JavaScript 权威指南

# 第一章 类型、值和变量

**function** write(str){  
 document.write(str + '<br>');  
}

## 1.1 数字

/\*判断是否为NaN\*/

**var** a = NaN;  
write(a!=a); //true 方法一：NaN本身并不等于NaN  
write(isNaN(a)); //true 方法二：使用全局函数  
  
/\*负0值\*/  
write(0 === (-0)); //true 负0值和正0值严格相等  
  
/\*二进制浮点数\*/  
**var** x = .2 - .1;  
**var** y = .3 - .2;  
write(x);//0.1  
write(y);//0.09999999999999998

## 1.2 文本

/\*字符串\*/  
**var** s = 'hello';  
**var** newS = s.substring(2);  
write(s); //hello  
write(newS); //llo

//通常字符串和数字等本身是不能改变的，所以方法返回的肯定是新的字符串，原来的字符串不变

## 1.3 布尔值

## 1.4 null和undefined

/\*null和undefined\*/  
//null表示空对象  
//undefined表示未初始化的值  
**var** o = {};  
write(o==**null**); //false  
write(**typeof null**); //object  
write(**null** == undefined); //true  
write(**null** === undefined); //false  
//null和undefined都不包含任何属性和方法  
//undefined表示系统型的，出乎意料的或类似错误的值的空缺  
//null表示程序级的，正常的或在意料之内的值的空缺

## 1.5 全局对象

## 1.6 包装对象

/\*包装对象\*/  
**var** s = 'hello world!';  
**var** world = s.substring(s.indexOf(' ')+1);  
write(world); //world!  
  
//字符串是基本数值类型，怎么会有方法呢？  
//执行过程：  
// 1.调用new String(s)的方式转换成对象  
// 2.由于这个对象继承了字符串的方法，用来处理属性的引用，即引用字符串的方法  
// 3.引用结束，销毁新创建的对象  
// 数字和布尔值也具有各自的方法，通过Number()和Boolean()构造函数创建临时对象，方法的调用均来自临时对象  
// null和undefined没有包装对象，访问属性会造成类型错误  
// 以上过程类似于  
**var** a = **new** String(s);  
write(a); //hello world!  
**var** world = a.substring(a.indexOf(' '));  
write(world); //world!  
write(a); //hello world!  
a = **null**;  
write(a); //null

**var** str = 'hello world!';  
str.len = 4;  
**var** t = str.len; //创建了一个字符串的临时变量。并赋值一个属性，随后对象立即销毁  
write(t);//undefined  
//在读字符串、数字和布尔值的属性时，表现的像对象一样，但是如果试图给属性赋值，则会忽略这个操作  
//修改只是发生在临时对象上，而这个临时对象并未继续保留下来

//存取字符串、数字或布尔值的属性时临时创建的对象叫做包装对象

//基本数值类型和和显式包装对象的区别  
**var** s = 'hello world';  
**var** S = **new** String(s);  
write(S == s);//true 这里只是在必要的时侯将包装对象转换成原始值，所以才会相等  
write(S === s);//false  
write(**typeof** s); //string  
write(**typeof** S); //object

## 1.7 不可变原始值和可变的对象的引用

/\*不可变原始值和可变的对象引用\*/  
//不可变值包括undefined、null、布尔值、数字和字符串  
//可变值是对象

//不可变值得任何方法都是返回新的不可变值，而不会改变原来的不可变值  
**var** s = 'hello';  
**var** S = s.toUpperCase();  
write(s);//hello,原来的值不变  
write(S);//HELLO

//可变值  
**var** o = {x:1};  
o.x = 2; //值变化  
o.y = 1; //添加一个新的属性  
  
**for**(**var** key **in** o){  
 write(key);  
 write(o[key]);  
}  
/\*  
x  
2  
y  
1  
\*/

/\*不可变原始值和可变的对象引用\*/  
//不可变值包括undefined、null、布尔值、数字和字符串  
//可变值是对象  
  
//不可变值的比较是值的比较，字符串比较特殊  
//需要比对长度、以及每个索引的字符都相等  
  
//对象的比较并非值得比较  
**var** x = {a:1};  
**var** y = {b:2};  
write(x==y);//false 因为是不同的副本  
  
**var** X = {a:1};  
**var** Y = X; //并没有为Y创建一个副本，而是使Y和X指向了同一个副本  
write(X == Y);//true 同一个副本的引用，浅拷贝  
write(X === Y);//true  
  
**var** a = [1,2];  
**var** b = [1,2];  
write(a == b);//false  
  
//深拷贝，创建一个新的副本,拷贝每一个属性  
**var** a = {x:1,y:2};  
**var** b = {};  
  
**for**(**var** key **in** a){  
 b[key] = a[key];  
 write(b[key]);// 1 2  
}

//数组的比较  
**function** isEqualArray(a,b){  
 **function** isArray(arr){  
 **if**(Object.prototype.toString.call(arr) !== '[object Array]'){  
 **return true**;  
 }  
 }  
 **for**(**var** i= 0,len=arguments.length; i<len; i++){  
 **if**(isArray(arguments[i])){  
 **return false**;  
 }  
 }  
 **if**(a.length != b.length){  
 **return false**;  
 }  
 **for**(**var** i= 0,len= a.length; i<len; i++){  
 **if**(a[i] != b[i]){  
 **return false**;  
 }  
 }  
 **return true**;  
}  
  
write(isEqualArray([1,2,3],{}));//false  
write(isEqualArray([1,2,3],[1]));//false  
write(isEqualArray([1,2,3],[1,2,3]));//true

## 1.8 类型转换

/\*显式类型转换\*/  
//使用Boolean() Number() String() Object()  
//不通过new运算符调用函数时就是作为类型转换函数  
**var** num = Number('3');//字符串'3'转换成数字3  
write(**typeof** num);//number  
  
**var** num1 = **new** Number('3');  
write(**typeof** num1);//object  
  
/\*隐式类型转换\*/  
**var** num = 1;  
**var** str = num + "";//等价于String(x)  
write(**typeof** str);//string   
  
**var** num1 = +str;//,+str的+是正号的意思，等价于Number(x),也可以写成 str – 0

//var num1 = -str 则表明取相反数，改变符号位  
write(num1);//1  
write(**typeof** num1);//number  
  
**var** b = !!num;//等价于Boolean(x)  
write(**typeof** b);//boolean  
write(b);//true

/\*对象转换为原始值\*/  
//toString()  
**var** o = {x:1,y:2};  
write(o.toString());//[object Object]  
**var** arr = [1,2,3,4];  
write(arr.toString());//1,2,3,4  
write((**function** A(){}).toString());//function A(){}

write(**new** Date().toString());

//Tue Apr 19 2016 08:49:15 GMT+0800 (中国标准时间)

//valueOf() 默认将对象转换为表示它的原始值  
write(arr.valueOf());//1,2,3,4  
write(o.valueOf());//[object Object]  
write((**function** A(){}).valueOf());//function A(){}  
write(**new** Date().valueOf());//1460968086219 返回1970年1月1日以来的毫秒数

//对象到字符串的转换过程  
//1.如果对象具有toString()方法，则调用这个方法  
//2.并没有toString()方法或者方法并不返回一个原始值，那么就会调用valueOf()方法  
//3.如果存在这个方法，则调用它，返回值是原始值那么转换为字符串，并返回这个字符串结果  
//4.否则无法从toString()和valueOf()获得一个原始值，因此这时它将抛出异常

//对象到数字的转换过程  
//1.如果对象具有valueOf()方法，返回一个原始值，将这个原始值转换为数字并返回数字  
//3.否则如果对象具有toString()方法，返回一个原始值，转换成数字并返回  
//4.否则无法从toString()和valueOf()获得一个原始值，因此这时它将抛出异常

## 1.9 变量声明和提升

/\*变量声明和提升\*/

**function** test(o){  
 **var** i = 0;  
 write(j); //undefined  
 **if**(**typeof** o == 'object'){  
 **var** j = 0;  
 write(k); //undefined  
 **for**(**var** k=0; k<10; k++){ //c语言中这里就是块级作用域  
 write(k); //0,1,2,3,4,5,6,7,8,9  
 }  
 write(k); //10  
 }  
 write(j); //0  
}  
test({});

//i,j,k在整个test函数内都是有定义的

//如果养成了声明并初始化的习惯，那么就可以通过typedef来检测变量有没有初始化

**var** a;  
write(**typeof** a);//undefined  
write(**typeof** b);//undefined

//但是需要注意的是未声明的值也是undefined

//所以只有在养成声明并初始化的习惯时，能判断的是这个变量有三种情况

1. 声明并初始化 不为undefined
2. 未声明，为undefined
3. 声明但是未初始化，为undefined

//JS的函数作用域是指在函数内声明的所有变量在函数体内始终是可见的。

//变量在声明前已经可用,JS的这个特性被非正式的称为声明提升

//即JS中函数里声明的所有变量（不涉及赋值）都被提升至函数体的顶部

**var** global = 'window';  
**function** test(){  
 write(global); //window 此时是全局变量  
}  
test();

**var** global = 'window';  
**function** test(){  
 write(global); //undefined 此时是函数内的局部变量  
 **var** global;  
}  
test();

类似于

**var** global = 'window';  
**function** test(){  
 **var** global;  
 write(global); //undefined 此时是函数内的局部变量  
}  
test();

**var** global = 'window';  
**function** test(){  
 write(global); //undefined 此时是函数内的局部变量  
 **var** global = 'global';  
}  
test();

类似于

**var** global = 'window';  
**function** test(){  
 **var** global;  
 write(global); //undefined 此时是函数内的局部变量  
 global = 'global';  
}  
test();

**var** global = 'window';  
**function** test(){  
 write(global); //undefined 此时是函数内的局部变量  
 **var** global = 'global';  
 write(global); //global  
}  
test();

变量和函数声明的提升

**var** a = 1;  
**function** b(){  
 a = 10;  
 **return**;  
 **function** a(){}  
}  
b();  
alert(a); //1 注意不是10 实际上等同于

**var** a = 1;  
**function** b(){  
 **function** a(){}  
 a = 10; //此处将函数名a指向10，而var a = 1不会受影响  
 **return**;  
}

b();  
alert(a); //1 注意不是10  
**函数的声明比变量的声明具有高的优先级**。  
javascript，变量有4种基本方式进入作用域：

1 语言内置：所有的作用域里都有this和arguments；(译者注：经过测试arguments在全局作用域是不可见的)

2 形式参数：函数的形式参数会作为函数体作用域的一部分；

3 函数声明：像这种形式：function foo(){}；

4 变量声明：像这样：var foo；

函数声明和变量声明总是会被解释器悄悄地被“提升”到方法体的最顶部。

变量赋值并没有被提升，只是声明被提升了。但是，函数的声明有点不一样，函数体也会一同被提升。

最需要牢记在心的是变量解析顺序。前面给出的命名进入作用域的4种方式，变量解析的顺序就是列出来的顺序。

函数的声明比变量的声明具有高的优先级。以下两个程序说明了这种情况  
**function** f(){  
 **var** a;  
 **function** a(){}  
 write(a); //function a(){}  
}  
f();  
  
**function** fn(){  
 **function** a(){}  
 **var** a;  
 write(a); //function a(){}  
}  
fn();

但是要区别于以下两种情况：

**function** f(){  
 **var** a = 1;  
 **function** a(){}  
 write(a); //1  
}  
f();  
  
**function** fn(){  
 **function** a(){}  
 **var** a = 1;  
 write(a); //1  
}  
fn();

/\*命名函数\*/  
//foo(); //Uncaught TypeError: foo is not a function  
bar(); //有效  
//baz(); //Uncaught TypeError: baz is not a function  
//spam(); //Uncaught ReferenceError: spam is not defined  
  
**var** foo = **function**(){}; // foo指向匿名函数  
**function** bar(){}; // 函数声明  
**var** baz = **function** spam() {}; // 命名函数，只有baz被提升，spam不会被提升。  
  
foo(); //有效  
bar(); //有效  
baz(); //有效  
//spam(); //Uncaught ReferenceError: spam is not defined  
  
//可以给一个函数一个名字。如果这样的话，它就不是一个函数声明，  
//同时，函数体定义里面的指定的函数名( 如果有的话,如下面的spam, 译者注)将不会被提升,　而是被忽略。  
//上面的spam被忽略,baz和foo一样不会被提升

1、全局作用范围内，var与不var是有区别的，没有var的变量是不会被提升的

alert(a); //Uncaught ReferenceError: a is not defined  
a = 1; //虽然是全局的，但是没有var的写法，变量是不会被提升的

2.eval中创建的局部变量是不会被提升的。

**var** a = 2;  
**function** f(){  
 write(a); //2 eval中的a没有被提升  
 eval('var a = 1;');  
 write(a); //1  
}  
f();  
write(a); //2

## 1.10 变量的作用域

/\*作为属性的变量\*/  
**var** a = 3; //使用var声明一个变量是不可配置的，无法delete删除  
**delete** a;  
write(**typeof** a); //number  
  
b = 4; //未使用var，虽然也是全局变量，但是属性是可以配置的  
**delete** b;  
write(**typeof** b); //undefined

/\*作用域链\*/  
//顶层代码：（全局执行环境）  
//1.全局对象  
  
//不包含嵌套的函数体内：（局部执行环境）  
//1.定义函数参数和局部变量的对象  
//2.全局对象  
  
//在一个嵌套的函数体内 （局部执行环境）  
//1.嵌套的函数体内的参数和局部变量的对象  
//2.嵌套函数所在的外部函数的参数和局部变量的对象  
//3.全局对象  
  
//作用域链搜索规则  
//1.搜索当前函数体内（当前环境所在对象）的属性和方法  
//2.未找到则向上搜索，包含该对象的对象所在环境的属性和方法  
//3.最后是全局对象的属性和方法  
//4.如果未找到，则抛出一个引用错误异常

# 第二章 表达式和运算符

## 2.1 属性访问表达式

/\*属性访问表达式\*/  
//1.express.id  
//2.express[express]  
  
//1.使用第一种.表达式访问属性，则属性的名称必须是合法的标识符，并且要知道属性的确切名字  
//2.使用第二种[]表达式访问属性，则属性的名称可以是保留字或者包含空格和标点符号或者数字（对于数组而言）  
//并且如果属性名是通过运算得出的值而不是固定的值时，必须使用[]写法。

## 2.2 对象创建表达式

/\*对象创建表达式\*/  
**var** o = **new** Object();  
write(**typeof** o);//object  
**var** obj = **new** Object; //如果构造函数没有参数，则可以省略()  
write(**typeof** obj);//object

## 2.3 运算符概述

/\*运算符优先级\*/  
//属性访问表达式和调用表达式的优先级要比所有运算符都要高  
**var** str = 'this is a str';  
write(str.split(' ')[2]); //a  
//可以知道先是属性访问表达式，然后是调用表达式，然后又是属性访问表达式  
write(**typeof** str.split(' ')[2]); //string  
//可以知道先是属性访问表达式，然后是调用表达式，然后又是属性访问表达式  
//最后是typeof运算符/\*运算顺序\*/  
**var** a = 1;  
**var** b = a++ +a;  
write(b); //3  
//1.计算b  
//2.计算a++   
//3.计算a   
//4.计算 a++ +a

## 2.4 算术表达式

/\*算术表达式\*/  
//在JS中，所有数字都是浮点型的  
**var** a = 5/3;  
write(a);//1.6666666666666667

/\*+ 运算符\*/  
//1.如果操作数是对象，对象遵循对象到原始值的转换规则转换为原始值  
//日期通过toString()方法执行转换,其他对象通过valueOf()的方法转换  
//由于valueOf()方法多数对象都不具备，因此他们会通过toString()方法来执行转换  
//2.如果转换为原始值后，其中一个操作数是字符串，那么另一个操作数也转换成字符串，进行字符串连接  
//3.否则，两个操作数转换为数字，进行加法操作  
  
write(1+2);//3  
write('1'+'2');//12  
  
write(1+{});//1[object Object]  
//1.对象调用toString()方法  
write({}.toString());//[object Object]  
//2.调用toString()方法后变成了字符串，所以字符串进行字符串连接  
//3.1[object Object]  
write(**true**+**true**);//2  
  
write(2+**null**);//2 null转换成数字0  
write(2+undefined);//NaN undefined转换为NaN的做法  
  
//一下依赖于运算符的优先级  
write(1+1+'hell');//2hell 从左到右的结合性  
write(1+(1+'hell'));//11hell

/\*位运算符\*/  
write(0x1234 & 0x00FF);//0x34  
write(0x1234 | 0x00FF);//0x12FF

## 2.5 关系表达式

/\*关系表达式\*/  
**var** arr=[1,2,3,4,5];  
**var** arr1=[1,2,3,4,5];  
write(arr == arr1);//false 需要用前面的数组相等检测方法  
  
//一般对象转换为原始值，都是优先调用valueOf()方法，在尝试toString()方法  
//除了日期类，日期类只使用toString()方法

/\*关系表达式\*/  
//==和===  
//1.===  
//1.两个值类型不同，则不想等  
//2.如果两个值是null或者undefined，则不相等  
//3.如果两个值都是布尔值true或者false，则相等  
//4.值是数字，且相等，则相等，0和-0也同样相等  
//5.两个值是字符串，则对应位上的16位数完全相等则相等。

//==和===  
//2.==  
//1.如果操作数类型相同，则和严格相等的比较规则一样。  
//2.否则，如果操作数类型不同，

1）如果一个是null，一个是undefined，则相等

2）如果一个是数字，一个是字符串，字符串转换成数字，然后使用转换后的值进行比较

3) 如果其中一个值是true，则转换成1再进行比较，如果其中一个值是false，则转换成0在进行比较

4) 如果值是对象，另一个数字或字符串，则将对象转换为原始值，在比较。

//比较运算符  
//1.如果操作数是对象，对象转换为原始值，进行比较  
//2.转换为原始值以后，如果两个操作数都是字符串，那么将依照字母表顺序进行比较  
//3.如果至少有一个不是字符串，那么两个操作数都转换成数字进行数值比较，如果一个操作数是NaN，那么比较操作符总是返回false

**if**(2 > '13'){ // '13'转换成13  
 write(1);  
} **else**{  
 write(2); //输出 2 按数字比较  
}

//加号运算符和比较运算符的行为有所不同，前者更偏向于字符串，其中一个是字符串则进行字符串连接操作  
//后者更偏向于数字，如果两个操作数都是字符串才会进行字符串的比较

## 2.6 逻辑表达式

**var** a = 1;  
**var** b = 1;

**if**(a==b)alert(1);  
//等同于  
(a==b) && alert(1)  
//只有&&前面的条件满足了才会去执行后面的条件

/\*||\*/  
**function** copy(o,p){  
 p = p || {}; //如果没有传入任何对象，则使用一个新创建的对象  
 write(p);  
}  
  
copy({});//[object Object]  
copy({},[1,2,3]);//1,2,3

## 2.7 表达式计算

/\*eval\*/  
//1.直接调用eval,总是在调用它的上下文作用域内执行  
//2.其他的间接调用则使用全局对象作为其上下文作用域  
  
**var** geval = eval; //使用别名调用eval将是全局eval，这算是间接调用  
**var** x = 'x global';  
**var** y = 'y global';  
**function** f(){  
 **var** x = 'x local';  
 eval("x += ' changed'"); //直接eval改变了局部变量的值  
 **return** x;  
}  
**function** g(){  
 **var** y = 'y local';  
 geval("y += ' changed'"); //间接调用改变了全局变量的值  
 **return** y;  
}  
console.log(f(),x);//x local changed x global  
console.log(g(),y);//y local y global changed

//所以更可能会使用全局eval而不是局部eval

## 2.8 其他运算符

/\*delete\*/  
**var** o = {x:1,y:2};  
**var** a = [1,2,3];  
**delete** a[1];  
write(a.length); //3 长度仍然没有变化  
write(a[1]);//undefined  
**function** A(){}  
write(**delete** A);//false function语句定义的函数和函数参数不能删除  
write(A);//function A(){}  
**function** B(a){  
 write(**delete** a);//false 函数参数不能删除  
 write(a);//2  
}  
B(2);

# 第三章 语句

## 3.1 条件语句

**function** f(n){  
 **switch**(n){  
 **case** '1': write(1);  
 **case** '2': write(2);  
 **case** '3': write(3);  
 **default** : write(4);**break**;  
 }  
}  
  
f(1); //4  
f('1'); //1 2 3 4  
//case 语句按照'==='运算符进行比较

## 3.2 循环

/\*for-in 语句\*/  
**var** o = {x:1,y:2,z:3};  
**var** a = [], i = 0;  
**for**(a[i++] **in** o); /\*empty\*/  
write(a.toString()); //x,y,z 利用数组存储o对象的属性名

//for-in只遍历可枚举的属性，需要注意的是利用原型链继承的属性也是可以遍历的!

/\*return 语句\*/  
//1.没有返回值的函数被调用时，其实会返回undefined  
**function** A(){}  
write(A()); //undefined  
//2.只返回return的函数被调用时，返回的也是undefined  
**function** B(o){  
 **if**(!o){  
 **return**;  
 }  
}  
write(B({})); //undefined

## 3.3 其他语句类型

"ust strict";  
*/\*\*  
 \* 1.严格模式下禁止使用with语句  
 \* 2.严格模式下所有变量都要先声明（使用var），非严格模式是给全局对象新添加一个新属性  
 \* 3.在严格模式下调用的函数的this值是undefined,非严格模式下调用的是全局对象[object Window]  
 \* 4.在严格模式下给只读属性赋值和给不可扩展的对象创建新成员都将抛出一个类型的异常，非严格模式不会报错  
 \* 5.在严格模式中，传入eval()的代码不能再调用程序所在的上下文中声明变量或定义函数，而在非严格模式中可以这样做。  
 \* 6.在严格模式中，arguments对象拥有传入函数值的静态副本（不只是指向了传入的参数对象的引用，而是另外开辟了一个副本）  
 \* 而在非严格模式中arguments对象的数组元素和函数参数都是指向了同一个值的引用  
 \* 7.严格模式下delete运算符后跟随非法的标识符将抛出语法错误异常  
 \* 8.严格模式中，在一个对象直接量中定义两个或多个同名属性将产生语法错误  
 \* 9.在严格模式中，函数声明中存在两个或多个同名的参数将产生语法错误  
 \* 10.在严格模式中不允许使用八进制整数直接量。  
 \* 11.严格模式限制了对调用栈的检测能力，arguments.caller和arguments.callee都会抛出类型错误异常  
 \*/*

/\*第一条\*/

"use strict";  
  
**with** (location){ //Uncaught SyntaxError: Strict mode code may not include a with statement  
 alert(1);  
}

/\*第二条\*/

"use strict";  
  
a = 2;  
alert(a); //Uncaught ReferenceError: a is not defined

/\*第三条\*/

**function** fn(){  
 **return this**;  
}  
  
**function** f(){  
 "ust strict";  
 **return this**;  
}  
  
write(fn());//[object Window] 非严格模式  
write(f());//undefined

/\*第五条\*/

**function** fn(){  
 eval('var i = 0');  
 write(i); //0  
}  
  
**function** f(){  
 "use strict";  
  
 eval('var i = 0');  
 write(i); //Uncaught ReferenceError: i is not defined  
}  
  
fn();  
f();

# 第四章 对象

## 4.1 创建对象

/\*原型\*/  
*/\*\*  
 \* 1.没有原型的对象，Object.prototype  
 \* 2.通过new和构造函数调用创建对象的原型就是构造函数的prototype属性的值，例如通过 new Array()创建的对象的原型就是Array.prototype  
 \* 3.所有内置构造函数都具有一个继承自Object.prototype的原型，例如Date.prototype继承自Object.prototype  
 \*/*

/\*对象创建的方法\*/  
  
*/\*\*  
 \* 1.对象直接量 {}  
 \* 2.通过new创建对象  
 \* 3.Object.create()  
 \*/*//Object.create()  
//创建一个新对象，第一个参数是这个对象的原型，所以传入的是原型对象  
**var** o1 = Object.create({x:1,y:2});  
write(o1.x); //1  
  
//创建一个没有原型的对象  
**var** o2 = Object.create(**null**);  
  
//创建类似于{} 或者new Object()的空对象  
**var** o3 = Object.create(Object.prototype);  
  
  
//一下函数可以模拟原型继承  
**function** inherit(p){  
 //**if**(p==**null**) **return**;  
 **if**(Object.create){  
 **return** Object.create(p); //p是返回对象的原型对象  
 }  
  
 **var** t = **typeof** p;  
 **if**((t != "object") && (t != "function")) **return**;  
 **function** f(){} //构造函数  
 f.prototype = p; //将其原型对象属性设置为p  
 **return new** f(); //使用f()创建一个p的继承对象  
}  
  
//以上函数返回的新对象继承了参数对象的属性  
//需要注意的是与Object.create()函数不同的是，inherit()函数不能通过传入null原型来创建对象  
  
  
**var** o = {x:1, y:2};  
**var** o\_child = inherit(o);  
  
o\_child.x = 2; //我的猜测，这个是对象自身的属性，即为对象o\_child创建了一个属性x并赋值2  
  
write(o.x); //1 原型对象的属性  
write(o.y); //2 原型对象的属性  
write(o\_child.x); //2 //这个是o\_child自身的属性  
write(o\_child.y); //2 //这个不是o\_child自身的属性，而是继承来的属性  
  
o.y = 222;  
write(o\_child.y); //222 这个是o\_child继承来的属性，发现原型对象的属性值变了，那么继承来的属性值也变了  
  
o\_child.y = 4; //为o\_child对象创建了一个属性值y并赋值4  
write(o\_child.y); //4 这个是o\_child自身的属性  
  
o.y = 222222; //修改了原型对象的属性值  
write(o\_child.y); //4 根据原型链的搜索机制，先搜索对象的自身属性，如果未搜索到则搜索继承来的属性  
  
  
write(o.x); //1  
write(o.y); //222222  
//需要说明的是通过o\_child对象并不能来修改o的属性值，但是可以继承o的属性值

//一下函数可以模拟原型继承  
**function** inherit(p){  
 **if**(p==**null**) **return**;  
 **if**(Object.create){  
 **return** Object.create(p); //p是返回对象的原型对象  
 }  
  
 **var** t = **typeof** p;  
 **if**((t != "object") && (t != "function")) **return**;  
 **function** f(){} //构造函数  
 f.prototype = p; //将其原型对象属性设置为p  
 **return new** f(); //使用f()创建一个p的继承对象  
}  
  
  
//判断实例对象的属性是否是原型对象继承来的属性  
**function** hasPrototypeKey(obj,key){  
 **return** !obj.hasOwnProperty(key) && (key **in** obj);  
}  
  
//obj.hasOwnProperty(key) 如果有自身的属性，那么返回false,表明该对象并不是继承来的属性  
//key in obj不仅能够遍历自身的属性，还能够遍历原型对象的属性  
//所以如果自身属性不存在而 key in obj存在,那么说明是原型对象继承来的属性  
**var** o = {x:1, y:2};  
**var** o\_child = inherit(o);  
  
write(hasPrototypeKey(o\_child,'x')); //true 是原型对象的属性  
o\_child.x = 122;  
write(hasPrototypeKey(o\_child,'x')); //false 不是原型对象的属性

## 4.2 属性的查询和设置

/\*属性的查询和设置\*/  
//.运算符后不能是保留字和关键字  
//[]运算符内可以是保留字和关键字，也可以是表达式，但是表达式必须返回字符串或可以转换成一个字符串的值  
//当时数组的[]也可以是数字

/\*属性访问错误\*/  
  
*/\*\*  
 \* 1.如果对象不存在属性，那么返回undefined  
 \* 2.如果对象都不存在，访问一个不存在的对象的属性时抛出错误  
 \* 3.null和undefined都没有属性，会报错  
 \*/***var** o = {};  
write(o.z); //undefined  
write(a.k); //Uncaught ReferenceError: a is not defined  
  
//undefined没有属性  
write(o.z.length); //Uncaught TypeError: Cannot read property 'length' of undefined

//解决上面的抛出异常的方法  
//使用 && 短路行为

**var** o = {};  
**var** len = o && o.z && o.z.length; //o.z的undefined转换为false，后面的就不执行了  
o.z = [1,2];  
**var** len = o && o.z && o.z.length;  
write(len); //2  
//&& 短路行为  
*/\*\*  
 \* 1.计算左操作数，如果左操作数是假值，整个表达式的结果一定是假值，简单的返回左操作数的值，不计算右操作数的值  
 \* 2.如果左操作数是真值，那么&&运算符将计算右操作数的值并将其返回作为整个表达式的计算结果  
 \*/***var** obj = {x:333};  
write(obj && obj.x); //333 返回的是obj.x

//内置构造函数的原型是只读的  
Object.prototype = 0; //赋值并不会报错  
write(Object.prototype);//[object Object]  
  
//严格模式中会报错  
Object.prototype = 0; //赋值并不会报错  
write(Object.prototype);  
//Uncaught TypeError: Cannot assign to read only property 'prototype' of function Object() { [native code] }  
  
*/\*\*  
 \* 给对象o设置属性p会错误的情况  
 \* 1.o中p是只读的，不能给只读属性重新赋值  
 \* 2.o中的属性p是继承属性，且它是只读的，不能通过同名自有属性覆盖只读的继承属性  
 \* 3.o中不存在自有属性p，o没有使用setter方法继承属性p，并且o的可扩展性是false  
 \*/*

## 4.3 删除属性

/\*删除属性\*/  
//delete断开属性和宿主对象的联系，而不会去操作属性中的属性  
**var** a = {p:{x:1}};  
**var** b = a.p;  
**delete** a.p;  
write(b.x); //1

//删除的属性的引用依然存在，所以可能造成内存泄漏  
//解决办法是遍历属性中的属性，依次删除  
**delete** a.p.x;  
write(b.x); //undefined  
  
write(a.p); //[object Object]  
**delete** a.p;  
write(a.p); //undefined

//delete删除成功或没有任何副作用（比如删除不存在的属性）返回true  
write(**delete** 1); //true  
write(**delete** a); //true

//"use strict";

## 4.4 检测属性

/\*检测属性\*/  
  
//in 运算符 如果对象是自有属性或继承属性中包含这个属性则返回true

注意和for-in的区别  
**var** o = {x:1};  
write('x' **in** o); //true  
write('toString' **in** o); //true o继承的toString属性  
  
//hasOwnProperty 检测对象的自有属性，必须是可枚举的属性  
write(o.hasOwnProperty('x')); //true  
write(o.hasOwnProperty('toString')); //false   
//propertyIsEnumerable() 自身属性且可枚举性为true才返回true  
**var** o = inherit({y:2});  
o.x = 1;  
write(o.propertyIsEnumerable('x')); //true  
write(o.propertyIsEnumerable('y')); //false  
  
write(Object.prototype.propertyIsEnumerable('toString')); //false 不可枚举

## 4.5 枚举属性

//把p中可枚举属性复制到o中，并返回o  
//o和p中含有同名属性，则覆盖o中的属性  
**function** extend(o,p){  
 **for**(var prop **in** p){ //遍历p中的所有的可枚举属性  
 o[prop] = p[prop]; //将属性添加至o中  
 }  
 **return** o;  
}  
  
**var** o = {};  
**var** p = {x:1,y:2};  
extend(o,p);  
write(o.x); //1  
  
**var** o1 = {x:222};  
extend(o1,p);  
write(o1.x); //1 覆盖了o1的同名属性

//把p中可枚举属性复制到o中，并返回o  
//o和p中含有同名属性，则o中的属性不受影响  
**function** merge(o,p){  
 **for**(var prop **in** p){ //遍历p中的所有属性  
 **if**(o.hasOwnProperty(prop)) **continue**; //过滤o中已经存在的属性  
 o[prop] = p[prop]; //将属性添加至o中  
 }  
 **return** o;  
}  
  
**var** o = {};  
**var** p = {x:1,y:2};  
merge(o,p);  
write(o.x); //1  
  
**var** o1 = {x:222};  
merge(o1,p);  
write(o1.x); //222 保留原来的属性

//o和p中含有同名属性，删除o中的属性  
**function** substract(o,p){  
 **for**(**var** prop **in** p){ //遍历p中的所有属性  
 **delete** o[prop]; //删除一个不存在的属性不会报错  
 }  
 **return** o;  
}  
  
  
**var** p = {x:1,y:2};  
**var** o1 = {x:222};  
substract(o1,p);  
  
write(o1.x); //undefined 删除了o1中和p的同名属性x

//合并o和p中的属性，如果有同名属性，使用p中的属性  
**function** union(o,p){  
 **return** extend(extend({},o),p);  
}

//列举o中可枚举的自有属性  
**function** keys(o){  
 **if**(**typeof** o !== 'object') **return**;  
 **var** arr = [];  
 **for**(**var** prop **in** o){  
 **if**(o.hasOwnProperty(prop)){  
 arr.push(prop);  
 }  
 }  
 **return** arr;  
}  
  
  
**var** p = {x:1,y:2};  
  
**var** prop = keys(p);  
write(prop.toString()); //x,y

//Object.keys() 和keys()方法类似  
write(Object.keys(p)); //x,y

//Object.getOwnPropertyNames() 返回对象的所有属性名称，还包括不可枚举属性  
write(Object.getOwnPropertyNames(Object.prototype));  
//constructor,toString,  
//toLocaleString,valueOf,  
//hasOwnProperty,isPrototypeOf,  
//propertyIsEnumerable,\_\_defineGetter\_\_,  
//\_\_lookupGetter\_\_,\_\_defineSetter\_\_,  
//\_\_lookupSetter\_\_,\_\_proto\_\_

## 4.6 属性getter和setter

/\*属性getter和setter\*/  
**var** serialNum = {  
 $n:0,  
  
 //返回当前值然后自增  
 **get** next(){  
 **return this**.$n ++;  
 },  
  
 **set** next(n){  
 **if**(n >= **this**.$n) **this**.$n = n;  
 }  
};  
  
**var** q = inherit(serialNum);  
write(q.$n); //0  
write(q.next); //0  
write(q.next); //1  
write(q.next); //2

## 4.7 属性的特性

/\*属性的特性\*/  
*/\*\*  
 \* 数据属性的特性  
 \* 1.值 value  
 \* 2.可写性 writable  
 \* 3.可枚举性 enumerable  
 \* 4.可配置性 configurable  
 \*/  
  
/\*\*  
 \* 存储器属性特性  
 \* 1.读取 get  
 \* 2.写入 set  
 \* 3.可枚举性 enumerable  
 \* 4.可配置性 configurable  
 \*/*//Object.getOwnPropertyDescriptor() 获得属性描述符 参数数对象和对象的属性  
**var** o = {x:1,y:2};  
write(Object.getOwnPropertyDescriptor(o,'x')); //[object Object]  
  
**var** parm = Object.getOwnPropertyDescriptor(o,'x');  
//configurable: true  
//enumerable: true  
//value: 1  
//writable: true  
  
**var** ramdon = {  
 $n:0,  
 **get** next(){  
 **return this**.$n++;  
 }  
};  
  
parm = Object.getOwnPropertyDescriptor(ramdon,'next');  
//configurable: true  
//enumerable: true  
//get: next()  
//set: undefined  
  
//getOwnPropertyDescriptor只能获取自身的属性  
  
//Object.defineProperty() 参数 要修改的对象，要创建或修改的属性的名称，以及属性描述符对象  
  
**var** obj = {y:1};  
  
//添加一个不可枚举的数据属性x  
Object.defineProperty(obj,'x',{  
 value:1,  
 writable:**false**, //不可写  
 enumerable:**false**, //不可枚举  
 configurable:**true**});  
  
obj.x = 3;  
write(obj.x); //1 因为不可写，严格模式下回报错  
  
write('x' **in** obj); //true 尽管x是不可枚举的  
  
**var** arr = Object.keys(obj);  
write(arr.toString()); //y 没有x,因为x属性不可枚举  
  
//属性是可配置的，所以可以通过这种方式进行修改  
Object.defineProperty(obj,'x',{  
 value:1111  
});  
  
write(obj.x); //1111  
  
//将x从数据属性修改为存取器属性  
Object.defineProperty(obj,'x',{  
 **get**: **function**(){  
 **return** 0;  
 }  
});  
  
write(obj.x); //0

*/\*\*  
 \* 1.对象不可扩展，可以编辑已有的属性，不能添加新属性  
 \* 2.属性不可配置，不能修改它的可配置性和可枚举性  
 \* 3.存取器属性是不可配置的，不能修改getter和setter方法，也不能转为数据属性  
 \* 4.数据属性如果不可配置，则不能转换为存取器属性  
 \* 5.数据属性不可配置，则不能将它可写性从false修改为true，可以从true修改为false  
 \* 6.数据属性不可配置不可写，则不能修改它的值，但是可配置但不可写属性的值可以修改的  
 \* （修改的方法必须使用函数defineProperty，其实是先标记为可写，然后修改值，然后标记为不可写）  
 \*/*

## 4.8 对象的三个属性

/\*原型属性\*/  
*/\*\*  
 \* 1.通过对象字面量创建的对象，原型是Object.prototype  
 \* 2.通过new创建的对象使用构造函数的prototype属性作为它们的原型  
 \* 3.通过Object.create()创建的对象使用第一个参数，可以是null作为它们的原型  
 \*/*//isPrototypeOf  
**var** a = {};  
write(Object.prototype.isPrototypeOf(a)); //true  
  
**var** d = **new** Date();  
write(Date.prototype.isPrototypeOf(d)); //true  
write(Object.prototype.isPrototypeOf(d)); //true  
write(Object.prototype.isPrototypeOf(Date.prototype)); //true 这就是原型链了  
//d继承至Date.prototype Date.prototype继承至Object.prototype  
  
  
//Object.getPrototypeOf  
write(Object.getPrototypeOf(a)); //[object Object]  
Object.getPrototypeOf(d); //Date  
  
//d.constructor是创建这个对象的构造函数Date(),它有一个属性prototype指向原型对象  
write(d.constructor.prototype === Date.prototype); //true  
  
//\_\_prototype\_\_  
//IE和Opera未实现这个属性  
  
  
**var** o = {x:1};  
**var** o\_child = Object.create(o);  
//使用constructor.prototype检测对象原型的方式并不可靠  
write(o\_child.constructor.prototype); //[object Object]  
write(Object.getPrototypeOf(o\_child) === o); //true

/\*类属性\*/  
  
//[object Class]  
//ECMAScript3和ECMAScript5都没有提供设置这个属性的方法  
//只有一种间接的方法可以查询它,默认的toString()方法  
//由于很多对象继承的toString()方法重写了，为了正确调用toString()方法  
//必须间接调用Function.call()方法  
  
//返回传递给它的任意对象的类  
**function** classOf(o){  
 **if**(o === **null**) **return** 'Null';  
 **if**(o === undefined) **return** 'Undefined';  
 **return** Object.prototype.toString.call(o).slice(8,-1); //-1会加上length,其实就是length-1  
}  
  
//Object.prototype.toString.call(o) '[Object String]' 需要截取第八位到倒数第二位，也就是length-1  
  
write(classOf(**null**)); //Null  
write(classOf(undefined)); //Undefined  
write(classOf("")); //String  
write(classOf(1)); //Number  
write(classOf(**false**)); //Boolean  
write(classOf({})); //Object  
write(classOf([])); //Array  
write(classOf(/./)); //RegExp  
write(classOf(**new** Date())); //Date  
write(classOf(window)); //global 客户端宿主对象  
write(classOf(**new** Function())); //Function  
**function** F(){}  
write(classOf(**new** F())); //Object

//通过内置构造函数创造的对象，包含类属性，与构造函数的名称相匹配  
//通过对象直接量和Object.create创建的对象的类属性是"Object"  
//自定义的构造函数创建的对象也是一样，类属性是"Object"  
//所以对于自定义的类来说，没办法通过类属性来区分对象的类

/\*可扩展性\*/

*/\*\*  
 \* 1.对象的可扩展性用于表示是否可以给对象添加新属性  
 \* 2.Object.isExtensible()来判断对象是否可扩展  
 \* 3.Object.preventExtensions()用来设置不可扩展的对象  
 \* 但是如果给原型添加属性，这个不可扩展的对象同样会继承这些新属性  
 \*/***var** o = {};  
o.x = 1;  
write(o.x); //1  
write(Object.isExtensible(o)); //true  
Object.preventExtensions(o);  
write(Object.isExtensible(o)); //false  
o.y = 1;  
write(o.y); //undefined  
  
Object.prototype.aaa = 1;  
write(o.aaa); //1 仍然可以继承属性

## 4.9 对象序列化

/\*对象序列化\*/  
**var** o = {x:1,y:2};  
**var** s = JSON.stringify(o); write(s); //{"x":1,"y":2}  
  
**var** p = JSON.parse(s);  
write(p.x); //1  
write(p.y); //2 p是o的深拷贝

## 4.10 对象的方法

/\*对象方法\*/  
  
*/\*\*  
 \* 1.toString()方法  
 \* 2.toLocaleString()方法  
 \* 3.toJSON()方法  
 \* 4.valueOf()方法  
 \*/*//toLocalString方法  
*/\*\*  
 \* 1.Object默认的toLocalString()方法仅调用toString()方法  
 \* 2.Date和Number类的toLocalString()方法可以对数字、日期和事件做本地化的转换  
 \* 3.Array的的toLocalString()方法和toString()方法不同的是每个数组元素会调用toLocaleString方法转换为字符串  
 \* 而不是调用各自的toString()方法  
 \*/*//toJSON方法  
*/\*\*  
 \* 1.Object.prototype实际上没有定义toJSON方法  
 \* 2.JSON.stringify()方法会调用toJSON()方法  
 \*/***var** o = {x:1};  
//write(Object.prototype.toJSON(o)); //Uncaught TypeError: Object.prototype.toJSON is not a function  
  
**var** date = **new** Date();  
write(date.toJSON()); //2016-04-22T02:57:44.866Z  
  
//valueOf方法  
*/\*\*  
 \* 1.往往是将对象转换为某种原始值而非字符串的时候才会调用它  
 \* 2.尤其是转换为数字  
 \*/*

# 第五章 数组

## 5.1 创建数组

*/\*\*  
 \* 1.数组可能是稀疏的，数组元素的索引不一定要连续  
 \* 2.数组都有一个length属性，针对非稀疏数组，就是数组的个数，针对稀疏数组，则比所有元素的索引要大  
 \* 3.用数字索引来访问数组元素一般来说比访问常规的对象属性要快的多  
 \*/*

/\*创建数组\*/  
*/\*\*  
 \* 1.数组直接量 []  
 \*/***var** arr = [,,];  
write(arr.length); //2，注意不是3， 数组直接量的语法允许有可选的结尾的逗号  
  
**var** arr1 = [1,2,3,4,5,];  
write(arr1.length); //5，注意不是6  
  
*/\*\*  
 \* 2.使用构造函数 Array()  
 \*/***var** a = **new** Array();  
**var** b = **new** Array(10);  
**var** c = **new** Array(1,2,3,4,5);  
write(a.toString()); //  
write(b.toString()); //,,,,,,,,, 数组长度为10的空数组  
write(c.toString()); //1,2,3,4,5

## 5.2 数组元素的读和写

//数组是对象的特殊形式，用方括号访问数组元素就像用方括号访问对象的属性一样  
//JS将指定的数字索引值转换成字符串  
//索引值1变成'1'，然后将其作为属性名来使用  
**var** o = {};  
o[1] = 'new';

//所有的索引都是属性名，但只是0~2^32-2之间的整数属性名才是索引  
//所有的数组都是对象，但如果使用的属性是数组的索引，数组的特殊行为就是将根据需要更新它们的length属性值  
//可以使用负数或非正数来索引数组，此时数值转换为字符串，字符串作为属性名来使用，而非数组的索引  
//同样，使用了非负整数的字符串，就当作数组索引，而非对象属性  
  
**var** a = [];  
a[-1.23] = **true**; //将创建名为'-1.23'的属性  
write(a.length); //0 不会更新length属性  
a['hello'] = 11;  
write(a.length); //0 不会更新length属性  
a[999] = 1; //先将999转化为'999'，然后作为属性名来使用，也作为索引  
write(a.length); //1000 作为索引，而不只是作为属性名使用，更新了length属性  
a["1000"] = 0; //a[1000]是一样的  
write(a.length); //1001

//所有的索引都是属性名，但只是0~2^32-2之间的整数属性名才是索引  
//所有的数组都是对象，但如果使用的属性是数组的索引，数组的特殊行为就是将根据需要更新它们的length属性值  
//可以使用负数或非正数来索引数组，此时数值转换为字符串，字符串作为属性名来使用，而非数组的索引  
//同样，使用了非负整数的字符串，就当作数组索引，而非对象属性  
  
**var** a = [];  
a[-1.23] = **true**; //将创建名为'-1.23'的属性  
write(a.length); //0 不会更新length属性  
a['hello'] = 11;  
write(a.length); //0 不会更新length属性  
a[999] = 1; //先将999转化为'999'，然后作为属性名来使用，也作为索引  
write(a.length); //1000 作为索引，而不只是作为属性名使用，更新了length属性  
a["1000"] = 0; //a[1000]是一样的  
write(a.length); //1001  
//事实上数组索引仅仅是对属性名的一种特殊类型

## 5.3 稀疏数组

/\*稀疏数组\*/  
**var** a = **new** Array(5);  
write(a.length); //5  
a = [];  
write(a.length); //0

a[1000] = 0;  
write(a.length); //1001

//非稀疏数组  
**var** a = [,,,];  
write(a.length); //3  
write(a[0]); //undefined  
write(a[1]); //undefined  
write(a[2]); //undefined  
  
write(0 **in** a); //false????  
  
//稀疏数组  
**var** a1 = **new** Array(3);  
write(0 **in** a1); //false  
  
**var** a2 = [,];  
**var** a3 = [undefined]; //非稀疏数组  
  
write(0 **in** a2); //false  
write(0 **in** a3); //true  
  
**var** a4 = [1,,3]; //稀疏数组  
write(a4.length); //3

## 5.4 数组长度

/\*数组长度\*/  
**var** a = [];  
a[1000] = 1;  
write(a.length); //1001 length属性自增  
  
**var** a1 = [1,2,3,4,5];  
a1.length = 2;  
write(a1[2]); //undefined 索引值大于等于length的元素删除了

## 5.5 数组元素的添加和删除

/\*添加\*/  
  
**var** a = [1];  
a.push(2,3);  
write(a.toString()); //1,2,3  
a[a.length] = 4;  
write(a.toString()); //1,2,3,4  
  
a.unshift(0);  
write(a.toString()); //0,1,2,3,4  
  
/\*删除\*/  
  
**delete** a[3];  
write(a.toString()); //0,1,2,,4 变成了稀疏数组  
write(a.length); //5 length属性不变  
  
a.pop();  
write(a.toString()); //0,1,2, 还是稀疏数组  
write(a.length); //4 length属性变  
  
a.shift();  
write(a.toString()); //1,2, 还是稀疏数组  
write(a.length); //3 length属性变  
  
a.pop();  
write(a.toString()); //1,2 不是稀疏数组  
write(a.length); //2 length属性变

## 5.6 数组遍历

## 5.7 多维数组

## 5.8 数组的方法

/\*join()\*/  
//数组和字符串的转换  
  
**var** a = [1,2,3,4,5];  
write(a.join(',')); //1,2,3,4,5  
write(a.join()); //1,2,3,4,5  
  
write(a.join('-')); //1-2-3-4-5  
  
**var** str = '1-2-3-4-5';  
write(str.split('-')); //1,2,3,4,5 数组  
  
  
/\*reverse()\*/  
//逆序数组 改变的是原数组  
**var** a = [1,2,3,4,5];  
a.reverse();  
write(a); //5,4,3,2,1

/\*sort()\*/  
//以字母顺序排序，有必要先转换为字符串，改变的是原数组  
  
**var** b = ['a','c','b'];  
write(b.sort()); //a,b,c  
write(b); //a,b,c b数组变了  
  
//数组中有undefined则排到末尾  
  
**var** c = [1,2,88,3,4,5];  
  
c.sort(**function**(a,b){  
 **return** a-b; //数字小的排在前面  
});  
  
  
write(c); //1,2,3,4,5,88  
  
//不区分大小写排列字母  
**var** d = ['A','b','C','d'];  
  
d.sort(**function**(s,t){  
 **var** a = s.toLowerCase();  
 **var** b = t.toLowerCase();

//直接用一句 return a-b; 可以代替以下三句  
 **if**(a < b) **return** -1; //字母小的排在前面  
 **if**(a > b) **return** 1;  
 **return** 0;  
});  
  
  
write(d); //A,b,C,d

/\*concat()\*/  
//创建并返回新的数组  
  
**var** a = [1,2,3];  
**var** b = a.concat(4,5);  
write(a); //1,2,3 a本身不会变  
write(b); //1,2,3,4,5  
  
/\*slice()\*/  
//创建并返回新的子数组  
  
//第一参数 起始位置 包含  
//第二参数 末尾 不包含 如果省略，则到数组尾部  
  
a = [1,2,3,4,5,6,7,8,9];  
  
b = a.slice(6);  
write(a); //1,2,3,4,5,6,7,8,9  
write(b); //7,8,9 包含索引为6的项  
  
b = a.slice(-1);  
write(b); //9 等于 a.slice(8) 包含索引为8的项 即包含a[8]  
  
b = a.slice(1,4);  
write(b); //2,3,4 包含a[1],但是不包含a[4]

/\*splice()\*/ //插入和删除元素的通用方法  
//会修改调用的数组  
  
//第一参数 插入或删除的起始位置  
//第二参数 应该删除的元素个数 如果省略，则从起始点到末尾的元素都删除  
//从第三个参数开始是要插入的元素  
  
**var** c = [1,2,3,4,5,6,7,8,9];  
c.splice(5); //删除包含c[5]及之后的所有项  
write(c); //1,2,3,4,5 c数组本身改变了  
  
c.splice(2,2); //删除包含c[2]连续2项  
write(c); //1,2,5  
  
c.splice(2,0,'a'); //在索引为2位置插入'a',删除0项  
write(c); //1,2,a,5  
  
c.splice(2,2,[1,2],3); //c[2]开始删除两项 剩下1 2 , 在c[2]插入两项  
write(c); //1 2 [1 2] 3

/\*push() pop()\*/  
/\*unshift()　shift()\*/

//"use strict";  
  
/\*push() pop()\*/  
/\*unshift()　shift()\*/  
  
/\*toString() toLocalString()\*/  
write([1,2,3].toString()); //1,2,3  
write([1,['a','b'],3].toString()); //1,a,b,3  
write([1,['a','b'],3].toLocaleString()); //1,a,b,3

## 5.9 ECMAScript 5中数组的方法

//大多数方法都接受一个函数并对数组的每个元素调用一次该函数  
//如果稀疏数组，对不存在的元素不调用传递的函数  
//一般情况下，对调用的函数使用三个参数：1.数组元素 2.元素的索引 3.数组本身  
//通常只要一个参数，后两个参数是可以省略的  
  
/\*forEach()\*/  
  
**var** data = [1,2,3,4,5];  
  
**var** sum = 0;  
  
data.forEach(**function**(value){  
 sum += value;  
});  
  
write(sum); //15  
  
data.forEach(**function**(item,index,arr){  
 arr[index] = item + 1;  
});  
  
write(data); //2,3,4,5,6

/\*map()\*/  
//调用数组的每个元素传递给指定的函数，并返回一个数组，包含该函数的返回值  
  
**var** a = [1,2,3];  
**var** b = a.map(**function**(x){  
 **return** x\*x;  
});  
  
write(b); //1,4,9  
  
  
//传递给map的调用方式和传递给forEach的调用方式一样  
//与forEach不一样的是，map()函数应该有返回值，map返回的是新数组，不修改调用的数组  
//如果是稀疏数组，则返回的也是稀疏数组

/\*filter()\*/  
  
//filter() 方法返回的数组元素是调用的数组的一个子集  
//该函数返回true或false  
//调用判定函数就像调用forEach()和map()一样  
//如果返回值是true，那么传递给判定函数的元素就是数组子集的成员，会添加到被返回的数组中。  
  
a = [1,2,3,4,5];  
b = a.filter(**function**(x){  
 **return** x < 3; //注意返回的是true 就添加到子集中， 传入的x是第一个参数 数组元素，省略了另外一个参数  
});  
  
write(b); //1,2 跳过了稀疏数组，返回的总是稠密数组  
  
//压缩稀疏数组  
**function** dense(arr){  
 **if**(classOf(arr) !== 'Array') **return**;  
 **var** den = arr.filter(**function**(){**return true**;});  
 **return** den;  
}  
  
**var** arr = [1,,2];  
  
write(dense(arr)); //1,2

/\*every()和some()\*/  
//对数组的元素进行逻辑判定，返回true或false  
  
**var** arr = [1,2,3,4,5];  
**var** bool = arr.every(**function**(x){  
 **return** x < 10;  
});  
  
write(bool); //true 全部满足才行  
  
bool = arr.some(**function**(x){  
 **return** x > 10;  
});  
   
write(bool); //false 部分满足就行

/\*reduce()和reduceRight()\*/  
//将数组元素进行组合，生成单个值  
  
**var** a = [1,2,3,4,5];  
**var** sum = a.reduce(**function**(x,y){  
 **return** x+y;  
},0);  
  
write(sum); //15 第二个参数传递给函数的初始值  
  
// x y  
//第一次 0(传入的初始值) 1 (数组的第一个元素)  
// (0+1) = 1 2  
// 1+2 = 3 3  
// ...  
  
  
//如果没有第二个参数，那么默认的第一个x值是数组的第一个元素  
  
sum = a.reduce(**function**(x,y){  
 **return** x+y;  
});  
write(sum); //15 没有第二参数

/\*indexOf()和lastIndexOf()\*/  
  
**var** a = [1,2,3,2,5];  
  
write(a.indexOf(6)); //-1  
write(a.indexOf(2)); //1 索引 a[1] = 2  
write(a.lastIndexOf(2)); //3 从后往前索引，发现a[3] = 2 所以是3  
  
  
//查找数组a中所有出现x的值的索引，以数组的形式返回  
**function** findAll(a,x){  
 **var** results = [];  
  
 len = a.length;  
 pos = 0;  
  
 **while**(pos < len){  
 pos = a.indexOf(x,pos);  
 **if**(pos === -1) **break**;  
 results.push(pos);  
 pos ++;  
 }  
 **return** results;  
}  
  
  
//indexOf的第二个参数是需要索引的起始位置，第一个参数是需要索引的值  
//indexOf返回的是找到的第一个元素的索引  
//利用第二个参数就可以查找除了第一个以外匹配的值  
  
a = [1,2,3,4,5,5,65,6,7,8,7,6,5,5];  
write(findAll(a,5)); //4,5,12,13  
write(a[4]); //5

## 5.10 数组类型

//判断数组  
**function** isArray(o){  
 **if**(Array.isArray){  
 **return** Array.isArray(o);  
 } **else**{  
 **return typeof** o === "object" && Object.prototype.toString.call(o) === "[object Array]"; //注意toString没有()调用  
 }  
}  
  
write(isArray([])); //true

## 5.11 类数组对象

/\*类数组对象\*/  
//arguments对象就是一个类数组对象  
//类数组对象  
**var** a = {  
 '0': 'a',  
 '1': 'b',  
 '2': 'c',  
 length:3  
};  
  
//间接使用Function.call方法调用  
  
**var** str = Array.prototype.join.call(a,'-'); //第一个参数是调用的上下文环境，第二个参数是传入join方法的参数  
write(str); //a-b-c  
  
Array.prototype.splice.call(a,2); //删除包含a[2]之后的所有项  
write(a.toString()); //只剩下a[0] 和a[1]  
write(a.length); //2  
write(a[0]); //a  
write(a[1]); //b  
  
  
//var str1 = Array.join(a,'--'); //google不支持  
//write(str1);  
  
Array.join = Array.join || **function**(o,str){  
 **return** Array.prototype.join.call(o,str);  
};  
  
**var** str1 = Array.join(a,'--');  
write(str1); //a--b

## 5.12 作为数组的字符串

/\*作为数组的字符串\*/  
  
**var** s = 'test';  
write(s[1]); //e  
  
**var** str = Array.prototype.join.call(s,'-');  
write(str); //t-e-s-t  
  
//字符串是只读的，不能通过数组的push pop等方法改变它

# 第六章 函数

## 6.1 函数定义

/\*函数定义\*/  
  
//函数声明不能出现在循环,条件判断或者try/catch/finally以及with语句里  
//函数定义表达式可以出现在JS代码的任何地方

## 6.2 函数调用

/\*函数调用\*/  
//判断严格模式与非严格模式  
**var** strict = (**function**(){  
 **return** !**this**; //非严格模式下，是Window全局对象 是不是匿名函数的原因？？？  
}());  
write(strict); //false

"use strict";  
 **return** !**this**; //严格模式下，调用的上下文则是undefined  
}());  
  
write(strict); //true

/\*方法调用\*/  
//方法调用和函数调用有一个重要的区别，调用上下文  
//方法调用使用该方法对应的对象作为调用上下文，函数体可以用this引用该对象  
  
**var** o = {  
 a:1,  
 b:2,  
 add: **function**(){  
 **this**.result = **this**.a + **this**.b;  
 },  
  
 f: **function**(){  
 **this**.c = **this**.b - **this**.a;  
 }  
};  
  
o.add();  
write(o.result); //3 this指代o对象  
  
//方括号属性访问  
o['f']();  
write(o.c); //1  
  
//方法的链式调用  
//当方法不需要返回值时最好直接返回this  
//这样只要指定一次调用的对象即可  
  
//例如  
//a.f().f1().f2().f3()  
//f f1 f2 f3都是a的方法

//this值  
//如果嵌套的函数作为方法调用，其this值指向调用它的对象  
//如果作为函数调用，其this值不是全局对象就是undefined  
  
**var** o = {  
 m:**function**(){  
 **var** self = **this**;  
 console.log(**this**); //指代o  
 f();  
  
 **function** f(){  
 console.log(**this** === o); //false this指代全局对象window，因为这里作为函数调用  
 console.log(self === o); //true  
 }  
 }  
  
};  
  
o.m();  
  
  
//调用构造函数  
//构造函数通常不使用return关键字，通常初始化一个新对象  
//构造函数函数体执行完毕时，会显式返回，  
//如果显式的使用return语句返回一个对象，那么调用表达式的值就是这个对象  
//如果使用return语句但没有指定返回值，或者返回一个原始值，那么这时将忽略返回值  
//同时使用新对象作为调用结果  
  
**function** A(){  
 **this**.x = 1;  
 **this**.y = 2;  
 **return**; //此时return被忽略  
}  
  
**var** a = **new** A();  
write(a); //[object Object]  
  
**function** B(){  
 **return** 1; //此时return被忽略  
}  
  
**var** b = **new** B();  
write(b); //[object Object]  
  
**var** b1 = B();  
write(b1); //1 当做普通的函数调用

**function** C(){  
 **this**.x = 1;  
 **this**.y = 2;  
  
 **var** o = {};  
 o.x = 3;  
 **return** o;  
}  
  
**var** o1 = **new** C();  
write(o1.x); //3 return的是o对象

//间接调用  
//call和apply可以用来间接调用函数  
//显示指定调用所需的this值  
//任何函数可以作为任何对象的方法来调用，哪怕这个函数不是那个对象的方法

## 6.3 函数的实参和形参

/\*函数的实参和形参\*/  
**function** getPropertyNames(o,a){  
 a = a || []; //这里形参a相当于var a; 否则如果不预先声明，这句会报错  
 //　|| 第一个是真值的话，就返回第一个,否则返回第二个  
  
 **for**(**var** prop **in** o){  
 a.push(prop);  
 }  
 **return** a;  
}  
  
**var** o = {x:1};  
  
**var** a = getPropertyNames(o); //第二个参数省略  
write(a.toString()); //x  
  
  
**var** o1 = {y:1};  
getPropertyNames(o1,a); //追加a数组的内容  
write(a.toString()); //x,y

//arguments  
//非严格模式  
//类数组对象 它其实是一个对象，只是碰巧具有以数字为索引的属性  
**function** f(x){  
 write(x); //5  
 arguments[0] = undefined;  
 write(x); //undefined  
}  
  
f(5);  
//严格模式  
**function** f1(x){  
 "use strict";  
 write(x); //5  
 arguments[0] = undefined; //严格模式下不能给arguments赋值  
 write(x); //5  
}  
f1(5);

## 6.4 作为值的函数

/\*作为值的函数\*/  
  
**function** square(x){  
 **return** x\*x;  
}  
  
//这个定义创建一个新的函数对象，并将其赋值给变量square,函数的名字实际上看不见的  
//square仅仅是变量的名字，这个变量指代函数对象  
  
**var** s = square; //s和square指代同一个函数  
  
write(square(4)); //16  
write(s(4)); //16  
  
//函数如果赋值为对象的某个属性，那么函数就称为方法  
//函数甚至不需要名字  
  
**var** a = [**function**(x){**return** x\*x;}, 20];  
write(a[0](a[1])); //400  
  
  
//将函数用作值  
  
**function** operate(value1, value2, f){  
 **return** f(value1,value2);  
}  
  
**var** i = operate(10,20, **function**(x,y){ //此刻函数就是一个实参对象，传递给f这个形参  
 **return** x+y;  
});  
  
write(i); //30

/\*自定义函数属性\*/  
  
fn.counter = 0; //函数对象的计数器属性，由于函数声明被提升，所以个可以使用函数对象的属性  
  
**function** fn(){  
 **return** fn.counter++;  
}  
  
  
write(fn()); //0  
write(fn()); //1  
write(fn()); //2  
write(fn.counter); //3

## 6.5 作为命名空间的函数

/\*作为命名空间的函数\*/  
  
//函数的作用域  
//在函数中声明的变量在整个函数体内都是可见的，包括嵌套的函数中  
//在函数的外部不可见  
  
//块级作用域  
(**function**(){  
 //模块代码  
})();  
  
//注意必须要圆括号，这样就解析为函数表达式，如果没有就解析为函数声明

**var** extend\_ieBug = (**function**(){  
 **for**(**var** p **in** {toString:**null**}){  
 //  
 **return function** extend(o){  
 **for**(**var** i=0,len=arguments.length; i<len; i++){  
 **var** source = arguments[i];  
 **for**(**var** prop **in** source){  
 o[prop] = source[prop];  
 }  
 }  
 **return** o;  
 }  
 }  
  
 //for-in如果不会枚举测试对象的toString属性  
 //这个函数显示测试Object.prototype中不可枚举的属性  
 **return function** patched\_extend(o){  
 **for**(**var** i=0,len=arguments.length; i<len; i++){  
 **var** source = arguments[i];  
 **for**(**var** prop **in** source){  
 o[prop] = source[prop];  
 }  
  
 //检测特殊属性  
 **for**(**var** j=0,len1=protoprops.length; i<len1; i++){  
 prop = protoprops[i];  
 **if**(source.hasOwnProperty(prop)){  
 o[prop] = source[prop];  
 }  
 }  
 }  
 };  
  
 **var** protoprops = ['toString','valueOf','constructor',  
 'hasOwnProperty','isPrototypeOf','toLocalString'];  
  
}()); //立即执行后,extend\_ieBug = extend  
  
**var** o = {x:1};  
extend\_ieBug(o);

## 6.6 闭包（理解不深）

/\*闭包\*/  
  
**var** scope = 'golbal scope';  
  
**function** checkScope(){  
 **var** scope = 'local scope';  
 **function** f(){**return** scope};  
 **return** f(); //返回的是局部变量scope的值，因为返回的是执行的f函数  
}  
  
  
write(checkScope()); //local scope  
  
  
  
**function** checkScope1(){  
 **var** scope1 = 'local scope';  
 **function** f(){**return** scope1};  
 **return** f;   
}  
  
write(checkScope1()()); //local scope  
  
//闭包可以捕捉到局部变量（和参数），并一直保存下来  
//如果存在嵌套的函数，函数都有各自对应的作用域链，并且这个作用域链指向一个变量绑定对象  
//如果函数定义了嵌套函数，并将它作为返回值返回或者存储在某处的属性里  
//这时就会有一个外部引用指向这个嵌套的函数，就不会被当做垃圾回收，它所指向的变量绑定对象也不会被当做垃圾回收  
//容易造成内存泄漏  
  
//闭包的作用可以使局部变量用作私有状态

## 6.7 函数属性、方法和构造函数

/\*函数属性、方法和构造函数\*/  
  
/\*函数的prototype属性\*/  
  
//每个函数都包含一个prototype属性，这个属性指向一个对象的引用  
//这个被引用的对象就叫做"原型对象"  
//每一个函数都包含一个不同的原型对象  
//当将函数用作构造函数的时候，新创建的对象会从原型对象上继承属性  
  
/\*函数的call和apply方法 Function.call\*/  
//call()和apply()的第一个实参是要调用函数的母对象，即调用上下文  
//在函数体内通过this来获得对它的引用  
  
//Object.prototype.toString.call()  
  
**function** f(){  
 write(**this**.x);  
}  
**var** o = {x:1};  
f.call(o); //1 this指代o  
f.apply(o);  
  
//在ECMAScript5中call和apply的第一个实参都会变成this的值，哪怕传入的实参是原始值甚至是null或undefined  
//在E3中传入null和undefined会被全局对象代替，而其他的原始值则会被相应的包装对象替代  
  
//对于call来说,第一个调用的上下文实参之后的所有实参就是要传入待调用函数的值  
**function** f1(x,y){  
 **var** sum = x + y + **this**.x;  
 write(sum);  
}  
  
f1.call(o,1,4); //6  
f1.apply(o,[1,4]); //6  
  
**var** max = Math.max(1,2,3,4,5,6,7);  
write(max); //7  
//这种情况下，如果要比较数组参数的最大值  
**var** arr = [1,2,3,4,5,6,7,1234];  
max = Math.max.apply(Math,arr);  
write(max); //1234  
  
//传入apply中的参数可以是类数组对象也可以是真实数组  
//例如arguments  
  
  
/\*函数的bind方法 Function.bind\*/  
//将函数绑定至某个对象  
  
**function** fn(y){  
 write(**this**.x + y);  
}  
  
**var** o1 = {x:1};  
  
**var** g = fn.bind(o1); //在函数fn上调用bind()方法并传入一个对象o1作为参数  
//这个方法将返回一个新的函数，这个新的函数会将原始函数fn当做o1的方法来使用  
g(4); //5 this指代o1  
g(7); //8 this指代o1  
  
//模拟bind函数  
  
**function** newBind(o,f){  
 **if**(f.bind) **return** f.bind(o);  
 **else return function**(){   
 **return** f.apply(o,arguments);  
 }  
}  
  
**var** g1 = newBind(o,fn);  
g1(100); //101  
  
//bind函数不仅仅将函数绑定至一个对象  
//传入bind()的实参也会绑定至this,这个附带的应用是一种常见的函数式编程技术  
//有时也叫 “柯里化(currying)”  
  
  
**var** sum = **function**(x,y){  
 write(x+y);  
};  
  
**var** succ = sum.bind(**null**,1); //1绑定形参x this绑定到null  
succ(3); //4 3指代y  
succ(103); //104  
  
**var** sum1 = **function**(y,z){  
 write(**this**.x + y + z);  
};  
**var** g = sum1.bind({x:1}); //绑定this  
g(10,10); //21  
  
//ECMAScript3中的方法  
**if**(!Function.prototype.bind){  
 Function.prototype.bind = **function**(o /\*, args\*/){  
 //将this值和arguments保存至变量中  
 //以便在嵌套函数中可以使用  
 **var** self = **this**;  
 **var** boundArgs = arguments;  
  
 //bind方法返回的是函数，闭包  
 **return function**(){  
 **var** args = [];  
 **var** i,len;  
  
 **for**(i=1,len = boundArgs.length; i<len; i++){ //第0个参数是对象o，所以i=1  
 args.push(boundArgs[i]); //这里是bind绑定函数中传入的参数  
 //例如var succ = sum.bind(null,1); 中的实参1  
 }  
 //这里的arguments是需要return返回的函数所传入的参数数组  
 **for**(i=0,len = arguments.length; i<len; i++){  
 args.push(arguments[i]); //这里是例如上面的g所传入的参数  
 //例如g(10,10) 则传入的是10 10  
 }  
  
 //例如Fun继承至Function.prototype  
 //Fun.bind = function(){ self = this}; 那么this指代Fun对象，bind是Fun的属性  
 //所以self指代要需要继承bind方法的函数对象  
 **return** self.apply(o,args);  
 };  
 };  
  
  
  
  
}  
  
  
//例如Fun继承至Function.prototype  
//Fun.bind = function(){ self = this}; 那么this指代Fun对象，bind是Fun的属性  
//所以self指代要需要继承bind方法的函数对象  
**var** obj = {x:1};  
obj.say = **function**(){  
 write(**this**.x); //this指代obj  
};  
obj.say(); //1

/\*函数的toString方法\*/  
**function** f(){  
  
}  
  
write(f.toString()); //function f(){ }  
  
  
/\*Function()构造函数\*/  
  
*/\*\*  
 \* 1. 允许动态创建并编译函数  
 \* 2. 影响执行效率  
 \* 3. 所创建的函数并不是使用词法作用域，函数体代码的编译总是会在顶层函数执行  
 \*/***var** scope = 'global';  
  
**function** constructFunction(){  
 **var** scope = 'local';  
 **return new** Function('return scope'); //无法捕捉局部作用域  
}  
  
write(constructFunction()()); //global  
  
  
//检测函数  
**function** isFunction(f){  
 **return** Object.prototype.toString.call(f) === '[object Function]';  
}  
  
write(isFunction(constructFunction)); //true

## 6.8 函数式编程

/\*使用函数处理数组\*/  
  
**var** sum = [1,2,3,4,5,5,5,7,8,9];  
  
**var** add = **function**(x,y){  
 **return** x + y;  
};  
  
**var** square = **function**(x){  
 **return** x\*x;  
};  
  
**var** aver = sum.reduce(add)/sum.length;  
write(sum.reduce(add)); //49  
write(aver); //4.9  
  
**var** dev = sum.map(**function**(x){  
 **return** x-aver;  
});  
  
write(dev);  
**var** stddev = Math.sqrt(dev.map(square).reduce(add)/(sum.length-1));  
write(stddev); //2.558211180579986

/\*高阶函数\*/

//返回一个新的可以计算f(g(...))的函数  
//返回的函数将它所有的实参传入g(),然后将g()的放回值传入f()  
//这里的this值是同一个值  
**function** compose(f,g){  
 **return function**(){  
 //f只需要传入一个参数，所以是call方法  
 //g需要传入多个参数，所以时apply方法  
 **return** f.call(**this**,g.apply(**this**,arguments));  
 }  
}  
  
**var** square = **function**(x){  
 **return** x\*x;  
};  
  
**var** sum = **function**(x,y){  
 **return** x+y;  
};  
  
**var** squareSum = compose(square,sum);  
write(squareSum(2,2)); //16

/\*不完全函数\*/ P197  
/\*记忆函数\*/ P199

# 第七章 类和模块

## 7.1 类和原型

/\*类和原型\*/  
  
//继承对象  
**function** inherit(p){  
 **if**(p == **null**) **return**; //undefined 和 null都会return  
 **if**(Object.create){  
 **return** Object.create(p); //p是返回对象的原型对象  
 }  
  
 **var** t = **typeof** p;  
 **if**((t != "object") && (t != "function")) **return**;  
 **function** f(){} //构造函数  
 f.prototype = p; //将其原型对象属性设置为p  
 **return new** f(); //使用f()创建一个p的继承对象  
}  
  
//类的所有实例对象都从同一个原型对象上继承属性  
//原型对象是类的核心  
  
  
//如果定义了一个原型对象，然后通过inherit()函数创建一个继承自它的对象  
//这样就定义了一个JavaScript类  
  
  
//一个简单的JavaScript类  
  
//这个工厂方法返回一个新的'范围对象'  
**function** range(**from**,to){  
  
 **var** r = inherit(range.methods); //创建一个对象，这个对象继承至range.methods  
 //range.methods当然也是一个原型对象  
  
  
 //这两个属性是不可继承的，每个对象都拥有唯一的属性  
 r.from = **from**;  
 r.to = to;  
  
 **return** r;  
}  
  
  
//原型对象 range.methods 需要注意的是这个对象挂载在函数对象上了  
//它所定义的方法为每个范围对象所继承  
  
range.methods = {   
 //以下函数都用到了from和to属性，而且使用了this关键字  
 //为了指代它们，使用this关键字来指代调用这个方法的对象  
 //任何类的方法都可以通过this来读取对象的属性  
 includes: **function**(x){  
 write(**this**.from <= x && x <= **this**.to) ;  
 },  
  
 forEach: **function**(f){  
 **for**(**var** x = Math.ceil(**this**.from); x <= **this**.to; x++){  
 //Math.ceil 执行向上舍入，即它总是将数值向上舍入为最接近的整数；  
 f(x);  
 }  
 },  
  
 toString: **function**(){  
 write("(" + **this**.from + '...' + **this**.to + ')') ;  
 }  
};  
  
  
**var** r = range(1,6); //使用range工厂方法创建了一个范围对象  
//r有两个自有不可继承的属性，这两个属性也是非共享的  
//r对象同时还有三个继承至原型对象的方法  
r.includes(9); //false 不在范围内  
r.includes(6); //true 在范围内  
r.forEach(write); //1,2,3,4,5,6  
r.toString(); //(1...6)

## 7.2 类和构造函数

/\*类和构造函数\*/  
  
//7.1展示的是定义类的其中一种方法，这种方法并不常用  
//因为它没有定义构造函数，构造函数用来初始化新创建的对象  
//构造函数的prototype属性被用做新对象的原型  
//同一个构造函数创建的所有对象都继承自一个相同的对象，因此它们都是同一个类的成员  
  
//使用构造函数代替7.1中的工厂函数  
//使用构造函数来定义范围类Range  
  
//这是一个构造函数，用以初始化新创建的范围对象  
//这里并没有创建并返回一个对象，仅仅是初始化  
**function** Range(**from**,to){  
 **this**.from = **from**;  
 **this**.to = to;  
}  
  
//所有的'范围对象'都继承自这个对象  
//注意，属性的名字必须是prototype  
Range.prototype = {  
 includes: **function**(x){  
 write(**this**.from <= x && x <= **this**.to) ;  
 },  
  
 forEach: **function**(f){  
 **for**(**var** x = Math.ceil(**this**.from); x <= **this**.to; x++){  
 //Math.ceil 执行向上舍入，即它总是将数值向上舍入为最接近的整数；  
 f(x);  
 }  
 },  
  
 toString: **function**(){  
 write("(" + **this**.from + '...' + **this**.to + ')') ;  
 }  
};  
  
**var** r = **new** Range(1,6);  
  
r.includes(9); //false 不在范围内  
r.includes(6); //true 在范围内  
r.forEach(write); //1,2,3,4,5,6  
r.toString(); //(1...6)  
  
//常见的编程约定：定义构造函数既是定义类，并且类名首字母要大写

/\*构造函数和类的标识\*/  
  
//原型对象是类的唯一标识,当且仅当两个对象继承自同一个原型对象时，它们才是属于同一个类的实例  
//构造函数不能作为类的唯一标识，因为两个构造函数的原型对象可能是同一个  
//那么这两个构造函数创建的实例是属于同一个类的  
  
//构造函数的名字通常用作类名  
//当使用instanceof运算符来检测对象是否属于某个类时会用到构造函数  
write(r **instanceof** Range); //true 因为r继承自Range.prototype,所以返回true  
write(r **instanceof** Object); //true 因为Range.prototype又继承自Object.prototype  
  
//实际上instanceof运算符并不会检测r是否由Range()构造函数初始化而来  
//而是会检查r是否继承自Range.prototype  
//不过instanceof语法强化了'构造函数是类的公有标识'的概念  
  
/\*constructor属性\*/  
  
//上例中，Range.prototype定义为一个新对象,这个对象包含了类所需要的方法  
//任何JS函数都可以用作构造函数，并且调用构造函数是需要用到prototype属性的  
//每个JavaScript函数（Function.bind()方法返回的函数除外）都自动拥有一个prototype属性  
//这个属性的值是一个对象，这个对象包含一个唯一一个不可枚举的属性constructor  
//constructor属性的值是一个函数对象  
**var** F = **function**() {}; //函数对象  
**var** p = F.prototype; //F的原型对象  
**var** c = p.constructor; //与原型相关联的函数  
write(c === F); //true  
  
//p是F的原型对象，实例对象继承的constructor均指代它们的构造函数  
**var** o = **new** F();  
write(o.constructor === F); //true o是实例对象，constructor属性继承自F对应的原型对象  
  
  
*/\*\*  
 Range.prototype = {  
 includes: function(x){  
 write(this.from <= x && x <= this.to) ;  
 },  
  
 forEach: function(f){  
 for(var x = Math.ceil(this.from); x <= this.to; x++){  
 //Math.ceil 执行向上舍入，即它总是将数值向上舍入为最接近的整数；  
 f(x);  
 }  
 },  
  
 toString: function(){  
 write("(" + this.from + '...' + this.to + ')') ;  
 }  
 }; \*/*//需要注意的是，上述Range.prototype对象是使用一个新对象重写预定义的Range.prototype对象  
//新定义的原型对象不含有constructor属性  
write(Range.prototype.constructor === Range); //false  
  
//这样使用Range构造函数创建的实例对象就没有继承constructor属性  
//因为原型对象中这个属性不存在  
**var** r1 = **new** Range(1,6);  
write(r1.constructor === Range); //false  
  
//补救措施一  
Range.prototype = {  
 constructor: Range, //显示设置构造函数反向引用  
 includes: **function**(x){  
 write(**this**.from <= x && x <= **this**.to) ;  
 },  
  
 forEach: **function**(f){  
 **for**(**var** x = Math.ceil(**this**.from); x <= **this**.to; x++){  
 //Math.ceil 执行向上舍入，即它总是将数值向上舍入为最接近的整数；  
 f(x);  
 }  
 },  
  
 toString: **function**(){  
 write("(" + **this**.from + '...' + **this**.to + ')') ;  
 }  
};  
  
write(Range.prototype.constructor === Range); //true  
write(r1.constructor === Range); //false  
**var** r2 = **new** Range(1,6);  
write(r2.constructor === Range); //true

//补救措施二  
**function** Range(**from**,to){  
 **this**.from = **from**;  
 **this**.to = to;  
}  
  
write(Range.prototype.constructor === Range); //true  
//每个JavaScript函数都自动拥有一个prototype属性  
//默认的原型对象也具有constructor属性，不可枚举的  
//不要重写预定义的原型对象  
//仅仅在预定义原型对象的基础上增加方法或者属性  
Range.prototype.includes = **function**(x) {  
 write(**this**.from <= x && x <= **this**.to) ;  
};



## 7.3 JavaScript中Java式的类继承

*/\*\*  
 \* 定义个类分三步走  
 \* 1.定义一个构造函数，并设置初始化新对象的实例属性  
 \* 2.给构造函数的prototype对象定义实例的方法  
 \* 3.给构造函数定义类字段和类属性  
 \*/*//定义一个构造函数，并设置初始化新对象的实例属性  
**function** Person(name,age){  
 **this**.name = name || '';  
 **this**.age = age || 0;  
}  
  
//给构造函数的prototype对象定义实例的方法  
Person.prototype.sayInfo = **function**(){  
 write(**this**.name + '-' + **this**.age);  
};  
//类字段和类方法是直接定义为构造函数的属性  
  
//给构造函数定义类字段和类属性  
//类属性中的常量，常量默认是大写字母  
Person.ZXK = **new** Person('victor',26);  
//类属性中的私有字段，私有字段使用\_下划线约定  
Person.\_minAge = 18; //成年人标准  
  
//类方法  
Person.isAdult = **function** (p) {  
 **if**(p.age >= Person.\_minAge){ //私有字段其实只用在类方法中，不对外公开  
 **return true**;  
 }  
 **return false**;  
};  
//当然这里的私有字段其实对外也看得见，这并不是我们想要的  
//所以可以使用闭包里的局部变量来模拟  
  
**var** p = **new** Person('zxk',35);  
write(Person.isAdult(p)); //true  
p.sayInfo(); //zxk-35

## 7.4 类的扩充

/\*类的扩充\*/  
  
**var** Person = **function**() {};  
  
Person.prototype.say = **function**(){  
 write('name');  
};  
  
**var** p = **new** Person(); //创建一个实例对象  
p.say(); //name  
  
  
Person.prototype.sayAge = **function**(){  
 write('age');  
};  
  
p.sayAge(); //age 创建对象之后原型属性发生改变，也会影响到继承这个原型的所有实例对象

## 7.5 类的类型

/\*类和类型\*/  
  
//三种检测任意对象的类的技术 instanceof运算符 constructor属性 构造函数的名字  
  
/\*instanceof运算符\*/  
//构造函数是类的公共标识，原型是唯一的标识  
//尽管instanceof运算符的右操作数是构造函数，但计算过程实际上是检测了对象的继承关系  
//而不是创建对象的构造函数  
  
/\*constructor属性\*/  
  
  
/\*构造函数的名称\*/  
  
  
//可以判断值类型的type()函数  
  
  
*/\*\*  
 \* 以字符串形式返回o的类型  
 \* 如果o是null，返回null，如果是NaN，则返回'nan'  
 \* 如果typeof所返回的值不是'object',则返回这个值  
 \* 如果o的类不是'Object',则返回这个值  
 \* 如果o包含构造函数并且构造函数具有名称，则返回这个名称  
 \* 否则，一律返回'Object'  
 \*  
 \*/*Function.prototype.getName = **function**(){  
 //有些JS实现中为函数对象提供了一个非标准的属性name  
 **if**('name' **in this**) **return this**.name; //这里this指代F构造函数  
 **return this**.name = **this**.toString().match(/function\s\*([^(]\*)\(/)[1];  
};  
  
  
**function** type(o){  
 **var** t,c,n; //type,class name  
  
 **if**(o === **null**) **return** 'null';  
  
 **if**(o!== o) **return** 'nan';  
  
  
 **if**((t = **typeof** o) != 'object') **return** t;  
  
 **if**((c = classOf(o)) !== 'Object') **return** c;  
  
  
 **if**(o.constructor && **typeof** o.constructor === "function" &&  
 (n = o.constructor.getName()))  
 **return** n;  
  
  
 **function** classOf(o){  
 **return** Object.prototype.toString.call(o).slice(8,-1); //-1会加上length,其实就是length-1  
 }  
}  
  
**var** a = 1;  
write(type(a)); //number  
  
**var** b = **new** Date();  
write(type(b)); //Date  
  
**function** F(){};  
**var** c = **new** F();  
write(type(c)); //F  
  
//有些JS实现中为函数对象提供了一个非标准的属性name  
write(F.prototype.constructor.name); //F  
  
//当然也存在缺陷  
//并不是所有的对象都具有constructor属性  
**function** sum(a,b){  
 **return this**.x + a + b;  
}  
  
**var** o = {x:1};  
**var** k = sum.bind(o);  
write(k(3,2)); //6  
  
write(type(k)); //function  
  
**var** Fn = **function**(x,y){ //匿名的构造函数  
 **this**.r = x;  
 **this**.i = y;  
};  
  
write(type(Fn)); //function  
write(Fn.getName()); //"" 返回的是空字符串

/\*鸭式辩型\*/  
//不要关注对象的类是什么，而是关注对象能做什么 p215

## 7.6 JavaScript中的面向对象的技术

## 7.7 子类

/\*子类\*/  
//类B可以继承自类A，那么将A称为父亲  
//将B称为子类  
  
  
  
//B的原型对象继承自A的原型对象  
//B.prototype = inherit(A.prototype);  
  
**function** defineSubclass(superclass, //父类构造函数  
 constructor,//新的子类构造函数  
 methods, //实例方法：复制至原型中  
 statics) //类属性：复制至构造函数中  
{  
 //创建子类的原型对象  
 //子类的原型对象是父类构造函数的一个实例对象!!!!  
 constructor.prototype = inherit(superclass.prototype);  
 //因为重写了预定义的原型对象，所以需要重新设定constructor属性  
 constructor.prototype.constructor = constructor;  
   
 //像常规类一样赋值方法和类属性  
 **if**(methods) extend(constructor.prototype,methods);  
 **if**(statics) extend(constructor,statics);  
   
 //返回这个类  
 **return** constructor;  
}  
  
  
//通过父类构造函数的方法来做到这一点  
Function.prototype.extend = **function**(constructor,methods,statics){  
 **return** defineSubclass(**this**,constructor,methods,statics);  
};  
  
//另外一种实现子类的方式  
**function** Father(){}  
  
**function** Son(name){  
 **this**.name = name;  
}  
  
Father.prototype.add = **function**(){  
  
};  
  
Father.prototype.remove = **function**(){  
  
};  
  
Father.prototype.size = **function**(){  
  
};  
  
//子类的原型对象是父类构造函数的一个实例对象!!!!  
Son.prototype = inherit(Father.prototype);  
//类似于 Son.prototype = new Father()  
  
//给原型添加属性  
//如果有同名的属性就覆盖Father.prototype中的同名属性  
//这样的话将继承至父类的核心方法全部替换掉了  
  
extend(Son.prototype,{  
 constructor:Son,   
 add:**function**(){}, //覆盖了父类的方法  
 remove:**function**(){},  
 size:**function**(){},  
});  
  
  
//动态的从Father类继承方法，如果给Son.prototype添加新的方法,Son的所有实例就会立即拥有这个方法。

//"use strict";  
  
/\*构造函数和方法链\*/  
  
//以上的Son类虽然继承了父类的方法，但是由于重写了方法，所以覆盖了继承的方法  
//往往定义子类时，我们并不需要完全替换掉他们，而是对父类的行为进行修改或扩充  
  
//为了做到这一点，构造函数和子类的方法需要调用或链接到父类构造函数和父类方法  
  
  
//例如  
**function** Father(name,age){  
 **this**.name = name;  
 **this**.age = age;  
}  
  
  
Father.prototype.sayAge = **function**(){  
 write(**this**.age);  
};  
  
Father.prototype.sayName = **function**(){  
 write(**this**.name);  
};  
  
Father.prototype.add = **function**(t){  
 t = **this**.name + '-' + **this**.age + '-' + t;  
 write(t);  
};  
  
  
//Son是Father的子类  
**function** Son(){  
 //仅链接到父类  
 //作为普通函数调用父类的构造函数来初始化该构造函数调用创建的对象  
 Father.apply(**this**,arguments);  
}  
  
//设置为Father的子类  
Son.prototype = inherit(Father.prototype);  
//重写了预定义的prototype属性，需要重新定义constructor属性  
Son.prototype.constructor = Son;  
  
  
//父类的add方法中并没有对t进行值检测，重写父类的方法  
Son.prototype.add = **function**(t){  
 **if**(t == **null**) **return**; //如果t是null或者undefined  
  
 //调用父类的add()方法以执行实际的操作  
 **return** Father.prototype.add.apply(**this**,arguments);  
};  
  
  
**var** s = **new** Son('victor',23);  
write(s.name); //victor  
write(s.age); //23  
  
**var** f = **new** Father('hugo',24);  
write(f.name); //hugo  
write(f.age); //24  
  
s.add('hello'); //victor-23-hello  
s.add(**null**); //''  
f.add('hello'); //victor-23-hello  
f.add(**null**); //hugo-24-null

//类工厂和方法链  
  
//函数返回具体fatherClass类的子类  
//并重写该类的add()方法用以对添加的元素做特殊的过滤  
  
**function** filteredSonClass(fatherClass,fiter){  
 **var** constructor = **function**(){ //子类构造函数  
 fatherClass.apply(**this**,arguments);  
 };  
  
 **var** proto = constructor.prototype = inherit(fatherClass.prototype);  
 proto.constructor = constructor;  
 proto.add = **function**(t) {  
 **if**(!fiter(t)) **return**;  
 //调用父类的方法  
 fatherClass.prototype.add.apply(**this**,arguments);  
 };  
  
 **return** constructor;  
}  
  
  
**function** F(name,age){  
 **this**.name = name;  
 **this**.age = age;  
}  
  
  
F.prototype.add = **function**(t){  
 write(t);  
};  
  
  
**var** S = filteredSonClass(F,**function**(x){  
 **return typeof** x === 'string'; //只能显示字符串  
});  
  
**var** s = **new** S('victor',23);  
write(s.age); //23  
write(s.name); //victor  
s.add(1); //''  
s.add('string');//string

/\*组合VS子类\*/  
//组合优于继承  
/\*类的层次结构和抽象类\*/

## 7.8 ECMAScript5中的类

/\*定义不可变的类\*/  
  
//实现的构造函数也可以用作工厂函数,这样不论调用函数之前是否带有new关键字，都可以正确的创建实例  
  
**function** Range(**from**,to){  
  
 **var** props = {  
 form: {value:**from**, enumerable:**true**, writable:**false**, configurable:**false**},  
 to:{value:to, enumerable:**true**, writable:**false**, configurable:**false**}  
 };  
  
  
 **if**(**this instanceof** Range){ //使用new运算符创建的对象所指代的this当然属于Range类  
  
 //方法调用和函数调用不一样  
 //方法调用this指代的是调用该方法的对象  
 //函数调用this值不是全局对象就是undefined  
 //如果使用new运算符，this当然指代Range类创建的对象？？？  
 Object.defineProperties(**this**,props); //定义多个属性特性  
 }  
  
 **else  
 return** Object.create(Range.prototype,props); //属性由props指定  
 //返回的是一个实例对象，该实例对象的原型对象是Range.prototype  
}  
  
  
**var** r = **new** Range(1,6); //this = Range{}  
**var** R = Range(1,6); //this = Window  
  
write(r.form); //1 构造函数  
write(r.to); //6  
  
write(R.form); //1 工厂方法  
write(R.to); //6  
  
  
//如果用同样的方法给Range.prototype对象添加属性  
//那么我们需要给这些属性设置他们的特性  
//因为我们无法识别出他们的可枚举性，可写性或者可配置性，这些属性默认值都是false  
Object.defineProperties(Range.prototype,{  
 includes:{  
 value: **function**(x){**return** x;}  
 },  
  
 add:{  
 value: **function**(x){**return** x\*x;}  
 }  
});

//"use strict";  
  
  
/\*定义不可变的类\*/  
  
**var** arr = [1,2,3,4];  
**var** arrS = arr.splice(2);  
write(arr); //1,2 删去包含a[2]及之后的所有项  
write(arrS); //3,4 返回的是删去的项  
  
  
//属性描述符工具函数  
//将o指定的名字的属性(或所有)  
//如果只传一个对象参数,则操作该对象所有的属性  
//如果传入两个以上参数，则操作指定的名字的属性  
//将这些属性设置为不可写的和不可配置的  
**function** freezeProps(o){  
 **var** props = (arguments.length == 1) ? Object.getOwnPropertyNames(o)  
 : Array.prototype.splice.call(arguments,1);//虽然删除了包含arguments[1]的项，但是返回的正好是删除的项  
 props.forEach(**function**(p){ //遍历属性值,每个属性都会遍历这个函数  
 //忽略不可配置的属性  
 **if**(!Object.getOwnPropertyDescriptor(o,p).configurable) **return**;  
 Object.defineProperty(o,p,{  
 writable:**false**,  
 configurable:**false** });  
 });  
  
 **return** o;  
}  
  
//将这些属性设置为不可枚举的和可配置的  
**function** hideProps(o){  
 **var** props = (arguments.length == 1) ? Object.getOwnPropertyNames(o)  
 : Array.prototype.splice.call(arguments,1);  
  
 props.forEach(**function**(p){  
 //忽略不可配置的属性  
 **if**(!Object.getOwnPropertyDescriptor(o,p).configurable) **return**;  
 Object.defineProperty(o,p,{  
 enumerable:**false** });  
 });  
  
 **return** o;  
}  
  
//Object.defineProperty()和Object.defineProperties()两个函数可以用来创建新属性  
//也可以修改已有的属性，创建新属性时默认的属特性值都是false  
  
**function** Range(**from**,to){  
 **this**.from = **from**;  
 **this**.to = to;  
 freezeProps(**this**); //将这个对象的所有属性设置为不可变  
}  
  
**var** r = **new** Range(1,7);  
write(r.from); //1  
  
r.from = 2;  
write(r.from); //1 因为属性不可写  
  
write(Object.getOwnPropertyDescriptor(r,'from').writable); //false  
  
write(r.to); //7  
r.to = 10000;  
write(r.to); //7  
  
//使用不可枚举属性定义原型  
Range.prototype = hideProps({  
 constructor:Range,  
 sayFrom:**function**(){  
 write(**this**.from);  
 }  
});

/\*封装对象的状态\*/  
  
  
//私有状态  
  
//在经典的面向对象的编程中，经常需要将对象的某个状态封装或隐藏在对象内  
//只有通过对象的方法才能访问这些状态，对外只暴露一些重要的状态变量可以直接读写  
  
//允许声明类的‘私有’实例字段，这些私有实例字段只能被类的实例方法访问，且在类的外部是不可见的  
  
//我们可以通过将变量(或参数) 闭包 在一个构造函数内来模拟实现私有实例字段  
  
//在构造函数内定义函数（函数可以访问构造函数内部的需要私有的参数和变量）  
//并将这个函数赋值给新创建的对象的属性  
  
**function** Range(**from**,to){  
 //this.from = from; //相当于公有属性，外部可以访问  
 //this.to = to;  
  
 //定义存取器函数来返回值  
 //这些值都保存在闭包中  
 **this**.from = **function**(){  
 **return from**; //私有属性，只有访问对象的方法from()时才能访问它，对外不可见  
 };  
  
 **this**.to = **function**(){  
 **return** to;  
 };  
}  
  
  
//这让from和to字段看起来是不可修改的,而this.from和this.to是可以修改的  
//这里让from和to这些私有字段只读不可写  
  
**var** r = **new** Range(100,200);  
write(r.from()); //100  
write(r.to()); //200  
  
//当然也并不是不可写的,只是需要如下这么做  
r.from = **function**() {**return** 1000};  
write(r.from()); //1000  
//如上所示，构造函数中的变量和参数可以用作它创建对象的私有状态  
//缺点是可以重写这些存取器方法  
//如上面的 r.from = function() {return 1000};  
  
//使用getter和setter方法来改进  
  
**function** Range(**from**,to){  
 **if**(**from** > to) **return**;  
  
 //定义存取器的方法  
 **function** getFrom(){  
 **return from**;  
 }  
 **function** getTo(){  
 **return** to;  
 }  
 **function** setFrom(f){  
 **if**(f <= to) **from** = f;  
 }  
 **function** setTo(t){  
 **if**(t >= **from**) to = t;  
 }  
  
 //使用取值器的属性设置为可枚举，不可配置（防止重写方法）  
 Object.defineProperties(**this**,{  
 **from**:{**get**:getFrom, **set**:setFrom, enumerable:**true**, configurable: **false**},  
 to:{**get**:getTo, **set**:setTo, enumerable:**true**, configurable: **false**}  
 });  
}  
  
  
**var** r = **new** Range(1,7);  
  
write(r.from); //1  
write(r.to); //7  
  
r.from = 2;  
write(r.from); //2  
  
r.from = 88; //大于r.to  
write(r.from); //2 仍然不会变  
  
  
write(r.from); //2

//需要注意的是这种做法造成了更多的系统开销，使用闭包来封装类的状态的类一定  
//比不适用封装的状态变量的等价类运行速度更慢，占用更多的内存

/\*防止类的扩展\*/  
  
//和freezeProps()一样，可以通过Object.freeze()来把对象的属性设置为只读的和不可配置的  
  
**var** o = {x:1};  
Object.freeze(o); //冻结对象o,只读不可配置 不可写 不可扩展  
o.x = 100;  
write(o.x); //1 并不能改变o.x的属性值 不可写  
**delete** o.x;  
write(o.x); //1 由于不可配置，所以不能删除属性  
  
//Object.preventExtensions(o); //防止对象的扩展  
o.z = 1;  
write(o.z); //undefined 当然也是不可扩展的  
  
**var** obj = {x:11};  
Object.seal(obj); //使对象不可扩展，不可配置， 但是可写  
  
obj.y = 1;  
write(obj.y); //undefined 不可扩展  
  
**delete** obj.x;  
write(obj.x); //11 不可删除 因为不可配置  
  
obj.x = 111;  
write(obj.x); //111 但是可写

## 7.9 模块

/\*模块\*/  
  
//例如CommonJS使用的require()函数  
//不同的模块必须避免修改全局执行上下文  
//所以模块应当尽可能少的定义全局标识，理想状况是所有的模块都不应当定义超过一个全局标识  
  
/\*用作命名空间的对象\*/  
  
//在模块创建过程中避免污染全局变量的一种方法是使用一个对象作为命名空间  
//它将函数和值作为命名空间对象属性存储起来，而不是定义全局函数和变量  
//例如本书p115中Person构造函数,sayInfo存储为Person.prototype的属性，这些方法不是全局的  
  
**var** father = {}; //命名空间  
  
father.Father = **function**(name,age){ //构造函数  
 **this**.name = name;  
 **this**.age = age;  
};  
  
  
  
**var** F = father.Father; //导入到另外一个文件的全局命名空间中  
**var** f = **new** F('victor',23);  
write(f.name); //victor  
write(f.age); //23

//模块对外导出一些共用API,这些API是提供给其他程序员使用的,包括函数，类，属性和方法  
//但是模块的实现往往需要一些辅助函数和方法  
//这些函数和方法并不需要在函数外部可见  
//比如P116页的Person.\_minAge私有字段  
  
//可以将模块定义在某个函数的内部来实现  
//函数的作用域  
//在函数中声明的变量在整个函数体内都是可见的，包括嵌套的函数中  
//在函数的外部不可见  
  
//块级作用域  
(**function**(){  
 //模块代码  
})();  
  
  
//声明全局变量Person,使用一个函数的返回值给它赋值  
//函数定义后立即执行  
//返回值赋值给Person  
//注意它是一个函数表达式，因此函数'invocation'并没有创建全局变量  
  
**var** Person = (**function** invocation(){//第一行代码  
  
 **function** Person(name,age){ //这个构造函数是一个局部变量  
 **this**.name = name;  
 **this**.age = age;  
 }  
  
 //原型方法  
 Person.prototype.sayInfo = **function**(){  
 Info(); //调用了这个辅助函数  
 write(**this**.name + '-' + **this**.age);  
 };  
  
 //辅助函数和变量  
 //不属于模块的共有API,隐藏在这个函数的作用域内  
 //因此我们不必将它们定义为Person的属性,P116页的类方法isAdult()就是定义为Person的属性  
 **function** Info(){  
 write(str);  
 }  
 **var** str = '这是一个辅助函数';  
  
 //这个模块的共有API是Person构造函数  
 //我们需要把这个函数从私有命名空间中导出来  
 //以便在外部可以使用它，我们通过返回构造函数来导出它  
 //它变成第一行代码所指的表达式的值  
 **return** Person;  
}()); //立即执行  
  
  
  
**var** p = **new** Person('victor',25);  
write(p.name); //victor  
write(p.age); //25  
p.sayInfo(); //这是一个辅助函数 victor-25  
  
//一旦将模块代码封装进一个函数，就需要一些方法导出其共用API  
//以便在模块函数的外部调用它们  
//上面的例子中模块函数返回构造函数  
//这个构造函数随后赋值给一个全局变量  
  
//将值返回表明API已经导出在函数作用域之外  
  
//上面只是一个类，如果包含多个类等，则可以返回命名空间对象  
  
  
//创建一个全局变量用来存放集合相关的模块  
**var** collections;  
  
**if**(!collections) {  
 collections = {};  
}  
  
//定义Family模块  
collections.family = (**function** namespace(){  
 //这里定义多种’集合‘类，使用局部变量和函数  
 //例如Person类  
 **function** Person(name,age){  
 **this**.name = name;  
 **this**.age = age;  
 }  
  
 //Person类的子类Father类  
 //使用Function.prototype.extend()方法来定义子类  
 **var** Father = Person.extend(  
 **function** Father(job) {  
 **this**.job = job;  
 }//constructor 子类的构造函数  
 );  
  
  
 //Mother类  
 //var Mother =  
  
 //省略很多其他类  
 //以及这些类的原型对象方法以及辅助函数和变量  
  
  
  
 //返回的是一个对象  
 //这个对象叫做命名空间对象  
 //这个对象的属性都是以上定义的类  
 **return** {  
 Perosn: Person,  
 Father: Father  
 //后面还有许多类似的类  
 };  
}()); //立即执行  
  
  
  
//另外一种类似的技术是将模块函数当做构造函数，通过new来调用  
**var** collections;  
  
**if**(!collections) {  
 collections = {};  
}  
  
  
//定义Family模块  
  
  
**var** a = (**new function** Person(name){ //先是一个立即执行的构造函数，然后使用new  
 **this**.name = name;  
}('victor'));  
  
write(a.name); //victor  
  
  
  
*/\*\*  
 \* new function namespance(){}()  
 \*/*collections.family = (**new function** namespace(){  
 //这里定义多种’集合‘类，使用局部变量和函数  
 //例如Person类  
 **function** Person(name,age){  
 **this**.name = name;  
 **this**.age = age;  
 }  
  
 //Person类的子类Father类  
 //使用Function.prototype.extend()方法来定义子类  
 **var** Father = Person.extend(  
 **function** Father(job) {  
 **this**.job = job;  
 }//constructor 子类的构造函数  
 );  
  
  
 //Mother类  
 //var Mother =  
  
 //省略很多其他类  
 //以及这些类的原型对象方法以及辅助函数和变量  
  
  
  
 */\*\*返回的是一个对象  
 //这个对象叫做命名空间对象  
 //这个对象的属性都是以上定义的类  
 return {  
 Person: Person,  
 Father: Father  
 //后面还有许多类似的类  
 };\*/* //不要return  
 **this**.Person = Person;  
 **this**.Father = Father;  
  
}()); //立即执行  
  
  
//另外一种替代的方法  
  
**var** collections;  
  
**if**(!collections) {  
 collections = {};  
}  
  
collections.family = {};  
  
(**function** namespace(){  
 //这里定义多种’集合‘类，使用局部变量和函数  
 //例如Person类  
 **function** Person(name,age){  
 **this**.name = name;  
 **this**.age = age;  
 }  
  
 //Person类的子类Father类  
 //使用Function.prototype.extend()方法来定义子类  
 **var** Father = Person.extend(  
 **function** Father(job) {  
 **this**.job = job;  
 }//constructor 子类的构造函数  
 );  
  
  
 //Mother类  
 //var Mother =  
  
 //省略很多其他类  
 //以及这些类的原型对象方法以及辅助函数和变量  
  
 //这样就不需要return了  
 className.family.Father = Father;  
 collections.family.Person = Person;  
  
}()); //立即执行

# 第八章 正则表达式

## 8.1 正则表达式的定义

/\*正则表达式的定义\*/  
  
**var**　pattern = /s$/; //RegExp对象  
**var** pattern\_copy = **new** RegExp("s$"); //与上面一样

# 第九章 JavaScript的子集和扩展

## 9.1 常量和局部变量

/\*常量和局部变量\*/  
  
//const 常量  
**const** pi = 3.14;  
pi = 3.15;  
write(pi); //3.14  
  
//const pi = 4; //重新定义常量会报错Uncaught TypeError: Identifier 'pi' has already been declared  
  
**var** pi = 4; //也会报错 Uncaught TypeError: Identifier 'pi' has already been declared  
  
//var和const行为非常类似，都会提升声明  
  
//let 局部变量  
  
**function** oddSum(n){  
  
  
 **var** total;  
 **var** result = [];  
  
 **for**(**let** x=1; x<=n; x++){ //let只在这个for循环内有定义，不会提升声明  
 **let** odd = 2\*x -1;  
 total += odd;  
 result.push(total);  
 }  
  
 write(x); //undefined  
  
 **return** result;  
  
}  
  
write(oddSum(5));

# 第十章 服务器端JavaScript

# 第十一章 Web浏览器中的JavaScript

## 11.1 客户端JavaScript

/\*客户端的JavaScript\*/  
  
//window.location = "http://www.baidu.com"; //跳转到百度首页  
  
//给定一段时间后触发一个回调  
  
setTimeout(**function**(){  
 alert(100);  
},1000); //1s后执行  
  
  
  
**var** a = 100;  
setTimeout(**function**(){  
 alert(a); //1100 回调函数中的后执行  
},0);  
  
a = 1100; //先执行

## 11.2 在HTML里嵌入JavaScript

/\*<script>标签\*/

<body>  
 <h1>Digital Time</h1>  
 <p id="clock"></p>  
<script src="lib.js" type="text/javascript"></script>  
<script src="test.js" type="text/javascript"></script>  
  
<script>  
 **function** displayTime(){  
 **var** clk = document.getElementById('clock');  
 **var** now = **new** Date();  
 clk.innerHTML = now.toLocaleTimeString();  
 setTimeout(displayTime,1000);  
 }  
 window.onload = displayTime;  
</script>

## 11.3 JavaScript程序的执行

<body>  
 <h1>Digital Time</h1>  
 <p id="clock"></p>  
  
 *<!--document.write 是同步载入的程序  
 在HTML解析器遇到<script>元素时，它默认必须先执行脚本,然后恢复文档的解析和渲染  
 这对于内联脚本没什么问题,即在HTML页面中使用<script>标签  
 但是如果源代码是一个由src属性指定的外部文件  
 这意味着脚本后面的文档部分在下载和执行脚本之间，都不会出现在浏览器中  
 意思是文档在文本内容已经载入，但是未被浏览器引擎解析为DOM树  
 而DOM树的生产是受JavaScript代码执行影响的，JavaScript代码会'阻塞'页面UI的渲染  
   
 脚本的执行在默认情况下是同步和阻塞的  
 <script>标签可以有defer和async属性  
 defer和async属性都像在告诉浏览器连接进来的脚本不会使用document.write  
 也不会生成文档内容  
   
 defer属性使得浏览器延迟脚本的执行，直到文档的载入和解析完成才可以操作  
 async属性使得浏览器可以尽快的执行脚本，而不用在下载脚本时阻塞文档解析  
 如果同时拥有两个属性，浏览器会遵从async属性并忽略defer属性  
 异步脚本可能会无序执行  
 -->* <script src="lib.js" type="text/javascript"></script>  
 <script src="test.js" type="text/javascript"></script>  
</body>  
</html>

*<!--客户端JavaScript时间线  
 1.Web浏览器创建Document对象,并且开始解析Web页面，  
 解析HTML元素和它们的文本内容后添加到Element对象和Text节点到文档中  
 这个阶段，document.readystate属性的值是'loading'  
 2.当HTML解析器遇到没有async和defer属性的<script>元素时，它把这些元素添加  
 到文档中，然后执行行内或外部脚本，这些脚本会同步执行，并且在脚本下载和执行时  
 解析器会暂停，这样脚本就可以用document.write()来把文本插入到输入流中  
 解析器恢复时这些文本会成为文档的一部分.  
 3.当解析器遇到设置了async属性的<script>元素时，开始下载脚本文本，并继续解析文档  
 脚本会在它下载完成后尽快执行，但是解析器没有停下来等它下载，异步脚本禁止使用  
 document.write()方法  
 4.文档解析完，document.readyState属性变成了'interactive'  
 5.按脚本在文档中出现的顺序执行所有有defer属性的脚本，异步脚本可能也会在这个时间执行  
 延迟脚本能访问完整的文档树，禁止使用document.write()方法  
 6.浏览器在Document对象上触发了DOMContentLoaded事件，这标志着程序执行从同步脚本执行  
 阶段转换到了异步时间驱动阶段，这时异步脚本能可能没有执行完成  
 7.文档已经完全解析完成，浏览器可能还在等待其他内容载入，比如图片，当所有的内容载入完成  
 并且所有的异步脚本完成载入和执行，document.readState属性改变成'complete'  
 Web浏览器触发Window对象上的load事件  
 8.之后就是调用异步事件,以异步响应用户输入事件，网络事件，计时器过期等  
-->*

## 11.4 兼容性和互用性

## 11.5 可访问性

## 11.6 同源策略

*<!--同源策略  
   
 文档的来源包含协议、主机、以及载入文档的URL端口  
 1.从不同的服务器（主机）载入的文档具有不同的来源  
 2.通过同一主机的不同端口载入的文档具有不同的来源  
 3.使用http协议载入的文档和https协议载入的文档具有不同的来源，即使他们来自同一个服务器  
   
 脚本本身的来源和同源策略并不相关，相关的是脚本所嵌入的文档的来源  
   
 解决方法  
 1.多个子域的大站点：home.example.com和orders.example.com多域名站点  
 使用Document对象的domain属性，默认情况下domain存放的载入文档的服务器的主机名  
 如果两个窗口包含的脚本把domain设置成了相同的值，那么这两个窗口就不再受同源策略的约束  
 它们可以互相读取对方的属性 例如以上两个域名载入的文档中的脚本可以把它们的document.domain  
 属性都设置成example.com，这样一来文档就有了同源性，可以互相读取属性  
 2.跨域资源共享  
 服务器用头信息显示的列出源，或使用通配符来匹配所有的源并允许由任何地址请求文件  
 例如用新的'Origin:'请求头和新的响应头来扩展HTTP，允许跨域HTTP请求  
 这样XMLHttpRequest就不会被同源策略所限制了  
 3.跨文档消息  
 允许来自一个文档的脚本可以传递文本消息到另一个文档里的脚本，而不管脚本里的资源是否不同  
 调用Window对象上的postMessage()方法,可以异步传递消息事件可以使用onmessage时间句处理程序函数来处理它）  
 到窗口的文档里  
   
-->*

# 第十二章 Window对象

## 12.1 计时器

/\*客户端的JavaScript\*/  
  
/\*Window对象\*/  
  
//方法： 计时器  
  
//setTimeout() setInterval() 返回一个值,传给clearInterval()去掉后续函数的调用  
  
  
**function** invoke(f,start,interval,end){  
 **if**(!start){  
 start = 0;  
 }  
  
 **if**(arguments.length <= 2){  
 setTimeout(f,start); //如果只定义了两个参数或者小于两个，则若干毫秒后调动f  
 } **else**{  
 setTimeout(repeat,start); //若干毫秒后调用repeat  
  
 **function** repeat(){  
 **var** h = setInterval(f,interval); //循环调用f  
  
 **if**(end){  
 setTimeout(**function**(){  
 clearInterval(h);  
 },end);  
 }  
  
 }  
  
 }  
}  
  
  
**var** a = 3;  
  
**function** f(){  
 write(a);  
}  
  
invoke(f,0); //7  
invoke(f); //7  
//invoke(f,0,1000); //7  
  
a = 7;  
  
invoke(f); //以上结果全变成100了  
  
a = 100;  
  
  
setTimeout('alert(a)',1000); //100 字符串会在指定的超时时间或间隔之后进行求值  
//相当于执行eval()  
  
//以上还用0毫秒超时时间来调用setTimeout()  
//指定的函数并不会立刻执行，相反，会把它放到队列中，  
//等到前面出于等待状态的时间处理程序全部  
//执行完成后，才会立即调用它  
  
  
//例如  
  
**var** o = 1;  
  
setTimeout(**function**(){  
 write(o);  
 write(111); //1000 第四个执行  
},1000);  
  
  
setTimeout(**function**(){  
 write(o);  
 write(222); //1000 第一个执行  
},0);  
  
  
setTimeout(**function**(){  
 write(o);  
 write(333); //1000 第五个执行  
},1000);  
  
  
setTimeout(**function**(){  
 write(o);  
 write(444); //1000 第二个执行  
},0);  
  
  
setTimeout(**function**(){  
 write(o);  
 write(555); //1000 第三个执行  
},500);  
  
  
o = 1000;

## 12.2 Window.location

/\*解析URL\*/  
write(window.location === document.location); //true  
  
//window对象的location属性引用的是Location对象  
//表示该窗口中当前显示的文档的URL  
write(window.location); //http://localhost:63342/JS%E6%B5%8B%E8%AF%95%E4%BB%A3%E7%A0%81/test.html  
write(window.location.href); //http://localhost:63342/JS%E6%B5%8B%E8%AF%95%E4%BB%A3%E7%A0%81/test.html  
  
//location对象的href属性是一个字符串，包含URL的完整文本  
//Location对象的toString()方法返回href属性的值，  
//在会隐式调用toString()的情况下，可以使用location代替location.href  
  
write(window.location.protocol); //http:  
write(window.location.host); //localhost:63342  
write(window.location.hostname); //localhost  
write(window.location.port); //63342  
write(window.location.pathname); // /JS%E6%B5%8B%E8%AF%95%E4%BB%A3%E7%A0%81/test.html  
write(window.location.search); // '' 返回？之后的url  
write(window.location.hash); // ''  
  
//解析url：var str = 'http://s.weibo.com/weibo/Aralic?topnav=1&wvr=6'  
//获得参数名和参数值：  
  
**var** str = 'http://s.weibo.com/weibo/Aralic?topnav=1&wvr=6';  
  
  
  
**function** urlArgs(str){  
 **var** args = {};  
 **var** arr = [];  
  
 //var query = location.search.substring(1); //查找字符串并去掉'?'  
 **if**(str) {  
 **if** (str.indexOf('?') === -1) {  
 **return**;  
 } **else** {  
 arr = str.substring(str.indexOf("?") + 1).split('&');  
  
 **for** (**var** i = 0, len = arr.length; i < len; i++) {  
 args[(arr[i].split('='))[0]] = (arr[i].split('='))[1];  
 }  
 }  
 }  
  
 **return** args;  
}  
  
  
**var** args = urlArgs(str);  
  
  
  
  
//以上是自己写的，改进方法  
**function** urlArgs\_copy(str){  
  
 **var** args = {};  
 **var** arr = [];  
  
 **var** l = str.indexOf('?');  
  
 **if**(l == -1){  
 **return** {}; //返回空对象  
 } **else**{  
 arr = str.substring(l+1).split('&'); //没有&应该也可以  
  
 **for**(**var** i=0,len = arr.length; i<len; i++){  
 **if**(arr[i].indexOf('=') == -1){  
 **continue**;  
 } **else**{  
 args[(arr[i].split('='))[0]] = (arr[i].split('='))[1];  
 }  
 }  
 }  
  
 **return** args;  
}  
  
  
  
str = 'http://s.weibo.com/weibo/Aralic?topnav=1&wvr=6';  
args = urlArgs\_copy(str);  
write(JSON.stringify(args)); //{"topnav":"1","wvr":"6"}  
  
str = 'http://s.weibo.com/weibo/Aralictopnav=1&wvr=6';  
args = urlArgs\_copy(str);  
write(JSON.stringify(args));//{}  
  
str = 'http://s.weibo.com/weibo/Aralic?topnav=1';  
args = urlArgs\_copy(str);  
write(JSON.stringify(args)); //{"topnav":"1"}  
  
  
str = 'http://s.weibo.com/weibo/Aralic?topnav1&wvr=6';  
args = urlArgs\_copy(str);  
write(JSON.stringify(args)); //{"wvr":"6"}  
  
  
  
//如果获取浏览器本身的location对象的属性值  
location.search = "?page=" + 1;  
  
**function** locationUrlArgs(){  
 **var** args = {};  
 **var** arr = [];  
  
 **var** query = location.search.substring(1); //查询url串并去掉‘？’  
  
 arr = query.split('&');  
  
 **for**(**var** i=0,len=arr.length; i<len; i++){  
 **if**(arr[i].indexOf('=') === -1){  
 **continue**;  
 } **else**{  
 args[(arr[i].split('=')[0])] = (arr[i].split('='))[1];  
 }  
 }  
  
 **return** args;  
}  
  
**var** args = locationUrlArgs();  
write(JSON.stringify(args)); //{"page":"1"}

//substring  
**var** s = 'hello world';  
write(s.substring(4)); //o world 返回包含s[4]在内的所有项  
//indexOf  
write(s.indexOf('33')); //-1  
write(s.indexOf('w')); //6 因为s[6] = w  
write(s[6]); //w  
//如果要得到w之后的字符串  
write(s.substring(s.indexOf('w') +1)); //orld  
//如果要得到包含w及之后的字符串  
write(s.substring(s.indexOf('w'))); //world

/\*载入新的文档\*/  
  
  
//location.replace 载入新文档，会从浏览历史中把当前文档删除  
  
//location.replace('test\_copy.html'); //此时跳转到了此页面  
  
//传入的是相对url,相对于当前所在文档的的目录，就像应用超链接  
  
//更传统的方法是使用location属性  
  
//location = 'test.html'; //可以是相对的也可以绝对的URL  
  
location.search = "?page=" + 1;  
  
**function** locationUrlArgs(){  
 **var** args = {};  
 **var** arr = [];  
  
 **var** query = location.search.substring(1); //查询url串并去掉‘？’  
  
 arr = query.split('&');  
  
 **for**(**var** i=0,len=arr.length; i<len; i++){  
 **if**(arr[i].indexOf('=') === -1){  
 **continue**;  
 } **else**{  
 args[(arr[i].split('=')[0])] = (arr[i].split('='))[1];  
 }  
 }  
  
 **return** args;  
}  
  
**var** args = locationUrlArgs();  
write(JSON.stringify(args)); //{"page":"1"}

## 12.3 Window.history

/\*浏览历史\*/  
//Window对象的history属性引用的是该窗口的History对象  
//history.back() 后退  
//history.forward() 前进

## 12.4 Window.Navigator

## 12.5 Window.Screen

/\*屏幕\*/  
  
write(window.screen.width); //1920 实际上检测是显示器屏幕大小  
write(window.screen.height); //1080  
  
write(window.screen.availWidth); //1920 实际可用大小  
write(window.screen.availHeight); //1040

## 12.6 对话框

/\*对话框\*/  
//confirm prompt alert  
//confirm和prompt会产生阻塞 弹出对话框前代码会停止运行  
//alert 也会阻塞，但并不总是这样  
//showModalDialog 模态对话框

## 12.7 作为Window对象属性的文档元素

/\*作为Window对象属性的文档元素\*/  
  
*/\*\** <p *id="p1">this is first page!*</p>  
 <p *id="p2">this is first page!*</p>  
 *\*/*//如果HTML文档中用id属性来为元素命名  
//如果window对象没有此名字的属性  
//window对象会赋予一个属性，它的名字是id属性的值  
//这个属性的值指向标识文档元素的HTMLElement对象  
  
write(window.p1); //[object HTMLParagraphElement]  
write(window.p1.outerHTML); //this is first page!  
  
//在HTML文档中使用id属性会成为可以被脚本访问的全局对象  
//但是如果window对象有此属性，就会覆盖当前的id值对应的全局变量  
  
**var** p2 = 1;  
write(window.p2); //1 注意不是id为p2的HTMLElement对象了 属性被覆盖了  
  
  
//比如id的名字为 history navigator location screen的元素，就不会以全局变量的形式出现  
//因为这些已经存在window.history等属性了  
  
//最好不要这样引用  
//因为window的默认属性会隐藏和覆盖带id的属性  
//最好使用getElementById来显示查找元素

## 12.8 多窗口和窗体

/\*多窗口和窗体\*/  
/\*打开和关闭窗口\*/  
//window.open 参数  
//1. 要在新窗口中显示的URL,如果省略，会使用空页面URL about:blank  
//2. 新打开窗口的名字，如果是已存在的窗口名字，会跳转到那个窗口  
//如果省略名字，会使用\_blank打开一个新的、未命名的窗口  
  
//窗口名字  
//不仅允许open方法引用已存在的窗口，还可以作为<a> <from>元素上  
//HTML target属性的值 用来表示引用的文档 target属性值可以为\_blank \_top \_parent  
//write(window.name); //''  
  
//3. 以逗号分隔的列表，包含大小和各种属性 用以表明新窗口如何打开  
  
//可能默认的情况下会被浏览器拦截弹出，因为广告商滥用这种技术  
//var w = window.open('test\_copy.html','copy',"width=400,height=500");  
  
//open的返回值是代表新创建的窗口的window对象  
//w.alert("111"); //新窗口会弹alert  
  
//w.location = 'http:www.baodu.com'; //指向新的网址  
// http://localhost:63342/JS%E6%B5%8B%E8%AF%95%E4%BB%A3%E7%A0%81/www.baodu.com  
//好像不行  
  
/\*setTimeout(function(){  
 w.close();  
},5000); //5s钟后关闭新窗口  
  
//window.close(); //关闭当前窗口 此时新窗口就关闭不了了，因为当前窗口已经退出  
//即setTimeout已经不起作用了  
  
setTimeout(function(){  
 window.close();  
},6000); //5s钟后关闭当前窗口  
\*/  
  
window.close(); //关闭了自己  
  
  
//var w = window.open('test\_copy.html'); //此时省略了窗口名字 第二个参数  
//使用了指定的名字\_blank打开一个新的未命名的窗口  
  
//var w = window.open(); ///about:blank 打开了一个空页面  
  
  
//var w = window.open('test\_copy.html','copy');  
  
**var** w = window.open('test\_copy.html');  
  
write(w.opener !== **null**); //true  
  
write(w.opener === window); //true  
//该属性指向的是打开新窗口的当前窗口  
  
write(w.open().opener === w); //true w窗口中又打开了一个新窗口，那么该新窗口的opener属性指向的是打开该窗口所在脚本的w窗口

//这样的话两个窗口之间就可以互相引用对方的window全局对象

# 第十三章 脚本化文档DOM

## 13.1 DOM概览

/\*脚本化文档\*/  
  
//每一个Web浏览器窗口、标签页和框架由一个Window对象所表示，每个Window对象  
//有一个document属性引用了Document对象  
//Document对象表示了窗口的内容  
//Document对象是一个巨大的API中的核心对象，叫做文档对象模型 Document Object Model DOM  
//它代表和操作文档的内容

<!DOCTYPE html>  
<html>  
<head lang="en">  
 <title>this is title</title>  
</head>  
  
<body>  
 <h1> this is h1</h1>  
 <p> this <i>this is i</i> is p</p>  
 <script src="lib.js" type="text/javascript"></script>  
 <script src="test.js" type="text/javascript"></script>  
</body>  
</html>



以上HTML页面的DOM表示树状图

一个节点之上的直接节点是其父节点，例如p就是i节点的父节点

相同父节点下的子节点还是兄弟节点，比如h1 p 和script

Body后面的所有节点都是body的后代节点

上图中的每一个方框就是文档的一个节点，表示一个Node对象

Document节点：树形的根部是Document节点，它代表整个文档

Element节点：代表HTML元素的节点是Element节点 如head title h1 p body script body i

Text节点：代表文本的节点是Text节点 如”this is title”

Document Element Text是Node的子类 Document和Element是两个重要的DOM类，文章大部分内容都是讲述它们的属性和方法



通用的Document和Element类型与HTMLDocument和HTMLElement类型之间是有严格的区别的。Document类型代表一个HTML和XML文档，Element类型代表该文档中的一个元素。HTMLDocument和HTMLElement子类只是针对HTML文档和元素。

## 13.2 选取文档元素

//使用全局变量的document来引用Document对象  
//为了操作文档中的元素，必须通过某种方式获取这些引用文档元素的Element对象  
*/\*\*  
 \* 1.用指定的id属性  
 \* 2.用指定的name属性  
 \* 3.用指定的标签名字  
 \* 4.用指定的Css类  
 \* 5.匹配指定的Css选择器  
 \*/*/\*通过id选取元素\*/ //获取单个时是 get Element By Element没有s  
  
//引用Document对象的getElementById('idName') 方法  
  
//查找多个id  
//接收一个任意多的字符串参数  
//返回一个对象，它把这些id映射到对应Element对象  
  
**function** getElements(/\*id1,id2,id3...\*/){  
 **var** elements = {};  
 **var** id;  
 **var** elt;  
  
 **for**(**var** i=0,len=arguments.length;i<len; i++){  
 id = arguments[i];  
 elt = document.getElementById(id);  
 **if**(elt){  
 elements[id] = elt;  
 }  
 }  
 **return** elements;  
}  
  
  
/\*通过名字获取元素\*/ //注意获取多个时是get Elements By 有s  
//name属性和id不同，可以不是唯一的  
  
//document.getElementsByName('Name')  
//Document类型代表一个HTML和XML文档，  
//Element类型代表该文档中的一个元素。  
//HTMLDocument和HTMLElement子类只是针对HTML文档和元素。  
  
//需要注意的是getElementsByName方法定义在HTMLDocument类中，而不在Document类中  
//所以它只针对HTML文档可用，在XML文档不可用  
//它返回一个NodeList对象，类似于包含若干Element对象的只读数组  
  
//尽量不要讲同样的字符串同时用作名字和id  
//在id中，该方法也返回id属性匹配指定值的元素  
  
/\*通过标签名选取元素\*/  
//document.getElementsByTagName('tagName')  
//返回的是包含所有<tagName>元素的只读的类数组对象  
//var firstSpan = document.getElementsByTagName('p')[0];  
  
//document.getElementsByTagName('\*') 获取所有的元素  
**var** allElements = document.getElementsByTagName('\*');  
  
//Element类也定义了getElementByTagName()方法  
//其原理和Document版本是一样的  
  
*/\*\*  
 \** <h1> *this is h1*</h1>  
 <p> *this* <i>*this is i1*</i> *is* <i>*this is i2*</i> *p*</p>  
 *\*/*//获取p元素的所有后代元素i  
**var** p = document.getElementsByTagName('p'); //返回的是一个类数组对象 调用了Document类的方法  
**var** i = p[0].getElementsByTagName('i'); //调用了Element类的getElementByTagName方法  
//但是在浏览器里调试发现得到的是HTMLCollection类的数组对象  
/\*通过CSS类选取元素\*/  
//返回值时一个NodeList对象  
**var** warnings = document.getElementsByClassName('warning');  
**var** doubleClass = document.getElementsByClassName('warning danger');  
  
/\*通过CSS类选择器选取元素\*/  
//querySelectorAll() querySelector()  
  
//这两个方法在Element节点中也有定义

## 13.3 文档结构和遍历

/\*文档结构和遍历\*/  
  
  
/\*作为节点数的文档\*/  
//Document对象、它的Element对象和文档中表示文本的Text对象都是Node对象  
//Node对象的属性  
*/\*\*  
 \* 1. parentNode  
 \* 该节点的父节点，需要注意的是Document对象是没有父节点的，所以是null  
 \* 2. childNodes  
 \* 只读的类数组对象(NodeList对象)，它是该节点的子节点的实时表示  
 \* 3. firstChild lastChild  
 \* 该节点的子节点中的第一个和最后一个，如果该节点没有子节点，则是null  
 \* 4. nextSibling previoursSibling  
 \* 该节点的兄弟节点中的前一个和后一个  
 \* 5. nodeType  
 \* 该节点的类型  
 \* 1 代表Element节点  
 \* 3 代表Text节点  
 \* 8 代表Comment节点  
 \* 9 代表Document节点  
 \* 11 代表DocumentFragment节点  
 \*  
 \* 6. nodeValue  
 \* Text节点或Comment节点的文本内容  
 \*  
 \* 7. nodeName  
 \* 元素的标签名，以大写形式表示  
 \*/  
  
  
/\*\*  
 \*  
 \** <!DOCTYPE *html>* <html>  
 <head *lang="en">* <title>*this is title*</title>  
 </head>  
  
 <body>  
 <h1 *class="warning danger"> this is h1*</h1>  
 <p> *this* <i *class="innerI">this is i1*</i> *is* <i>*this is i2*</i> *p*</p>  
 <script *src="lib.js" type="text/javascript">*</script>  
 <script *src="test.js" type="text/javascript">*</script>  
 </body>  
 </html>  
 *\*/***var** html = document.childNodes[1]; //html  
**var** head = document.childNodes[1].childNodes[0]; //head  
**var** Head = document.childNodes[1].firstChild; //head  
**var** html = document.childNodes[1].firstChild.parentNode; //html  
**var** text = document.childNodes[1].childNodes[1]; //text 需要注意的是如果有空格，子节点中就会有Text节点  
//理想我们需要的是body  
  
**var** HTML = document.childNodes[1].nodeName; // 'HTML'  
**var** htmlNodeType = document.childNodes[1].nodeType; //1 代表Element节点  
  
/\*作为元素树的文档\*/  
//作为节点树的文档容易将焦点集中到不是元素的文本（元素之间的空白等）上  
//如果想把焦点集中到文档中的元素上时，我们可以使用另一个更有用的API  
//它将文档看作是Element对象树，忽略部分文档：Text和Comment节点  
  
//类似childNodes, Element对象的children属性只包含Element对象  
//注意Text和Comment节点没有children属性  
//所以Node.parentNode属性不可能返回Text和Comment节点  
//任何Element的parentNode总是另一个Element  
  
//Element对象的子属性和兄弟属性,类似于Node对象  
*/\*\*  
 \* 1.firstElementChild lastElementChild  
 \* 代表子Element对象  
 \*  
 \* 2.nextElementSibling previousElementSibling  
 \* 代表兄弟Element  
 \*  
 \* 3.childElementCount  
 \* 子元素的属性 和children.length值相等  
 \*  
 \*/*//可移植的文档遍历函数  
  
*/\*\*  
 \* 返回元素e的第n层祖先元素，如果不存在祖先元素或者不是Element  
 \* (例如Document或者DocumentFragment)则返回null  
 \* 如果n为0，则返回e本身，如果n为1，则返回其父元素  
 \* 如果n大于1，则返回e的祖先元素  
 \*/***function** parentElement(e,n){  
 **if**(n === undefined){  
 n = 1;  
 }  
 **while**(n-- && e){  
 e = e.parentNode;  
 }  
  
 **if**(!e || e.nodeType !==1) { //nodeType不为Element  
 **return null**;  
 }  
 **return** e;  
}  
  
  
*/\*\*  
 \* 返回e元素的第n个兄弟元素  
 \* 如果n为正，则返回后续的第n个兄弟元素  
 \* n为负，则返回前面的第n个兄弟元素  
 \* n为零，则返回e本身  
 \*/***function** siblingElement(e,n){  
 **while**(e && n!==0){   
 **if**(n > 0) { //则返回后续的第n个兄弟元素  
 **if**(e.nextElementSibling) { //Element对象  
 e = e.nextElementSibling;  
 } **else**{  
 **for**(e = e.nextSibling; e && e.nodeType !== 1; e=e.nextSibling);   
//Node对象的属性 并判断是不是Element对象   
//如果e.nodeType ！==1 即如果不是Element对象 则继续使 e=e.nextSibling  
//否则如果是Element,则因为找到了下一个兄弟节点是Element对象  
//所以退出循环  
 /\*空循环\*/  
 }  
 n--; //循环执行n遍  
 } **else**{ //返回前面的第n个兄弟元素  
 **if**(e.previousElementSibling){  
 e = e.previousElementSibling;  
 } **else**{  
 **for**(e = e.previousSibling; e && e.nodeType !==1; e=e.previousElementSibling);  
 }  
 n++;  
 }  
 }  
 **return** e;  
}  
  
  
*/\*\*  
 \* 返回元素e的第n个子元素，如果不存在则返回null  
 \* 负值代表从后往前数 0表示第一个子元素，而-1表示最后一个 -2表示倒数第二个  
 \*/***function** childElement(e,n){  
 **if**(e.children){  
 **if**(n < 0){  
 n += e.chileren.length;  
 }   
   
 **if**(n < 0) {  
 **return null**;  
 }  
   
 **return** e.children[n];  
 }  
   
   
 //如果e没有children数组  
   
 **if**(n >= 0){  
 **if**(e.firstElementChild) {  
 e = e.firstElementChild;  
 } **else**{  
 **for**(e = e.firstChild; e && e.nodeType !== 1; e = e.nexSibling);  
 }  
   
 **return** siblingElement(e,n); //返回第一个子元素的第n个兄弟元素  
 } **else**{ //n < 0   
 **if**(e.firstElementChild){  
 e = e.firstElementChild;  
 } **else**{  
 **for**(e = e.lastChild; e && e.nodeType !==1; e = e.previousSibling);  
 }  
   
 **return** siblingElement(e, n+1);   
 }  
}

## 13.4 属性

/\*属性\*/  
  
/\*HTML属性作为Element属性\*/  
  
  
//HTML元素由一个标签和一组称为属性（attribute）的名/值对组成  
//例如<a>标签的href属性  
//HTML元素的属性值在代表这些元素的HTMLElement对象的属性中是可用的  
//DOM中还定义了另外一些API来获取HTML属性值  
  
/\*HTML属性作为Element的属性\*/  
  
//表示HTML文档元素的HTMLElement对象定义了读写属性  
//它们映射了元素的HTML属性  
  
**var** a = document.getElementById('link');  
write(a.href);  
write(a.href); //http://www.baidu.com/  
write(a.id === 'link'); //true  
  
//当然也可以为form表单元素设置action和method属性等  
  
//HTML属性名不区分小大写，但是JS中则是小写，如果有不止一个单词，则使用的驼峰法命名的规则的  
//例如tabIndex  
//需要注意的是没有定义任何从元素中删除属性的方法 ,delete是无效的  
  
//获取和设置非标准的HTML属性  
//如上所示 HTMLElement和其子类型定义了一些属性，对应于元素的标准HTML属性  
//Element类型还定义了getAttribute()和setAttribute()方法来查询和设置非标准的HTML属性  
  
write(a.getAttribute("HREF")); //获取href属性 http://www.baidu.com  
  
a.setAttribute('href','http//:www.github.com');  
  
//该方法的属性名不区分大小写  
  
//hasAttribute() removeAttribute()  
  
  
/\*数据集属性\*/  
  
//可以使用以上两个方法来读和写非标准属性的值  
//但是为此付出的代价是文档将不再是合法有效的HTML  
//HTML5任意以‘data-’为前缀的小写属性名字都是合法的,将不会对其元素的表现产生任何影响  
//定义了一种标准的附加额外数据的方法  
//HTML5在Element对象上定义了dataset属性，该属性代表一个对象，它的各个属性对应于去掉  
//前缀的‘data-’属性  
//dataset.x 就是‘data-x’属性的值  
  
//dataset.jqueryTest就是 ‘data-jqury-test’属性的值  
  
  
*/\*\*  
 \** <h1 *id="para" data-title='this is' class="warning danger"> this is h1*</h1>  
 *\*  
 \*/***var** p = document.getElementById('para');  
**var** dataset = p.dataset;  
write(dataset.title); //this is  
  
/\*作为Attr节点的属性\*/  
  
//最后一种使用Element的属性的方法 Node类定义了attributes属性  
//针对非Element对象的任何节点，该属性为null  
//它代表元素的所有属性 同时attributes属性是只读的类数组对象  
  
**var** pAttr = document.getElementById('para').attributes[0]; //第一个属性id

## 13.5 元素的内容

/\*元素的内容\*/  
  
/\*作为HTML的元素内容\*/  
//读取Element的innerHTML属性作为字符串标记返回那个元素的内容  
//在元素上设置该属性调用了Web浏览器的解析器，用新字符串内容解析展现形式替换元素当前内容  
//由于是Element的的属性而非HTMLElement的属性 所以在XML元素也可以使用该属性  
//通常设置innerHTML效率非常高，因为Web浏览器很擅长解析HTML  
//但是对innerHTML属性用 '+=' 操作符重复追加一小段文本通常效率很低下  
//因为它们既要序列化也要解析  
//HTML5标准说innerHTML应该在Document节点以及Element节点上工作正常，但这还未被普遍支持  
  
//查询outerHTML返回的HTML或XML标记的字符串包含被查询元素的开头和结尾标签  
//当设置outerHTML时，元素本身被新的内容所替换  
//只有Element节点定义了outerHTML属性，Document节点则没有  
  
*/\*\*  
 \** <p *id="para">  
 this* <i *class="innerI">  
 this is i1* </i>  
 *is* <i>  
 *this is i2* </i>  
 *p* </p>  
 *\*  
 \*/***var** p = document.getElementById('para');  
write(p.innerHTML); //this this is i1 is this is i2 p  
write(p.outerHTML); //this this is i1 is this is i2 p  
  
write(p.innerHTML.toString()); //this this is i1 is this is i2 p  
  
  
  
/\*作为纯文本的元素内容\*/  
//有时需要查询纯文本形式的内容，或者在文档中插入纯文本（不必转义HTML标记中使用的尖括号和&符号）  
//使用Node的textContent属性来实现  
  
p = document.getElementById('para');  
write(p.textContent); //this this is i1 is this is i2 p  
  
//除了IE都支持，在IE中用Element的innerText属性来代替  
  
*/\*\*  
 \* 如果1个参数，返回元素的textContext或innerText  
 \* 如果2个参数，用text参数设置元素的textContext或innerText  
 \*/***function** textContext(e,text){  
 **var** content = e.textContent;  
  
 **if**(text === undefined){ //如果text未定义  
 **if**(content !== undefined){  
 **return** content; //返回元素的textContent  
 } **else**{  
 **return** e.innerText; //返回IE的属性  
 }  
 } **else**{  
 **if**(content !== undefined){  
 e.textContent = text; //注意不要使用content = value  
 } **else**{  
 e.innerText = text;  
 }  
 }  
}  
  
  
  
  
  
  
//textContent属性就是将指定元素的所有后代Text节点简单的串联在一起  
//innerText和textContent有一些不同，不返回<script>元素的内容  
//忽略多余的空白。并试图保留表格格式  
//<script>元素中如果设置type = text/x-custiom-data  
//就表明了脚本不可执行的JavaScript代码，JS解析器将忽略该脚本，但元素仍然存在于文档中  
//它的text属性还将返回数据给你  
  
/\*作为Text节点的元素内容\*/  
//在考虑元素的内容时，通常感兴趣的是它的Text节点  
  
//递归遍历元素的子节点，连接后代节点中的所有Text节点的文本  
//nodeValue保存Text节点的内容  
  
//返回e的纯文本内容，递归进入其子元素  
//该方法的效果类似于textContent属性  
  
**function** textContent(e){  
 **var** child,type,s='';  
  
 **for**(child = e.firstChild; child != **null**; child = child.nextSibling){  
 type = child.nodeType;  
  
 **if**(type === 3 || type === 4){ //Text和CDATASection节点  
 s += child.nodeValue; //累加Text节点中文本的内容  
 } **else if**(type === 1){ //如果是Element 则继续递归调用  
 s += textContent(child); //递归  
 write(s); //this this is i1  
 //this this is i1 is this is i2  
 //this this is i1 is this is i2 p  
 }  
 }  
 **return** s;  
}  
  
  
*/\*\*  
 \** <p *id="para">  
 this* <i *class="innerI">  
 this is i1* </i>  
 *is* <i>  
 *this is i2* </i>  
 *p* </p>  
 *\*  
 \*/***var** p = document.getElementById('para');  
write(textContent(p)); //this this is i1 is this is i2 p  
  
  
//nodeValue属性是可读可写的，设置它可以改变Text或CDATASection节点所显示的内容  
//递归把n的后代子节点中的所有Text节点内容转换为大写形式  
//CharacterData定义data属性 和nodeValue的文本相同  
  
  
**function** upcase(n){ //n指代node e指代element  
 **if**(n.nodeType === 3 || n.nodeType === 4){ //如果n是text节点或者CATA节点  
 n.data = n.data.toUpperCase();  
 }  
  
 **else**{  
 **for**(**var** i=0,len=n.childNodes.length; i<len; i++){ //遍历该节点的子节点  
 upcase(n.childNodes[i]); //递归调用  
 }  
 }  
}  
  
upcase(p);//THIS THIS IS I1 IS THIS IS I2 P

## 13.6 创建、插入和删除节点

//从指定的URL异步加载和执行脚本  
  
**function** loadasync(url){  
 **var** head = document.getElementsByTagName('head')[0]; //查找文档中的head标签  
 **var** s = document.createElement('script'); //创建一个<script>元素  
 s.src = url; //设置它的src属性值  
 head.appendChild(s); //将该<script>插入到head中  
}  
  
//操作多个节点时的一种捷径 DocumentFragment 文档碎片  
  
  
/\*创建节点\*/  
  
//创建Element节点  
//createElement 传递元素的标签名，对HTML文档来说不区分大小写  
  
//创建Text节点  
**var** newTextNode = document.createTextNode('text node content');  
  
//复制已经存在的节点  
  
//cloneNode() 传递参数true能够递归的复制所有的后代节点 传递参数false只是执行一个浅复制  
  
**var** p = document.getElementById('para');  
//该方法是节点的方法  
**var** p\_deep\_copy = p.cloneNode(**true**); //返回该节点的深拷贝副本  
  
**var** body = document.getElementsByTagName('body')[0];  
body.appendChild(p\_deep\_copy); //this this is i1 is this is i2 p  
  
  
**var** p\_copy = p.cloneNode(**false**); //浅复制  
body.appendChild(p\_copy);

//节点的创建和插入  
  
/\*插入节点\*/  
//appendChild() 和 insertBefore()  
  
//appendChild在需要插入的Element节点上调用的，插入指定的节点使其成为那个节点的最后一个子节点  
//insertBefore两个参数，第一个是带插入的节点,第二个是已存在的节点，新节点将插入该节点的前面  
//该方法在新节点的父节点上调用  
//第二个参数必须是该节点的父节点的子节点 如果是null，行为和appendChild一样  
  
//数字索引位置插入节点的简单函数  
**function** insertAt(parent,child,n){  
 **if**(n < 0 || n > parent.childNodes.length){ //检测n值的准确性  
 **return false**;  
 } **else if**(n === parent.childNodes.length){ //如果插入的是最后一个位置  
 parent.appendChild(child); //如果插入的是之前的任何位置  
 } **else**{  
 parent.insertBefore(child, parent.childNodes[n]);  
 }  
}  
  
  
//insertAt(p,p\_deep\_copy,1);  
//appendChild或insertBefore也可以将已存在的节点再次插入，那个节点将自动从它原来的位置删除并在新的位置重新插入  
  
  
//表格的行排序  
  
//根据指定表格每行第n个单元格的值，对第一个<tbody>中的行进行排序  
*/\*\** <table *id='tb'>* <caption>*价格表*</caption>  
 <thead>  
 <tr>  
 <th>*瓜种*</th>  
 <th>*价格*</th>  
 </tr>  
 </thead>  
 <tbody>  
 <tr>  
 <th>*东瓜*</th>  
 <th>*100*</th>  
 </tr>  
 <tr>  
 <th>*西瓜*</th>  
 <th>*98*</th>  
 </tr>  
 <tr>  
 <th>*南瓜*</th>  
 <th>*99*</th>  
 </tr>  
 <tr>  
 <th>*北瓜*</th>  
 <th>*95*</th>  
 </tr>  
 <tr>  
 <th>*黄瓜*</th>  
 <th>*97*</th>  
 </tr>  
 </tbody>  
 <tfoot>  
 <tr>  
 <th>*总价*</th>  
 <th>*200*</th>  
 </tr>  
 </tfoot>  
 </table>  
 *\*/***function** sortTableRows(table,n,comparator){  
 //var tbody = document.getElementsByTagName('tbody')[0];  
 **var** tbody = table.tBodies[0]; //可能是隐式创建的  
 **var** rows = tbody.getElementsByTagName('tr'); //tbody中的所有行 HTMLCollection[5]  
 //以上得到的是一个类数组  
 rows = Array.prototype.slice.call(rows,0); //真实数组 Array[5]  
  
 //这里rows是真正的数组了 基于第n个td元素的值进行排序  
 rows.sort(**function**(row1,row2){  
 **var** ceil1 = row1.getElementsByTagName('td')[n]; //获得第n个单元格  
 **var** ceil2 = row2.getElementsByTagName('td')[n]; //两行都是  
  
 **var** val1 = ceil1.textContent || ceil1.innerText;  
 **var** val2 = ceil2.textContent || ceil2.innerText;  
  
 **if**(comparator){  
 **return** comparator(val1,val2); //按照特定的函数比较  
 }  
 **return** val2-val1; //降序排列  
 });  
  
 **for**(**var** i=0,len=rows.length; i<len; i++){  
 tbody.appendChild(rows[i]); //重新排列  
 }  
  
}  
**var** table = document.getElementById('tb');  
  
sortTableRows(table,1); //根据

  
//查找表格的地n个th元素，使它可单击  
**function** makeSortTable(table,n){  
 **var** thead = table.tHead;  
 **var** th = thead.getElementsByTagName('th');  
 **if**(n<0 || n>th.length) {  
 **return false**;  
 }  
  
 th[n].onclick = **function**(){  
 sortTableRows(table,n);  
 }  
}  
  
**var** table = document.getElementById('tb');  
makeSortTable(table,1);

/\*删除和替换节点\*/  
  
//removeChild方法时从文档树中删除一个节点  
//在其父节点上调用  
//需要删除的子节点作为方法参数传递给它  
  
//例如在文档中删除n节点  
//n.parentNode.removeChild(n)  
  
//replaceChild 删除一个字节点，并用新的节点替换它  
// 在父节点上调用它，参数一是新节点，参数而是待删除的节点  
  
//用一个新的<b>元素替换n元素,并使n元素成为<b>元素的子节点  
  
**function** embolden(n){  
 **if**(**typeof** n == 'string'){  
 n = document.getElementById(n);  
 }  
 **var** parent = n.parentNode; //获取n的父节点  
   
 **var** b = document.createElement('b'); //创建一个b元素  
   
 parent.replaceChild(b,n); //用b元素替换n元素  
   
 b.appendChild(n); //将n元素添加到b元素的子节点中  
}  
  
  
/\*使用DocumentFragment\*/  
  
//DocumentFragment是一种特殊的Node,它作为节点的临时容器  
//创建一个DocumentFragment  
  
**var** frag = document.createDocumentFragment();  
  
//DocumentFragment是独立的,而不是任何其他文档的一部分  
//它的parentNode总是null  
//它可以有任意多的子节点  
//可以用appendChild insertBefore等方法来操作它们  
//DocumentFragment的特殊之处在于它使得一组节点被当做一个节点看待  
//调用appendChild insertBefore或replaceChild()传递一个DocumentFragment  
//其实是将该文档片段的所有子节点插入到文档中,而非片段本身  
  
//倒序排列节点n的子节点  
  
**function** reverse(n){  
 **var** f = document.createDocumentFragment();  
  
 **while**(n.lastChild){  
 f.appendChild(n.lastChild);  
 }  
  
 n.appendChild(f);  
}  
  
  
**var** ul = document.getElementById('ul');  
reverse(ul);

C:\Users\Victor.Zxk\AppData\Local\Temp\637.tmp.png

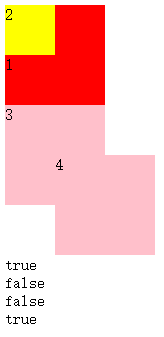
C:\Users\Victor.Zxk\AppData\Local\Temp\4DB2.tmp.png

## 13.7 文档和元素的几何形状和滚动（重要）

/\*文档坐标和视口坐标\*/  
//有两个不同的点作为坐标系的原点  
//1.文档坐标：元素X和Y的坐标可以相对于文档的左上角  
//2.视口坐标（窗口坐标）：相对于在其中显示的文档的视口的左上角（有滚动条的情况下滚动了页面后的位置）  
  
//如果没有滚动条，文档左上角就是视口左上角  
//文档坐标 = 视口坐标 + 滚动条偏移量  
//文档坐标在用户滚动时不会发生变化，而视口坐标是会变化的  
  
//浏览器窗口中滚动条的位置 pageXOffset pageYOffset  
// IE: scrollLeft scrollTop  
  
//查询窗口滚动条的位置  
**function** getScrollOffsets(w){  
 w = w || window; //使用指定的窗口，不带参数则是使用当前窗口  
  
 **if**(w.pageXOffset != **null**){ //除了IE8或者更早的版本以外其他浏览器都可用  
 **return** {  
 x: w.pageXOffset, //滚动条的位置  
 y: w.pageYOffset  
 };  
 }  
  
 **var** d = w.document;  
 **if**(document.compatMode == 'CSS1Compat'){ //对标准模式下的IE（或任何浏览器）  
 **return**{  
 x: d.documentElement.scrollLeft,  
 y: d.documentElement.scrollTop //正常情况下需要查询文档的根节点d.documentElement来获取这些属性值  
 };  
 }  
  
 **return**{ //怪异模式  
 x: d.body.scrollLeft, //怪异模式下必须在文档的body元素上查询它们  
 y: d.body.scrollTop  
 };  
}  
  
  
//浏览器窗口的视口大小 innerWidth innerHeight  
// IE: clientWidth clientHeight  
  
//查询窗口的视口大小  
**function** getViewPortSize(w){  
 w = w || window;  
  
 **if**(w.innerWidth){ //除了IE8或者更早的版本以外其他浏览器都可用  
 **return**{  
 x: w.innerWidth,  
 y: w.innerHeight  
 };  
 }  
  
 **var** d = document;  
 **if**(document.compatMode == "CSS1Compat"){ //对标准模式下的IE（或任何浏览器）  
 **return**{  
 x: d.documentElement.clientWidth,  
 y: d.documentElement.clientHeight  
 };  
 }  
  
 **return** {//怪异模式  
 x: d.body.clientWidth,  
 y: d.body.clientHeight  
 };  
}

/\*查询元素的几何尺寸\*/  
  
//getBoundingClientRect() 这个方法在IE5中引入，所有浏览器都能实现  
//不需要参数，返回一个有left right top bottom属性的对象  
//left和top属性表示元素的左上角的X和Y坐标  
//right和bottom表示元素的右下角的X和Y坐标

//元素的位置  
//返回元素在视口坐标中的位置  
//如果需要转换为文档坐标（用户滚动浏览器窗口以后），需要加上滚动的偏移量  
  
**function** getElementPosition(e){  
 **var** box = e.getBoundingClientRect(); //返回元素在视口坐标中的位置  
  
 **var** offset = getScrollOffsets(); //获取滚动坐标  
  
 **return** {  
 //返回元素左上角的视口坐标位置  
 //getBoundingClientRect返回的是一个有left right top和bottom对象的属性  
 view\_x : box.left,  
 view\_y : box.top,  
 //返回元素左上角的文档坐标位置  
 dom\_x: box.left + offset.x,  
 dom\_y: box.top + offset.y  
 };  
}  
//getBoundingClientRect() 这个方法在IE5中引入，所有浏览器都能实现  
//不需要参数，返回一个有left right top bottom属性的对象  
//left和top属性表示元素的左上角的X和Y坐标  
//right和bottom表示元素的右下角的X和Y坐标  
  
//阿里笔试题目 实现一个函数 用于检测文档中两个可见的矩形元素是否有重叠  
  
**function** ieOverlap(elem1, elem2){  
  
 //1.获取两个矩形在视口中的位置,注意不是文档坐标是视口坐标  
 **var** e1Box = elem1.getBoundingClientRect(); //get获取bounding边界框Client客户端Rect矩形  
 **var** e2Box = elem2.getBoundingClientRect();  
  
 //2.获取两个矩形各自的大小  
 **var** e1Width = e1Box.width || (e1Box.right - e1Box.left);  
 **var** e2Width = e2Box.width || (e2Box.right - e2Box.left);  
 **var** e1Height = e1Box.height || (e1Box.bottom - e1Box.top);  
 **var** e2Height = e2Box.height || (e2Box.bottom - e2Box.top);  
  
 //多数浏览器的getBoundingClientRect方法还会返回width和height属性  
 //这两个属性包含元素的边框和内边距  
 //需要注意的是getBoundingClientRect方法返回的矩形对象并不是实时的  
 **if**(((e1Width + e2Width) > (e2Box.right - e1Box.left)) &&  
 ((e1Height + e2Height) > (e2Box.bottom - e1Box.top))){  
 **return true**; //有重叠  
 } **else**{  
 **return false**; //没有重叠  
 }  
}  
  
  
**var** div1 = document.getElementById('1');  
**var** div2 = document.getElementById('2');  
**var** div3 = document.getElementById('3');  
**var** div4 = document.getElementById('4');  
  
write(ieOverlap(div1,div2)); //true  
write(ieOverlap(div1,div3)); //false  
write(ieOverlap(div2,div3)); //false  
write(ieOverlap(div3,div4)); //true



//包含元素的内边距和边框，不包含元素的外边距  
**function** getElementSize(e){  
 **var** box = e.getBoundingClientRect(); //返回元素在视口坐标中的位置  
  
 //原始的IE可能并没有实现width和height属性  
 **return** {  
 width: box.width || (box.right - box.left),  
 height: box.height || (box.bottom - box.top)  
 };  
}

/\*判定元素在某点\*/  
  
//判定某个点上的位置上是什么元素  
//elementFromPoint()  
//传入X Y参数，使用视口坐标而非文档坐标  
  
//已经被target属性替代了，不常用了  
  
/\*滚动\*/  
//可以设置scrollLeft 和 scrollTop的值来滚动浏览器  
//另一种方法是 Window对象的scrollTop()方法  
//接收一个点的X和Y（注意是文档坐标不是视口坐标）参数来设置滚动条的偏移量  
//如果想滚动使得文档中的某个元素可见，可以利用getBoundingClientRect()方法获取元素的视口坐标  
//加上滚动偏移量转化为文档坐标，然后使用scrollTop()方法

/\*关于元素尺寸、位置和溢出的更多信息\*/

//getBoundingClientRect()方法在所有当前浏览器上都有定义  
//但是要支持老浏览器，必须使用更老的技术来判定元素的尺寸和位置  
//元素的尺寸：任何HTML元素的只读属性offsetWidth offsetHeight  
//元素的文档坐标: 所有的HTML元素都有 offsetLeft 和 offsetTop 返回元素的X和Y坐标 注意是文档坐标  
//当然这个属性对于已经定位元素的后代元素和一些表格单元等，这些属性返回的坐标是相对于祖先元素而非文档  
//offsetParent属性指定这些属性所相对的父元素 如果为null offsetLeft 和 offsetTop是文档坐标  
  
//之前获取文档坐标的方法 方法一  
**function** getElementPosition\_1(e){  
 **var** box = e.getBoundingClientRect(); //返回元素在视口坐标中的位置  
  
 **var** offset = getScrollOffsets(); //获取滚动坐标  
  
 **return** {  
 //返回元素左上角的视口坐标位置  
 //getBoundingClientRect返回的是一个有left right top和bottom对象的属性  
 view\_x : box.left,  
 view\_y : box.top,  
 //返回元素左上角的文档坐标位置  
 dom\_x: box.left + offset.x,  
 dom\_y: box.top + offset.y  
 };  
}  
  
//方法二  
**function** getELementPosition\_2(e){  
  
 **var** dom\_x = 0;  
 **var** dom\_y = 0; //返回的是窗口坐标  
  
 **while**(e != **null**){  
 dom\_x += e.offsetLeft; //相对于父元素的左上角的偏移量  
 dom\_y += e.offsetTop;  
 e = e.offsetParent; //向上遍历父元素，如果为null 该元素的坐标就是文档坐标  
 }  
  
 **return** { //返回文档坐标 这个量是累加偏移量得到的  
 x: dom\_x,  
 y: dom\_y  
 };  
}

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | 浏览器 | X | Y |
| 滚动条偏移  scrollLeft可读可写 | 正常 | window.pageXOffset | \_YOffset |
| IE标准 | window.document.documenElement.scrollLeft | \_scrollTop |
| IE怪异 | window.document.body.scrollLeft | \_scrollTop |
| 视口大小  它们不包含边框大小，只包含内容和它的内边距，如果有滚动条，也不包含 | 正常 | window.innerWidth | innerHeight |
| IE标准 | window.document.documenElement.clientWidth | \_clientHeight |
| IE怪异 | window.document.body. clientWidth | \_clientHeight |
| element.getBoundingClientRect()  返回的坐标包含元素的边框和内边距  element是具体的元素  视口坐标，不是文档坐标 | 正常 | left right  元素左上角的x坐标 | top bottom  元素右下角的y坐标 |
| 正常 | width  元素的width大小 | height  元素的height大小 |
| IE | .right - .left  元素的width大小 | .bottom - ,top  元素的height大小 |
| 如果element.getBoundingClientRect()方法不存在  element的属性  element是具体的元素 |  | offsetLeft  相对于定位元素或表格单元等父元素的x坐标，即如果offsetParent不为null，则相对于父元素，如果为null，则相对于文档，如果需要获取文档坐标，则需要遍历元素的offsetParent,直至其属性为null，然后累加这些偏移量 | offsetTop |
|  | offsetWidth  **与**clientWidth不同的是该属性包含边框大小 | offsetHeight |

## 13.8 HTML表单

/\*HTML表单\*/  
  
/\*选取表单和表单元素\*/  
//document.querySelectorAll('#shipping input[type="radio"]');  
//选取id为'shipping'的表单中所有的单选按钮  
  
//document.querySelectorAll('#shipping input[type="radio"][name="method"]');  
//选取id为'shipping'的表单中所有名字为'method'的单选按钮  
  
//也可以用如下方法  
//var inputs = document.getElementById('form').getElementsByTagName('input');  
  
//当然还可以产用如下方式  
//有name和id属性的form可以用以下任何方式来选取  
*/\*\*  
 \** <form *name="input" method="get">  
 Username:* <input *type="text" name="user">* <input *type="submit" value="Submit">* </form>  
 *\*/*//访问表单  
**var** form = window.input; //不可靠，不要使用  
**var** form1 = document.input; //仅当表单有name属性时可用  
**var** form2 = document.forms.input; //显示访问有name或id的表单  
**var** form3 = document.forms[0]; //不可靠，访问文档中的第一个表单  
  
//访问表单元素  
**var** user = document.forms.input[0]; //访问名为input的表单的第一个元素  
**var** user1 = document.forms.input.user; //访问名为input的表单的第一个name为user的元素  
write(user === user1); //true  
**var** user2 = document.input.user; //当表单有name为input,而不只是有id为input  
write(user2 === user1); //true  
  
//如果要明确的访问一个表单元素，可以US噢因表单对象的elements属性  
  
**var** user3 = document.forms.input.elements[0];  
write(user3 === user2);//true  
  
**var** user4 = document.forms.input.elements.user;  
write(user4 === user3); //true  
  
//当用name来索引时返回值往往是一个类数组对象  
  
/\*表单和元素的属性\*/  
  
/\*表单和元素的事件处理程序\*/

## 13.9 其他文档特性

/\*其他文档特性\*/  
  
/\*document.write()\*/  
  
  
*/\*\*  
 \* 重要理解：  
 \* 只有在解析文档时(刷新或者第一次加载HTML页面时)才能使用write()方法输出HTML到当前文档中  
 \* 由于脚本的执行是文档解析的一部分，所以能够在*<script>元素的顶层代码中调用document.write()  
 *\* 但是如果不是在文档解析时候能够执行的document.write()方法  
 \* 例如要触发事件才能执行document.write()方法 那么会擦除当前文档和它包含的脚本  
 \*/*//可以在<script>元素的顶层代码中调用document.write()  
//因为脚本的执行是文档解析流程的一部分  
  
//但是如果将document.write()放在一个函数的定义中  
//该函数的调用是从一个时间处理程序中发起的  
//产生的结果未必是你想要的  
//事实上它会擦除当前文档和它包含的脚本  
//同理在设置了defer和async属性的脚本中不要使用document.write()  
  
/\*  
记住，在载入页面后，浏览器输出流自动关闭。  
在此之后，任何一个对当前页面进行操作的document.write()方法将打开—个新的输出流，  
它将清除当前页面内容(包括源文档的任何变量或值)。  
因此，假如希望用脚本生成的HTML替换当前页面，  
就必须把HTML内容连接起来赋给一个变量，  
使用一个 document.write()方法完成写操作。  
不必清除文档并打开一个新数据流，  
一个document.write()调用就可完成所有的操作。  
\*/  
  
  
/\*  
关于document.write()方法还有一点要说明的是它的相关方法document.close()。  
脚本向窗口(不管是本窗口或其他窗口)写完 内容后，必须关闭输出流。  
在延时脚本的最后一个document.write()方法后面，  
必须确保含有document.close()方法，  
不这样做 就不能显示图像和表单。  
并且，任何后面调用的document.write()方法只会把内容追加到页面后，  
而不会清除现有内容来写入新值。  
\*/  
  
*/\*\*  
 \** <form *name="input" method="get">  
 Username:* <input *type="text" name="user">* <input *type="submit" onclick="f()" value="Submit">* </form>  
 *\*/***function** f(){  
 write('1234'); //清除了当前的页面  
 write('<h1>' + 'this is a' + '</h1>');  
}

# 第十四章 脚本化CSS

## 14.1 重要的CSS属性

/\*CSS盒模型和定位细节\*/  
  
//外边距和绝对定位的元素无关  
//width和height只指定了元素内容区域的尺寸  
//它不包含元素的内边距和边框（或外边距）  
//为了确定有边框元素在屏幕上的全尺寸，需要把宽度加上左右两边的内边距和左右两个边框宽度  
//把元素的高度加上上下两边的内边距和上下两个边框的宽度  
  
  
//left和top  
//如果一个绝对定位的元素嵌套在一个定位的容器元素中  
//left和top属性指定了从容器边框内侧到定位元素边框外侧的距离  
//需要注意的是这些属性不是从容器内容区域的左上角开始度量  
//而是从容器内边距的左上角开始的

<head lang="en">  
 <title>this is title</title>  
 <style>  
 .outer{  
 position: absolute;  
 width: 200px;  
 height: 200px;  
 border: 5px solid yellow;  
 padding: 20px;  
 }  
  
 .inner{  
 position: absolute;  
 width: 50px;  
 height: 50px;  
 left:0;  
 top: 0;  
 background: green;  
 z-index: -1;  
 }  
  
  
 </style>  
</head>  
<body>  
  
 <div class="outer">  
 这个上面和左面是padding哦,这里是内容区  
 <div class="inner">  
  
 </div>  
 </div>



可以清楚的知道相对于边框定位，而不是内容区

//标准的CSS盒模型规定width和height样式属性给定内容区域的尺寸，并且不包含内边距和边框  
//可以称此盒模型为‘内容盒模型’  
//在IE6之前和IE6-8的怪异模式下显示一个页面时（缺少<！DOCTYPE>或有一个不够严格的doctype时）  
//width和height属性确实包含内边距和边框宽度的

//CSS3引进了box-sizing属性，默认是‘content-box’ 是标准盒模型  
//如果设置box-sizing属性为 border-box 则会使用IE盒模型，包含边框和内边距

/\*元素显示和可见性\*/  
//visibility display  
//display：none 受影响的元素根本不显示，也没有布局  
//visibility和display属性之间的差别可以从它们对使用静态或相对定位的元素的影响中看到  
//对于一个常规布局流中的元素，设置visibility为hidden使得元素不可见，但是文档布局中仍保留了它的空间  
//类似的元素可以重复隐藏或显示而不改变文档布局  
//但是如果元素是display属性设置none,在文档布局中不再给它分配空间,它各边的元素会合拢  
//就当它从来不存在 display在创建展开和折叠轮廓的效果时很有用  
  
//visibility和display对绝对和固定定位的元素的影响是等价的  
//因为这些元素都不是文档布局的一部分，然而在隐藏和显示定位元素时一般首选visibility属性

/\*用CSS定位元素\*/  
//如果想相对于一个属性常规文档流中的容器绝对定位一个元素  
//则将容器的position指定为relative,top和left指定为0px  
//这就让容器变成了动态定位，但它仍保留在文档流中的原来的位置

/\*overflow 和 clip\*/

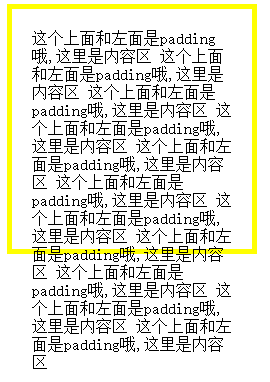
<div class="outer">  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
 这个上面和左面是padding哦,这里是内容区  
</div>

<style>  
 .outer{  
 position: absolute;  
 width: 200px;  
 height: 200px;  
 border: 5px solid yellow;  
 padding: 20px;

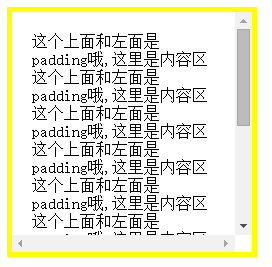
overflow: visible; /\*默认值\*/

}  
</style>

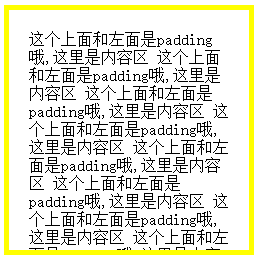
Overflow默认值，visible 内容可以溢出并绘制在元素的边框外面



<style>  
 .outer{  
 position: absolute;  
 width: 200px;  
 height: 200px;  
 border: 5px solid yellow;  
 padding: 20px;  
 overflow: scroll; /\*设置成滚动\*/  
 }  
</style>



<style>  
 .outer{  
 position: absolute;  
 width: 200px;  
 height: 200px;  
 border: 5px solid yellow;  
 padding: 20px;  
 overflow: hidden; /\*隐藏溢出的部分\*/  
 }  
</style>



<style>  
 .outer{  
 width: 200px;  
 height: 200px;  
 border: 5px solid yellow;  
 padding: 20px;  
 clip: rect(100px 25px 25px 25px); /\*定义裁剪区域，相对于元素边框的左上角\*/  
 /\*top right bottom left\*/  
 }  
</style>

## 14.2 脚本化内联样式

<p id="p1">this is p</p>

<style> /\*内联样式表\*/  
 #p1{  
 font-size: 60px;  
 color: red;  
 }  
</style>

//更改单独的文档元素的style属性  
//设置元素的文本  
//改变了文档样式表 即在<head>中的<style>样式  
**var** p = document.getElementById('p1');  
p.style.fontSize = '100px'; //font-size 在脚本中是驼峰命名法的名字属性  
p.style.color = 'yellow';

假设加入嵌套样式表

<p id="p1" style="font-size: 10px">元素中的style是嵌套样式表</p>  
<script src="lib.js" type="text/javascript"></script>  
<script src="test.js" type="text/javascript"></script>



<script src="lib.js" type="text/javascript"></script>  
<script src="test.js" type="text/javascript"></script>  
<p id="p1" style="font-size: 10px">元素中的style是嵌套样式表</p>

C:\Users\Victor.Zxk\AppData\Local\Temp\3E72.tmp.png

为什么颜色是红色？嵌套样式表中没有改变字体颜色，嵌套样式表因为优先级高，所以覆盖了JS脚本中的font-size属性可以理解，但是颜色为什么是内联样式中的red？？？

//脚本中的style属性可以理解为代表内联样式  
//元素的内联样式只是在设置样式的时候有用，如果需要查询元素的样式，就要使用计算样式  
//读取内联样式特别困难，对复合属性来说，在真正使用这些值的时候，代码不得不包含非同寻常的CSS解析能力  
//有时，作为单个字符串值来设置或查询元素的内联样式反而比作为CSSStyleDeclaration对象更加简单  
//可以使用元素的getAttribute()和setAttribute方法或CSSStyleDeclaration的cssText属性来实现  
  
**var** cssText = p.getAttribute('style'); //"font-size: 100px; color: yellow;"  
**var** cssText1 = p.style.cssText;//"font-size: 100px; color: yellow;"

/\*脚本化内联样式\*/  
  
  
/\*CSS动画\*/  
//使用setTimeout()或setInterval()重复调用函数来修改元素的内联样式达到目的  
  
//第一参数 目标元素e 或者元素id 相对定位  
//第二参数 e震动的距离 默认是10px  
//第三参数 震动时间 默认是500ms  
//第四参数 动画完成时的回调函数  
**function** shake(e,distance,time,oncomplete){  
 **if**(**typeof** e === 'string'){  
 e = document.getElementById(e);  
 }  
  
 **if**(!distance){  
 distance = 10;  
 }  
  
 **if**(!time){  
 time = 500;  
 }  
  
 **var** originStyle = e.style.cssText;  
 e.style.position = 'relative';  
 **var** startTime = (**new** Date()).getTime(); //返回距 1970 年 1 月 1 日之间的毫秒数。  
 //write(startTime);  
  
 animate();  
  
  
 **function** animate(){  
 **var** nowTime = (**new** Date()).getTime();  
 **var** timeLong = nowTime - startTime; //消耗的时间  
 **var** precent = timeLong/time; //消耗的时间百分数  
  
 **if**(precent < 1){ //时间没到  
 //使用正弦函数将完成的比例乘以4pi  
 //4pi是两个正弦周期，所有有两次正负值，所以来回往复两次  
 **var** x = distance \* Math.sin(precent \* 4 \* Math.PI);  
 e.style.left = x + 'px'; //因为使用相对定位，所以使用left属性值  
 //每25ms尝试运行函数，如果小于25ms则以小于25ms运行直至0ms为止  
 setTimeout(animate, Math.min(25,time - timeLong));  
 }  
 **else**{  
 e.style.cssText = originStyle;  
 **if**(oncomplete) {  
 oncomplete(e);  
 }  
 }  
 }  
}  
  
  
  
  
  
//将e从完全不透明淡出到透明  
//假设e是完全不透明的  
  
**function** fadeOut(e,time,complete){  
 **if**(**typeof** e === 'string'){  
 e = document.getElementById(e);  
 }  
  
 **if**(!time){  
 time = 500;  
 }  
  
 **var** startTime = (**new** Date()).getTime();  
 //var originCssText = e.style.cssText;  
 **var** ease = Math.sqrt;  
  
  
 animate();  
  
 **function** animate(){  
 **var** nowTime = (**new** Date()).getTime();  
 **var** timeLong = nowTime - startTime;  
 **var** precent = timeLong / time;  
  
 **if**(precent < 1) {  
 e.style.opacity = String(1 - ease(precent)); //精巧的非线性 缓动函数 一开始淡出比较快，后面淡出比较慢  
 setTimeout(animate,Math.min(25,time-timeLong));  
 } **else**{  
 e.style.opacity = '0';  
 **if**(complete){  
 complete(e);  
 }  
 }  
 }  
}  
//当然CSS3的过度模板定义了在样式表中指定动画效果的方式

## 14.3 查询计算出的样式

/\*查询计算出的样式\*/  
  
//元素style属性代表了元素的内联样式  
//它覆盖了所有的样式表  
  
  
//getComputedStyle() 计算样式 该方法是浏览器窗口对象的方法  
  
//e.style 内联样式 只是在设置样式的时候有用，入股偶要查询元素的样式，就要使用计算样式  
  
//计算样式  
**var** btn = document.getElementById('bt');  
**var** width = window.getComputedStyle(btn,"").width; //参数一是要获取其计算样式的元素，第二个参数是必须的  
//通常是null或者空字符串  
  
write(width); //200px  
  
//内联样式  
btn.style.width = '400px'; //可行

## 14.4 脚本化CSS类

/\*脚本化CSS类\*/

.p1{  
 font-size: 100px;  
 color: red;  
  
}  
  
.p1.p2{  
 font-size: 50px;  
 color: blue;  
}  
  
.p2{  
 font-size: 50px;  
 color: yellow;  
}

**function** addClass(e,cla){  
 e.className = cla;  
}  
  
**function** clearClass(e){  
 e.className = '';  
}  
  
**var** p = document.getElementById('p');  
  
setTimeout(**function**(){  
 clearClass(p);  
},1000);  
  
addClass(p,'p1');  
  
//className属性只能假定0个或1个类名  
//多个类名时候就无法工作  
addClass(p,'p2'); //覆盖了p1 类 没有.p1.p2两个类  
  
//HTML5解决了这个问题，为每个元素定义classList属性  
//一个只读的类数组对象  
//add和remove方法可以添加和清除一个类名  
//toggle表示如果不存在类名就添加一个  
//否则，就删除它  
//contains检测class属性是否包含一个指定的类名  
//可能不是所有的浏览器都支持classList属性

**var** classList = p.classList; //'p2'  
classList.add('p1'); //'p1,p2'两个类名  
  
//类数组对象转化为数组对象  
**var** arr = Array.prototype.slice.call(classList,0); //Array[2]  
write(arr); //p2,p1  
  
classList.remove('p1');  
  
arr = Array.prototype.slice.call(classList,0); //Array[2]  
write(arr); //p2

## 14.5 脚本化样式表

# 第十五章 事件处理

## 15.1 事件类型

/\*事件类型\*/  
  
//新事件来源  
*/\*\*  
 \* 1. 3级DOM事件事件  
 \* 2. HTML5规范 比如：历史管理 拖放 跨文档通信 视频和音频的播放  
 \* 3. 基于触摸和JS的移动设备的出现  
 \*/*/\*传统的事件类型\*/  
  
*/\*\*  
 \* 1.表单事件  
 \* submit reset click focus blur change and so on  
 \* focus 和 blur 事件不会冒泡  
 \* 其他所有表单事件都可以冒泡  
 \*  
 \* 2.Window事件  
 \* load 当文档和其所有外部资源（比如图片）完全加载并显示给用户时就会触发它  
 \* unload 当用户离开当前文档转向其他文档时就会触发它 可以用于保存用户的状态 不能用于取消用户转向其他地方  
 \* beforeunload 能提供询问用户是否确定离开当前页面的机会  
 \* onerror  
 \* focus blur 事件也能用做Window事件 当浏览器窗口从操作系统中得到或失去键盘焦点时会触发它们  
 \* resize和scroll事件 调整浏览器窗口大小或滚动它是会触发  
 \* scroll也可以在任何设置了CSS的overflow属性的元素上触发  
 \* 3.鼠标事件  
 \* 鼠标事件会冒泡直到文档最顶层  
 \* clientX clientY 鼠标在窗口坐标中的位置  
 \* button which 按下的是鼠标键的哪一个  
 \* 对于click事件，detail属性指定了其是单击、双击还是三击  
 \*  
 \* mousemove 移动或拖动鼠标时触发  
 \* mousedown mouseup 按下或释放鼠标时触发 可以探测和响应鼠标的拖动  
 \* click 在mousedown和mouseup事件队列之后，浏览器还会触发click事件  
 \* dbclick 连续两次单击鼠标 跟在第二个click事件之后的是dbclick事件  
 \* contextmenu 鼠标右击获取浏览器显示上下文菜单之前或触发该事件  
 \* mouseover 移动鼠标指针使它悬停在新元素上时触发  
 \* mouseout 移出某个元素时触发  
 \* mousewheel 滚动鼠标滚轮时触发  
 \*  
 \* 4.键盘事件  
 \* keyCode  
 \* keydown keyup keypress  
 \*/*/\*3级DOM事件\*/  
  
/\*HTML5事件\*/  
//播放音频和视频的事件  
//拖放API  
//历史管理机制  
//定义了表单验证机制，验证失败时会在表单元素上触发invalid事件  
//离线WEB应用的支持  
//message事件进行异步通信 扩文档通信API 允许一台服务器上的脚本和另外一台服务器上的脚本交换信息  
//WEB存储事件  
  
/\*触摸屏和移动设备事件\*/

## 15.2 注册事件处理程序

/\*注册事件处理程序\*/  
  
  
/\*设置JavaScript对象属性为事件处理程序\*/  
  
//通过设置事件目标的属性为所需事件处理程序函数  
//事件处理程序属性的名字由'on'后面跟着事件名组成 onclick onchange onload等等  
//注意区分大小写，都是小写  
//例如  
  
window.onload = **function**(){ //onload事件  
};  
  
  
//这种事件处理程序注册技术适用于所有浏览器的所有常用事件类型  
//事件处理程序属性的缺点是其设计都是围绕着  
//假设每个事件目标对于每种事件类型将最多只有一个处理程序  
//如果想编写能够在任意文档中都能使用的脚本库代码，  
//更好的方式是使用一种不修改或覆盖任何已有注册处理程序的技术 (例如addEventListener())  
  
/\*设置HTML标签属性为事件处理程序\*/  
  
/\*addEventListener()\*/  
//除了IE8和之前的版本外，所有浏览器都支持标准事件模型  
//任何能成为事件目标的对象 这些对象包括Window对象 Document对象和所有的文档元素  
//接受三个参数  
//参数一 要注册处理程序的事件类型 这个事件类型是字符串 不包括用于设置事件处理程序属性的前缀 on  
//参数二 指定事件的类型发生时应该调用的函数  
//参数三 最后一个参数是布尔值 通常传递false 如果相反传递了true 那么函数就注册为捕获事件处理程序  
//传递false默认就是冒泡  
  
**var** btn = document.getElementById('bt');  
btn.onclick = **function**(){  
 alert('this is onclick');  
};  
  
btn.addEventListener('click',**function**(){  
 alert('this is addEventListener');  
},**false**); //默认冒泡  
  
//两个alert都会弹，说明这两个注册事件都触发了  
//说明用'click'作为第一个参数调用addEventListener()不会影响onclick属性的值  
  
  
btn.addEventListener('click',**function**(){  
 alert('this is addEventListener 1');  
},**false**); //默认冒泡  
  
btn.addEventListener('click',**function**(){  
 alert('this is addEventListener 2');  
},**false**); //默认冒泡  
  
//通过addEventListener()可以为同一个对象注册同一个事件类型的多个处理程序函数  
//以上alert都会弹  
  
  
btn.onclick = **function**(){  
 alert('this is onclick 1');  
};  
  
//覆盖了前面的onclick属性 alert('this is onclick');不会弹了 最后弹 alert('this is onclick 1');  
//addEventListener按照注册的顺序调用  
  
//removeEventListener从对象中删除处理程序  
  
/\*attachEvent() detachEvent()\*/  
  
//IE9之前的IE不支持addEventListener和removeEventListener  
//IE5及以后的版本定义了类似的方法attachEvent和detachEvent  
  
  
//IE事件模型不支持事件捕获 所以以上两个函数只有两个参数：事件类型和处理程序函数  
//与addEventListener不同的是事件类型名字中attachEvent带on  
//这点与事件处理程序属性类似  
//与addEventListener相似的是attachEvent允许相同的事件处理程序函数注册多次  
  
//兼容性写法  
**var** eventUtil = {  
 addEvent: **function**(element,eventType,handler){  
 **if**(element.addEventListener){ //DOM3级  
 element.addEventListener(eventType,handler,**false**); //默认冒泡  
 } **else if**(element.attachEvent){ //兼容IE  
 element.attachEvent(('on'+eventType),handler); //默认冒泡  
 } **else**{  
 element['on'+ eventType] = handler; //DOM0级  
 }  
 }  
};

## 15.3 事件处理程序的调用

/\*事件处理程序的参数\*/  
  
**function** handler(){  
}  
  
window.onload = handler;  
  
//以上的handler就是一个事件处理程序  
//调用事件处理程序时，把事件对象(event)作为它们的一个参数  
//在IE8及以前的版本中，通过设置属性注册事件处理程序(例如上面的window.onload)  
//调用它们时并未传递事件对象 需要通过全局对象window.event来获取事件对象  
  
**function** handler(event){  
 event = event || window.event; //获取事件对象  
}  
//非IE浏览器使用event参数来构造函数 而IE在构造函数时没有要求参数

//如果在这样的函数中使用event标识符，那么引用的正式window.event

/\*事件处理程序的运行环境\*/  
**var** btn = document.getElementById('bt');  
btn.onclick = **function**(){  
 alert(**this** === btn); //true this指代事件目标，如果把btn看做对象，调用对象的方法，this当然也是指代那个对象  
};  
  
//调用addEventListener 也可以把this指代事件目标  
  
//确保处理程序一直作为事件目标的方法调用  
  
**function** addEvent(target,eventType,handler){  
 **if**(target.addEventListener){  
 target.addEventListener(eventType,handler,**false**);  
 } **else if**(target.attachEvent){  
 target.attachEvent('on'+eventType,**function**(event){  
 //把处理程序作为事件目标的方法调用  
 //传递事件对象  
 **return** handler.call(target,event);  
 });  
 } **else** {  
 target['on' + eventType] = handler;  
 }  
}  
  
  
//使用这种方法注册的时间处理程序不能删除，因为传递给attachEvent函数的handler不是一个引用  
//详细见JavaScript高级编程

/\*事件处理程序的作用域\*/  
//通过HTML属性来注册事件处理程序被转换为能存取全局变量的顶级函数而非任何本地函数  
  
/\*  
 <button onclick = fun()> </button> 这是在HTML中绑定事件处理程序的方法叫通过HTML属性来注册事件处理程序  
 \*/  
//通过HTML属性定义的时间处理程序能好像本地变量一样使用目标对象 容器对象<form>和Document对象的属性  
//通过这种方式创建的处理程序函数，其代码近似于使用with语句修改后的作用域链  
/\*  
function(event){  
 with(document){  
 with(this.form || {}){  
 with(this){  
 //这样的话可以快捷的使用document对象的属性或者form属性或者this对象的属性  
 //例如可以直接使用tagName 替代 this.tagName  
 //使用getElementById() 替代 document.getElementById()  
 //对于表单元素则可以直接引用id替换表单元素  
 //这里是编码  
 }  
 }  
 }  
}  
\*/

/\*事件处理程序的返回值\*/  
  
//通过设置对象属性或HTML属性注册事件处理程序的返回值非常有意义  
//返回值false就是告诉浏览器不要执行这个事件相关的默认操作  
//例如表单提交按钮的onclick事件处理程序能返回false阻止浏览器提交表单  
  
  
  
  
**function** submit(){  
 **return false**;  
}  
  
window.onbeforeunload = **function**(){  
 **return** '是否跳转到新页面？';  
};  
  
//onbeforeunload事件处理程序的返回值也非常有意义 当浏览器要跳转到新页面时触发这个事件  
//如果事件处理程序中返回一个字符串，那么它将出现在询问用户是否想离开当前页面的标准对话框  
  
//事件处理程序的返回值中对通过属性注册的处理程序有意义  
//使用addEventListener()或者attachEvent()注册事件处理程序转而必须调用preventDefault()方法或设置事件对象的  
//returnValue属性

/\*事件处理程序的调用顺序\*/  
  
*/\*\*  
 \* 1.通过设置对象属性或HTML属性注册的处理程序一直优先调用  
 \* 2.使用addEventListener()注册的处理程序按照它们的注册顺序调用  
 \* 3.使用attachEvent()注册的处理程序可能按照任何顺序调用 所以代码不应该依赖调用顺序  
 \*/***var** btn = document.getElementById('bt');  
  
btn.addEventListener('click',submit1,**false**);  
  
  
  
  
  
//事件一  
btn.onclick = **function**(){  
 alert('onclick event'); //3 后厨发  
};  
  
//事件二  
**function** submit1(){  
 alert('addEventListener event'); //2.后触发  
}  
  
  
  
//事件三  
**function** submit(){  
 alert('HTML event'); //1 先触发  
}  
  
  
//发现三个一起存在时 事件三不会触发 事件二先触发 事件一后触发  
//如果去掉事件一， 那么事件三先触发 事件二后触发

/\*事件传播\*/  
  
//当事件目标是window对象或其他一些单独的对象时（比如XMLHttpRequest）  
//浏览器简单的通过调用对象上适当的处理程序响应事件  
//当事件目标是文档或文档元素时，情况比较复杂  
  
//在调用在目标元素上注册事件处理函数后，大部分事件会‘冒泡’到DOM树根  
//调用目标父元素的事件处理程序，然后调用在目标的祖父元素上注册事件处理程序  
//这会一直到Document对象 最后到达Window对象  
  
//事件冒泡在大量单独文档元素上注册处理程序提供了替代方案  
//即在共同的祖先元素上注册一个处理程序来处理所有的事件  
  
//例如可以在form元素上注册change事件处理程序来取代  
//在表单的每个元素上注册change事件处理程序  
  
  
//尽管发生在文档元素上的大部分事件都会冒泡，值得注意的是focus blur和scroll事件不会冒泡  
//文档元素上的load事件会冒泡，但它会在document对象上停止冒泡而不会传播到Window对象  
//只有当整个文档都加载完毕时才会触发Window的load事件  
  
//事件传播的三个阶段  
*/\*\*  
 \* 1.第三阶段 事件冒泡  
 \* （所谓事件冒泡即在调用目标元素的父元素的事件处理程序  
 \* 然后调用在目标的祖父元素上注册的事件处理程序，会一直到Document对象，最后到Window对象）  
 \* 2.第二阶段 目标对象本身的事件处理程序调用时  
 \* 3.第一阶段 发生在目标处理程序调用之前，成为‘捕获’阶段  
 \* 例如addEventListener()把第一个布尔值作为其第三个参数 如果是true，  
 \* 那么事件处理程序被注册为捕获事件处理程序，会在事件传播的第一个阶段调用  
 \*  
 \* 事件传播的捕获阶段像是反向的事件冒泡  
 \* 最先调用Window对象的捕获处理程序 然后是Document对象的捕获处理程序 接着是body对象的  
 \* 再然后是DOM树向下直到调用事件目标的父元素的捕获事件处理程序  
 \* 在目标对象本身上注册的捕获事件处理程序不会被调用  
 \*  
 \* 事件捕获提供了在事件没有送达目标之前查看它们的机会  
 \* 事件捕获能用于程序调试，或用于后面介绍的事件取消技术  
 \* 过滤掉事件从而使目标事件处理程序绝不会被调用  
 \* 事件捕获捕获常用语处理鼠标拖放 因为要处理的拖放的位置不能是这个元素内部的子元素  
 \*/***var** btn = document.getElementById('bt');  
  
//第一种测试  
  
document.body.addEventListener('click',**function**(){  
 alert("捕获阶段，body的click！");  
},**true**); //捕获阶段 第一个触发  
  
btn.addEventListener('click',**function**(){  
 alert("目标对象的本身事件处理程序，button的click！");  
},**false**); //目标对象的本身事件处理程序 第二个触发  
  
document.body.addEventListener('click',**function**(){  
 alert("冒泡阶段，body的click！");  
},**false**); //冒泡阶段 第三个触发  
  
  
  
//这里说明是按顺序执行的 如果和上面的合并  
//先执行alert("捕获阶段，body的click！");  
  
//其实就是执行目标对象的本身事件处理程序包括捕获和冒泡  
//接着执行 alert("目标对象的本身事件处理程序，button的click！");  
//接着按顺序执行下面几个  
  
//最后执行alert("冒泡阶段，body的click！");  
  
btn.addEventListener('click',**function**(){  
 alert("addEventListener　冒泡2");  
},**false**);  
  
btn.addEventListener('click',**function**(){  
 alert("addEventListener 捕获2");  
},**true**);  
  
  
btn.addEventListener('click',**function**(){  
 alert("addEventListener　冒泡1");  
},**false**);  
  
btn.addEventListener('click',**function**(){  
 alert("addEventListener 捕获1");  
},**true**);  
  
btn.addEventListener('click',**function**(){  
 alert("addEventListener　冒泡2");  
},**false**);  
  
btn.addEventListener('click',**function**(){  
 alert("addEventListener 捕获2");  
},**true**);

/\*事件取消\*/  
  
//取消事件操作  
  
  
//属性注册的事件处理程序的返回值能用于取消事件的浏览器默认操作  
//在支持addEventListener()的浏览器中，能通过调用事件对象的preventDefault()  
//方法取消事件的默认操作 不过在IE9之前的IE中，可以通过设置事件对象的returnValue属性为false  
//达到同样的目的  
  
**function** cancelHandler(event){  
 **var** event = event || window.event; // window.event用于IE  
  
 **if**(event.preventDefault){ //标准技术  
 event.preventDefault();  
 } **else if**(event.returnValue){ //IE  
 event.returnValue = **false**;  
 } **else**{  
 **return false**; //用于处理使用对象属性注册的处理程序  
 }  
}  
  
//取消事件传播  
  
//调用事件对象的一个stopPropagation()方法阻止事件继续传播  
//如果在同一个对象上定义了其他处理程序，剩下的处理程序将依旧被调用  
//但是任何其他对象上的事件处理程序将不会被调用  
  
//可以在事件传播期间的任意三个阶段上调用  
//能工作在捕获阶段 事件目标本身中和冒泡阶段  
  
//IE9之前的IE不支持stopPropagation()方法 IE事件对象有一个cancelBubble属性  
//设置这个属性为true可以阻止事件进一步传播  
//（IE8不支持事件传播的捕获阶段，所以冒泡是唯一的待取消的事件传播）  
  
//调用事件对象的一个stopImmediatePropagation()方法阻止事件继续传播  
//如果在同一个对象上定义了其他处理程序，剩下的处理程序也要被阻止  
//并且任何其他对象上的事件处理程序也将不会被调用  
//兼容性写法  
**var** eventUtil = {  
 //注册事件  
 addEvent: **function**(element,eventType,handler){  
 **if**(element.addEventListener){ //DOM3级  
 element.addEventListener(eventType,handler,**false**); //默认冒泡  
 } **else if**(element.attachEvent){ //兼容IE  
 element.attachEvent(('on'+eventType),handler); //默认冒泡  
 } **else**{  
 element['on'+ eventType] = handler; //DOM0级  
 }  
 },  
  
 //移除事件  
 removeEvent: **function**(element,eventType,handler){  
 **if**(element.removeEventListener){ //DOM3级  
 element.removeEventListener(eventType,handler,**false**); //默认冒泡  
 } **else if**(element.detachEvent){ //兼容IE  
 element.detachEvent(('on'+eventType),handler); //默认冒泡  
 } **else**{  
 element['on'+ eventType] = handler; //DOM0级  
 }  
 },  
  
 //获取事件对象event  
 getEvent: **function**(event){  
 **return** event || window.event;  
 },  
  
 //取消事件的默认行为  
 cancelHandler: **function**(event){  
 **if**(event.preventDefault){  
 event.preventDefault();  
 } **else if**(event.returnValue){  
 event.returnValue = **false**; //IE  
 } **else**{  
 **return false**; //针对DOM0级  
 }  
 },  
  
 //取消事件传播,不包括目标对象的其他注册事件  
 stopPropagation: **function**(event){ //Propagation 传播的意思  
 **if**(event.stopPropagation){  
 event.stopPropagation();  
 } **else** {  
 event.cancelBubble = **true**; //IE  
 }  
 },  
  
 //获取目标对象  
 getTarget: **function**(event){  
 **return** event.target || event.srcElement; //IE  
 }  
};

## 15.4 文档加载事件

P459

## 15.5 鼠标事件

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | 浏览器 | X | Y |
| 滚动条偏移  scrollLeft可读可写 | 正常 | window.pageXOffset | \_YOffset |
| IE标准 | window.document.documenElement.scrollLeft | \_scrollTop |
| IE怪异 | window.document.body.scrollLeft | \_scrollTop |
| 视口大小  它们不包含边框大小，只包含内容和它的内边距，如果有滚动条，也不包含 | 正常 | window.innerWidth | innerHeight |
| IE标准 | window.document.documenElement.clientWidth | \_clientHeight |
| IE怪异 | window.document.body. clientWidth | \_clientHeight |
| element.getBoundingClientRect()  返回的坐标包含元素的边框和内边距  element是具体的元素  视口坐标，不是文档坐标 | 正常 | left right  元素左上角的x坐标 | top bottom  元素右下角的y坐标 |
| 正常 | width  元素的width大小 | height  元素的height大小 |
| IE | .right - .left  元素的width大小 | .bottom - ,top  元素的height大小 |
| 如果element.getBoundingClientRect()方法不存在  element的属性  element是具体的元素 |  | offsetLeft  相对于定位元素或表格单元等父元素的x坐标，即如果offsetParent不为null，则相对于父元素，如果为null，则相对于文档，如果需要获取文档坐标，则需要遍历元素的offsetParent,直至其属性为null，然后累加这些偏移量 | offsetTop |
|  | offsetWidth  **与**clientWidth不同的是该属性包含边框大小 | offsetHeight |
| event的属性  事件对象 |  | event..clientX  鼠标的视口X坐标 | event.clientY |

//一旦将模块代码封装进一个函数，就需要一些方法导出其共用API  
//以便在模块函数的外部调用它们  
//上面的例子中模块函数返回构造函数  
//这个构造函数随后赋值给一个全局变量  
//将值返回表明API已经导出在函数作用域之外  
//上面只是一个类，如果包含多个类等，则可以返回命名空间对象

**静态定位(static) ：**

      static，无特殊定位，它是html元素默认的定位方式，即我们不设定元素的position属性时默认的position值就是static，它遵循正常的文档流对象，对象占用文档空间，该定位方式下，top、right、bottom、left、z-index等属性是无效的。

/\*JavaScript高级编程的方法\*/  
//单例对象。

div#outer{  
 width: 100px;  
 height: 100px;  
 background: red;  
 position: relative;  
 /\*fixed absolute relative 都可行 正常文档流的static不行\*/  
}

**var** DragDrop = **function**(){ //返回的是一个对象的匿名函数  
  
 //私有变量  
 **var** dragging = **null**;  
 // var myDiv = document.getElementById("myDiv");  
 diffX = 0; //diffX必须是全局变量  
 diffY = 0;  
  
  
 //私有属性  
 **function** handlerEvent(event){  
 //获取事件和目标  
 event = eventUtil.getEvent(event);  
 **var** target = eventUtil.getTarget(event);  
  
 //确定事件类型  
 **switch** (event.type){  
 **case** "mousedown":  
 **if**(target.className.indexOf("draggable") > -1){ //按元素查找索引值,没有找到子字符串则返回-1  
 dragging = target;  
 diffX = event.clientX - target.offsetLeft;  
 diffY = event.clientY - target.offsetTop;  
 }  
 **break**;  
  
 **case** "mousemove":  
 **if**(dragging != **null**){  
  
 //指定位置  
 dragging.style.left = (event.clientX - diffX) + "px";  
 dragging.style.top = (event.clientY - diffY) + "px";  
 }  
 **break**;  
  
 **case** "mouseup":  
 dragging = **null**;  
 **break**;  
  
 **default** :  
 dragging = **null**;  
 **break**;  
 }  
 }  
  
  
 //公共接口  
 //特权、公有方法和属性  
 **return**{  
 enable:**function**(){  
 eventUtil.addEvent(document,"mousedown",handlerEvent); //使用document而不是div的原因是事件委托，详细看事件委托  
 eventUtil.addEvent(document,"mousemove",handlerEvent);  
 eventUtil.addEvent(document,"mouseup",handlerEvent);  
 },  
  
 disable:**function**(){  
 eventUtil.removeEvent(document,"mousedown",handlerEvent);  
 eventUtil.removeEvent(document,"mousemove",handlerEvent);  
 eventUtil.removeEvent(document,"mouseup",handlerEvent);  
 }  
 }  
}(); //立马执行  
  
  
DragDrop.enable(); //启动拖动

//自己写的拖动事件

//单例对象。  
**var** drag = (**function**(){  
  
 //变量初始化  
 **var** dragging = **null**; //一个需要拖动的对象  
 **var** diff = { //鼠标偏移element对象的像素值  
 x: 0,  
 y: 0  
 };  
  
  
 **function** handler(event){  
 **var** event = event || window.event;  
 **var** target = event.target || event.srcElement;  
  
 **switch**(event.type){  
 **case** 'mousedown':  
 //判断元素是否是一个可拖动对象  
 **if**(target.classList){  
 **if**(target.classList.contains('draggable')){  
 dragging = target; //正在拖动的对象是可拖动的对象  
 }  
 } **else if**(target.className.indexOf('draggable') !== -1){  
 dragging = target;  
 }**else**{  
 **break**;  
 }

//如果可拖动  
 **if**(dragging){  
 //计算鼠标偏移元素左上角的像素  
 //需要注意的是并不是真正的偏移左上角的像素  
 //event.clientX是视口坐标  
 //这里假设target.offsetParent = null;  
 //即target这个目标元素的父元素没有定位元素或者target不是表格单元  
 //所以target.offsetLeft是相对于文档左上角的偏移量  
 //所以是文档坐标  
 //真正的偏移量是 event.clientX + 滚动条偏移量 - target.offsetLeft  
 //这里暂且叫做相对偏移元素左上角的像素  
 diff.x = event.clientX - target.offsetLeft;  
 diff.y = event.clientY - target.offsetTop;  
  
 //添加事件  
 addEvent(document,'mousemove',handler);  
  
  
 }

break;  
  
  
 **case** 'mousemove':  
 **if**(dragging != **null**){  
  
 //移动元素  
 //元素的文档坐标 = 鼠标的视口坐标 - 相对偏移量  
 dragging.style.left = (event.clientX - diff.x) + 'px';  
 dragging.style.top = (event.clientY - diff.y) + 'px';  
  
 //添加事件  
 addEvent(document,'mouseup',handler);  
 }  
 **break**;  
 **case** 'mouseup':  
  
 dragging = **null**; //释放拖动对象  
  
 //取消事件  
 removeEvent(document,'mousemove',handler);  
 removeEvent(document,'mouseup',handler);  
 **break**;  
  
 **default**:  
 dragging = **null**;  
 **break**;  
 }  
 }  
  
  
 **function** addEvent(element,type,handler){  
 //事件委托  
 **if**(element.addEventListener){  
 element.addEventListener(type,handler,**false**);  
 } **else if**(element.attachEvent){  
 element.attachEvent('on'+type,handler);  
 } **else**{  
 element['on'+type] = handler;  
 }  
 }  
  
 **function** removeEvent(element,type,handler){  
 //取消委托  
 **if**(element.removeEventListener){  
 element.removeEventListener(type,handler,**false**);  
 } **else if**(element.detachEvent){  
 element.detachEvent('on'+type,handler);  
 } **else**{  
 element['on'+type] = **null**;  
 }  
 }  
  
  
 **return** {  
 enable:**function**(){  
 //事件委托给document  
 addEvent(document,"mousedown",handler);  
 },  
  
 disable:**function**(){  
 //取消委托  
 removeEvent(document,'mousedown',handler);  
 }  
 }  
  
  
  
})();  
  
  
drag.enable();  
drag.disable();

## 15.6 鼠标滚轮事件（未学）

## 15.7 拖放事件（未学）

## 15.8 文本事件（未学）

## 15.9 键盘事件（未学）

# 第十六章 脚本化HTTP

//ajax  
//异步XML和JavaScript  
//特点是使用脚本操纵HTTP和Web服务器进行数据交换 不会导致页面重载  
//Comet  
//与ajax相反，Web服务器发起通信并异步发送消息到客户端  
//如果Web应用需要响应服务端发送的消息，则它会使用Ajax技术发送或请求数据  
//在Comet中，服务端向客户端‘推’数据  
  
//<script>元素的src属性能设置URL并发起HTTP GET请求，使用<script>元素实现脚本  
//操纵HTTP是非常吸引人的，因为它们可以跨域通信而不受限于同源策略  
  
//通常使用基于<script>的Ajax传输协议时，服务器的响应产用JSON编码的数据格式  
//当执行脚本时，JavaScript解析器能自动将其‘解码’  
//由于使用JSON数据格式，因此这种Ajax传输协议也叫'JSONP'

## 16.1 使用XMLHttpRequest

/\*XMLHttpRequest\*/  
//浏览器在XMLHttpRequest类上定义了它们的HTTP API  
//这个类的每一个实例都表示一个独立的请求/响应对  
//这个对象的属性和方法允许指定请求细节和提取响应数据  
//使用HTTP API必须做的第一件事情就是实例化XMLHttpRequest对象  
  
**var** request = **new** XMLHttpRequest();  
//也可以重用已存在的XMLHttpRequest 但注意这将会终止之前通过该对象挂起的任何请求  
  
  
  
/\*一、指定请求\*/  
/\*一、指定请求\*/  
/\*一、指定请求\*/  
/\*一、指定请求\*/  
  
  
//一个HTTP请求分四步  
*/\*\*  
 \* 1.HTTP请求方法或‘动作’  
 \* 2.正在请求的URL  
 \* 3.一个可选的请求头集合（首部），其中可能包含身份验证信息  
 \* 4.一个可选的请求主体  
 \*/*/\*1.指定请求方法和URL\*/  
**var** request = **new** XMLHttpRequest();  
//创建XMLHttpRequest对象实例后，下一步是发起HTTP请求的下一步是调用  
//实例的open()方法去指定这个请求的两个必须部分:方法和URL  
request.open("GET", //请求方法  
 'test.html'); //请求的URL  
  
//第一个参数是HTTP请求或动作 不区分大小写，通常使用大写  
//GET 用于常规请求，它适用于当URL完全指定请求资源  
//当请求对服务器没有任何副作用（即数据不存储在服务器的数据库上）以及当服务器的响应是可缓存  
//POST方法通常用于HTML表单，  
//它在请求主体中包含额外数据（表单数据）且这些数据常存储到服务器上的数据库中（副作用）  
//相同URL重复请求POST请求从服务器得到响应可能不同，同时不应该缓存使用这个方法的请求  
  
//第二个参数如果指定绝对URL、协议、主机、端口通常必须匹配所有文档的对应内容  
//跨域请求通常会报错  
//（但是当服务器允许跨域请求时，二级XMLHttpRequest规范会允许它）  
  
/\*2.指定请求头\*/  
//注意如果有请求头的话 GET请求可以不设置请求头  
request.setRequestHeader("Content-Type","text/plain");  
//POST请求需要'Content-Type'头指定请求主体的MIME类型  
//相同的头调用setRequestHeader多次，新值不会取代之前指定的值  
//相反，HTTP请求将包含这个头的多个副本或这个头将指定多个值  
//有些头是不能指定的，为了防止伪造  
///可能包含身份验证，例如给open传递第三四五个参数，比如用户名和密码等后续讲解  
  
/\*3.指定可选的请求主体并向服务器发送\*/  
//如果是get请求，则没有请求主体 所以应该传递null或省略这个参数  
request.send(**null**);  
  
//POST请求通常拥有主体，同时要匹配使用setRequestHeader()指定的‘Content-Type’头  
//HTTP请求的各部分有指定的顺序，请求方法和URL首先到达，然后是请求头，然后是请求主体  
//XMLHttpRequest API的设计似乎使每个方法都将写入网络流，这意味着  
//setRequestHeader方法的调用必须在调用open()方法之后和send()方法之前  
//否则将抛出异常  
  
  
/\*总体思想\*/  
  
//用POST方法发送纯文本给服务器，并忽略服务器返回的任何响应  
**function** postMessage(msg)  
{  
 //一个请求实例，新请求  
 **var** xmlHttp = **new** XMLHttpRequest();  
 //请求的方法和URL  
 xmlHttp.open('GET','/test.html');  
 //请求头集合  
 xmlHttp.setRequestHeader('Content-Type','text/plain;charset=UTF-8');  
 //请求主体  
 xmlHttp.send(msg);  
 //请求完成，忽略任何响应和任何错误  
 //调用最后的send()方法才开始启动网络，启动请求  
 //但是它等待服务器的响应时并不阻塞  
 //这就是异步处理HTTP响应  
}  
  
  
  
  
  
/\*二、取得响应\*/  
/\*二、取得响应\*/  
/\*二、取得响应\*/  
/\*二、取得响应\*/  
  
//服务器返回的HTTP响应  
*/\*\*  
 \* 1.一个数字或文字组成的状态码，用来显示请求的成功和失败  
 \* 2.一个响应头集（首部）  
 \* 3.响应主体  
 \*/*//XMLHttpRequest实例对象的属性和方法  
*/\*\*  
 \* 1.status statusText属性  
 \* 以数字或者文本的形式返回HTTP状态码  
 \* 状态码  
 \* 100-199 接收的请求正在处理。  
 200-299 请求成功。  
 200 OK 请求成功并正常处理  
 204 No Content 请求处理成功,但是没有资源可返回  
 206 Partial Content 请求处理成功，范围请求，只返回其中的某部分  
 300-399 重定向  
 301 永久性重定向  
 302 临时性重定向  
 303  
 304 服务器资源未改变  
 400-499 客户端的错误  
 400 语义有误，当前请求无法被服务器理解。  
 401 当前请求需要用户验证  
 403 服务器已经理解请求，但是拒绝执行它。  
 404 服务器上没有请求的资源  
 500-599 服务器错误。  
 500 服务器错误  
 503 服务不可用  
  
  
 \* 2.使用getResponseHeader()和getAllResponseHeader()查询响应头  
 \* XMLHttpRequest会自动处理cookie，它会从返回集合中过滤掉cookie头  
 \*  
 \* 3.响应主体可以从responseText属性中得到文本形式的，从responseXML属性中得到Document形式的  
 \*  
 \*/*//XMLHttpRequest对象通常异步使用：发送请求后，send()方法立即返回 直到响应返回  
//否则前面列出的方法和属性都无效  
//为了在响应准备就绪时得到通知 必须监听XMLHttpRequest对象上的readystatechange事件  
//同时需要理解readyState属性  
  
//XMLHttpRequset的readyState值  
*/\*\*  
 \* UNSENT 0 open()尚未调用  
 \* OPENED 1 open()已调用  
 \* HEADERS\_RECEIVED 2 接收到头信息  
 \* LOADING 3 接收到响应主体  
 \* DONE 4 响应完成  
 \*/*//理论上每次readyState属性的改变都会触发readystatechange事件  
//但实际上当该属性值变为0或者1时可能没有触发readystatechange事件  
//当调用send()时，即使readyState仍处于OPENED状态，也通常触发它  
//某些浏览器在LOADING状态时触发多次事件来给出进度反馈  
//readyState属性值为4或服务器的响应完成时，所有浏览器都触发readystatechange事件  
//由于在响应完成之前也会触发事件，所以事件处理程序应该一直检验readState值  
  
//为了监听readystatechange事件，需要把事件处理程序设置为XMLHttpRequest对象的  
//onreadystatechange属性  
//当然也可以使用addEventListener() 或attachEvent()  
//但由于通常每个请求只需要一个处理程序，所以设置onreadystatechange更容易  
  
//获取HTTP响应的onreadystatechange  
  
//发出一个GET请求获得指定的URL的内容  
//当它响应成功到达，验证是否是纯文本  
//如果是，把它传递给指定回调函数  
  
**function** getText(url,callback){  
 **var** request = **new** XMLHttpRequest(); //创建新请求  
 request.open('GET',url); //指定请求方法和待获取的URL  
  
 //由于是get请求，忽略请求头  
  
 request.onreadystatechange = **function**(){ //定义事件处理程序  
 //如果请求完成  
 **if**(request.readyState === 4){ //服务器响应完成  
 **if**(request.status === 200){ //服务器请求成功  
 **var** type = request.getResponseHeader('Content-Type'); //获取响应头  
 **if**(type.match(/^text/)){ //确保响应是文本  
 callback(request.responseText);  
 }  
 }  
 }  
  
 };  
  
 request.send(**null**); //立即发送请求  
}  
  
  
/\*1.同步响应\*/  
  
//异步处理HTTP响应是最好的方式  
//然而XMLHttpRequest也支持同步响应  
//如果把false作为第三个参数传递给open() 那么send()方法将阻塞直到请求完成  
//在这种情况下不需要使用事件处理程序  
//一旦send()返回，仅需要检查XMLHttpRequest对象的status和responseText属性  
  
//getText的同步版本  
  
//发起同步的HTTP GET请求以获得指定的URL内容  
//返回响应文本 或如果请求不成功或响应不是文本就报错  
**function** getTextSync(url){ //少了一个参数callback  
 **var** request = **new** XMLHttpRequest();  
 request.open('GET',url,**false**); //如果是true或者省略就是异步响应  
 request.send(**null**);  
  
 **if**(request.status !== 200){  
 **return false**;  
 }  
  
 **var** type = request.getResponseHeader('Content-Type');  
 **if**(!type.match(/^text/)){  
 **return false**;  
 }  
 **return** request.responseText;  
}  
  
//同步请求应该避免它们,因为JavaScript是单线程的，当send()方法阻塞时，它通常会导致整个浏览器UI冻结  
//如果连接的服务器响应慢，那么用户的浏览器将冻结  
  
/\*2.响应解码\*/  
//XML  
//如果服务器发送XML或XHTML文档作为其响应，能通过responseXML属性获得一个解析形式的XML文档  
//这个属性的值是一个Document对象  
  
//JSON  
//如果服务器想发送诸如对象或数组这样的结构化数据作为其响应，它应该传输JSON编码的字符串数据  
//接收它时,可以把responseText属性传递给JSON.parse()  
//注意 JSON.stringify(obj); //将对象转化为JSON格式的字符串  
// JSON.parse(string); //将一个JSON字符串转化为JavaScript对象  
  
//解析HTTP响应  
//发起HTTP响应获取指定的URL内容  
//当响应到达时，解析为XML的Document对象 解析后的JSON对象  
//或字符串形式传递给回调函数  
  
**function get**(url,callback){  
 **var** request = **new** XMLHttpRequest();  
 request.open('GET',url);  
  
 request.onreadystatechange = **function**(){  
 **if**(request.readyState === 4){  
 **if**(request.state === 200){  
 **var** type = request.getResponseHeader('Content-Type');  
 **if**(type.indexOf('xml') !== -1 && request.responseXML){  
 callback(request.responseXML); //XML Document对象响应  
 } **else if**(type === 'application/json'){  
 callback(JSON.parse(request.responseText)); //JSON响应  
 } **else**{  
 callback(request.responseText); //字符串响应  
 }  
 }  
 }  
 };  
  
 request.send(**null**);  
}  
  
//检查该响应的'Content-Type'头专门处理'application/json'影响  
//可能还希望特殊编码的另一个响应类型'application/javascript' 或 'text/javascript'  
//可以使用XMLHttpRequest请求JavaScript脚本，然后执行全局eval()执行这个脚本  
//但是根本没必要  
//因为<script>元素本身操纵HTTP脚本的能力完全可以实现加载并执行脚本  
//<script>元素能发起跨域HTTP请求，而XMLHttpRequest API则禁止  
  
  
  
  
/\*额外话题,请求资源\*/  
*/\*\*  
 \* 1.访问网页时,发送HTTP请求，通常是GET请求，包括请求方法,请求URL,协议版本（例如HTTP/1.1）,可选的请求首部  
 \* 2.DNS解析域名  
 \* 3.服务器响应请求并返回 发送响应 包括响应协议版本，响应的状态码，可选的响应头部，响应主体等  
 \* 4.  
 \*/  
  
/\*\*  
 \* GET:获取资源  
 \* 用来请求访问已被统一资源标识符(URI)识别的资源 指定的资源经服务器解析后返回响应内容  
 \* 如果请求的资源是本文，保持原样返回  
 \* 如果是像CGI（通用网关接口）那样的程序，则返回经过执行后的输出结果  
 \*  
 \* POST:传输实体主体  
 \* GET方法也可以传输实体的主体，但一般不用GET方法进行传输 而是用POST方法  
 \* POST主要的目的并不是获取响应的主体内容  
 \*  
 \*/*/\*三、编码请求主体\*/  
/\*三、编码请求主体\*/  
/\*三、编码请求主体\*/  
/\*三、编码请求主体\*/  
  
/\*1.表单编码的请求\*/  
  
//HTTP POST请求包括一个请求主体 即request.send()中需要发送的主体  
  
//表单编码的请求 表单编码格式的MIME类型 application/x-www-form-urlencoded  
//此种类型的编码并不需要HTML表单 我们实际上并不需要直接使用表单  
  
//表单编码在WEB上如此广泛的使用，同时所有服务器的编程语言都能得到良好的支持  
//所以非表单的表单编码通常也是容易实现的事情  
  
//一个简单表单的编码像如下字符串  
//find=pizza&zipcode=02134&radius=1km 用&符号分开名/值对  
  
  
//模拟表单提交的Ajax,希望发送给服务器的可能是一个JavaScript对象  
//以上这个简单的表单编码变成JavaScript对象的表单编码形式可能是  
  
  
  
  
//以下函数将JS对象转换为 find=pizza&zipcode=02134&radius=1km  
*/\*\*  
 \* {  
 \* find:pizza,  
 \* zipcode:02134,  
 \* reaius = '1km'  
 \* }  
 \*/*//  
  
//data类似于如上的JavaScript对象  
//data 使用HTTP请求的编码对象  
**function** encodeFormData(data){  
 **if**(!data) **return** "";  
 **var** pairs = []; //为了保存 名=值 对  
  
 **for**(**var** name **in** data){  
 **if**(!data.hasOwnProperty(name)) **continue**; //如果不是自己的属性，是继承来的，跳过  
 **if**(**typeof** data[name] === 'function') **continue**; //跳过方法  
 **var** value = data[name].toString(); //转换为字符串  
 name = encodeURIComponent(name.place('%20',"+")); //+替换空格  
 value = encodeURIComponent(value.place('%20',"+")); //空格 URL中的空格可以用+号或者编码 %20  
 pairs.push(name + '=' + value);  
 }  
 **return** pairs.join('&'); //数组转换为字符串  
}  
  
  
//使用表单编码数据发起一个HTTP POST请求  
**function** postFormData(url,data,callback){  
 **var** request = **new** XMLHttpRequest();  
 request.open('POST',url);  
  
 request.onreadystatechange = **function**(){  
 **if**(request.readyState === 4){  
 **if**(request.state === 200){ //或者这个if可以省略  
 callback(requset);  
 }  
 }  
 };  
 request.setRequestHeader('Content-Type','application/x-www-form-urlencoded'); //设置Content-Type  
 request.send(encodeFormData(data)); //发送表单编码数据  
}  
  
  
//表单数据同样可以通过GET请求来提交  
//当提交表单的目的是为了执行只读查询 GET比POST更合适  
//GET没有请求主体，所以要发送的表单编码数据'负载'要作为URL（后跟一个问号）的查询部分  
  
//使用表单编码数据发起一个HTTP GET请求  
**function** getFormData(url,data,callback){  
 **var** request = **new** XMLHttpRequest();  
 request.open("GET", url+"?"+encodeFormData(data));  
  
 request.onreadystatechange = **function**(){  
 **if**(request.readyState === 4){  
 **if**(request.state === 200){ //或者这个if可以省略  
 callback(request);  
 }  
 }  
 };  
  
 request.send(**null**);  
}  
  
  
  
/\*2.JSON编码请求\*/  
//近年来，作为WEB交换格式的JSON已经得到普及  
  
  
//使用JSON编码主体发起一个HTTP POST请求  
//data 使用HTTP请求的编码对象  
**function** postJSON(url,data,callback){  
 **var** request = **new** XMLHttpRequest();  
  
 //url统一资源定位符  
 request.open('POST',url);  
 request.onreadystatechange = **function**(){  
 **if**(request.readyState === 4){  
 callback(request);  
 }  
 };  
 request.setRequestHeader('Content-Type','application/json');  
 request.send(JSON.stringify(data)); //将data对象转换为JSON格式的字符串  
}  
  
  
/\*3.multipart/form-data请求\*/  
  
//如果HTML表单同时包含文件上传元素和其他元素时  
//浏览器不能使用普通的表单编码而必须使用  
//称为 'multipart/form-data'的特殊Content-Type来用POST方法提交表单  
  
//XHR2定义了新的FormData对象  
//然后多次调用这个对象的append()方法把个体（可以是字符串、File或Blob对象）添加到请求中  
//可以省略设置Content-Type头，send()会自动设置Content-Type头  
**function** postFormData2(url,data,callback){  
 **if**(**typeof** FormData === 'undefined'){ //如果FormData不存在，该方法不能用  
 **return false**;  
 }  
  
 **var** request = **new** XMLHttpRequest();  
 request.open('POST',url);  
  
 request.onreadystatechange = **function**(){  
 **if**(request.readyState === 4 && callback){  
 callback(request);  
 }  
 };  
  
 **var** formData = **new** FormData;  
 **for**(**var** name **in** data){  
 **if**(!data.hasOwnProperty(name)) **continue**;  
 **if**(**typeof** data[name] === 'function') **continue**;  
 formData.append(name,data[name]);  
 }  
  
 //无需设置Content-Type send会自动设置这个首部  
 request.send(formData);  
}  
  
  
  
/\*4.HTTP进度事件\*/  
  
/\*5.中止请求和超时\*/  
  
//可以通过调用XMLHttpRequest对象的abort()方法来取消正在进行的HTTP请求  
**function** timedGetText(url,timeout,callback){  
 **var** outed = **false**; //超时标志  
 **var** request = **new** XMLHttpRequest();  
  
 **var** timeID = setTimeout(**function**(){  
 outed = **true**; //超时  
 request.abort(); //中止HTTP请求  
 },timeout);  
  
 request.onreadystatechange = **function**(){  
 **if**(request.readyState !== 4) **return**; //如果请求没有完成则返回 请求有很多个阶段，详细看前面说明  
 **if**(outed) **return**; //超时则返回  
 clearTimeout(timeID); //取消超时调用 即取消setTimeout中的回调函数  
 **if**(request.status === 200){  
 callback(request);  
 }  
 };  
   
 request.send(**null**);  
}

## 16.2 跨域HTTP请求

/\*跨域HTTP请求\*/  
  
  
//同源策略  
//文档必须来源于相同的协议、主机、端口、  
//否则不同来源的文档就不能互相通信  
  
  
  
//因为同源策略,浏览器不允许原始脚本查找跨域文档的内容  
//使用XMLHttpRequest 文档内容都是通过responseText属性暴露  
//所以同源策略不允许XMLHttpRequest进行跨域请求  
//需要注意的是<script>标签的src属性并未真正受限于同源策略  
//它加载并执行任何来源的脚本  
  
//跨域请求的灵活性使得<script>元素成为取代XMLHttpRequest的主流Ajax传输协议

/\*CORS(Cross-Origin Resource Sharing 跨域资源共享)\*/  
//插入话题CORS(跨域资源共享) 允许跨域访问网站 IE8中是XDomainRequest支持  
//如果浏览器支持XMLHttpRequest的CROS且实现跨域请求的网站（服务器）启动了CORS允许跨域请求  
//那么同源策略将不放宽而跨域请求就会正常工作  
  
//安全细节  
//1. 如果给XMLHttpRequest的open()方法传入用户名和密码，即进行身份验证  
//那么它们绝对不会通过跨域请求发送 （有可能被分布式密码攻击破解）  
//2. 跨域请求通常也不会包含任何的用户证书 cookie和HTTP身份验证令牌（token）不会作为请求的内容发送  
//且任何跨域响应接收的cookie都会丢弃  
//3. 必须在用send()发送请求前设置XMLhttpRequest的withCredentials (Credentials 得到信用)  
//属性为true，并且测试withCredentials的存在性是测试浏览器是否支持CORS的一种方法  
//查询有href属性但是没有title属性的所有a元素  
//注册onmouseover事件处理程序  
//这个程序使用XMLHttpRequest HEAD请求取得链接资源的详细信息  
//然后把这些详细信息设置为链接的title属性  
  
eventUtil.addEvent(window,'load',CORSHandler);  
  
  
  
**function** CORSHandler(){  
 //查看是否支持CORS  
 **var** isSupportsCORS = ((**new** XMLHttpRequest()).withCredentials !== undefined);  
  
 //遍历文档中的所有链接  
 **var** links = document.getElementsByTagName('a');  
  
 **for**(**var** i=0,len=links.length; i<len; i++){  
 **var** link = links[i];  
 **if**(link.title) **continue**; //过滤有title的link  
 **if**(link.href) **continue**; //跳过超链接中的锚点  
  
 //如果这是一个跨域链接  
 **if**(link.host !== location.host || link.protocol !== location.protocol){  
 link.title = "站外链接"; //假设不能跨域访问得到更多信息  
 **if**(!isSupportsCORS){  
 **continue**; //如果没有CORS支持就退出  
 }  
 }  
  
 **if**(link.addEventListener){  
 link.addEventListener('mouseover',mouseOverHandler,**false**);  
 } **else**{  
 link.attachEvent('onmouseover',mouseOverHandler);  
 }  
 }  
  
  
 **function** mouseOverHandler(e){  
 **var** event = e || window.event;  
 **var** target = event.target || event.srcElement;  
  
  
 **var** url = target.href; //注意这个站点是访问不是本站点 是跨域的站点  
  
 **var** req = **new** XMLHttpRequest();  
 req.open('HEAD',url); //跨域访问  
  
 req.onreadystatechange = **function**(){  
 **if**(req.readyState !== 4) **return**;  
 **if**(req.status === 200){  
 **var** type = req.getResponseHeader("Content-Type");  
 **var** size = req.getResponseHeader("Content-Length");  
 **var** date = req.getResponseHeader("Last-Modified");  
  
 target.title = "类型:" + type + "\n" + "大小:" + size + "\n" +  
 "时间:" + date;  
  
 }  
 };  
  
 req.send(**null**);  
  
 //如果只是想一次获取这些头信息  
 **if**(link.removeEventListener){  
 link.removeEventListener('mouseover',mouseOverHandler,**false**);  
 } **else**{  
 link.detachEvent('onmouseover',mouseOverHandler);  
 }  
 }  
}

## 16.3 跨域访问之JSONP(重点)

/\*跨域HTTP请求\*/  
  
  
  
/\*借助<script>发送HTTP请求：JSONP(JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。)\*/  
  
//<script>元素可以作为一种Ajax传输机制  
//只需设置<script>元素的src属性 （假如它还没插入到document中，需要插入进去）  
//然后浏览器就会发送一个HTTP请求以下载src属性所指向的URL  
//使用<script>元素进行Ajax传输的一个主要原因是，它不受同源策略的影响  
//因此可以使用它们从其他的服务器请求数据  
//第二个原因是包含JSON编码数据的响应体会自动解码  
//这种使用<script>元素作为Ajax传输的技术称为JSONP  
//若HTTP请求所得到的响应是经过JSON编码的，则适合使用该技术  
//P代表填充 或 前缀  
  
//假设你写了一个服务，它处理GET请求并返回JSON编码的数据  
//同源的文档可以在代码中使用XMLHttpRequest和JSON.parse()  
//假如在服务器上启动了CORS,在新的浏览器下，跨域的文档也可以使用XMLHttpRequest享受该服务  
//如果不支持CORS,在一些旧的浏览器下面 跨域文档只能通过<script>元素访问这个服务  
//使用JSONP,JSON响应数据是合法的JavaScript代码，当它到达时浏览器将执行它  
  
//当通过<script>调用数据时 响应内容必须用JavaScript函数名和圆括号包裹起来  
//如果请求响应，发送的的JSON响应如下  
  
handlerResponse(  
 [1,2,{"buckle":"my shoe"}]  
);  
//解释一下，这里把它看作一个可执行的函数  
//handlerResponse是函数名，然后执行了这个带参数的函数  
//这样请求响应后，这段代码会成为<script>的内容 它先判断JSON编码后的数据  
//然后把它传递给handlerResponse()函数  
  
  
  
  
  
  
  
//为了可行起见，我们必须通过某种方式告诉服务，它正在从一个<script>元素调用  
//必须返回一个JSONP响应  
//例如在url中添加一个查询参数来实现 比如追加 ?json（或&json）  
  
//在实际中，支持JSONP的服务（服务器中）不会强制指定客户端必须实现的回调函数名称  
//比如上面的handlerResponse  
  
//可以由客户端设置查询参数的值，来指定一个函数名  
//然后使用函数名去填充响应  
  
//  
  
  
//根据指定的url发送一个JSONP请求  
//然后把解析得到的响应数据传递给回调函数  
//在url中添加一个名为jsonp的查询参数 用于指定该请求的回调函数的名称  
  
  
getJSONP.counter = 0; //计数器  
  
**function** getJSONP(url,callback){  
  
 **var** cbnum = "cb" + getJSONP.counter++;  
 **var** cbname = "getJSONP." + cbnum; //作为JSONP函数的属性  
  
 //将回调函数名称以表单编码的形式添加到URL的查询部分中  
 //使用JSONP作为参数名  
 //可能使用其他参数 例如callback  
 **if**(url.indexOf("?") === -1){ //url没有查询部分  
 url += "?jsonp=" + cbname; //作为查询部分添加参数  
 } **else**{  
 url += "&jsonp=" + cbname; //作为新的参数添加它  
 }  
  
 //创建<script>参数用于发送请求  
 **var** script = document.createElement('script');  
  
 //定义将被脚本执行的回调函数  
 //回调函数作为getJSONP()函数的一个属性存储起来  
 getJSONP[cbnum] = **function**(response){  
 **try**{  
 callback(response); //处理响应数据  
 }  
  
 **finally**{ //即使回调函数或响应抛出错误  
 **delete** getJSONP[cbnum]; //删除该回调函数  
 script.parentNode.removeChild(script); //移除script元素  
 }  
 };  
  
 //立即触发HTTP请求  
 script.src = url; //设置脚本的URL  
 document.body.appendChild(script); //把它添加到文档中,正是因为插入了文档所以触发了一个HTTP请求  
}  
  
  
  
//重新理解  
  
//1.首先<script>元素可以作为一种Ajax传输机制  
//怎么理解这种机制呢，只要在文档的<script>中加载一个立即执行的函数，并且这个函数的参数是JSON编码格式  
//并且这个函数在当前文档的脚本中已经定义了，那么只要跨域去服务器请求这个<script>的带JSON编码格式的立即执行的函数  
//把它加载到当前的文档中，那么这个立即执行的函数就成为了一个请求响应后的回调函数，它将跨域请求来的JSON格式的数据作为参数  
//进行执行  
  
//2.<script>元素的src属性不受同源策略的限制，所以可以进行跨域请求数据  
  
//3.需要注意的是请求的是一段代码，这段代码是一个实参是带JSON编码格式的立即执行的函数  
//例如 handlerResponse([1,2,{"buckle":"my shoe"}]);  
//只要当前文档的脚本中存在这个handlerResponse函数，就可以立即执行这个函数了  
  
  
//注意请求的是一段合法的JavaScript代码，这段代码到达时浏览器立即执行它  
//需要注意的是返回的是一个立即执行的函数代码，  
//而不是JSON数据，如果只是JSON数据的话没有执行那么这段数据并不能立马实时的得到应用  
  
//4.这个是一个动态的<script>标签，当我们需要数据的时候我们先动态的创建这个<script>标签  
//然后设置这个<script>标签需要访问的url,当我们把它插入文档中时这个<script>就向服务程序  
//发送了一个HTTP GET请求，请求的是一段JavaScript代码，这段代码的实参也就是响应的数据是经过  
//JSON格式编码的，然后立即执行这个函数（假设文档中已经存在了这个定义过的函数）,然后执行完  
//后也就是请求了数据并使用了数据后，从文档中移除这个<script>标签  
  
  
  
  
  
  
  
//脚本的安全性  
//为了使用<script>元素进行Ajax传输  
//必须允许Web页面可以执行远程服务器发送过来的任何JavaScript代码  
//对于不可信的服务器，不应该采用JSONP技术，当与可信的服务器通信时，要提防攻击者可能进入服务器中  
//然后黑客会接管你的网页，运行他自己的代码  
//并显示他想要的内容

## 16.4 try catch finally

*/\*\*  
 \* 1、不管有木有出现异常，finally块中代码都会执行；  
 2、当try和catch中有return时，finally仍然会执行；  
 3、如果catch和try中有return，finally是在return后面的表达式运算后（待决操作）执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，不管finally中的代码怎么样，返回的值都不会改变，任然是之前保存的值），所以函数返回值是在finally执行前确定的；  
 4、finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值。  
 \*/  
  
  
/\*\*  
 ry-catch-finally 的存在使得控制流转移问题比看起来要复杂一点点。我们先从实际代码建立直观认识，然后以 ECMA-262 中的描述来加深理解。  
  
 首先 finally 具有「覆盖」控制流转移操作的能力——用新的控制流操作取代「待决」的控制流转移操作。如下所示：  
 \*/***function** test () {  
 **try** {  
 // 下面的 return 语句想要做控制流转移，然而  
 // 由于此处代码处于 try 块中，并且存在相应的 finally 块  
 // 因此编译器知道当前控制流转移操作不应当立即完成  
 // 应当先执行完 finally 块之后再做决定  
 // 于是当前控制流转移只是简单的先记录下来  
 // 作为「待决」(pending) 操作  
 **return** 1;  
 }  
 **finally** {  
 // finally 块中如果出现了新的控制流转移操作  
 // 那么将丢弃之前的「待决」操作，不再执行  
 // 而仅仅执行此处新的控制流转移操作  
 // 我们可以简单的将这一行为理解为 finally 具有  
 // 「覆盖」(override)控制流转移操作的能力  
 **return** 2;  
 }  
}  
  
// 显示 2  
write(test ()); //2  
  
write("");//空行  
  
  
  
//其次，这种「覆盖」是可嵌套的：  
**function** test1 () {  
 **try** {  
 **try** {  
 **return** 1;  
 }  
 **finally** {  
 // 丢弃 return 1 操作  
 // 改为执行 return 2 操作  
 **return** 2;  
 }  
 }  
 **finally** {  
 // 丢弃 return 2 操作  
 // 改为执行 return 3 操作  
 **return** 3;  
 }  
}  
  
// 显示 3  
write(test1 ());  
  
write("");  
  
//另外，要注意到除了 return 外，break、continue、throw 也都属于控制流转移操作  
//因此 finally 块也能处理它们，下面的代码演示了这一点：  
  
**function** test2(){  
 **for** (**var** i = 0; i < 3; ++ i) {  
 **try** {  
 write(i);  
 **break**;  
 }  
 **finally** {  
 // 将原本的 break 操作扭转为 continue 操作  
 **continue**;  
 }  
 }  
}  
  
test2(); // 0 1 2  
write("");  
  
  
//也许这一点会令你略感惊讶——在不使用catch块的情况下， throw抛出的异常也能够被扭转：  
**function** test3() {  
 **try** {  
 **throw new** Error('oops')  
 }  
 **finally** {  
 // 注意我仅仅是使用 finally 块  
 // 而没有用任何 catch 块就将一个  
 // 本该抛出的异常硬生生丢弃了！  
 **return** 3;  
 }  
}  
  
  
  
write(test3()); //3  
write("");  
  
  
*/\*\*  
 \* 好了，到这里，我们已经从代码层面直观的认识了 finally 块对控制流的「覆盖」能力。  
 接下来我会从 ECMA-262 的角度做更进一步的解释。  
  
 按照 ECMA-262 规范（以下简称「规范」）的描述，try-catch-finally 语句实际上有三种可能的形式：  
 // 完整的 try-catch-finally 链条  
 try {} catch(e) {} finally {}  
  
 // 没有 finally  
 try {} catch(e) {}  
  
 // 没有 catch  
 try {} finally {}  
  
 try-finally 形式的语句按照下述过程执行：  
 先执行 try 块中的语句，一直到结束或者遇到控制流转移请求。无论何种情况，都将其状态记录下来，记为 B  
 再执行 finally 块中的语句，一直到结束或者遇到控制流转移请求。无论何种情况，都将其状态记录下来，记为 F  
 检查 F，如果它并不是一个新的控制流转移请求，那么忽略 F 执行 B 即可  
 否则，忽略 B 执行 F  
 \*/***var** s = "1";  
**try**{  
 s += "2";  
} **catch**(e){  
 s += "3";  
} **finally**{  
 s += "4";  
}  
  
write(s); //124 catch没有执行  
  
**var** s1 = "1";  
**try**{  
 s1 += "2";  
 **throw new** Error('999999');  
} **catch**(e){  
 s1 += "3";  
} **finally**{  
 s1 += "4";  
}  
  
write(s1); //1234  
  
  
**var** s2 = "1";  
**try**{  
 **throw new** Error('999999');  
 s2 += "2"; //此语句不执行  
} **catch**(e){  
 s2 += "3";  
} **finally**{  
 s2 += "4";  
}  
  
write(s2); //134  
  
  
  
  
  
**function get**(){  
 **var** s1 = "1";  
  
 **try**{  
 s1 += "2";  
 **return** s1; //说明return是 12  
 //这是一个待决(pending) 操作 于是当前控制流转移只是简单的先记录下来  
 //finally没有出现新的控制流转移操作(break continue return throw) 所以执行这个待决操作  
 //之前s1的值12和这个操作一样被保留了下来，并不会显示124 但是要注意最终的s1的值是124  
 } **catch**(e){  
 s1 += "3";  
 } **finally** {  
 s1 += "4";  
 //alert(s1); //124 finally是执行的，但是返回12  
 }  
  
 **return** 10; //此句不会执行  
}  
  
  
write(**get**()); //12  
  
  
  
**function** get1(){  
 **var** s1 = "1";  
  
 **try**{  
 s1 += "2";  
 **return** s1; //词句是待决操作  
  
 } **catch**(e){  
 s1 += "3";  
 } **finally** {  
 s1 += "4";  
 **return** s1; //124 finally是执行的 由于finally出现新的控制流转移操作，所以抛弃待决操作  
 }  
  
 **return** 10; //此句不会执行  
}  
  
  
write(get1()); //124  
  
  
  
**function** get2(){  
 **var** s1 = "1";  
  
 **try**{  
 s1 += "2";  
 **throw new** Error('1');  
  
 } **catch**(e){  
 s1 += "3";  
 **return** s1; //同样是待决操作  
 } **finally** {  
 s1 += "4";  
 **return** s1; //124 finally是执行的 由于finally出现新的控制流转移操作，所以抛弃待决操作  
 }  
  
 **return** 10; //此句不会执行  
}  
  
  
write(get2()); //1234  
  
  
**function** get3(){  
 **var** s1 = "1";  
  
 **try**{  
 s1 += "2";  
 **throw new** Error('1');  
 **return** s1;  
  
 } **catch**(e){  
 s1 += "3";  
 **return** s1; //同样是待决操作 但是返回的是123 因为执行的是待决操作  
 } **finally** {  
 s1 += "4"; //此举执行了 s1 = 1234  
 }  
  
 **return** 10; //此句不会执行  
}  
  
  
write(get3()); //123

## 16.5 基于服务器端推送事件的Comet技术

/\*基于服务器端推送事件的Comet技术\*/  
  
  
//在服务器端推送事件的标准草案中定义了一个EventSource对象  
//简化了Comet应用程序在编写可以传递一个URL给EventSource()构造函数 然后返回实例上监听消息事件  
  
**var** ticker = **new** EventSource("/chat"); //传递的是一个url  
  
ticker.onmessage = **function**(e){  
 **var** type = e.type;  
 **var** data = e.data;  
 //处理事件类型和事件的字符串数据  
};  
  
//服务器发送数据给客户端时,就会触发这个onmessage事件，data是服务器发送给客户端的数据  
//type的默认值是message  
  
//服务器端推送事件的协议很简单 客户端（创建一个EventSource对象时会）建立一个到服务器的连接  
//服务器保持这个连接处于打开状态  
  
//Comet架构的一个常见应用是聊天应用  
//聊天客户端通过XMLHttpRequest向聊天室发送新的消息  
//也可以通过EventSource对象订阅聊天消息  
  
  
  
//通过EventSource的简易聊天客户端  
  
window.onload = **function**(){  
 //UI细节  
 **var** nick = prompt('Enter your nickname');  
 **var** input = document.getElementById('input');  
 input.focus(); //设置键盘焦点  
  
 //通过EventSource注册新消息通知  
 **var** chat = **new** EventSource("/chat");  
 chat.onmessage = **function**(e){  
 **var** msg = e.data;  
 **var** node = document.createTextNode(msg); //创建一个文本节点,放入服务器端推送的数据  
 **var** div = document.getElementById('div'); //创建一个容器  
 div.appendChild(node); //将文本插入容器  
 document.body.insertBefore(div,input); //该方法在新节点的父节点上调用，第一参数待插入节点，第二参数是插入节点之后的已存在节点  
 };  
  
 //使用XMLHttpRequest把用户的消息发送给服务器 注意这里不需要监听事件的回调函数了  
 input.onchange = **function**(){ //用户完成输入  
 **var** msg = nick + ":" + input.value;  
 **var** req = **new** XMLHttpRequest();  
 req.open('POST',"/chat");  
 req.setRequestHeader('Content-Type','text/plain,charset=UTF-8');  
 req.send(msg);  
 input.value = "";  
 };  
};  
  
  
  
//如果浏览器不兼容EventSource  
//使用XMLHttpRequest模拟  
  
window.onload = **function**(){  
 //如果EventSoucre不存在  
 **if**(window.EventSource === undefined){  
 window.EventSource = **function**(url){  
 **var** req;  
 **var** aborted; //设置为true表示放弃连接  
 **var** type = **null**; //检查响应类型  
  
 //创建一个XHR对象  
 **if**(window.XMLHttpRequest){  
 req = **new** XMLHttpRequest();  
 } **else if**(window.ActiveXObject){ //IE  
 req = **new** ActiveXObject('Microsoft.XMLHTTP');  
 }  
  
 //定义事件处理程序  
 req.readystatechange = **function**(){  
 **switch**(req.readyState){  
 **case** 3: processData(); **break**; //如果在下载过程中会为readyState=3时触发readystatechange事件  
 **case** 4: reconnect(); **break**;  
 }  
 };  
  
  
 //触发一个永久连接  
 connect();  
  
 **function** connect(){  
 type = **null**;  
 req.open('GET',url);  
 req.setRequestHeader('Content-Type',"no-cache"); //所有的内容都不会缓存在网页中  
 req.send();  
 }  
  
  
 //如果连接正常关闭，等待一秒钟尝试连接  
 **function** reconnect(){  
 **if**(aborted) **return**; //如果为ture 表示不连接  
 **if**(req.status >= 300) **return**; //在报错之后不进行重连操作  
 setTimeout(connect,1000);  
 }  
  
 **function** processData(){  
 **if**(!type){ //如果没有准备好,先检查响应类型  
 type = req.getResponseHeader('Content-Type');  
 **if**(type != 'text/event-stream'){  
 aborted = **true**; //放弃重新连接  
 req.abort();   
 **return**;  
 }  
 }  
   
 //接下来是数据处理部分  
 }  
 };  
  
 }  
};  
  
  
  
//服务器端设计  
  
//当客户端发起一个URL为/chat的GET请求时，会用一个数组来保存响应数据流并保持连接处于打开状态  
//当客户端针对POST请求时，它会将响应的主体部分作为一条聊天消息使用并写入数据  
//将接收到的数据发送到每个已打开的响应数据流上（客户端）上  
//需要注意的是每20s发送一条注释到客户端 这样它们就不会关闭连接再重连了  
  
  
//这里是需要发送的静态文件  
**var** clientUi = require('fs').readFileSync('test.html');  
**var** js = require('fs').readdirSync('test.js');  
  
//每20s发送一条注释到客户端  
setInterval(**function**(){  
 clients.forEach(**function**(client){  
 client.write(":ping?n");  
 });  
},20000);  
  
  
//创建一个新的服务器  
**var** server = **new** http.Server();  
  
//当接收到一个新的请求  
server.on('request',**function**(request,response){  
 //解析请求的url  
 //如果是/ 发送静态文件clientUi和js  
   
 //如果是/chat之外的路由 ，发送404  
   
 //否认这如果是post  
 **if**(request.method === 'POST'){  
 //获取数据  
 request.on('data',**function**(data){  
  
 });  
  
 //请求完成时，广播数据  
 request.on('end',**function**(){  
 //发送200状态吗  
 //广播数据  
 });   
 } **else**{ //如果是get  
   
   
   
 //如果客户端关闭了连接，  
 request.connect.on('end',**function**(){  
   
 //从数组中去掉客户端对象  
 });  
   
   
 //将响应的客户端对象加入数组  
   
   
 }  
   
   
   
   
});  
  
  
//启动服务器 监听8000端口  
server.listen(8000);

# 第十七章 客户端存储

## 17.1 Web存储

/\*客户端存储\*/  
  
  
//客户端存储遵循 '同源策略'  
  
*/\*\*  
 \* WEB存储: localStorage 和 sessionStorage  
 \* cookie:每一次HTTP请求都会把数据传输到服务器端，适合存储少量文本数据  
 \* IE UserData  
 \* 离线Web应用  
 \* Web数据库  
 \* 文件系统API  
 \*/*/\*Web存储\*/  
//window对象的属性 window.localStorage   
//localStorage 和 sessionStorage 存储的有效期和作用域不同  
  
**var** name = localStorage.username;  
name = localStorage['username'];  
  
**if**(!name){  
 name = prompt('what is your name');  
 localStorage.username = name;  
}  
  
//JSON存储  
**var** data = {  
 name:'vicor',  
 age:23  
};  
  
localStorage.data = JSON.stringify(data);  
  
**var** l\_data = JSON.parse(localStorage.data);  
  
  
//localStorage存储的数据是永久有效的  
//localStorage的作用域是限定在文档源级别的 文档源通过协议、主机名、端口三者来确定  
//同源的文档间共享同样的localStorage数据  
  
//sessionStorage有效期和存储数据的脚本所在的最顶层窗口或者是浏览器标签页是一样的  
//一旦窗口或者标签页被永久关闭了，那么所有的数据也被删除了，  
//需要注意的是当前浏览器具备重新打开最近关闭的标签页恢复上一次浏览器的会话功能，所以有效期可能更长  
//sessionStorage的作用域也限定在文档源中 不仅如此，sessionStorage的作用域还限定在窗口中  
//如果同源的文档渲染在不同的浏览器标签页中，那么它们互相之间拥有的是各自的sessionStorage数据，无法共享  
//哪怕两个标签页是同一个页面，运行的是同一个脚本也不行  
//需要注意的是基于窗口作用域说的是顶级窗口，如果一个浏览器的标签页包含两个<iframe>元素，它们所包含的文档是同源的  
//可以共享数据  
  
/\*存储API\*/  
  
localStorage.setItem('x',5); //以'x'的名字存储一个值  
**var** x = localStorage.getItem('x');  
alert(x); //5  
  
//枚举所有存储的名值对  
**for**(**var** i=0;i<localStorage.length;i++){ //length表示所有名/值对的总数  
 **var** name1 = localStorage.key(i);  
 **var** value = localStorage.getItem(name1);  
 write(name1 + ":" + value);  
}  
//data:{"name":"vicor","age":23}  
//x:5  
  
  
localStorage.removeItem("x");  
  
//枚举所有存储的名值对  
**for**(**var** i=0;i<localStorage.length;i++){ //length表示所有名/值对的总数  
 **var** name1 = localStorage.key(i);  
 **var** value = localStorage.getItem(name1);  
 write(name1 + ":" + value);  
}  
  
//data:{"name":"vicor","age":23}  
  
localStorage.setItem('x',5); //以'x'的名字存储一个值  
localStorage.clear(); //全部删除  
write(localStorage.length); //0

## 17.2 cookie

/\*cookie\*/  
  
//读写Document对象的cookie属性  
//document.cookie  
  
  
//cookie是指WEB浏览器存储的少量数据，cookie最早设计为被服务端所用  
//从最底层来看，作为HTTP协议的一种扩展实现它  
//cookie数据会自动在WEB浏览器和WEB服务器之间传输  
//cookie技术通过在请求和响应报文中写入Cookie信息来控制客户端的状态  
//服务器端可以读写存储在客户端的cookie值  
  
write(navigator.cookieEnabled); //true 允许使用cookie 并不是所有浏览器都实现了  
  
//cookie有效期和作用域  
//cookie默认的有效期很短暂，它只能持续在WEB浏览器的会话期间  
//一旦用户关闭浏览器，cookie保存的数据就丢失了  
//和sessionStorage的作用域不同的是，cookie的作用域并不是局限在浏览器的单个窗口中  
//它的有效期和整个浏览器的进程的有效期一致  
//如果想要延长cookie的有效期，可以通过设置max-age属性  
//一旦设置了有效期，浏览器就会将cookie数据存储在一个文件中，直到过了指定有效期才会删除该文件  
  
//作用域通过文档源和文档路径来确定  
//默认情况下cookie的作用域是通过文档源和文档路径来确定的  
//主要和它创建的WEB页面有关，并对该WEB页面以及和该WEB页面同目录或者子目录的其他WEB页面可见  
//例如在http://www.example.com/catalog/order.html中创建了cookie  
//在http://www.example.com/catalog/index.html是可见的  
  
//作用域通过cookie的path和domain属性是可以配置的  
  
//将cookie的路径path属性设置成 '/'等于是让cookie和localStorage拥有同样的作用域  
//比如设置成 /catalog 那么对于 http://www.example.com/catalog/order.html和  
//http://www.example.com/catalog/wid/order.html都是可见的  
  
//cookie的作用域默认由文档源限制  
//大型网站的子域之间能够互相共享cookie,是因为可以设置cookie的domain属性  
//比如order.example.com域下的服务器想要读取catalog.example.com域下的服务器  
//比如在order.example.com域下创建了一个cookie，设置path属性为 / domain属性为.example.com  
//那么该cookie在其他子域下也是可见的  
  
//domain默认是当前WEB服务器的主机名  
//cookie的域只能设置为当前服务器的域  
  
//属性secure是布尔类型的属性，默认以不安全的形式传递的（http协议）  
//一旦标识为安全的，只能当服务器和浏览器通过https或者其他的安全协议连接才能传递它  
  
  
//局限性，部分浏览器对单个cookie大小仍然有4KB的限制  
//现代浏览器语序cookie总数超过300个  
  
//保存cookie  
**var** name = 'victor';  
**var** age = 23;  
  
**var** cookie = name + "=" + encodeURIComponent(age);  
  
cookie += ';max-age=' + (2\*60\*60\*24); //2天 单位是秒 60\*60\*24是一秒  
//如果想要设置其他值  
//追加 ;path=path ;domain=domain ;secure=secure  
  
  
//读取cookie  
**function** getCookie() {  
 **var** cookies = {};  
 **var** all = document.cookie;  
  
 **if** (all === "") {  
 **return** cookies;  
 }  
  
 **var** list = all.split(",");  
 **for** (**var** i = 0, len = list.length; i < len; i++) {  
 **var** cookie = list[i];  
 **if** (cookie.indexOf("=") !== = -1) {  
 **var** name = (cookie.split("="))[0];  
 **var** value = (cookie.split("="))[1];  
 value = decodeURIComponent(value);  
 cookies[name] = value;  
 }  
 }  
   
 **return** cookies

;  
}   
//实现基于cookie存储的API  
//构造函数  
**function** cookieStorage(maxage,path){  
 **var** cookie = (**function**(){  
 **var** cookies = {};  
 **var** all = document.cookie;  
  
 **if** (all === "") {  
 **return** cookies;  
 }  
 **var** list = all.split(",");  
 **for** (**var** i = 0, len = list.length; i < len; i++) {  
 **var** cookie = list[i];  
 **if** (cookie.indexOf("=") !== = -1) {  
 **var** name = (cookie.split("="))[0];  
 **var** value = (cookie.split("="))[1];  
 value = decodeURIComponent(value);  
 cookies[name] = value;  
 }  
 }  
 **return** cookies;  
 }());  
  
  
  
 //将所有cookie的名字存储到一个数组中 私有属性  
 **var** keys = [];  
 **for**(**var** key **in** cookie){  
 keys.push(key);  
 }  
  
  
 //存储cookie的个数 公有属性  
 **this**.length = keys.length;  
  
  
 //公有方法  
 **this**.getItem = **function**(name){  
 **return** cookie[name] || **null**;  
 };  
  
  
 **this**.key = **function**(n){  
 **if**(n<0 || n > keys.length) **return null**;  
 **return** keys[n];  
 };  
  
  
 **this**.setItem = **function**(name){  
  
  
 //忽略代码  
  
 **this**.length ++;  
  
  
 };  
  
 **this**.removeItem = **function**(){  
 //忽略代码  
  
 **this**.length --;  
  
 };  
  
  
 **this**.clear =**function**(){  
  
  
 //忽略代码  
 **this**.length = 0;  
 }  
  
}

## 17.3 IE userData

/\*IE userData持久化数据\*/  
  
//有效期 除非手动删除，否则永久有效  
//也可以设置expires属性来设置它的过期时间  
  
//作用域限制在和当前文档同目录的文档中，cookie在所在目录下的子目录也有效  
//允许存储的数据量比cookie大，比localStorage以及sessionStorage数据量要小

# 第十八章 HTML5 API

## 18.1 postMessage

*/\*\*  
 \* 1.getElementsByClassName querySelectorAll() dataset属性  
 \* 2.classList属性  
 \* 3.XMLHttpRequest Level 2 跨域HTTP请求 EventSource API  
 \* 4.Web存储API（localeStorage sessionStorage） 离线WEB应用  
 \* 5.*<audio> <video> <canvas> *SVG图形  
 \*  
 \*/  
  
  
/\*\*  
 \* 1.地理位置API  
 \* 2.历史记录管理  
 \* 3.跨域消息传递 -  
 \* 4.Web Worker -  
 \* 5.类型化数组和ArrayBuffer -  
 \* 6.Blob -  
 \* 7.文件系统API -  
 \* 8.客户端数据库 -  
 \*/*/\*跨域消息传递\*/  
  
//如果窗口中的内容是来自于不同的源，WEB浏览器（遵循同源策略）不会允许访问其他窗口中的文档内容  
//大部分情况下，浏览器还不允许脚本读取其他窗口的属性或调用其它窗口的方法  
//有一个window方法允许来自非同源脚本调用 postMessage()方法  
  
//通过异步消息传递的方式 在来自不同源的脚本之间传递消息（包括IE8都实现了）  
  
//这项技术称为 跨文档消息传递  
//这是一个window对象的方法  
//所以又称为窗口消息传递  
  
//第一个参数是字符串，可以使用JSON格式 第二个参数是一个字符串 指定目标窗口的源  
//包括协议 主机名 以及URL端口部分 如果传递给任何窗口 可以设置为 \* 如果要指定和当前窗口同源的话 简单的使用 /  
  
//在目标窗口的Window对象上就会触发一个message事件  
//在脚本中可以定义通知message事件的处理程序函数 调用该事件处理程序的时候会传递给它一个拥有如下属性的事件对象  
  
  
// data 作为第一个参数传递给postMessage()方法的消息内容副本  
// source 消息源自的Window对象  
// origin 一个url形式的字符串指定消息来源