*(){  
 this.property = true;  
}  
*SuperType*.prototype.getSuperValue = function () {  
 return this.property;  
};  
function *SubType*(){  
 this.subproperty=false;  
}  
// 通过重写原型，实现继承， 必须是SuperType的实例，实例指向的是原型，而非构造函数  
*SubType*.prototype = new *SuperType*();  
*SubType*.prototype.getSubValue = function(){return this.subproperty;};  
  
var *instance* = new *SubType*();  
console.log(*instance*.getSuperValue()); // true;  
console.log(*instance*.getSubValue()); // false

①确定原型和实例的关系

console.log(*instance* instanceof Object); // true  
console.log(*instance* instanceof *SuperType*); // true  
console.log(*instance* instanceof *SubType*); // true  
console.log(Object.isPrototypeOf(*instance*)); // true  
console.log(*SuperType*.isPrototypeOf(*instance*)); // true  
console.log(*SubType*.isPrototypeOf(*instance*)); // true

③定义方法

不管如何，给原型添加共有的属性和方法的代码一定要放在替换原型语句之后，共有的属性和方法；

不能使用字面量为子类添加新的方法或属性

## 继承--构造函数

①原型链继承存在问题：父类中的实例属性成为子类原型属性；创建子类时，不能向父类中传递属性

可以使用借用构造函数来解决问题

function *SuperType*(*name*) {  
 this.name = *name*;  
}  
  
function *SubType*(){  
 *SuperType*.call(this, "Nicholas");  
 this.age = 29;  
}

②问题：因为调用的是父类中的构造函数，那么父类原型定义的属性和方法对子类是不可见的

## 组合继承

将原型链和构造函数相结合：使用原型链实现对原型属性和方法的继承，借用构造函数实现对实例属性的继承

function *SuperType*(*name*) {  
 this.name = *name*;  
 this.colors =["red","blue","green"];  
}  
*SuperType*.prototype.sayName = function () {  
 console.log(this.name);  
};  
  
function *SubType*(*name*, *age*){  
 *SuperType*.call(this, *name*); // 变量的作用域为this 第二次调用  
 this.age = *age*;  
}  
// 继承方法  
*SubType*.prototype = new *SuperType*(); // 第一次调用SuperType()  
*SubType*.constructor = *SubType*;  
*SubType*.prototype.sayAge = function(){console.log(this.age);};

instanceof和isPrototypeOf()能够识别基于组合继承创建的函数

问题：无论在什么情况下，都会调用两次父类的构造函数

## 最好的--寄生组合式继承方式

function *inheritPrototype*(*subType*, *superType*) {  
 // 浅拷贝  
 var prototype = Object(*superType*.prototype); // 创建对象：创建父类原型的副本  
 prototype.constructor = *subType*; // 增强对象：添加constructor属性  
 *subType*.prototype = prototype; // 指定对象  
}  
// 使用  
*inheritPrototype*(*SubType*, *SuperType*);

# 五、函数表达式

## 1.函数声明和函数表达式

① 函数声明一个最重要的特征就是函数声明提升。所以函数声明可以放在调用它的语句的后面

*sayHi*();  
function *sayHi*(){console.log("hi");}

② 函数表达式：与其他表达式一样，使用前需要先赋值，像上面的例子使用，会导致错误。

var *sayHi* = function () {  
 console.log("express");  
};  
function *sayHi*(){console.log("hi");}  
*sayHi*(); // hi

## 2.递归函数正确写法

// 非严格模式, 将函数名赋给其他变量，或者使用函数名作为变量，都不会错误  
function *factorial*(*num*){  
 if(*num* <= 1){return 1;}  
 else return *num* \* arguments.callee(*num* - 1);  
}  
// 严格模式下，不能使用arguments.callee  
var *factorial* = (function f(*num*){  
 if(*num* <= 1){return 1;}  
 else return *num* \* *f*(*num* - 1);  
});

## 3.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

闭包会携带包含它的函数的作用域，因此会比其他函数占用更多的内存，过度使用会导致内存占用过多。

由于作用连的关系，下列代码可以正确执行

function *c*() { console.log(*t*);}  
var *t* = "name";  
*c*();

## 4.闭包与变量

作用链有一个副作用：闭包只能取得包含函数重任何变量的最后一个值

function *wrongFunction*(){  
 var result = [];  
 for(var i=0; i<10; i++){  
 result[i] = function(){ return i;}; // 每个i都是10  
 }  
 return result;  
}  
function *createFunction*(){  
 var result = [];  
 for(var i=0; i<10; i++){  
 result[i] = function(*num*){return function () {  
 return *num*;  
 } }(i); // 嵌套一层函数  
 }  
 return result;  
}

## 5.闭包中的this对象

**匿名函数**的执行环境具有全局性，因此this对象通常指向window，arguments也存在同样的问题，使用that=this解决

var *name* = "The Window";  
var *object* = {  
 name: "My Object",  
 getNameFunc:function () {  
 var name = "getNameFunc";  
 return function () {  
 return this.name;  
 }  
 }  
};  
console.log(*object*.getNameFunc()()); // 匿名函数具有全局性

## 6.模仿块级作用域

将函数声明包含在一对括号中时，表示它实际上是一个函数表达式。之所以使用括号是因为函数声明后边不能跟圆括号，而函数表达式可以

(function(){  
 // 这里是块级作用域  
})();

## 7.静态私有变量

在函数内部使用var定义的变量，外部不可访问。缺点：多查找作用域链中的一个层次，就会在一定程度上影响查找速度，正是使用闭包和私有变量的不足。

(function(){  
 //静态的、由所有实例共享的属性  
 var name ="";  
 // 没有var声明，Person是全局的, 严格模式下可以使用windo.Person  
 window.Person = function(*value*){  
 name = *value*;  
 };  
 Person.prototype.getName = function(){  
 return name;  
 };  
 Person.prototype.setName = function(*value*){  
 name=*value*;  
 };  
})();  
  
var *person* = new Person("Nicholas");  
console.log(*person*.getName());  
*person*.setName("Greg");  
console.log(*person*.getName());

## 8. 模块模式--单例

感觉模块模式和静态私有变量很相似，可以作为全局对象

var *singleton* = function(){  
 // 私有变量和私有函数  
 var privateValue = 10;  
 function *privateFunction*(){  
 return false;  
 }  
 // 特权/公有方法  
 return {  
 publicProperty : true,  
 publicMethod: function(){  
 privateValue ++;  
 return *privateFunction*();  
 }  
 }  
};

## 9. 增强的模块模式

适合必须是某种类型的实例的单例，同时还必须增加某些属性或方法

var *singlteonStrong* = function(){  
 // 私有变量和私有函数  
 var privateValue = 10;  
 function *privateFunction*(){  
 return false;  
 }  
  
 // 创建对象  
 var object = new CustomType();  
 //添加特权/公有属性和方法  
 object.publicProperty = true;  
 object.publicMethod = function(){  
 privateValue ++;  
 return *privateFunction*();  
 }  
 return object;  
};

# 六、BOM

BOM提供了很多对象用于访问浏览器的功能，BOM的核心是window对象。所有在全局作用域中声明的变量、函数都会成为window对象的属性和方法

每个框架都拥有自己的window对象，并且保存在frames集合中，可以使用top.frames[0]来访问

## window对象

* 1. 窗口位置：跨浏览器取得浏览器窗口相对于屏幕左边和上边的位置

var *leftPos* = (typeof window.screenLeft == "number") ?  
 window.screenLeft : window.screenX;  
var *topPos* = (typeof window.screenTop == "number") ?  
 window.screenTop : window.screenX;

* 1. 移动窗口

window.moveTo(200,300); // 将窗口移动到(200,300)  
window.moveBy(-50, 0); // 将窗口向左移动50像素

* 1. 窗口大小：只能准确取得页面视口的大小

var *pageWidth* = window.innerWidth;  
var *pageHeight* = window.innerHeight;  
if(typeof *pageWidth* != "number"){  
 var *doc* = document.documentElement || document.body;  
 *pageWidth* = *doc*.clientWidth;  
 *pageHeight* = *doc*.clientHeight;  
}  
// 有些浏览器下回默认禁用 chrome  
window.resizeTo(100, 100); // 调整窗口大小 100\*100  
window.resizeBy(100, 50); // 调整到200\*150

* 1. 弹出窗口是否被屏蔽检测：chrome浏览器默认屏蔽了非点击事件弹出的窗口

var *block* = false;  
try {  
 var *wroxWin* = window.open("http://www.sogou.com", "\_blank");  
 if (*wroxWin* == null) {*block* = true; }  
}catch(*ex*){  
 *block* = true;  
}

## 2．超时调用-- setTimeout

超时调用的代码都是在全局作用域中执行的，函数中的this指针在非严格模式下指向window对象，严格模式下是undefined

## 3．location对象

Location对象提供了与当前窗口中加载的文档有关的信息，window.location和document.location是一个对象

|  |  |  |
| --- | --- | --- |
| 属性名 | 例子 | 说明 |
| hash | #content | 返回URL中的hash，如果URL中不包含散列，返回”” |
| host | [www.sogou.com:80](http://www.sogou.com:80) | 返回服务器名称和端口号 |
| hostname | [www.sogou.com](http://www.sogou.com) | 返回服务器名称，不带端口号 |
| href |  | 返回当前页面的完整url |
| pathname | /WileyCD/ | 返回url中的目录或文件名 |
| port | 80 | 端口号 |
| protocol | https | 返回页面使用的协议 |
| search | ? \_ijt=6e1mjpg803 | 返回url中的查询字符串 |

1. 位置操作：每次修改location的属性，页面都会以新URL重新加载

// 立即打开url，并在浏览器的历史记录中生成一条记录  
location.assign(url);  
// 下面两个同assign完全等价  
window.location = 'https://www.sogou.com';  
location.href = 'https://www.sogou.com';

replace():上述修改href之后不会产生的历史记录，也不能回退到前一个页面

location.replace("https://www.sogou.com");  
  
location.reload(); // 重新加载 (有可能从缓存中加载)  
location.reload(true); // 重新加载(从服务器重新加载)

## 4．navigator对象

navigator对象最重要的属性就是userAgent

## 5.history

使用go()方法可以在用户的历史记录中任意跳转；

length属性，保存着历史记录的数量

history.go(-1); // 后退一页  
history.go(1); // 前进一页  
history.go("wrox.com"); // 跳转到最近的wrox.com页面

# 七、客户端检测

不到万不得已就不要使用客户端检测，只要能找到更通用的方法，就优先采用更通用的方法。先设计通用的方案，然后在使用特定于浏览器的技术增强方案。

## 能力检测

①能力检测的目标是识别浏览器是否支持某种特性，而不是识别浏览器。有两点需要注意：

先检测达成目的最常用的特性；必须测试实际要用的特性，一个特性存在，另外一个特性则不一定存在

function *getElement*(*clazz*, *tagName*){  
 if(document.getElementsByClassName){ // ie 678不支持该方法  
 return document.getElementsByClassName(*clazz*);  
 }else{  
 var tag = document.getElementsByTagName(*tagName*);  
 var tagAll = [];  
 for(var i=0; i<tag.length; i++){  
 for(var j=0, n=tag[i].className.split(' '); j<n.length; j++){  
 if(n[j].trim() == *clazz*){  
 tagAll.push(tag[i]);  
 break;  
 }  
 }  
 }  
 }  
 return tagAll;  
}

1. 更可靠的能力检测：检测某个属性存在并不能确定是否会按照适当的方式，因此更好的方式是检测该属性是否是函数

function *isHostMethod*(*object*, *property*) {  
 var t=typeof *object*[*property*];  
 return t=='function' || (!!(t=='object' && *object*[*property*])) || t=='unkown';  
}

## 用户代理检测

在服务器端，通过检测用户代理字符串来确定用户使用的浏览器是一种常用的做法，但是在客户端，用户代理检测一般是万不得已的做法，优先级应该排在能力检测或怪癖检测之后

var *client* = function () {  
 var engine = {  
 ie:0,  
 gecko:0,  
 webkit:0,  
 khtml:0,  
 opera:0,  
 // 具体的版本号  
 ver : null  
 };  
  
 var browser = {  
 ie : 0,  
 firefox : 0,  
 safari : 0,  
 konq : 0,  
 opera : 0,  
 chrome : 0,  
 ver :null  
 };  
 //在此检测呈现引擎、平台和设备  
 var ua = navigator.userAgent;  
 if(window.opera){  
 engine.ver = window.opera.version();  
 engine.opera = parseFloat(engine.ver);  
 }else if(/AppleWebKit\/(\S+)/.test(ua)){  
 engine.ver = RegExp["$1"];  
 engine.webkit = parseFloat(engine.ver);  
 // 确定是Chrome还是Safari  
 if(/Chrome\/(\s+)/.test(ua)){  
 browser.ver = RegExp["$1"];  
 browser.chrome = parseFloat(browser.ver);  
 }else if(/Version\/(\S+)/.test(ua)){  
 browser.ver = RegExp["$1"];  
 browser.safari = parseFloat(browser.ver);  
 }else { // 近似确定版本号  
 var safariVersion = 1;  
 if(engine.webkit < 100){  
 safariVersion = 1;  
 }else if(engine.webkit < 312){  
 safariVersion = 1.2;  
 }else if(engine.webkit < 412){  
 safariVersion = 1.3;  
 }else{  
 safariVersion = 2;  
 }  
 browser.safari = browser.ver = safariVersion;  
 }  
 }else if(/KHTML\/(\S+)/.test(ua) || /Konqueror\/([^;]+)/.test(ua)){  
 engine.ver = RegExp["$1"];  
 engine.khtml = browser.konq = parseFloat(engine.ver);  
 }else if(/rv:([^\)]+)\) Gecko\/\d{8}/.test(ua)){  
 engine.ver = RegExp["$1"];  
 engine.gecko = parseFloat(engine.ver);  
 if(/Firefox\/(\S+)/.test(ua)){ //确定是否为火狐浏览器  
 browser.ver = RegExp["$1"];  
 browser.firefox = parseFloat(browser.ver);  
 }  
 }else if(/MSIE ([^;]+)/.test(ua)){  
 engine.ver = RegExp["$1"];  
 engine.ie = parseFloat(engine.ver);  
 }  
  
  
 return {  
 engine: engine,  
 brower:browser  
 }  
};

# 八、DOM

## 1. 节点层次

①Node类型

DOM1中定义的Node接口，JavaScript中所有的节点类型都继承自Node类型。每个节点都有一个nodeType属性，Node定义了12个数值常量来表示：

Node.ELEMENT\_NODE(1), Node.ATTRIBUTE\_NODE(2), Node.TEXT\_NODE(3), Node.CDATA\_SECTION\_NODE(4), Node.ENTITY\_REFERENCE\_NODE(5), Node.ENTITY\_NODE(6), Node.PROCESSING\_INSTRUCTION\_NODE(7), Node.COMMENT\_NODE(8), Node.DOCUMENT\_NOD(9), Node.DOCUMENT\_TYPE\_NODE(10), Node.DOCUMENT\_FRAGMENT\_NODE(11), Node.NOTATION\_NODE(12)

// 由于IE中没有公开Node的构造函数，最好将nodeType属性与数字值进行比较  
if(someNode.nodeType == 1){ /\* do something\*/}

② 节点关系

每个节点都有一个childNodes属性，保存着NodeList对象，有数组的特性，但不是数组。而且NodeList对象是基于DOM结构动态执行查询的结果，DOM结构变化自动反应在NodeList中

var *firstChild* = someNode.childNodes[0];  
var *secondChild* = someNode.childNodes.item(1);  
// length属性是访问NodeList那一刻的值  
var *count* = someNode.childNodes.length;  
// 父节点，兄弟节点  
var *parent* = someNode.parentNode;  
var *pre* = someNode.previousSibling;  
var *next* = someNode.nextSibling;  
*firstChild* = someNode.firstChild;  
var *last* = someNode.lastChild;

④ 操作节点

**appendChild()**: 将新增的节点加到parentNode的最后，**如果新加入的节点已经是文档中的一部分，那就会将节点移动到新位置**。

**insertBefore()**: 接受连个参数，要插入的节点和作为参照的节点

**replaceChild()**: 要插入的节点和要替换的节点

**cloneNode()**: 接收一个参数，表示是否执行深复制。该方法不会复制节点上的JavaScript属性

var *deepList* = myList.cloneNode(true);  
console.log(*deepList*.length); // 3  
var *shallowList* = myList.cloneNode(false);  
console.log(*shallowList*.length); // 0

## 2. Document类型

① 基本信息

document对象是window对象的一个实例，可以将作为全局对象访问：

nodeType=9, nodeName=#document

②document.documentElement指向html元素

var html = document.documentElement;  
var *r* = html == document.childNodes[0] == document.firstChild; // true

document.body 指向body元素

document.URL取得请求完整的URL， document.domain 取得请求的domain，document.referrer 取得请求来源url，可能为空

//使用内嵌框架时，设置为相同的domain，可以跨域访问  
// 设置的domain必须是现在domain的松散，即取domain的一部分  
document.domain = "wrox.com"; // 松散，可以设置  
document.domain = "p2p.wrox.com"; // 紧绷，错误！

③ 查找元素

Document类型提供了两种方法getElementById()和getElementsByTagName().

如果页面中多个元素的ID值相同，getElementById()只返回文档中第一次出现的元素。IE7及较低版本有bug：name属性与ID匹配的表单元素也会被该方法返回

//<img src="myimage.gif" name="myImage">  
// 返回的是一个HTMLCollection对象  
var *images* = document.getElementsByTagName("img");  
console.log(*images*.length);  
console.log(*images*[0].src);  
console.log(*images*.item(0).src);  
// 通过neme属性取得项  
var *myImage* = *images*.namedItem("myImage");  
*myImage* = *images*["myImage"];  
// 只有HTMLDocument类型才有  
document.getElementsByName("color");

④ 特殊集合

document对象还有一些特殊的集合，这些集合都是HTMLCollection

document.anchors: 包含文档中所有带name特性的<a>标签

document.forms : 包含文档中所有的<form>元素

document.images: 包含文档中所有<img>标签

document.links : 包含文档中所有带href特性的<a>标签

⑤文档写入

在html中输出时，不可含有</script>标签，需要对其进行转义<\/script>

3. Element类型

特征： nodeType = 1, tagName/nodeName 的值为元素的标签名， nodeValue值为null， parentNode可能是Document或Element类型。

**在HTML中，标签名始终都回忆全部大写表示，XML标签名与源码中一致**

if(element.tagName.toLowerCase() =-= "div"){}