# 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，从索引处开始提取原数组的元素  
*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

## 2. 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;  
}

## 3. 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

## 4。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

### ② 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);}  
};

### ④ 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

### ⑤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;  
};

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);}  
};

## 6. 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);}  
 }  
}

## 7. 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());

## 8. 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*);  
 }  
}

## 9. 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

### ② 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

## 10. 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;  
}

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

## 11. 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

## 12. 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"){}

① HTML元素中都会含有一些属性：id、title、className，可以访问这些属性，也可以对其设值

var *div* = document.getElementById("myDiv");  
console.log(*div*.id + *div*.title + *div*.className);

② 每个元素都有一个或多个固定的特性，对其特性操作的方法有三种：getAttribute()、setAttribute()和removeAttribute

console.log(*div*.getAttribute("id") + *div*.getAttribute("title"));  
console.log(*div*.getAttribute("class"));

如果给定的特性不存在，getAttribute()返回null

getAttribute()在访问style和事件处理程序时，返回的结果与属性值并不对应，因此很少使用该方法

**在IE7及以前版本中，setAttribute()存在一些异常，通过这个方法设置class和style属性，没有任何效果，Firefox等浏览器也有这个问题，因此推荐使用属性来设置特性。**

**③ attributes属性**

atttibutes属性包含一个动态的集合—NamedNodeMap，该属性包含一系列节点，每个节点的nodeName就是特性的名称，而节点的nodeValue就是特性的值

var *id* = element.attributes.getNamedItem("id").nodeValue;  
element.attributes["id"].nodeValue = "someOtherId";  
// 遍历  
for(var *i*=0; *i*< element.attributes.length; ++*i*){  
 console.log(element.attributes[*i*].nodeValue);  
}

**④ 元素的子节点**

元素的子节点可以用childNodes获取，其包含了所有子节点，非IE浏览器中，不包含空白字符。 对于以下的html

<**ul** id=**"myMenu"** style=**"***position*: **absolute**;*visibility*: **hidden**;*background-color*: **silver**;**"**>  
 <**li**><**a** href=**"www.baidu.com"**>**百度**</**a**></**li**>  
 <**li**><**a** href=**"www.sogou.com"**>**搜狗**</**a**></**li**>  
 <**li**><**a** href=**"www.yahoo.com"**>**雅虎**</**a**></**li**>  
</**ul**>

// IE9及以下 下返回为3，其他都返回为7  
console.log(document.getElementById("myMenu").childNodes.length);

// 可以通过以下方式获取元素，但是获取的是ul的直接子元素

var *ul* = document.getElementById("myMenu");  
var *items* = *ul*.getElementsByTagName("li");

**⑤ Text类型**

nodeType=3, nodeName=”#text”, nodeValue=value

在修改文本节点时，字符串会经过HTML编码

document.createTextNode(): 创建新的文本节点

element.normalize(): 将所有文本节点合并成一个节点

## 4. DOM操作技术

浏览器充斥着隐藏的陷阱和不兼容问题

**①动态脚本：通过修改DOM动态添加的脚本；没有标准来探测是否加载完了**

var *script* = document.createElement("script");  
*script*.type = "text/javascript";  
*script*.src = "client.js";  
document.body.appendChild(*script*);  
  
// 为了兼容IE， Firfox，Opera和Safari  
*script* = document.createElement("script");  
*script*.type = "text/javascript";  
*script*.text = "function sayHi(){alert('hi');}";  
document.appendChild(*script*);

**② 动态样式**：同动态脚本相似，IE将<style>视为特殊的节点，不允许访问其子节点

var *style* = document.createElement("style");  
*style*.type = "text/css";  
try{  
 *style*.appendChild(document.createTextNode("body{background-color:red}"));  
}catch (*ex*){  
 *style*.styleSheet.cssText = "body{background-color:red}";  
}  
document.head.appendChild(*style*);

③ 使用NodeList

NodeList，NamedNodeMap和HTMLCollection都是动态的，即文档结构发生变化时，他们都会得到更新，所有NodeList对象都是在访问DOM文档时的实时运行的查询

// 会变成死循环  
var *divs* = document.getElementsByTagName("div");  
for(var *i*=0; *i*<*divs*.length; *i*++){  
 var *div* = document.createElement("div");  
 document.body.appendChild(*div*);  
}

**DOM操作往往是JavaScript程序中开销最大的部分，访问NodeList导致的问题最多，NodeList是动态的，每次访问，都会运行一次查询**

# 九、DOM扩展

对DOM的两个主要扩展是Selectors API 和 HTML5

## 1. 选择符API

两个核心方法：querySelector()和querySelectorAll()

① querySelector(): 接收一个选择符，返回与该模式匹配的第一个元素，如果没有返回null，如果传入了不支持的选择符，将会抛出错误。 (选择符也可以很复杂)

var *body* = document.querySelector("body");  
var *myDiv* = document.querySelector("#mydiv");  
var *selected* = document.querySelector(".selected");

② querySelectorAl(): 返回一个NodeList实例，该方法不是“动态”的，使用item()方法遍历取得的元素

var *ems* = document.querySelector("#mydiv").querySelectorAll("em");

③ matchesSelector(): 调用元素是否与该选择符匹配，目前支持不是很好

if(document.body.matchesSelector("body.page1")){}

## 2. 元素遍历

API为DOM元素添加了5个属性，利用这些属性不用担心空白文本节点：

childElementCount: 返回子元素（不包含文本节点和注释）的个数

firstElementChild: 指向第一个元素

lastElementChild: 指向最后一个元素

previousElementSibling: 指向前一个同辈元素

nextElementSibling: 指向后一个同辈元素

// 跨浏览器遍历某元素的子元素  
var *i*, *len*, *child* = element.firstChild;  
while(*child* != element.lastChild){  
 if(*child*.nodeType == 1){ // 检查是否为某些元素  
 processChild(*child*);  
 }  
 *child* = *child*.nextSibling;  
}  
  
//使用新属性，支持的浏览器有IE9+、Firefox3.5+、 Safari4+、chrome和Opera 10+  
*child* = element.firstElementChild;  
while(*child* != element.lastElementChild){  
 processChild(*child*);  
 *child* = *child*.nextElementSibling;  
}

## 3.HTML5

**①** 增加了getElementsByClassName(): 接收一个包含一个或多个类名的字符串，返回带有指定类的所有元素的NodeList， 同样具有性能问题

// 取得ID为mydiv的元素中带有类名selected的所有元素  
var *selects* = document.getElementById("mydiv").getElementsByClassName("selected");

**②** classList属性：添加、删除、替换类名

// 支持的浏览器有Firefox3.6+ 和Chrome  
*div*.classList.remove("disabled");// 删除"disabled" 类  
*div*.classList.add("current"); // 添加 "current" 类  
*div*.classList.toggle("user"); // 切换 "user" 类

③ 焦点管理

document.activeElement: 引用DOM中当前获得焦点的元素

document.hasFocus() : 用来确定文档是否获得了焦点

**④HTMLDocument的变化**

**readyState属性： loading 正在加载文档；comlete 已经加载完文档**

if(document.readyState == "complete"){}

**⑤head属性**

**HTML5新增了document.head属性**

var *head* = document.head || document.getElementsByTagName("head")[0];

## **4. 插入标记**

① innerHTML属性

在读模式下，innerHTML返回与调用元素的所有子节点对应的HTML标记；在写模式下，innerHTML会根据指定的值创建新的DOM树，然后完全替换元素原先所有的子节点。

通过innerHTML添加script元素不会执行，而大多数浏览器都支持插入style

不支持innerHTML的元素有： col, colgroup, frameset, head, html, style, table, tbody, thead, tfoot, tr

②outerHTML

outerHTML返回调用它的元素及其所有子节点的HTML标签，其他同innerHTML相似

③insertAdjacentHTM

// 作为前一个同辈元素插入  
element.insertAdjacentHTML("beforebegin","<p>Hello world!</p>");  
// 作为第一个子元素插入  
element.insertAdjacentHTML("afterbegin","<p>Hello world!</p>");  
// 作为最后一个子元素插入  
element.insertAdjacentHTML("beforeend","<p>Hello world!</p>");  
// 作为后一个同辈元素插入  
element.insertAdjacentHTML("beforeend","<p>Hello world!</p>");

④ scrollIntoView() 滚动浏览器窗口或某个容器元素，调用的元素就可出现在视口

中

## 5.内存与性能问

在删除带有时间处理程序或引用了其他JavaScript对象子树时，就有可能有内存占用问题。使用前述某个属性将含有事件处理程序的元素从文档中删除后，元素与事件绑定关系在内存中并没有删除，这种情况频繁出现，会导致页面内存占用数量增加。因此，在使用innerHTML、outerHTML属性时，最好手工删除被替换元素的所有事件处理程序和JavaScript对象属性。

## 6专有扩展

①文档模式：决定了怎么渲染页面。要强制浏览器以某种模式渲染页面，可以使用

<**meta** http-equiv=**"X-UA-COMPATIBLE"** content=**"IE=Edge"**>

document.documentMode; // 返回使用的文档模式

②children属性

var *childCount* = element.children.length;  
var *firstChild* = element.children[0];

③contains方法：该方法用来检测某个节点是否是该节点的后代节点

console.log(document.documentElement.contains(document.body));

DOM Level3 compareDocumentPosition()方法也可以确定节点间的关系

④innerText

设置innerText属性移除了原来的子节点，完全改变了DOM子树。此外，设置innerText会对文本中存在的HTML语法字符进行编码。

# 十、DOM变化

## 1. 增加了命名空间，并添加了NS结尾的方法

## 2. 样式

热河支持style特性的HTML元素在JavaScript中都有一个对应的style属性。，包含着通过HTML的style特性指定的所有样式，任何CSS属性都可以通过其属性名访问；对于短划线分割的CSS属性名，需要将其转换成驼峰大小写形式

|  |  |
| --- | --- |
| CSS属性 | JavaScript属性 |
| background-image | style.backgroundImage |
| color | style.color |
| display | style.display |
| font-family | style.fontFamily |
| float | style.cssFloat style.styleFloat(IE) |

var *myDiv* = document.getElementById("myDiv");  
*myDiv*.style.backgroundColor = "red";  
*myDiv*.style.width = "100px";  
*myDiv*.style.border = "1px solid black";

① DOM样式属性和方法

如果没有为元素设置style特性，style对象会包含一些默认值

// 多项变化  
*myDiv*.style.cssText = "width:25px; height: 100px; background-color: green";  
// length 和 item 可以迭代元素中定义的CSS属性名，使用getPropertyValue取得属性值  
for(var *i*=0, *len*=*myDiv*.style.length; *i*<*len*; *i*++){  
 var *prop* = *myDiv*.style[*i*]; // 或者 myDiv.style.item(i)  
 var *value* = *myDiv*.style.getPropertyValue(*prop*);  
}

// 移除属性  
*myDiv*.style.removeProperty("border");

② 计算的样式

计算样式包含从其他样式表层叠而来并影响到当前元素的样式信息。计算样式是只读的

// 计算样式  
var *computedStyle* = document.defaultView.getComputedStyle(*myDiv*, null);  
// IE 下  
*computedStyle* = *myDiv*.currentStyle;

## 3. 操作样式表

CSSStyleSheet类型表示的是样式表，包括通过<link>元素包含的样式表和<style>元素中定义的样式表，详细参数见P317

// 兼容IE浏览器，返回样式表  
function *getStyleSheet*(*element*) {  
 return *element*.sheet || *element*.styleSheet;  
}  
var *link* = document.getElementsByTagName("link")[0];  
var *sheet* = *getStyleSheet*(*link*);  
console.log(*sheet*.href);

## 4. 元素大小

①偏移量计算： offsetHeight, offsetWidth, offsetLeft, offsetTop

// 取得元素在页面上的左偏移量  
function *getElementLeft*(*element*){  
 var actualLeft = *element*.offsetLeft;  
 var current = *element*.offsetParent;  
 while(current != null){  
 actualLeft += current.offsetLeft;  
 current = current.offsetParent;  
 }  
 return actualLeft;  
}  
// 取得元素在页面上的上偏移量  
function *getElementTop*(*element*) {  
 var actualTop = *element*.offsetTop;  
 var current = *element*.offsetParent;  
  
 while(current !== null){  
 actualTop += current.offsetTop;  
 current = current.offsetParent;  
 }  
 return actualTop;  
}

②客户区大小

客户区大小指的是元素内容以及内边距所占据的空间大小：clientWidth和clientHeight，客户区的大小就是元素内部的大小。与偏移量相似，每次访问都要重新计算。

function *getViewport*() {  
 if(document.compatMode == "BackCompat"){  
 return {  
 width: document.body.clientWidth,  
 height: document.body.clientHeight  
 };  
 }else{  
 return {  
 width: document.documentElement.clientWidth,  
 height: document.documentElement.clientHeight  
 };  
 }  
}

③滚动大小

scrollHeight, scrollWidth,

scrollLeft: 被隐藏在内容区域左侧的像素数，通过设置可以改变元素的滚动位置, scrollTop: 被隐藏在内容区域上侧的像素数，通过设置可以改变元素的滚动位置。

在确定文档的总高度时，必须取得scrollWidth/clientWidth和scrollHeight/clientHeight中的最大值，才能保证跨浏览器的环境下得到精确的结果。

var *docHeight* = Math.max(document.documentElement.scrollHeight,  
 document.documentElement.clientHeight);  
var *docWidth* = Math.max(document.documentElement.scrollWidth,  
 document.documentElement.clientWidth);

④确定元素大小

function *getBoundingClientRect*(*element*) {  
 if(typeof arguments.callee.offset != "number"){  
 var scrollTop = document.documentElement.scrollTop;  
 var temp = document.createElement("div");  
 temp.style.cssText = "position:absolute;left:0;top:0;";  
 document.body.appendChild(temp);  
 arguments.callee.offset = -temp.getBoundingClientRect().top - scrollTop;  
 document.body.removeChild(temp);  
 temp = null;  
 }  
  
 var rect = *element*.getBoundingClientRect();  
 var offset = arguments.callee.offset;  
 return {  
 left: rec.left+ offset,  
 right: rec.right+ offset,  
 top: rect.top + offset,  
 bottom: rect.bottom + offset  
 };  
}

## 5. 遍历

DOM2级遍历和范围 定义了两个顺序遍历DOM结构的类型: NodeIterator 和 TreeWalker, 这两个类型能够基于给定的起点对DOM结构进行深度优先遍历操作

// 检测浏览器对DOM2级遍历能力的支持  
var *supportsTraverasls* = document.implementation.hasFeature("Traversal", "2.0");  
var *supportsNodeIterator* = (typeof document.createNodeIterator == "function");  
var *supportsTreeWalker* = (typeof document.createTreeWalker == "function");

① NodeIterator

使用document.createNodeIterator()创建实例，接受4个参数：

root: 搜索起点的数中的节点

whatToShow: 表示要访问哪些节点的数字代码, P328

filter: NodeFilter对象，或接受或拒绝某些特定节点的函数

entityReferenceExpansion: html中不用，设置为false

var *filter* = {  
 acceptNode : function (*node*) {  
 return *node*.tagName.toLowerCase() == "p" ?NodeFilter.FILTER\_ACCEPT:NodeFilter.FILTER\_SKIP;  
 }  
};  
var *iterator* = document.createNodeIterator(root, NodeFilter.SHOW\_ELEMENT, *filter*, false);

NodeIterator类型的两个主要方法： nextNode()和previousNode()

var *div* = document.getElementById("div1");  
var *iterator* = document.createNodeIterator(*div*, NodeFilter.SHOW\_ELEMENT, *filter*, false);  
var *filter1* = function (*node*) {  
 return *node*.tagName.toLowerCase() == "li" ?NodeFilter.FILTER\_ACCEPT:NodeFilter.FILTER\_SKIP;  
};  
var *iterators* = document.createNodeIterator(*div*, NodeFilter.SHOW\_ELEMENT, *filter1*, false);  
var *node* = *iterators*.nextNode();  
while(*node* != null){  
 console.log(*node*.tagName);  
 *node* = *iterators*.nextNode();  
}

② TreeWalker: NodeIterator的升级版本

parentNode(): 遍历到当前节点的父节点

firstChild(): 遍历到当前节点的一个子节点

lastChild(): 遍历到当前节点的最后一个子节点

nextSibling(): 遍历到当前节点的下一个同辈节点

previousSibling()：遍历到当前节点的上一个同辈节点

var *walker* = document.createTreeWalker(*div*, NodeFilter.SHOW\_ELEMENT, *filter1*, false);  
*walker*.firstChild();  
*walker*.nextSibling();  
var *node* = *walker*.firstChild();  
while(*node* !== null){  
 console.log(*node*.tagName);  
 *node* = *walker*.nextSibling();  
}

var *walker* = document.createTreeWalker(*div*, NodeFilter.SHOW\_ELEMENT, *filter1*, false);  
  
*walker*.firstChild();  
*walker*.nextSibling();  
var *node* = *walker*.firstChild();  
while(*node* !== null){  
 console.log(*node*.tagName);  
 *node* = *walker*.nextSibling();  
}

对于TreeWalker，NodeFilter.FILTER\_SKIP会跳过相应的节点继续前进到子树中的下一个节点，NodeFilter.FILTER\_REJECT则会跳过相应的节点

## 6. 范围选择

范围是选择DOM结构中特定的部分，然后再执行相应操作的一种手段，使用范围选区可以在删除文档中某些部分的同时，保持文档结构的格式良好，或复制文档中的相应部分

# 十一、事件

JavaScript与HTML之间的交互就是通过事件实现的。事件可以使用侦听器来预定事件，以便事件发生时执行相应的代码

## 1. 事件流

事件流描述的是从页面中接受事件的顺序： 事件冒泡：事件开始时有最具体的元素开始接受，然后逐级向上传播到较不具体的节点；事件捕获：与事件冒泡相反，由最不具体的节点到最具体的节点。

DOM2级事件规定事件流包含三个阶段：事件捕获阶段、处于目标阶段和事件冒泡阶段

## 2. 事件处理程序

原生的JavaScript中，事件处理程序名字以“on”开头。

<**input** type=**"button"** value=**"click me"** onclick=**"***showMessage*()**"**/>

在上述onclick处理程序中，扩展了作用域，在这个函数内部可以像访问局部变量一样访问document元素及该元素本身的成员；如果该元素是一个表单元素，则作用域中还会包含访问表单元素的入口。

<**form** method=**"post"**>  
 <**input** type=**"text"** name =**"username"** value=**""**>  
 <**input** type=**"button"** value=**"Echo Username"** onclick=**"**alert(username,*value*)**"**/>  
</**form**>

缺点：首先存在时差问题，在解析JavaScript程序之前触发事件；扩展事件处理程序的作用域在不同浏览器中会导致不同结果；HTML与JavaScript代码紧密耦合。

### ①DOM0级事件处理程序

通过JavaScript指定事件处理程序，即将一个函数赋值给一个事件处理程序属性。事件处理程序中的this引用的是当前元素

// DOM0级事件处理程序  
var *btn* = document.getElementById("myBtn");  
// 绑定事件  
*btn*.onclick = function () {  
 console.log(this.id);  
};  
// 删除  
*btn*.onclick = null;

### ②DOM2级事件处理程序

DOM2级定义了两个方法处理事件：addEventListener()和removeEventListener()，它们接受三个参数：要处理事件名、事件处理程序和布尔值。布尔值是true，表示在捕获阶段调用事件处理程序；如果是false表示在冒泡阶段调用事件处理程序。

移除时出入的参数与添加处理程序时使用的参数一致，意味着通过addEventListener()添加的匿名程序无法移除。

事件处理程序会按照其添加的先后顺序，依次执行。

// DOM2 级事件处理程序  
*btn*.addEventListener("click", function(){  
 console.log(this.*id*);  
},false);  
*btn*.addEventListener("click", function () {  
 console.log("Hello");  
});  
  
var *handler* = function(){  
 console.log(this.*id*);  
};  
*btn*.addEventListener("click",*handler*,false);  
// removeEventListener中的参数必须与addEventListener的参数一致，否则会移除失败，尤其是handler  
*btn*.removeEventListener("click",*handler*,false);

### ③ IE事件处理程序

attachEvent()和detachEvent(): 接受两个参数：事件处理程序名称(以”on”开头)和事件处理函数，且其添加到到冒泡阶段。使用attachEvent()方法时，事件处理程序会在全局作用域中运行。

*btn*.attachEvent("onclick",function(){  
 console.log("hello");  
});

## 3. 事件对象

在触发DOM上的某个时间时，会产生一个事件对象event，这个对象包含着所有与事件有关的信息。

### ①DOM中的事件对象

无论程序使用什么方法，兼容DOM的浏览器都会将一个event对象传入到事件处理程序中。

|  |  |  |
| --- | --- | --- |
| 属性/方法 | 类型 | 说明 |
| bubbles | Boolean | 表明事件是否冒泡 |
| cancelable | Boolean | 表明是否可以取消事件的默认行为 |
| currentTarget | Element | 事件处理程序当前正在处理事件的那个元素 |
| defaultPrevent | Boolean | 为true表示已调用preventDefault() |
| eventPahse | Integer | 调用事件处理程序的阶段：1捕获阶段，2处于目标阶段，3冒泡阶段 |
| preventDefault | Function | 取消事件的默认行为 |
| stopImmdiatePropagation | Function | 取消事件进一步捕获或冒泡，组织任何事件处理程序被调用 |
| stopPropagation() | Function | 取消事件的进一步捕获或冒泡 |
| target | Element | 事件目标 |
| type | String | 触发事件的类型 |
| view | AbstractView | 与事件相关的抽象视图。等同于发生事件的window对象。 |

## 4. UI事件

### ① load事件

当页面完全加载之后(包括所有图像、JavaScript文件、CSS文件等外部资源),就会触发window上的load事件。也可以在img标签上加入onload事件。

EventUtil.addHandler(window, "load", function(){  
 var image = document.createElement("img");  
 // 要在指定src属性之前先指定事件  
 EventUtil.addHandler(image, "load",function(*event*){  
 *event* = EventUtil.getEvent(*event*);  
 console.log(EventUtil.getTarget(*event*).src);  
 });  
 document.body.appendChild(image);  
 // 新图像只要设置了src属性就会下载  
 image.src = "simle.gif";  
});

IE9+等浏览器支持<script>、<link>元素上的load事件，只设置其src属性不会下载对应的文件，还要把元素添加到文档中才行。

### ② unload事件

当文档被完全卸载后触发unload事件，只要用户从一个页面切换到另一个页面就会触发。利用这个事件最多的情况就是清除引用，避免内存泄漏。

### ③ resize事件

当浏览器窗口被调整到一个新的高度或宽度时，就会触发resize事件。IE、Safari、Chrome和Opera会在浏览器窗口变化了1像素时就触发resize事件，Firefox则只会在用户停止调整时才会触发。因此要注意不要在该事件处理程序中加入大计算量的代码，防止性能问题。

### ④ scroll事件

scroll事件会在文档被滚动期间重复触发，故要尽量保持事件处理程序的代码简单

// srcoll  
*EventUtil*.addHandler(window, "scroll", function () {  
 if(document.compatMode == "CSS1Compat"){  
 console.log(document.documentElement.scrollTop);  
 }else{  
 console.log(document.body.scrollTop);  
 }  
});

## 5. 焦点事件

blur: 在元素失去焦点时触发，不会冒泡

focus：在元素获取焦点时触发，这个事件不会冒泡

focusin：在元素获得焦点时触发，该事件会冒泡

focusout: 在元素是去焦点时触发

当焦点从页面中一个元素移动到另一个元素时，会触发下列事件：

focusout\focusin\blur\DOMFoucesOut\foucs\DOMFocusIn

即使focus和blur不冒泡，也可以在捕获阶段侦听到他们

## 6. 鼠标与滚轮事件

只有在同一个元素上相继触发mousedown和mouseup事件，才会触发click事件

click: 在用户点击主鼠标按钮或按下回车键时触发

dbclick: 在用户双击主鼠标按钮时触发

mousedown: 在用户按下热议鼠标按钮时触发

mouseenter: 在鼠标光标从元素外部首次移动到元素范围之内时触发，该事件不冒泡，移动到后代元素上时也不触发

mouseleave: 在位于元素上方的鼠标移动到元素范围外时触发

mousemove: 鼠标指针在元素内部移动时触发，键盘不能触发

mouseout: 从一个元素上方移动到另一个元素时触发，另一个元素可以是子元素、父元素、兄弟元素

mouseover: 鼠标光标从元素外部首次移入该元素边界之内时触发

mouseup: 用户释放鼠标按钮时触发

### ① 鼠标坐标位置

鼠标事件是在浏览器视口中的特定位置发生的。clientX和clientY表示事件发生时鼠标指针在视口中的水平和垂直坐标。pageX和pageY表示鼠标在页面中的位置；screenX和screenY属性表示的是鼠标相对于整个屏幕的坐标信息。这些位置信息都保存在event中

### ② 修改键Shift Ctrl Alt Meta(windows键盘中的Windows键，Mac中的Cmd)

DOM规定了4个属性来判断修改键是否按下：shiftKey, ctrlKey, altKey,metaKey. 如果为true，则相应的按键被按下

*EventUtil*.addHandler(document, "click", function(){  
event = *EventUtil*.getEvent(event);  
 if(event.shiftKey){  
 console.log("shift is pushed down");  
 }  
});

### ③ 检测鼠标按钮

event对象存在一个button属性，表示按下或释放按钮，该属性有3个值：0表示主鼠标按钮（左键）， 1表示中间的鼠标按钮（滚轮），2表示次鼠标按钮（右键）

Event .addHandler(*div*, "mousedown", function (*event*) {  
 *event* = *EventUtil*.getEvent(*event*);  
 console.log(*EventUtil*.getButton(*event*));  
});

## 7. 键盘与文本事件

有三个键盘事件：

keydown: 当用户按下键盘上的任意键时触发，如果按住不放，会重复触发此事件

keypress: 当用户按下键盘上的字符键时触发，如果按住不放，会重复触发此事件

keyup： 当用户释放键盘上的按建时触发

发生keydown和keyup事件时，event对象的keyCode属性包含一个与键盘按键相对应的代码 P380

Event.addHandler(textbox, "keypress", function (*event*) {  
 *event* = *EventUtil*.getEvent(*event*);

// 使用 String.fromCharCode() 将code转换成实际字符  
 console.log(String.fromCharCode(*EventUtil*.getCharCode(*event*)));  
});

textInput事件只会在用户按下能够出入实际字符的键才会触发，而且只有在可编辑区域才能触发

当DOM发生变动时，能够触发DOM2级的变动事件

## 8. HTML5事件

### ① contextmenu 事件

该事件用于表示何时显示上下文菜单，以便取消默认的上下文菜单而提供自定义的菜单

<**script**>  
 *EventUtil*.addHandler(window, "load", function (*event*) {  
 var div = document.getElementById("context");  
  
 *EventUtil*.addHandler(div, "contextmenu", function (*event*) {  
 *event* = *EventUtil*.getEvent(*event*);  
 *EventUtil*.preventDefault(*event*);  
 var menu = document.getElementById("menu");  
 menu.style.left = *event*.clientX + "px";  
 menu.style.top = *event*.clientY + "px";  
 menu.style.visibility = "visible";  
 });  
  
 *EventUtil*.addHandler(document, "click", function (*event*) {  
 document.getElementById("menu").style.visibility = 'hidden';  
 })  
 });  
</**script**>

### ② beforeunload 事件

让开发人员有可能在页面卸载前阻止这一操作，这一事件在浏览器卸载页面之前触发，可以通过它来取消卸载并使用原有页面

*EventUtil*.addHandler(window, "beforeunload", function (*event*) {  
 *event* = *EventUtil*.getEvent(*event*);  
 var message = "I'am really going to miss you if you go";  
 // 给用户传递消息： IE Fiefox  
 *event*.returnValue = message;  
 // 对Safari和Chrome，需要返回。 Chrome下没有生效  
 return message;  
});

### ③DOMContentLoaded事件

load事件会在页面的所有资源加载完毕后触发，而DOMContentLoaded事件会在形成完整的DOM树之后触发，不理会图像、JavaScript文件等其他资源。

*EventUtil*.addHandler(document, "DOMContentLoaded", function (*event*) {  
 console.log("Content loaded");  
});

### ④ pageshow 和 pagehide 只有Firefox和Opera支持 p393

### ⑤ hashchange

在URL的参数列表发生变化时触发该事件

*EventUtil*.addHandler(window, "hashchange", function (*event*) {  
 console.log("current hash " + location.hash);  
});

## 9. 设备事件

### ① orientationchange事件

该事件是window事件，该事件发生时，可以通过window.orientation获取：0 表示肖像模式（竖屏）；90 表示向左旋转成横向模式；-90表示向右旋转成横向模式

*EventUtil*.addHandler(window, "load", function (*event*) {  
 var div = document.getElementById("mydiv");  
 div.innerHTML = "Current orientation is :" + window.orientation;  
 *EventUtil*.addHandler(window, "orientationchange", function (*event*) {  
 div.innerHTML = "Current orientation is :" + window.orientation;  
 });  
});

### ②触摸事件

touchstart： 当手指触摸屏幕时触发

touchmove : 当手指在屏幕上滑动时，连续触发

touchend : 当手指从屏幕上移开时触发

touchcancel: 当系统停止跟踪触摸时触发

以上这几个事件都会冒泡，也都可以用preventDefault()取消。每个触摸事件的event对象提供了鼠标事件中常见的属性：bubbles、cancelable、view、clientX、clientY、screenX、screenY、 detail、altKey、shiftKey、ctrlKey和metaKey

function *handleTouchEvent*(*event*) {  
 // 只跟踪一次触摸  
 if(*event*.touches.length == 1){  
 var output = document.getElementById("output");  
 switch (*event*.type){  
 case "touchstart":  
 output.innerHTML = "Touch start" + *event*.touches[0].clientX + ","  
 + *event*.touches[0].clientY ;  
 break;  
 case "touchend":  
 output.innerHTML = "Touch end" + *event*.changedTouches[0].clientX + ","  
 + *event*.changedTouches[0].clientY ;  
 break;  
 case "touchmove":  
 *event*.preventDefault(); // 阻止滚动  
 output.innerHTML = "Touch moved" + *event*.changedTouches[0].clientX + ","  
 + *event*.changedTouches[0].clientY ;  
 break;  
 }  
 }  
}

事件触发的先后顺序： touchstart mouseover mousemove mousedown mouseup click touchend

### ③ 手势事件： 两个手指的事件

gesturestart: 当一个手指已经按在屏幕上而另一个手指又触摸屏幕时触发

gesturechagne: 当触摸屏幕的任何一个手指的位置发生变化时触发

gestureend: 当任何一个手指从屏幕上移开时触发

每个触摸事件的event对象提供了鼠标事件中常见的属性：bubbles、cancelable、view、clientX、clientY、screenX、screenY、 detail、altKey、shiftKey、ctrlKey和metaKey。还有两个额外属性：rotation和scale。rotation：两个手指变化引起的旋转角度；scale： 两个手指间距离的变化情况

function *handleGestureEvent*(*event*) {  
 var output = document.getElementById("output");  
 switch (*event*.type){  
 case "gesturestart":  
 output.innerHTML = "Gesture started (rotation = " + *event*.rotation +  
 ", scale = " + *event*.scale + ")";  
 break;  
 case "gestureend":  
 output.innerHTML = "Gesture ended (rotation = " + *event*.rotation +  
 ", scale = " + *event*.scale + ")";  
 break;  
 case "gesturechange":  
 output.innerHTML = "Gesture changed (rotation = " + *event*.rotation +  
 ", scale = " + *event*.scale + ")";  
 break;  
 }  
}

## 10. 内存和性能

### ① 事件委托

对“事件处理程序过多”问题的解决方案就是事件委托。事件委托利用了事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。这样可以使整个页面占用的内存空间更少，能够提升整体性能。适合事件委托技术的事件包括： click、mouseup、 mousedown、 keydown、keyup和keypress。

var *list* = document.getElementById("myLinks");  
*EventUtil*.addHandler(*list*, "click", function (*event*) {  
 *event* = *EventUtil*.getEvent(*event*);  
 var target = *EventUtil*.getTarget(*event*);  
  
 switch (target.id){  
 case "doSomething":  
 document.title = "I changed the document's title.";  
 break;  
 case "goSomewhere":  
 location.href = "http://www.sogou.com";  
 break;  
 case "sayHi":  
 console.log("hi");  
 break;  
 }  
});

### ② 移除事件处理程序

每当将事件处理程序指定给元素时，运行中的浏览器代码与支持页面交互的JavaScript代码之间就会建立一个连接，连接越多，性能越差。使用事件委托技术，还可以在不需要时移除事件处理程序。

从文档中移除带有事件处理程序的元素时，事件处理程序可能还会在内存中；

## 11. 小结

在使用事件时，需要考虑如下一些内存和性能问题：

(1) 有必要限制一个页面中时间处理程序的数量

(2) 建立在事件冒泡机制上的事件委托技术，可有效减少事件处理程序的数量

# 十二、 表单脚本

## 1.表单的基础知识

推荐使用id来获取表单

### ① 提交表单

三种提交按钮写法

<**form**>  
 <**input** type=**"submit"** value=**"Submit Form"**/>  
 <**input** type=**"image"** src=**"../image/search.png"**/>  
 <**button** type=**"submit"**>**submit**</**button**>  
</**form**>

使用JavaScript操作表单

var *form* = document.getElementById("myForm");  
*EventUtil*.addHandler(*form*, "submit", function (*event*) {  
 // 取得事件对象  
 *event* = *EventUtil*.getEvent(*event*);  
 // 阻止默认事件  
 *EventUtil*.preventDefault(*event*);  
});  
  
// 使用JavaScript提交表单。 在调用submit()时不会触发submit事件  
*form*.submit();

防止重复提交： 第一次提交表单后禁用提交按钮；利用onsubmit事件处理程序取消后续表单提交操作。

### ② 重置表单

*EventUtil*.addHandler(*form*, "reset", function (event) {  
 event = *EventUtil*.getEvent(event);  
 *EventUtil*.preventDefault(event);  
});  
  
*form*.reset();

### ③ 表单字段

每个表单都有elements属性，该属性是表单中所有表单元素的集合。每个表单字段在elements集合中的顺序，与他们出现在标记中的顺序相同，可以按照位置和name特性来访问。

// 取得表单中的第一个字段  
var *field1* = *form*.elements[0];  
// 取得名为"textbox1"的字段  
var *filed2* = *form*.elements['textbox1'];  
// 取得表单中包含的字段数量  
var *filedCount* = *form*.elements.length;

### ④ 共有的表单字段属性

disabled: boolean，表示当前字段是否被禁用

form: 指向当前字段所属表单的指针，只读；

name: 当前字段的名称

readOnly: boolean, 表示当前字段是否只读

tabIndex: 表示当前字段的切换(tab)序号

value: 提交给服务器的值

// 把焦点设置到当前字段  
*field1*.focus();  
// 禁用字段  
*field1*.disable = true;

HTML5为表单字段新增一个autofocus属性，自动把焦点移动到相应的字段。

### ⑤ 共有的表单字段事件

处理支持鼠标、键盘、更改和HTML事件外，所有表单字段都支持下列3个事件：

blur: 当前字段失去焦点时触发

change: 对于<input> 和 <textarea> 元素，在它们失去焦点且value值改变是触发，对于<select>元素，在其选项改变时触发；

focus: 当前字段获得焦点时触发。

focus和change事件发生的先后不能保证。

*EventUtil*.addHandler(textbox, "change", function (event) {  
 event = *EventUtil*.getEvent(event);  
 var target = *EventUtil*.getTarget(event);  
  
 if(/[^\d]/.test(target.value)){  
 target.style.backgroundColor = "red";  
 }else{  
 target.style.backgroundColor = "";  
 }  
});

## 2. 选择文本

选择(select)事件: 选择文本框中的文本时触发 P420

### ① 过滤输入

屏蔽字符

EventUtil.addHandler(textbox, "keypress", function (*event*) {  
 *event* = EventUtil.getEvent(*event*);  
 var target = EventUtil.getTarget(*event*);  
 var charCode = EventUtil.getCharCode(*event*);  
 // 避免过滤掉 向上键、向下键、退格键和删除键， charCode > 9  
 if(!/\d/.test(String.fromCharCode(charCode)) && charCode > 9 && !*event*.ctrlKey){  
 EventUtil.preventDefault(*event*);  
 // do other things  
 }  
});

### ② 操作剪贴板

6个剪贴板事件：beforecopy、copy、beforecut、cut、beforepaste、paste

// 剪贴板  
getClipboardText: function (*event*) {  
 var clipboardData = (*event*.clipboardData || window.clipboardData);  
 return clipboardData.getData("text");  
},  
setClipboardText: function (*event*, *value*) {  
 if(*event*.clipboardData){  
 return *event*.clipboardData.setData("text/plain", *value*);  
 }else if(window.clipboardData){  
 return window.clipboardData("text", *value*);  
 }  
}

检测粘贴过来的值是否符合要求：

*EventUtil*.addHandler(textbox, "paste", function (*event*) {  
 *event* = *EventUtil*.getEvent(*event*);  
 var text = *EventUtil*.getClipboardText(*event*);  
 if(!/^\d\*$/.test(text)){  
 *EventUtil*.preventDefault(*event*);  
 }  
});

## 3. 表单序列化

在JavaScript中可以利用表单字段的type属性，连同name和value属性一起实现对表单的序列化。

对表单字段的名称和值进行URL编码，使用&号分割

不发送禁用的表单字段

只发送勾选的复选框和单选按钮

不发送type为“reset”和“button”的按钮

多选选择框中的每个选中的值单独一个条目

在点击提交按钮提交表单的情况下，也会发送提交按钮；否则不发送提交按钮

如果<option>元素中没有value特性，则是<option>元素的文本值

// 表单序列化  
function *serialize*(*form*) {  
 var parts = [],  
 field = null,  
 i,  
 len,  
 j,  
 optLen,  
 option,  
 optValue;  
 for(i=0, len=*form*.elements.length; i<len; i++){  
 field = *form*.elements[i];  
 switch (filed.type){  
 case "select-one":  
 case "select-multiple":  
 if(filed.name.length){  
 for(j=0, optLen=field.options.length; j<optLen; j++){  
 option = field.options[j];  
 if(option.selected){  
 optValue = "";  
 if(option.hasAttribute){  
 optValue = option.hasAttribute("value") ? option.value: option.text;  
 }else{  
 optValue = option.attributes["value"].specified ? option.value : option.text;  
 }  
 parts.push(encodeURIComponent(field.name) + "=" + encodeURIComponent(optValue));  
 }  
 }  
 }  
 break;  
 case undefined: // 字段集  
 case "file":  
 case "submit":  
 case "reset":  
 case "button":  
 break;  
 case "radio":  
 case "checkbox":  
 if(!field.checked){  
 break;  
 }  
 /\*执行默认动作\*/  
 default:  
 if(field.name.length){  
 parts.push(encodeURIComponent(field.name) + "=" + encodeURIComponent(field.value));  
 }  
 }  
 }  
 return parts.join("&");  
}

# 十三、使用canvas绘图

## 1. 基本用法

要使用<canvas> 元素，必须先设置width和height属性，指定可以绘图的区域大小。

if(drawing.getContext){ // 判断浏览器是否支持画布  
 var context = drawing.getContext("2d");  
}

使用toDataURL()方法可以导出在<canvas>元素上绘制的图像，该方法接受一个参数，即图像的MIME类型格式。绘制到画布上的图像要来自相同的域。

## 2. 2D上下文

### ① 填充和描边

context.fillStyle = “red” ： 填充为红色

context.strokeStyle = “#0000ff” : 描边

### ② 绘制矩形

与矩形有关的方法有： fillRect()、strokeRect()和clearRect()。这三个方法都接受四个参数： x、y、矩形的宽度和高度。

// 绘制红色矩形  
context.fillStyle = "#ff0000"; // 矩形内填充的颜色  
context.fillRect(10, 10, 50, 50); //矩形的大小  
//绘制半透明的蓝色矩形  
context.fillStyle = "rgba(0,0, 255, 0.5)";  
context.fillRect(30, 30, 50, 50);

描边：

// 绘制红色描边矩形  
context.strokeStyle = "#ff0000";  
context.strokeRect(10, 10, 50, 50);  
// 绘制半透明的蓝色描边矩形  
context.strokeStyle = "rgba(0,0, 255, 0.5)";  
context.strokeRect(30, 30, 50, 50)

### ③ 绘制路径

要绘制路径，首先要调用beginPath()方法

arc(x, y, radius, startAngle, endAngle, counterclockwise): 以(x, y)为圆心绘制一条弧线，弧线半径为radius，起始和结束角度分别为startAngle和endAngle.

arcTo(x1, y1, x2, y2, radius): 从上一点开始绘制一条弧线，到(x2,y2)为止，并且以给定的半径radius穿过(x1, y1)

lineTo(x,y): 从上一点开始绘制一条直线，到(x,y)为止

moveTo(x,y): 将绘图游标移动到(x,y), 不画线

rect(x,y,width,heigth): 从点(x,y)开始绘制一个矩形，宽高分别由width和height指定。

如果想绘制一条连接到路径起点的线条，可以调用closePath().调用context.stroke()对路径描边，context.fill()对路径填充

### ④ 绘制文本

fillText() 和 strokeText() P451

### ⑤ 变换

rotate(angle): 围绕圆点旋转图像angle弧度。

scale(scaleX, scaleY): 缩放图像

translate(x,y): 将坐标原点移动到(x,y)

### ⑥ 绘制图像

使用drawImage()方法绘制图像

context.drawImage(image, x,y, width, height, positonX,positionY)

### ⑦ 阴影

context.(shadowColor, shadowOffsetX, shadowOffsetY, shadowBlur)

### ⑧ 渐变

渐变由CanvasGradient实例表示。

if (drawing.getContext) {  
 var context = drawing.getContext("2d");  
 // 设置阴影  
 context.shadowOffsetX = 5;  
 context.shadowOffsetY = 5;  
 context.shadowBlur = 4;  
 context.shadowColor = "rgba(0, 0, 0, 0.5)";  
 // 绘制径向渐变  
 var radialGradient =

context.createRadialGradient(55,55,10,55,55,30);  
 radialGradient.addColorStop(0,"white");  
 radialGradient.addColorStop(1,"black");  
 // 绘制红色矩形  
 context.fillStyle = "#ff0000"; // 矩形内填充的颜色  
 context.fillRect(10, 10, 50, 50); //矩形的大小  
  
 // 绘制渐变矩形, 线性渐变  
 var gradient = context.createLinearGradient(30, 30, 70, 70);  
 gradient.addColorStop(0,"white");  
 gradient.addColorStop(0.5,"yellow");  
 gradient.addColorStop(1,"black");  
  
  
 context.fillStyle = radialGradient; // 绘制渐变图形  
 context.fillRect(30, 30, 50, 50);  
}

# 十四、 HTML5脚本

## 1. 跨文档消息传递 (XDM)

XDM 指的是来自不同域的页面间传递消息。例如www.sogou.com 域的页面与位于一个内嵌框架中的p2p.sogou.com域中的页面通信。

核心方法是 postMessage() P481

## 2. 原生拖放

拖动事件：dragstart, drag, dragend

某个元素被拖动到一个有效的放置目标上时，会依次触发：dragenter, dragover, dragleave或drop.

通过重写dragenter和dragover事件的默认行为，可以把任何元素变成有效的放置目标。

dataTransfer对象用来在拖放事件中传递数据。有两个传送数据的方法：

event.dataTransfer.setData(“text/URL”, data)

event.dataTransfer.getData(“text/URL”)

控制被拖动元素能够执行哪些放置行为的两个属性：

event.dataTransfer.dropEffec：只能在ondragenter事件处理程序中设置。

event.dataTransfer.effectAllowed：只能在ondragstart事件中设置。

## 3. 媒体类型

video和 audio

# 十五、错误处理

## 1. 错误类型

Error， EvalError，RangError,ReferenceError, SyntaxError, TypeError, URIError

## 2. 抛出错误

throw方式：

throw 12345

throw “Hello world!”

throw true

throw {name: “JavaScript”}

throw new Error(“something is wrong”)

可以继承 Error(): CustomError.prototype = new Error()

## 3. 常见的错误类型

### ① 类型转换错误

类型转换错误发生在使用某个操作符，或者使用其他可能会自动转换的数据类型的语言结构时发生。 在使用相等(==)和不相等(!=)操作符，或者在if、for及while等流控制语句中使用非布尔值时，最常发生类型转换错误。

在控制语句中使用非布尔值，是极为常见的一个错误来源。

### ② 数据类型错误

通过添加类型检测语句，可以确保函数不会那么容易出错。

在确切知道应该传入什么类型的情况下，最好使用instanceof来检测其数据类型。

基本类型使用typeof来检测，对象的值应该使用instanceof来检测，面向公众的API则必须无条件地执行类型检查，以确保函数始终能够正常执行。

// 在确切知道应该传入什么类型的情况下，最好使用instanceof来检测其数据类型  
// 基本类型使用typeof来检测，对象的值应该使用instanceof来检测  
function *reverseSOrt*(*values*){  
 if(*values* instanceof Array){ }  
}  
function *getQueryString*(*url*){  
 if(typeof *url* == "string"){}  
}

### ③ 通信错误

第一种通信错误与格式不正确的URL或发送的数据有关。最常见的问题是将数据发送给服务器之前，没有使用encodeURIComponent()对数据进行编码。

对于查询字符串，必须使用encodeURIComponent()编码。

第二种是使用Ajax通信时，服务器返回的数据错误。

## 4. 区分致命错误和非致命错误

非致命错误：

不影响用户的主要任务；只影响页面的一部分；可以恢复；重复相同操作可以消除错误

致命错误：

应用程序根本无法继续运行；错误明显影响到了用户的主要操作；会导致其他连带错误。

## 5. 将错误日志保存到服务器

function *logError*(*sev*, *msg*){  
 var img = new Image();  
 img.src = "log.do?sev" + encodeURIComponent(*sev*) +"&msg="  
 + encodeURIComponent(*msg*);  
}

## 6. 调试技术

### ① 将消息记录到控制台

console.log(message)

console.error(message)

console.info(message)

console.warn(message)

function *log*(*message*){  
 if(typeof console == "object"){  
 console.log(*message*);  
 }else if(typeof opera == "object"){  
 opera.postError(*message*);  
 }else if(typeof java == "object" && typeof java.lang == "object"){  
 java.lang.System.out.println(*message*);  
 }  
}

### ② 将消息记录到当前页面

在页面中开辟一小块区域，用来显示消息。适合不支持JavaScript控制台的IE7及更早版本。

function *log2*(*message*){  
 var console = document.getElementById("debuginfo");  
 if(console == null ){  
 console = document.createElement("div");  
 console.id = "debuginfo";  
 console.style.background = "#dedede";  
 console.style.border = "1px solid silver";  
 console.style.padding = "5px";  
 console.style.width = "400px";  
 console.style.position = "absolute";  
 console.style.right = "0px";  
 console.style.top = "0px";  
 document.body.appendChild(console);  
 }  
 console.innerHTML += "<p>" + *message* + "</p>";  
}

## 7. 常见的IE错误

### ① 操作终止

在IE8之前的版本中，存在一个最难调试的错误：操作终止。在修改尚未加载完成的页面时，就会发生操作终止错误。

JavaScript代码在页面尚未加载完毕时就要修改document.body， 通过appendChild，innerHTML修改该元素的父元素或祖先元素，而且<script>元素不是<body>元素的直接子元素。可以将appendChild换成insertBefore，或者把<script>作为<body>元素的直接子元素。

### ② 未找到成员

IE中的未找到成员错误，就是由于垃圾收集例程导致的。

document.onclick = function () {  
 var event = window.event;  
 setTimeout(function(){  
 event.returnValue = false;  
 }, 1000);  
};

### ③ 未知运行时错误

当使用innerHTML或outerHTML以下列方式指定HTML时，就会发生未知运行时错误：一是把块元素插入到行内元素时，一是访问表格任意部分的任意属性时。

# 十六、JSON

## 1. 语法

与JavaScript对象字面量相比，JSON对象有两个地方不一样。

首先，没有声明变量；其次，没有末尾的分号；对象的属性必须加双引号，字符串必须是双引号。

## 2. JSON解析

通过JSON对象的stringfy()方法序列化一个对象。在序列化JavaScript对象时，所有函数及原型成员都会被有意忽略，不体现在结果中。此外，值为undeifned的任何属性值也会被跳过。结果中都是值为有效的JSON数据类型的实例属性。

var *book* = {  
 title: "professional JavaScript",  
 authors: ["Nicholas C. Zakas"],  
 edition: 3,  
 year: 2011  
};  
var *jsonText* = JSON.stringify(*book*);

### ① 序列化选项-- 过滤结果

JSON.stringify()还可以接收另外两个参数：过滤器（数组或函数），表示是否在JSON字符串中保留缩进的参数

// 表示jsonTextFilter中只有title和edition属性  
// {"title":"professional JavaScript", "edition":3}  
var *jsonTextFilter* = JSON.stringify(*book*, ["title","edition"]);

如果是函数，则包含两个参数：属性名和值

// 如果函数返回时undefined，那么对应的属性会被忽略  
var *jsonTextFunction* = JSON.stringify(*book*, function (*key*, *value*) {  
 switch(*key*) {  
 case "authors":  
 return *value*.json(",")  
 case "year":  
 return 5000;  
 case "edition":  
 return undefined;  
 }  
});

### ② 序列化选项-- 字符串缩进

// 4个空格的缩进  
var *jsonTextInd* = JSON.stringify(*book*, null, 4);  
// 字符串缩进：空格替换成字符串  
var *jsonTextString* = JSON.stringify(*book*, null, "--");

③ toJson() 方法

JSON.stringify()方法不能够满足对象进行自定义序列化的需求，可以在对象中定义toJSON()方法，返回其自身的JSON数据格式。

序列化顺序

(1) 如果存在toJSON() 方法而且能通过它获取到有效值，则调用该方法，否则返回对象本身。

(2) 如果提供第二个参数，应用这个函数过来，传入函数过滤器的参数是第(1)步返回的值

(3) 对第(2)步返回的每个值进行相应的序列化

(4) 如果提供了第三个参数，执行相应的格式化。

## 3. 解析选项

JSON.parse()方法也可以接收另一个函数参数，同JSON.stringify() 接收的函数相同

var *bookParse* = JSON.parse(*jsonText*, function (*key*, *value*) {  
 if (*key* == "releaseDate"){  
 return new Date(*value*);  
 } else {  
 return *value*;  
 }  
});

# 十七、Ajax与Comet

Ajax技术的核心是XMLHttpReques对象(XHR)

## 1. XHR对象

IE7+、Firfox、Opera、Chrome和Safari都支持原生的XHR对象

var *xhr* = new XMLHttpRequest();

### ① XHR的用法--open and send method

使用XHR对象，第一个要调用的方法就是open()，它接受3个参数：要发送的请求类型（get、post等）、请求的URL和是否异步发送请求（false为同步）。

只能向同一个域中使用相同端口和协议的URL发送请求，调用open方法并不会真正发送请求。

*xhr*.open("get", "../html/drag.html", false);

*xhr*.send(null);

xhr.send方法接受一个参数，即要作为请求主体发送的数据。如果不发送数据，则必须传入null

同步请求的响应数据会自动填充到XHR对象的属性中：

responseText：作为响应主体被返回的文本

responseXML：如果响应的内容类型为”text/xml”或”application/xml”，使用该属性

status：响应的HTTP状态

statusText：HTTP状态的说明

发送异步请求时，要检测XHR对象的readyState属性：0，未初始化；1，启动，调用open方法，未调用send方法；3，接收到部分数据；4，接收完全部数据。

通常，我们只对readyState值为4进行检测，必须在调用open()方法之前指定onreadystatechange事件处理程序才确保跨浏览器兼容性。

xhr.onreadystatechange = function () {  
 if(xhr.readyState == 4) {  
 if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){  
 console.log(xhr.responseText);  
 } else {  
 console.log("Request was unsuccessful: " + xhr.status);  
 }  
 }  
};

使用abort()取消异步请求：xhr.abort()，终止请求之后，还应该对XHR对象进行解引用操作。由于内存原因，不建议重用XHR对象。

### ②HTTP头部信息

设置头部信息：

xhr.setRequestHeader("MyHeader", "value");

获取头部信息：

var myHeader = *xhr*.getResponseHeader("MyHeader");  
var allHeaders = *xhr*.getAllResponseHeaders();

### ③ GET请求

对XHR来说，传入open()方法的URL末尾的查询字符串必须经过正确的编码，否则会发生错误。

### ④POST请求

POST请求应该把数据作为请求的主体提交，通过send()方法发送数据。可以通过将Content-Type头部信息设置为application/x-www-form-urlencoded来模拟表单提交内容。post请求比get请求的性能要低

## 2. 进度事件

为了确保正常执行，必须在调用open()方法之前添加 onprogress事件

*xhr*.onprogress = function (*evnet*) {  
 var divStatus = document.getElementById("status");  
 if (event.lengthComputable) {  
 divStatus.innerHTML = "Received " + event.position + " of " + event.totalSize + " bytes";  
 }  
};

## 3. 跨源资源共享

为了安全，跨域XHR对象有一些限制：

不能使用setRequestHeader()设置自定义头部

不能发送和接受cookie

调用getAllResponseHeaders()方法总会返回空字符串

建议： 对于本地资源，最好使用相对URL，访问远程资源时再使用绝对URL

## 4. Comet

Ajax是一种从页面向服务器请求数据的技术，而Comet则是一种服务器向页面推送数据的技术。有两种实现：长轮询和流。

长轮询：页面向服务器发送一个请求，然后服务器一直保持连接打开，直到有数据可发送。数据发送完成之后，浏览器关闭连接，随即又发起一个到服务器的新请求。

长轮询和短轮询最大的区别是 服务器如何发送数据。

HTTP流：它在页面的整个生命周期内只是用一个HTTP连接，浏览器向服务器发送一个请求，服务器保持连接打开，周期性的向浏览器发送数据。

// Commet， 长轮询  
function *createStreamingClient*(*url*, *progress*, *finished*){  
 var xhr = new XMLHttpRequest(),  
 received = 0;  
 xhr.open("get", *url*, true);  
 xhr.onreadystatechange = function () {  
 var result;  
  
 if(xhr.readyState == 3){  
 // 取得最新数据并调整计数器  
 result = xhr.responseText.substring(received);  
 received += result.length;  
 // 调用progress回调函数  
 *progress*(result);  
 } else if (xhr.readyState == 4) {  
 *finished*(xhr.responseText);  
 }  
 };  
 xhr.send(null);  
 return xhr;  
}  
  
var *client* = *createStreamingClient*("streaming.php", function (*data*) {  
 console.log("Received: " + *data*);  
}, function (*data*) {  
 console.log("done");  
});

### ① 向服务器发送事件 – SSE

SSE是围绕只读Comet交互退出的API。SSE API用于创建到服务器的单向连接，服务器通过这个连接可以发送热议数量的数据，服务器响应的MIMI类型必须是text/event-stream. SSE 支持短轮询，长轮询和HTTP流，且能在断开时自动重连。

// SSE server-sent events  
var *soucrce* = new EventSource("myevent.php");  
source.onmessage = function (*event*) {  
 var data = *event*.data;  
 // 处理数据  
};

### ② 事件流

响应的格式是纯文本，只有包含data:的数据行后边有空行时，才会触发message事件

### ③ web sockets

Web Sockets的目标是在一个单独的持久连接上提供全双工、双向通信。Web Socket使用了自定义的协议，所以未加密的连接是ws://，加密的连接是wss://。

WebSocket有一个表示当前状态的readyState属性：

WebSocket.OPENING(0), WebSocket.OPEN(1), WebSocket.CLOSING(2), WebSocket.CLOSE(3)

WebSocket只能发送纯文本数据，对非文本数据必须进行序列化。

var *socket* = new WebSocket("ws://www.example.com/focus.php");  
*socket*.send("hello world!");  
*socket*.onmessage = function (*event*) {  
 var data = *event*.data;  
 // other operate  
};  
*socket*.close();

其他事件：

open: 在成功建立连接时触发

error: 在发生错误时触发，连接不能持续

close: 在连接关闭时触发

onmessage: 在服务器向客户端发送消息时触发

## 5. 跨域技术

### ① 图像ping

图像ping是与服务器进行简单、单向的跨域通信，请求数据通过查询字符串形式发送。

var img = new Image();  
img.onload = img.onerror = function () {  
 console.log("done!");  
};  
img.src = *url*;

### ② JSONP

JSONP由两部分构成：回调函数和数据。JSONP是从其他域中加载代码执行，如果其他域不安全，很可能会包含恶意代码。因此一般用在自已运维的web服务。

var script = document.createElement("script");  
script.src = *url* + (*url*.indexOf("?") == -1 ? "?":"&") + "callback=" + *callback* ;  
document.body.insertBefore(script, document.body.firstChild);

# 十八、 高级技巧

1. 安全类型检查

2. 作用域安全的构造函数

3. 惰性载入函数

4. 函数绑定

5. 函数壳里化

6. 防篡改对象

7. 定时器

8. 函数节流

9. 自定义事件--观察者模式

# 十九、 离线应用与客户端存储

## 1. 离线检测

使用navigator.onLine来检测设备是否离线，HTML5还定义了两个事件：online和offline，当网络从离线变为在线或者从在线变为离线时触发。

## 2. Cookie

### ① 限制

cookie存储在客户端计算机上，只能被设置cookie的域名访问。

cookie的总数根据浏览器的不同，有不同的个数限制，最好保持在30以下

浏览器对cookie的大小也有限制，最好将整个cookie长度限制到4095B

### ② cookie的构成

名称：cookie的名称不区分大小写

值：cookie的名和值必须经过URL编码

域： cookie对哪个域是有效的

路径：对于指定路径有效

失效时间：GMT格式的日期，如果是以前的时间，则cookie会被立即删除

安全标志：指定后，cookie只有在使用SSL连接时才发送到服务器

例子：

Set-Cookie: name=value; domain=.wrox.com; path=/; secure

### ③ 子Cookie

为了绕开单域名下cookie数限制，可以使用子cookie

name=nam1=value1&name2=value2&name3=value3;

### ④ cookie的思考

HTTP专有的cookie可以从浏览器或者服务器设置，但是只能从服务器读取

由于所有的cookie都会由浏览器作为请求头发送，所以在cookie中存储大量信息会影响到特定域的请求性能。

一定不要在cookie中存储重要和敏感的数据

var *CookieUtil* ={  
 get: function (*name*) {  
 var cookieName = encodeURIComponent(*name*) + "=",  
 cookieStart = document.cookie.indexOf(cookieName),  
 cookieValue = null;  
 if (cookieStart > -1) {  
 var cookieEnd = document.cookie.indexOf(";", cookieStart);  
 if (cookieEnd == -1) {  
 cookieEnd = document.cookie.length;  
 }  
 cookieValue = decodeURIComponent(document.cookie.substring(cookieStart+cookieName.length, cookieEnd));  
 }  
 return cookieValue;  
 },  
   
 getUseReg: function (*name*) {  
 var arr,reg=new RegExp("(^| )"+*name*+"=([^;]\*)(;|$)");  
 if(arr=document.cookie.match(reg))  
 return decodeURIComponent(arr[2]);  
 else  
 return null;  
 },  
  
 set: function (*name*, *value*, *expires*, *path*, *domain*,*secure*) {  
 var cookieText = encodeURIComponent(*name*) + "=" +  
 encodeURIComponent(*value*);  
 if (*expires* instanceof Date){  
 cookieText += "; expires=" +*expires*.toGMTString();  
 }  
 if (*path*) {  
 cookieText += "; path=" + *path*;  
 }  
 if (*domain*) {  
 cookieText += "; domain=" + *domain*;  
 }  
 if (*secure*) {  
 cookieText += "; secure";  
 }  
  
 document.cookie = cookieText;  
 },  
 unset: function (*name*, *path*, *domain*, *secure*) {  
 this.set(*name*, "", new Date(0), *path*, *domain*, *secure*);  
 },  
 getSubCookie: function (*name*, *subName*) {  
 var subCookies = this.getAllCookie(*name*);  
 if (subCookies) {  
 return subCookies[*subName*];  
 } else {  
 return null;  
 }  
 },  
 getAllCookie: function (*name*) {  
 var cookieName = encodeURIComponent(*name*) + "=",  
 cookieStart = document.cookie.indexOf(cookieName),  
 cookieValue = null,  
 cookieEnd,  
 subCookies,  
 i,  
 parts,  
 result = {};  
 if (cookieStart > -1) {  
 cookieEnd = document.cookie.indexOf(";", cookieStart);  
 if (cookieEnd == -1){  
 cookieEnd = document.cookie.length;  
 }  
 cookieValue = document.cookie.substring(cookieStart+cookieName.length, cookieEnd);  
 if (cookieValue.length > 0) {  
 subCookies = cookieValue.split("&");  
 for (i=0, len=subCookies.length; i<len; i++) {  
 parts = subCookies[i].split("=");  
 result[decodeURIComponent(parts[0])] = decodeURIComponent(parts[i]);  
 }  
 return result;  
 }  
 }  
 return null;  
 },  
  
 setSubCookie: function (*name*, *subName*, *value*, *expires*, *path*, *domain*, *secure*) {  
 var subcookies = this.getAllCookie(*name*) || {};  
 subcookies[*subName*] = *value*;  
 this.setAllCookies(*name*, subcookies, *expires*, *path*, *domain*, *secure*);  
 },  
  
 setAllCookies: function (*name*, *subcookies*, *expires*, *path*, *domain*, *secure*) {  
 var cookieText = encodeURIComponent(*name*) + "=",  
 subcookieParts = new Array(),  
 subName;  
  
 for (subName in *subcookies*) {  
 if (subName.length > 0 && *subcookies*.hasOwnProperty(subName)) {  
 subcookieParts.push(encodeURIComponent(subName) + "=" + encodeURIComponent(*subcookies*[subName]));  
 }  
 }  
  
 if (subcookieParts.length > 0) {  
 cookieText += subcookieParts.join("&");  
 if (*expires* instanceof Date){  
 cookieText += "; expires=" +*expires*.toGMTString();  
 }  
 if (*path*) {  
 cookieText += "; path=" + *path*;  
 }  
 if (*domain*) {  
 cookieText += "; domain=" + *domain*;  
 }  
 if (*secure*) {  
 cookieText += "; secure";  
 }  
 } else {  
 cookieText += "; expires=" + (new Date(0)).toGMTString();  
 }  
  
 document.cookie = cookieText;  
 },  
 unsetSubCookie: function (*name*, *subName*, *path*, *domain*, *secure*) {  
 var subCookies = this.getAllCookie(*name*);  
 if(subCookies){  
 delete subCookies[*subName*];  
 this.setAllCookies(*name*, subCookies, null, *path*, *domain*, *secure*);  
 }  
 },  
  
 unsetAll: function (*name*, *path*, *domain*, *secure*) {  
 this.setAllCookies(*name*, null, new Date(0), *path*, *domain*, *secure*);  
 }  
};

## 3. Web存储机制

Web Storage的目的是克服由cookie带来的一些限制，当数据需要被严格控制在客户端，无须将数据返回到服务器时。

* 提供一种在cookie之外的存储会话数据的途径
* 提供一种存储大量可以跨会话存在的数据机制

### ① Storage类型

Storage类型提供了一下方法：

* clear(): 删除所有值
* getItem(name): 根据名字获取对应的值
* key(index): 获得index位置处的值的名字
* removeItem(name): 删除由name指定的名值对
* setItem(name, value): 为指定的name设置一个对应的值

### ② sessionStorage对象

sessionStorage对象存储特定于某个会话的数据，数据只保持到浏览器关闭

### ③ localStorage对象

要访问一个localStorage对象，页面必须来自同一个域名（子域名无效），使用同一种协议，在同一个端口上。

//使用方法存储数据  
localStorage.setItem("name", "wang");  
// 使用方法读取数据  
var *name* = localStorage.getItem("name");

# 二十、最佳实践

## 1. 可维护性

可理解性、直观性、可适应性、可扩展性、可调式性

分离应用逻辑和事件处理程序原则：

勿将event对象传给其他方法；值传递来自event对象中所需的数据

任何可以在应用层面的动作都应该可以在不执行任何事件处理程序的情况下进行

任何事件处理程序都应该能处理事件，然后将处理转交给应用逻辑

尊重对象所有权、尽可能避免全局变量

// 两个全局变量--避免  
var *name* = "Nicholas";  
function *sayName*() {  
 console.log(*name*);  
}  
// 一个全局变量-- 推荐  
var *MyApplication* = {  
 name: "Nicholas",  
 sayName: function () {  
 console.log(this.name);  
 }  
};

**避免与null进行比较**：

如果值应为一个引用类型，使用instanceof操作符检查其构造函数

如果值应为一个基本类型，使用typeof检查其类型

如果是希望对象包含某个特定的方法，则使用typeof操作符确保指定名字的方法存在于对象上

## 2. 性能

### ① 避免全局查找

随着作用域链的作用域数量的增加，访问当前作用域以外的变量的时间也在增加，而访问全局变量总比访问局部变量慢，因为需要遍历作用域链。

将在一个函数中会用到多次的全局对象存储为局部变量可以提高性能

### ② 在性能非常重要的地方必须避免使用with语句

### ③ 选择正确的方法

避免不必要的属性查找。

使用变量和数组要比访问对象上的属性更有效率。对象上的任何属性查找都要比访问变量或者数组花费的时间更长

一旦多次使用对象属性，应该将其存储在局部变量中。

如果可以使用数字化的数组位置访问，也可以使用命名属性，那么使用数字位置会更快。

优化循环：

* 减值迭代
* 简化终止条件
* 简化循环体
* 使用后测试循环 – do{}while()

展开循环：针对大数据集使用展开循环可以节省很多时间，但对于小数据集，额外的开销得不偿失

避免双重解释： 超时函数不要使用字符串： setTimeout(“console.log(‘123’)”, 500)

其他能提升性能的方法：

* 原生方法较快
* switch语句较快
* 位运算符较快

## 3. 最小化语句数

多个变量声明 var count=5, color=”blue” ... ;

插入迭代值： var name=values[i++];

尽量使用数组和对象字面量表达式来消除不必要的语句

## 4. 优化DOM交互

更改DOM显示页面非常消耗性能，要最小化现场更新

使用innerHTML处理大的DOM更改，要比标准的DOM方法创建更快

使用事件代理：用到了事件冒泡，添加监听的位置。

# 附录 A

/\*\*  
 \* Created by 王忠珂 on 2016/8/31.  
 \*/  
var *EventUtil* = {  
 addHandler: function (*element*, *type*, *handler*) {  
 if(*element*.addEventListener){  
 *element*.addEventListener(*type*, *handler*, false);  
 }else if(*element*.attachEvent){  
 *element*.attachEvent("on" + *type*, *handler*);  
 }else{  
 *element*["on" + *type*] = *handler*;  
 }  
 },  
  
 removeHandler: function (*element*, *type*, *handler*) {  
 if(*element*.removeEventListener){  
 *element*.removeEventListener(*type*, *handler*, false);  
 }else if(*element*.detachEvent){  
 *element*.detachEvent("on" + *type*, *handler*);  
 }else{  
 *element*["on" + *type*] = null;  
 }  
 },  
  
 getEvent: function (*event*) {  
 return *event* ? *event* : window.event;  
 },  
  
 getTarget: function (*event*) {  
 return *event*.target || *event*.srcElement;  
 },  
  
 preventDefault: function (*event*) {  
 if(*event*.preventDefault){  
 *event*.preventDefault();  
 }else{  
 *event*.returnValue = false;  
 }  
 },  
  
 stopPropagation: function (*event*) {  
 if(*event*.stopPropagation){  
 *event*.stopPropagation();  
 }else{  
 *event*.cancelBubble = true;  
 }  
 },  
  
 getButton: function (*event*) {  
 if(document.implementation.hasFeature("MouseEvents", "2.0")){  
 return *event*.button;  
 } else {  
 switch (*event*.button){  
 case 0:  
 case 1:  
 case 3:  
 case 5:  
 case 7:  
 return 0;  
 case 2 :  
 case 6:  
 return 2;  
 case 4:  
 return 1;  
 }  
 }  
  
 },  
  
 getCharCode: function (*event*) {  
 if(typeof *event*.charCode == "number")  
 return *event*.charCode;  
 else{  
 return *event*.keyCode;  
 }  
 },  
  
 // 剪贴板  
 getClipboardText: function (*event*) {  
 var clipboardData = (*event*.clipboardData || window.clipboardData);  
 return clipboardData.getData("text");  
 },  
 setClipboardText: function (*event*, *value*) {  
 if(*event*.clipboardData){  
 return *event*.clipboardData.setData("text/plain", *value*);  
 }else if(window.clipboardData){  
 return window.clipboardData("text", *value*);  
 }  
 },  
 bind: function (*fn*, *context*) {  
 return function () {  
 return *fn*.apply(*context*, arguments);  
 }  
 },  
 curryBind: function (*fn*, *context*) {  
 var args = Array.prototype.slice.call(arguments, 2);  
 return function () {  
 var innerArgs = Array.prototype.slice.call(arguments);  
 var finalArgs = args.concat(innerArgs);  
 return *fn*.apply(*context*, finalArgs);  
 }  
 },  
 /\*setTimeoutTest(callback, 1000, {name:"wang"});\*/  
 setTimeout: function (*fn*, *time*) {  
 var args = Array.prototype.slice.call(arguments, 2);  
 setTimeout(function () {  
 *fn*(args[0]);  
 setTimeout(arguments.callee, *time*);  
 }, *time*);  
 },  
 // 分时处理函数，防止执行时间过长，导致明显阻塞--Yielding Processes  
 chunk: function (*array*, *process*, *context*) {  
 setTimeout(function () {  
 var item = *array*.shift();  
 *process*.call(*context*, item);  
  
 if(*array*.length > 0) {  
 setTimeout(arguments.callee, 100);  
 }  
 }, 100);  
 }  
};  
// 函数绑定  
var *handler* = {  
 message: "Event handled",  
 handleClick: function (*event*) {  
 console.log(this.message);  
 }  
};  
// 保证作用域  
*EventUtil*.addHandler(window, "load", *handler*.handleClick.bind(*handler*) /\*EventUtil.bind(handler.handleClick, handler)\*/);  
*EventUtil*.addHandler(window, "load", *handler*.handleClick.bind(*handler*, "my-btn")/\*EventUtil.bind(handler.handleClick, handler, "my-btn")\*/);  
  
var *WebUtil* = {  
 xmlRequest: function (*url*) {  
 var xhr = new XMLHttpRequest();  
 // progress 事件  
 xhr.onload = function (*event*) {  
 if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){  
 console.log(xhr.responseText);  
 } else {  
 console.log("Request was unsuccessful： " + xhr.status);  
 }  
 };  
  
 xhr.onprogress = function (*evnet*) {  
 var divStatus = document.getElementById("status");  
 if (event.lengthComputable) {  
 divStatus.innerHTML = "Received " + event.position + " of " + event.totalSize + " bytes";  
  
 }  
  
 };  
 xhr.onreadystatechange = function () {  
 if(xhr.readyState == 4) {  
 if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){  
 console.log(xhr.responseText);  
 var myHeader = xhr.getResponseHeader("MyHeader");  
 var allHeaders = xhr.getAllResponseHeaders();  
 } else {  
 console.log("Request was unsuccessful: " + xhr.status);  
 }  
 }  
 };  
 xhr.open("get", "../html/drag.html", false);  
 xhr.setRequestHeader("MyHeader", "value");  
 xhr.send(null);  
  
  
 },  
  
 createStreamingClient: function (url, progress, finished){  
 var xhr = new XMLHttpRequest(),  
 received = 0;  
 xhr.open("get", url, true);  
 xhr.onreadystatechange = function () {  
 var result;  
  
 if(xhr.readyState == 3){  
 // 取得最新数据并调整计数器  
 result = xhr.responseText.substring(received);  
 received += result.length;  
 // 调用progress回调函数  
 progress(result);  
 } else if (xhr.readyState == 4) {  
 finished(xhr.responseText);  
 }  
 };  
 xhr.send(null);  
 return xhr;  
 },  
 createCORSRequest: function (method, url) {  
 var xhr = new XMLHttpRequest();  
 if ("withCredentials" in xhr){  
 xhr.open(method, url, true);  
 } else if (typeof XDomainRequest != "undefined") {  
 xhr = new XDomainRequest();  
 xhr.open(method, url);  
 } else {  
 xhr = null;  
 }  
 return xhr;  
 },  
 imgPing: function (url) {  
 var img = new Image();  
 img.onload = img.onerror = function () {  
 console.log("done!");  
 };  
 img.src = url;  
 },  
  
 jsonp: function (url, callback) {  
 var script = document.createElement("script");  
 script.src = url + (url.indexOf("?") == -1 ? "?":"&") + "callback=" + callback ;  
 document.body.insertBefore(script, document.body.firstChild);  
 },  
  
 sse: function (url) {  
 var soucrce = new EventSource(url);  
 source.onmessage = function (event) {  
 var data = event.data;  
 // 处理数据  
 };  
 },  
 websocket: function (url) {  
 var socket = new WebSocket("ws://www.example.com/focus.php");  
 socket.send("hello world!");  
 socket.onmessage = function (event) {  
 var data = event.data;  
 // other operate  
 };  
 socket.close();  
 }  
};  
  
var AdvancedTech = {  
 // 安全类型检测, 对于IE中以COM对象形式实现的任何函数，isFunction都会返回false  
 // 只对原生的toString()方法有效  
 isArray: function (value) {  
 return Object.prototype.toString.call(value) == "[object Array]";  
 },  
 isFunction: function (value) {  
 return Object.prototype.toString.call(value) == "[object Function]";  
 },  
 isRegExp: function (value) {  
 return Object.prototype.toString.call(value) == "[object RegExp]";  
 },  
 // 函数节流 -- 处理可能频繁发生的DOM事件 resize scroll  
 throttle: function (method, context) {  
 clearTimeout(method.tId);  
 method.tId = setTimeout(function () {  
 method.call(context);  
 }, 100);  
 }  
};  
  
// 观察者模式  
function EventTarget() {  
 this.handlers = {};  
}  
EventTarget.prototype = {  
 constructor: EventTarget,  
 addHandler: function (type, handler) {  
 if (typeof this.handlers[type] == "undefined"){  
 this.handlers[type] = [];  
 }  
 this.handlers[type].push(handler);  
 },  
  
 trigger: function (event) {  
 if (!event.target){  
 event.target = this;  
 }  
 if (this.handlers[event.type] instanceof Array) {  
 var handlers = this.handlers[event.type];  
 for (var i=0, len=handlers.length; i<len; i++) {  
 handlers[i](event);  
 }  
 }  
 },  
  
 removeHandler: function (type, handler) {  
 if (this.handlers[type] instanceof Array){  
 var handlers = this.handlers[type];  
 for (var i=0, len=handlers.length; i<len; i++) {  
 if (handlers[i] == handler){  
 break;  
 }  
 }  
 handler.splice(i,1);  
 }  
 }  
};  
  
// 作用域安全的构造函数  
function Person(name, age, job){  
 // 防止实例化时，忘记使用new  
 if (this instanceof Person){  
 this.name = name;  
 this.age = age;  
 this.job = job;  
 } else {  
 return new Person(name, age, job);  
 }  
}  
// 构造函数窃取结合使用原型链或者寄生组合方式  
function Student(name, age, job, cl) {  
 Person.call(this,name, age, job);  
 this.cl = cl;  
}  
Student.prototype = new Person();  
  
  
// 惰性载入函数: if判断只执行一次  
var createXHR = (function(){  
 if (typeof XMLHttpRequest != "undefined"){  
 return function () {  
 return new XMLHttpRequest();  
 };  
 } else if (typeof ActiveXObject != "undefined"){  
 return function (){  
 new ActiveXObject('MSXML2.XMLHttp.3.0');  
 }  
 } else {  
 return function () {  
 throw new Error("No XHR object available.");  
 }  
 }  
})();  
  
var CookieUtil ={  
 get: function (name) {  
 var cookieName = encodeURIComponent(name) + "=",  
 cookieStart = document.cookie.indexOf(cookieName),  
 cookieValue = null;  
 if (cookieStart > -1) {  
 var cookieEnd = document.cookie.indexOf(";", cookieStart);  
 if (cookieEnd == -1) {  
 cookieEnd = document.cookie.length;  
 }  
 cookieValue = decodeURIComponent(document.cookie.substring(cookieStart+cookieName.length, cookieEnd));  
 }  
 return cookieValue;  
 },  
   
 getUseReg: function (name) {  
 var arr,reg=new RegExp("(^| )"+name+"=([^;]\*)(;|$)");  
 if(arr=document.cookie.match(reg))  
 return decodeURIComponent(arr[2]);  
 else  
 return null;  
 },  
  
 set: function (name, value, expires, path, domain,secure) {  
 var cookieText = encodeURIComponent(name) + "=" +  
 encodeURIComponent(value);  
 if (expires instanceof Date){  
 cookieText += "; expires=" +expires.toGMTString();  
 }  
 if (path) {  
 cookieText += "; path=" + path;  
 }  
 if (domain) {  
 cookieText += "; domain=" + domain;  
 }  
 if (secure) {  
 cookieText += "; secure";  
 }  
  
 document.cookie = cookieText;  
 },  
 unset: function (name, path, domain, secure) {  
 this.set(name, "", new Date(0), path, domain, secure);  
 },  
 getSubCookie: function (name, subName) {  
 var subCookies = this.getAllCookie(name);  
 if (subCookies) {  
 return subCookies[subName];  
 } else {  
 return null;  
 }  
 },  
 getAllCookie: function (name) {  
 var cookieName = encodeURIComponent(name) + "=",  
 cookieStart = document.cookie.indexOf(cookieName),  
 cookieValue = null,  
 cookieEnd,  
 subCookies,  
 i,  
 parts,  
 result = {};  
 if (cookieStart > -1) {  
 cookieEnd = document.cookie.indexOf(";", cookieStart);  
 if (cookieEnd == -1){  
 cookieEnd = document.cookie.length;  
 }  
 cookieValue = document.cookie.substring(cookieStart+cookieName.length, cookieEnd);  
 if (cookieValue.length > 0) {  
 subCookies = cookieValue.split("&");  
 for (i=0, len=subCookies.length; i<len; i++) {  
 parts = subCookies[i].split("=");  
 result[decodeURIComponent(parts[0])] = decodeURIComponent(parts[i]);  
 }  
 return result;  
 }  
 }  
 return null;  
 },  
  
 setSubCookie: function (name, subName, value, expires, path, domain, secure) {  
 var subcookies = this.getAllCookie(name) || {};  
 subcookies[subName] = value;  
 this.setAllCookies(name, subcookies, expires, path, domain, secure);  
 },  
  
 setAllCookies: function (name, subcookies, expires, path, domain, secure) {  
 var cookieText = encodeURIComponent(name) + "=",  
 subcookieParts = new Array(),  
 subName;  
  
 for (subName in subcookies) {  
 if (subName.length > 0 && subcookies.hasOwnProperty(subName)) {  
 subcookieParts.push(encodeURIComponent(subName) + "=" + encodeURIComponent(subcookies[subName]));  
 }  
 }  
  
 if (subcookieParts.length > 0) {  
 cookieText += subcookieParts.join("&");  
 if (expires instanceof Date){  
 cookieText += "; expires=" +expires.toGMTString();  
 }  
 if (path) {  
 cookieText += "; path=" + path;  
 }  
 if (domain) {  
 cookieText += "; domain=" + domain;  
 }  
 if (secure) {  
 cookieText += "; secure";  
 }  
 } else {  
 cookieText += "; expires=" + (new Date(0)).toGMTString();  
 }  
  
 document.cookie = cookieText;  
 },  
 unsetSubCookie: function (name, subName, path, domain, secure) {  
 var subCookies = this.getAllCookie(name);  
 if(subCookies){  
 delete subCookies[subName];  
 this.setAllCookies(name, subCookies, null, path, domain, secure);  
 }  
 },  
  
 unsetAll: function (*name*, *path*, *domain*, *secure*) {  
 this.setAllCookies(*name*, null, new Date(0), *path*, *domain*, *secure*);  
 }  
};  
  
var *WebStorage* = {  
 getLocalStorage: function () {  
 if (typeof localStorage == "object") {  
 return localStorage;  
 } else if (typeof globalStorage == "object") {  
 return globalStorage;  
 } else {  
 throw new Error("Local storage not available");  
 }  
  
 }  
};

# 附录B：各种坑

1. 摘自：http://www.cnblogs.com/xxcanghai/p/5189353.html

function *Foo*() {  
 *getName* = function () { alert (1); };  
 return this;  
}  
*Foo*.getName = function () { alert (2);};  
*Foo*.prototype.getName = function () { alert (3);};  
var *getName* = function () { alert (4);};  
function *getName*() { alert (5);}  
  
//请写出以下输出结果：  
//答案：  
*Foo*.getName();//2 , 类的静态方法调用  
*getName*();//4, 变量声明提升  
*Foo*().getName();//1, 不使用new操作符，this指针指向window  
*getName*();//1, 全局变量定义覆盖  
new *Foo*.getName();//2， "."运算符优先级比new高， new (Foo.getName)(): 实际上将getName函数作为了构造函数来执行  
new *Foo*().getName();//3 (new Foo()).getName()  
new new *Foo*().getName();//3 (new (new Foo()).getName) ()

# 附录C：各种技巧

## 1. 使用外部变量的超时调用

for (var *i*=0; *i*<10; *i*++) {  
 setTimeout( function(*a*) {  
 return function () {  
 console.log(*a*);  
 }  
 }(*i*), 200);  
}

## 2. underscore中大量使用了惰性载入函数

// Generator function to create the findIndex and findLastIndex functions.  
var *createPredicateIndexFinder* = function(*dir*) {  
 return function(*array*, *predicate*, *context*) {  
 *predicate* = *cb*(*predicate*, *context*);  
 var length = getLength(*array*);  
 var index = *dir* > 0 ? 0 : length - 1;  
 for (; index >= 0 && index < length; index += *dir*) {  
 if (*predicate*(*array*[index], index, *array*)) return index;  
 }  
 return -1;  
 };  
};

## 3. call 和 apply的区别

call调用的时候传递的是单个参数，apply传递的是参数数组

foo.call(this, arg1, arg2,arg3) == foo.apply(this, arguments) == this.foo(arg1, arg2, arg3);

## 4. underscore 中的节流函数, 在指定时间内只调用一次

*\_*.throttle = function(*func*, *wait*, *options*) {  
 var timeout, context, args, result;  
 var previous = 0;  
 if (!*options*) *options* = {};  
  
 var *later* = function() {  
 previous = *options*.leading === false ? 0 : *\_*.now();  
 timeout = null;  
 result = *func*.apply(context, args);  
 if (!timeout) context = args = null;  
 };  
  
 var *throttled* = function() {  
 var now = *\_*.now();  
 if (!previous && *options*.leading === false) previous = now;  
 var remaining = *wait* - (now - previous);  
 context = this;  
 args = arguments;  
 if (remaining <= 0 || remaining > *wait*) {  
 if (timeout) {  
 clearTimeout(timeout);  
 timeout = null;  
 }  
 previous = now;  
 result = *func*.apply(context, args);  
 if (!timeout) context = args = null;  
 } else if (!timeout && *options*.trailing !== false) {  
 timeout = setTimeout(*later*, remaining);  
 }  
 return result;  
 };  
  
 *throttled*.cancel = function() {  
 clearTimeout(timeout);  
 previous = 0;  
 timeout = context = args = null;  
 };  
 return *throttled*;  
};

5. underscore 持续调用函数，只有最后一次被调用

*\_*.debounce = function(*func*, *wait*, *immediate*) {  
 var timeout, result;  
 var *later* = function(*context*, *args*) {  
 timeout = null;  
 if (*args*) result = *func*.apply(*context*, *args*);  
 };  
 var debounced = *restArgs*(function(*args*) {  
 if (timeout) clearTimeout(timeout);  
 if (*immediate*) {  
 var callNow = !timeout;  
 timeout = setTimeout(*later*, *wait*);  
 if (callNow) result = *func*.apply(this, *args*);  
 } else {  
 timeout = *\_*.delay(*later*, *wait*, this, *args*);  
 }  
  
 return result;  
 });  
 debounced.cancel = function() {  
 clearTimeout(timeout);  
 timeout = null;  
 };  
 return debounced;  
};

# 附录D:发现

1. 对function调用length，返回的值是arguments.length

function *f* (*func*, *boundArgs*) {} // f.length = 2