Contents

[CSS 2](#_Toc440635270)

[盒模型The box model 2](#_Toc440635271)

[元素布局 2](#_Toc440635272)

[元素定位 3](#_Toc440635273)

[JS 3](#_Toc440635274)

[< JavaScript 面向对象编程指南> 3](#_Toc440635275)

[函数也是数据？(该数据类型包含的是代码，且可执行) 3](#_Toc440635276)

[1. 对象 5](#_Toc440635277)

[2. 原型 7](#_Toc440635278)

[3. 继承 7](#_Toc440635279)

[4. 惯用法 9](#_Toc440635280)

[《Effective JavaScript》 11](#_Toc440635281)

[1. 全局变量与局部变量 11](#_Toc440635282)

[2. 闭包 11](#_Toc440635283)

[3. 变量声明提升间 12](#_Toc440635284)

[4. Javascript立即函数表达式 13](#_Toc440635285)

[5. 函数调用，方法调用及构造函数调用之间的区别？ 14](#_Toc440635286)

[6. 高阶函数 15](#_Toc440635287)

[7. 创建可变参数的函数一般提供两个版本 15](#_Toc440635288)

[8. 复用对象方法 15](#_Toc440635289)

[9. 高阶函数惯用法：函数curring 16](#_Toc440635290)

[10. JavaScript类的本质 16](#_Toc440635291)

[11. Javascript实现类继承的最佳实践 17](#_Toc440635292)

[12. Javascript实现存储私有数据 18](#_Toc440635293)

[13. 函数的this变量 19](#_Toc440635294)

[15. 面向函数编程 21](#_Toc440635295)

[16. 在类数组对象上尽量复用通过的数组方法 21](#_Toc440635296)

[17. undefined: JavaScript无法无法提供具体的值 22](#_Toc440635297)

[18. 选项对象 22](#_Toc440635298)

[19. 无状态API和有状态API 23](#_Toc440635299)

[20. 使用结构类型来实现多态性，从而避免使用继承 23](#_Toc440635300)

[21. Javascript无法实现重载（函数重名，但参数不同） 23](#_Toc440635301)

# CSS

参考http://learn.shayhowe.com/html-css/

## 盒模型The box model



We set a width property value of 400 pixels, but the actual width of our element is 492pixels. By default the box model is additive; thus to determine the actual size of a box we need to take into account padding, borders, and margins for all four sides of the box.

## 元素布局

元素display：block, inline, line-block, none

默认div, p, form, header, footer, section是block元素，span, a, img是inline元素

block, line-block元素有width, height, margin, border, padding属性；inline元素只有margin-left, margin-right, padding

document一般是自然从上到下排列，block占一行，若将一行分为几个block，常见如下方法：

方法1: The float property allow elements to be floated to the left or right of their parent element.

1. 行内的div赋float:left, width:30%, 行后赋clear:both

方法2:

1. 行内的div赋display:line-block,同时div之间加<!----> (为什么？remove the space between inline-block elements)

## 元素定位

static: 默认所有元素position:static,元素将按照它们在标记里出现的先后顺序出现在浏览器窗口里。

relative(相对定位)：若div赋position:relative, top:20px, left:20px,则div相对于本应在文档的位置位移(deltaX=top, deltaY=left)

absolute(绝对定位)：若div赋position:absolute, top:20px, left:20px,则div相对于最近的具有position:relative的父元素，若该父元素没有，则相对于body元素。把它摆放到容纳它的“容器”的任何位置。

Fixed(固定定位)：

overflow处理子元素尺寸超出父元素容器尺寸

visible: 全显示子元素

hidden: 隐藏溢出的内容，只显示在父元素的显示区域，用来对子元素内容进行“裁剪”

scroll: 显示滚动条

auto: 发生溢出时才显示滚动条

# JS

## < JavaScript 面向对象编程指南>

函数也是数据？(该数据类型包含的是代码，且可执行)

函数可以被赋给变量，从而被定义，删除，拷贝，或作为参数传给另一函数（作为回调函数）

var callback = function(a, b) {return a+b;}

function fun(a, b, c, callback) {

For (var i=0, ar=[]; i<3; ++i){

ar[i] = callback(arguments[i], 2);

}

return ar;

}

>>>callback （返回函数代码）

>>>delete callback （删除变量，存储函数代码片断）

>>>typeof callback （返回函数类型）

>>>callback(a,b) （执行函数）

浏览器特性探测技术(函数根据其当前所在的浏览器来重定义自己)

var callback = function() {

// 私有函数，用来初始化

function someSetup(){

}

//私有函数

function actualWork(){

}

someSetup() ;

return actualWork ;

}() ;

()表示自动执行，私有函数someSetup()执行初始化工作，然后返回actualWork函数赋给变量callback

Javascript没有{}作用域，有函数作用域，每个函数有自己的作用域链和词法作用域

函数作用域链:子函数可以访问自己的作用域，还可以访问父级作用域，还可以访问父级的父有作用域

function f() {

var b = 1 ;

function n() {

var c = 3 \* b ;

}

}

函数词法作用域：每个函数在被定义时（而非执行时）都会创建一个属于自己的环境

闭包1:返回子函数的父函数作用域中的变量或子函数

function f(){

var b = “b”;

return function() {

return b;

}

}

闭包2: 子函数赋给全局函数

var n;

function f() {

var b = “b” ;

n = function() {

return b ;

}

}

闭名3: 闭包只是引用局部变量（即拥有的是局部变量的连接，但不会记录局部变量的值）

function f() {

var a = [] ;

var i ;

for (i = 0 ; i < 3; i ++ ){

a[i] = function() {

return i ;

}

}

}

>>> var a = f() ;

创建三个闭包a[0](), a[1](), a[2](),它们都指向了一个共同的局部变量i, 但是闭包并不会记录它们的值，它们所拥有的只有一个i的连接（即引用），因此只能返回i的当前值。由于循环结束时i的值的为3,所以这三个函数都指向了这一共同值.

闭包的简单应用：

实现getter and setter: 变量在自调函数空间里，子函数赋给全局变量

var getValue, setValue;

(function(){

var secret = 0 ;

getValue = function() {

return secret ;

} ;

setValue = function(v) {

secret = v ;

} ;

})()

实现迭代器:

function setup(x){

var i = 0 ;

return function() {

return x[i++] ;

}

}

>>> var next = setup ([‘a’, ‘b’]) ;

>>>next()

对象

数组文本标识法: var a = [] ;

对象文本标识法: var o = {} ;

JavaScript是动态语言，所以它允许我们随时对现存对象的属性和方法进行修改。

* 创建对象的方式：

通过对象文本标识法直接创建对象: (实际上是由内建构造器Object()函数创建)

var o = {

name: ‘Rafaelo’,

sayName: function() { … }

} ;

通过构造器函数创建对象:

//定义构造器函数

function Hero(name){

this.name = name ;

this.sayName = function(){…} ;

}

//创建对象

var o = new Hero(‘Rafaelo’) ;

var o2 = new Hero(‘qzlin’) ;

重点：

* 一般将构造器函数首字母大写，以区别一般函数
* 构造器函数作为普通函数直接调用，那么this指的是全局对象window; 利用构造器函数创建对象，那么this指的是这个对象。
* 对象拥有构造器属性

o.constructor == Hero(name) ; //从而可以通过对象构造器属性创建另一对象

var o3 = new o.constructor(‘tom’) ;

* 工厂函数返回对象:

function factory(name) {

return {

name: name ;

}

}

* 内建对象：

数据封装类对象（Object, Array, Boolean, Number, String）

工具类对象(Math, Date, RegExp)

错误类对象（Error）

* 所有对象都继承自Object对象，并且拥有其所有的方法（如toString(),constructor）
* 函数也是对象，内建构造器是Function(),拥有如下属性:

toString() ; //返回函数源码

constructor ;

length ; //函数的输入参数个数

prototype ; //是一个对象，仅和构造器函数配合才有意义,

//因为JavaScript没有类概念，采用call or apply复用对象方法？

call() ;

apply() ;

//某对象

var my\_obj = { name: ‘current object’} ;

var other\_obj = {

name: ‘Ninja’,

say: function(who, role) {

return ‘I am ‘ + who + ‘ with ‘ + role;

}

}

//因为my\_obj没有方法say,不想自己实现，可以复用other\_obj对象的方法say(注意借用的方法say(who)的作用域会改为my\_obj对象)来执行

other\_obj.say.call(my\_obj, ‘Dude’, ‘Doctor’) ;

等同于

other\_obj.say.apply(my\_obj, [‘Dude’, ‘Doctor’]) ;

函数内有arguments变量，也是一个对象，含属性[i], length, callee, 基于callee属性可以实现递归函数

function(count){

if (count <５)　｛

alert(count) ;

arguments.callee(++count) ;

｝

}

原型

构造器函数 + 原型

//构造器函数，内赋属性和方法

function Gadget(name, color) {

this.name = name ;

this.color = color ;

this.draw = function() {

return … ;

}

}

//原型，添加属性和方法

Gadget.prototype.price = 100 ;

Gadget.prototype.getInfo = function() {

return …;

}

>>> var newtoy = new Gadget(‘webcam’, ‘black’) ;

>>>newtoy.price ;

分析：对象访问属性或方法时，JavaScript引擎会先遍历该对象的所有属性，如果没找到，JavaScript会查询创建当前对象的构造器函数的原型，如果仍没找到，就会继续搜索其原型的原型，直至最高级的父对象Object.

若重写某对象的prototype时，一定要重置相应的constructor属性. 否则已创建对象无法访问新的prototype，只能新创建的对象能访问。

Gadget.prototype = {

Price: 100 ,

getInfo = function() {…}

}

Gadget.prototype.constructor = Gadget ;

继承

* 通过创建父对象，赋给子对象构造器原型 (构造器工作模式—使用原型链模式)

function Shape(){

this.name = ‘shape’ ;

this.toString = function() { return this.name ; }

}

function Triangle(side, height) {

this.name = ‘Triangle ;

this.side = side ;

this.height = height ;

this.getArea = function() {..} ;

}

**Triangle.prototype = new Shape() ;**

**Triangle.prototype.constructor = Triangle ;**

* 通过父对象构造器函数的原型，赋给子对象构造器函数的原型 （构造器工作模式—原型拷贝模式, YUI, Extjs采用此方法）

**function extend(Child, Parent) {**

**//避免子对象修改影响父对象（因为共同引用），需要临时构造器**

**var F = function() {} ;**

**F.prototype = Parent.prototype ;**

**Child.prototype = new F() ;**

**Child.prototype.constructor = Child ;**

**Child.uber = Parent.prototype ;**

**}**

**extend(Triangle, Shape) ;**

* 通过对象拷贝 （对象工作模式--属性拷贝模式, 注意深拷贝=值传递与浅拷贝=引用传递）

**function deepCopy(parent, child){**

**var child = child || {} ;**

**for (var i in parent) {**

**if (typeof parent[i] === ‘object’) {**

**child[i] = (parent[i].constructor === Array) ? [] : {} ;**

**deepCopy(parent[i], child[i]) ;**

**} else {**

**child[i] = parent[i] ;**

**}**

**}**

**return child ;**

**}**

**var triangle = deepCopy(new Shape()) ;**

* 原型继承与属性拷贝的混合

**function objectPlus(o, stuff) {**

**//原型继承copy现存对象**

**function F() {}**

**F.prototype = o ;**

**var n = new F() ;**

**n.uber = o ;**

**//对其他对象stuff使用属性copy**

**for (var i in stuff) {**

**n[i] = stuff[i] ;**

**}**

**return n ;**

**}**

**var triangle = objectPlus(new Shape(), {**

**name: ‘Triangle’,**

**getArea: function() {…}**

**side: 0,**

**height: 0**

**}) ;**

* 复用构造器和属性拷贝 (构造器工作模式—原型模式—属性拷贝模式)

**function copyPrototype(Child, Parent) {**

**var p = Parent.prototype ;**

**var c = Child.prototype ;**

**for (var i in p) {**

**C[i] = p[i] ;**

**}**

**c.uber = p ;**

**}**

**function Triangle() {**

**Shape.apply(this, arguments) ;**

**}**

**copyPrototype(Triangle, Shape) ;**

JavaScript中没有类的概念，一般是*将构造器函数当做类*。所以一般选择构造器工作模式来解决问题。

惯用法

* 创建命名空间

var MYAPP = {} ;

MYAPP.namespace = function(name){

var parts = name.split('.') ;

var current = MYAPP ;

for (var i in parts){

var part = parts[i] ;

if (!current[part]) {

current[part] = {} ;

}

current = current[part] ;

}

};

MYAPP.namespace(‘event’) ;

MYAPP.namespace(‘dom.style’) ;

等价于

var MYAPP = {

event: {} ,

dom: {

style: {}

}

}

* 通过配置对象来替代多个参数，传给函数

MYAPP.dom.Button = function(text, type, color, border, font) {…}

new MYAPP.dom.Button(‘push’, null, ‘white’, null, ‘Arial’) ;

⬄

MYAPP.dom.Button = function(text, conf) {

var type = conf.type || ‘submit’ ;

var font = conf.font || ‘Verdana’ ;

}

* 私有属性和方法

在构造器中通过使用局部变量和函数的方式来实现类似的权限控制

MYAPP.namespace('dom') ;

MYAPP.dom.Button = function(text, conf){

//私有变量

var styles = {

font: 'Verdana',

border: '1px solid black',

color: 'black',

background: 'grey'

} ;

//私有函数

function \_setStyles(b) {

for (var i in styles){

b.style[i] = conf[i] || styles[i] ;

}

}

conf = conf || {} ;

var b = document.createElement('input') ;

b.type = conf['type'] || 'submit' ;

b.value = text ;

\_setStyles(b) ;

return b ;

};

* 自执行函数: 一般用于创建和返回对象

如果我们创建对象的过程很复杂，并且需要做一些初始化工作，那么我们就可以把第一部分相关的初始化工作设置为一个自执行函数，然后通过它来返回一个对象—它可以访问初始化部分定义的任何私有属性。

var MYAPP = {} ;

MYAPP.dom = function(){

// initialization code

function \_private(){

...

}

return {

getStyle: function(el, prop){

...

\_private() ;

},

setStyle: function(el, prop, value){

...

}

};

}() ;

链式调用：构造器函数（类）当中的方法返回this

## 《Effective JavaScript》

全局变量与局部变量

局部变量：函数内定义的变量如

function() {

var sum = 0 ;

}

全局变量： 函数外定义的或函数内定义，但没有var指示的变量如

function() {

sum = 0 ;

}

在Web浏览器中，全局对象被绑定到全局的window变量，console中访问this.sum或window.sum

闭包

* Javascript允许你引用在当前函数以外定义的变量

function makeSandwich() {

var magicIngredient = "peanut butter" ;

function make(filling){

return magicIngredient + " and " + filling ;

}

return make("jelly") ;

}

makeSandwich() ; // "peanut butter and jelly"

* 即使外部函数已经返回，当前函数仍然可以引用在外部函数所定义的变量

function makeSandwich() {

var magicIngredient = "peanut butter" ;

function make(filling){

return magicIngredient + " and " + filling ;

}

return make ;

}

var f = makeSandwich() ;

f("jelly") ; // "peanut butter and jelly"

f("bananas") ; // "peanut butter and bananas"

注意：JavaScript的函数值包含了比调用它们时执行所需要的代码还要多的信息。而且Javascript函数值还在内部存储它们可能引用的定义在其封闭作用域的变量。那些在其所涵盖的作用域内跟踪变量的函数被闭包。

* 闭包可以更新外部变量的值

注意：闭包存储的是外部变量的引用，而不是它们的值的副本。

function box() {

var val = undefined ;

return {

set: function(newVal) {val = newVal ; },

get: function() {return val ; }

};

}

var b = box() ;

b.set(98.6) ;

b.get(); // 98.6

变量声明提升间

Javascript没有块级作用域，只有函数作用域词法作用域

Javascript隐式地提升声明部分到封闭函数的顶部，而将赋值留在原地。惯用法是所有的var声明放置在函数的顶部。

function f() {

//...

//...

{

//...

var x = /\*...\*?

//...

}

}

function f() {

var x ;

//...

{

//...

x = /\*...\*/

//...

}

}

Javascript立即函数表达式

没有块级作用域，但可以用立即调用的函数表达式来创建块级作用域

function wrapElement(a) {

var result = [] ;

var i = 0 ;

for (i = 0, n = a.length ; i < n ; ++ i){

//在函数内部创建局部变量var j,并令该函数立即执行，等价于在函数体内的块级{}作用域定义变量j

(function(){

var j = i ;

result[i] = function() {return a[j] ; } ;

})() ;

}

return result ;

}

以上代码是等效的常见方法

function wrapElement(a) {

var result = [] ;

var i = 0 ;

for (i = 0, n = a.length ; i < n ; ++ i){

(function(j){

result[i] = function() {return a[j] ; } ;

})(i) ;

}

return result ;

}

函数声明与命名函数表达式区别？

//在程序的最顶层，该函数声明将创建一个名为double的全局函数

function double(x) {return x \* 2 ;}

//该命名函数表达式将函数绑定到变量f,而不是变量double

var f = function double(x) {return x \* 2 ;}

命名函数表达会导致很多问题，所以不值得使用。一般采用如下方式：

function find(tree, key){

if (!tree){

return null ;

}

if (tree.key === key){

return tree.value ;

}

return find(tree.left, key) ||

find(tree.right, key) ;

}

var f = find ;

重点：官方指定函数声明只能出现在其他函数或程序的最外层。编写可移植的函数的最好方式是始终避免将函数声明置于局部块或子语句中。

函数调用，方法调用及构造函数调用之间的区别？

* //函数调用

function hello(username){

return "hello " + username ;

}

hello("qzlin") ;

* //方法调用

var obj = {

username: "qzlin"

hello: function(){

return "hello " + this.username ;

}

};

obj.hello() ;

//复用对象方法

var obj2 = {

username: "Boo",

hello: obj.hello

};

//在方法调用中是调用表达式自身来确定this变量的绑定，所以obj2.hello方法中的this是obj2对象

obj2.hello() ;

//一个函数里有this变量，一般该函数作为对象方法或构造函数，如果直接作为函数调用，则会将全局对象作为接收者，即this是全局对象

* //构造函数使用

function User(name, passwordHash){

this.name = name ;

this.passwordHash = passwordHash ;

}

var u = new User("qzlin", "000");

(记住构造函数的主要职责是初始化该新对象)

高阶函数

将函数作参数或返回值的函数。将函数作为参数，通常该函数也称为回调函数，因为高阶函数随时调用它。常用于集合

var names = ["tom", "jerry", "puppy"].map(function(name){

return name.toUpperCase() ;

}) ;

创建可变参数的函数一般提供两个版本

//数组作为参数的版本

function averageOfArray(a){

for (var i= 0, sum= 0, n= a.length; i < n ; ++ i){

sum += a[i] ;

}

return sum / n ;

}

averageOfArray([2, 7, 2, 1]) ;

//可变参数的版本

function average(){

return averageOfArray(arguments) ;

}

复用对象方法

仅仅是方法借用，但方法内的this对象没有绑定，因此需要确定，如下：

var buffer = {

entries: [] ,

add: function(s){

this.entries.push(s) ;

},

join: function(){

this.entries.join() ;

}

}

//仅仅借用对象的方法，但没有绑定对象方法的this变量

["021", "-", "24115296"].forEach(buffer.add) ; //error: entries is undefined

// 集合的forEach函数提供绑定回调函数的this变量

["021", "-", "24115296"].forEach(buffer.add, buffer) ;

buffer.join() ;

//若函数没有提供，由需要自己封装一层函数，然后直接调用对象的方法（此时含对象）

["021", "-", "24115296"].forEach(function(s){

buffer.add(s) ;

}) ;

buffer.join() ;

//ES5标准库提供：函数对象的bind方法，第一个参数是作用域this对象，并产生一个以该对象调用方法的方式

["021", "-", "24115296"].forEach(buffer.add.bind(buffer)) ; //等价于上面的封装版本

buffer.join() ;

高阶函数惯用法：函数curring

实现函数委托 = 显示封装 = bind版本

//具体实现函数

function simpleURL(protocol, domain, path){

return protocol + "://" + domain + "/" + path ;

}

// 显示封装

var urls = paths.map(function(path){

return simpleURL("http", "siteDomain", path) ;

}) ;

//使用函数对象的bind来实现函数委托 （注意第一个参数是作用域，其它的是参数）

var urls = paths.map(simpleURL.bind(null, "http", "siteDomain")) ;

JavaScript类的本质

在许多语言中，每个对象是相关类的实例，该类提供在其所有实例间共享代码。相反,JavaScript并没有类的内置概念，对象是从其他对象中继承而来。

Javascript中的类本质上是一个构造函数与一个用在该类实例间共享方法的原型对象的结合(构造函数+原型)

将方法存储在实例对象中将创建该函数的多个副本，因为每个实例对象都有一份副本。因此最佳实践是：构造函数初始化对象，原型存储方法和类静态变量

function User(name, passwordHash){

this.name = name ;

this.passwordHash = passwordHash ;

}

//构造函数User的prototype属性用来设置新实例的原型关系

User.prototype.toString = function(){

return "[User " + this.name + "]";

} ;

User.prototype.checkPassword = function(password){

return hash(password) === this.passwordHash ;

};

关键点：

* 文档化构造函数期望使用new操作符调用是很重要的，尤其是在跨大型代码中共享构造函数和该构造函数来自一个共享库时。
* 获取原型对象的方法：Object.getPrototypeOf(u) === User.prototype ; //true

C.prototype用于建立由new C()创建的对象原型。

Object.getPrototypeOf(obj)是ES5中用来获取obj对象的原型对象的标准方法（不要使用obj.\_\_proto\_\_）

Javascript实现类继承的最佳实践

子类构造函数调用父类构造函数

子类原型继承父类原型

//父类Actor

function Actor(scene, x, y){

this.scene = scene ;

this.x = x ;

this.y = y ;

scene.register(this) ;

}

Actor.prototype.moveTo = function(x, y){

this.x = x ;

this.y = y ;

this.scene.draw() ;

};

Actor.prototype.exit = function(){

this.scence.unregister(this) ;

this.scene.draw() ;

};

Actor.draw = function(){

var image = this.scene.images[this.type] ;

this.scene.content.drawImage(image, this.x, this.y) ;

};

//子类SpaceShip

function SpaceShip(scene, x, y){

**Actor.call(this, scene, x, y) ; //子类构造函数调用父类构造函数**

this.points = 0 ;

}

//Object.create以一个原型对象作为参数，并返回一个继承自该原型对象的新对象

**SpaceShip.prototype = Object.create(Actor.prototype) ; //子类继承父类原型**

SpaceShip.prototype.type = "SpaceShip" ;

SpaceShip.prototype.scorePoint = function(){

this.points ++ ;

};

SpaceShip.prototype.left = function(){

this.moveTo(Math.max(this.x-10, 0), this.y) ;

};

SpaceShip.prototype.right = function(){

var maxWidth = this.scene.width - this.width() ;

this.moveTo(Math.min(this.x+10, maxWidth), this.y) ;

};

//游戏场景

function Scene(context, width, height, images){

this.context = context ;

this.width = width ;

this.height = height ;

this.images = images ;

this.actors = [] ;

}

Scene.prototype.register = function(actor){

this.actors.push(actor) ;

} ;

Scene.prototype.unregister = function(actor){

var i = this.actors.indexOf(actor) ;

if (i >= 0){

this.actors.splice(i, 1) ;

}

} ;

Scene.prototype.draw = function(){

this.context.clearRect(0, 0, this.width, this.height) ;

for (var a = this.actors, i = 0, n = a.length ; i < n; ++ i){

a[i].draw() ;

}

}

Javascript实现存储私有数据

和python一样，没有语法支持信息隐藏，如何实现存储私有数据？

编码规范，给私有属性名前置或后置一个下划线

使用闭包，在构造函数中以变量的方式来存储，并使对象方法引用这些变量

function User(name, passwordHash){

var \_role = "" ;

this.name = name ;

this.getRole = function(){

return \_role ;

};

this.setRole = function(role){

\_role = role ;

}

}

分析：在此构造函数中，this引用的变量或方法，都可以被外部直接引用，而构造函数里的变量（其作用域只在构造函数里）如\_role, name, passwordHash(别忘了构造函数输入参数也是函数里的变量)，外部无法引用，只能通过方法接口来引用。

函数的this变量

每个函数都有一个this变量的隐式绑定，该this变量的绑定值是在调用该函数时确定的（注：this变量是隐式绑定到最近的封闭函数）

function CSVReader(separators){

this.seperators = separators || [","] ;

this.regexp = new RegExp(this.seperators.map(function(sep){

return "\\" + sep[0] ;

}).join("|")) ;

}

CSVReader.prototype.read = function(str){

var lines = str.trim().split(/\n/) ;

return lines.map(function(line){

return line.split(this.regexp) ; //wrong this,因为这里的this是指调用函数map的对象lines

});

}

解决方法：

* 数组的map方法引入回调函数的this绑定（即引入作用域），从而改变回调函数作用域的this对象。

CSVReader.prototype.read = function(str){

var lines = str.trim().split(/\n/) ;

return lines.map(function(line){

return line.split(this.regexp) ;

}, this) ;

};

* 使用词法作用域的变量来改变this的混淆

CSVReader.prototype.read = function(str){

var lines = str.trim().split(/\n/) ;

**var self = this ;**

return lines.map(function(line){

return line.split(**self**.regexp) ;

}) ;

};

* 使用回调函数的bind方法改变回调函数内的this对象

CSVReader.prototype.read = function(str){

var lines = str.trim().split(/\n/) ;

return lines.map(function(line){

return line.split(this.regexp) ;

}.bind(this)) ;

};

关键点：

* 不要重用父类的属性名，因为子类和父类，如果属性名相同，那么该属性存储在实例对象上并命名为一个字符串，从而指向的是同一个属性。
* 如果一个程序库提供的对象的属性没有文档化或者明文标为内部属性，对于使用者来说，最好不要干涉那些属性。
* 避免使用轻率的猴子补丁（monkey-patching）：对象共享原型，因此每一个对象都可以增加、删除或修改原型的属性。Monkey-patching一般用来弥补平台的差距。

JavaScript中的对象特点

对象是JavaScript中最万能的数据结构，取决于不同的环境，对象可以表示一个灵活的键值记录，一个继承了方法的面向对象数据抽象，一个密集或稀疏的数组，或一个散列表。

JavaScript的对象操作总是以继承的方式工作，即使是一个空的对象字面量，也继承了Object.prototype的大量属性

var dict = {} ;

"toString" in dict ;

"valueOf" in dict ;

为防止原型污染，可以使用null原型:

var x = Object.create(null) ;

Object.getPrototypeOf(x) === null ; //true

注意与如下区别:

function C() {}

C.prototype = null ;

var x = new C() ;

Object.getPrototypeOf(x) === null ; // false ;

Object.getPrototypeOf(x) === Object.prototype ; // true ;

实现字典类

function Dict(elements){

this.elements = elements || {} ; //simple Object

this.hasSpecialProto = false ; // has "\_\_proto\_\_"" key ?

this.specialProto = undefined ; // "\_\_proto\_\_" element

}

Dict.prototype.has = function(key){

if (key === "\_\_proto\_\_"){

return this.hasSpecialProto ;

}

// own property only

return {}.hasOwnProperty.call(this.elements, key) ;

};

Dict.prototype.get = function(key){

if (key === "\_\_proto\_\_"){

return this.specialProto ;

}

// own property only

return this.has(key) ? this.elements[key] : undefined ;

};

Dict.prototype.set = function(key, val){

if (key === "\_\_proto\_\_"){

this.hasSpecialProto = true ;

this.specialProto = val ;

} else {

this.elements[key] = val ;

}

};

Dict.prototype.remove = function(key){

if (key === "\_\_proto\_\_"){

this.hasSpecialProto = false ;

this.specialProto = undefined ;

} else {

delete this.elements[key] ;

}

};

var dict = new Dict(

{

alice: 34,

bob: 24,

chris: 62

}) ;

面向函数编程

数组迭代优先使用for循环而不是for…in循环，迭代方法优于循环

* Array.prototype.forEach(callback): 数组遍历

var trimmed = [] ;

input.forEach(function(s){

trimmed.push(s.trim()) ;

}) ;

* Array.prototype.map(callback): 映射 实现y = f(x)

var trimmed = input.map(function(s){

return s.trim() ;

});

* Array.prototype.filter(callback): 过滤

var specialList = listings.filter(function(listing){

return listing.price >= min && listing.price <= max ;

});

在类数组对象上尽量复用通过的数组方法

function highlight(){

[].forEach.call(arguments, function(widget){

widget.setBackground("yellow") ;

});

}

var arrayLike = {0: "a", 1: "b", 2:"c", length: 3} ;

var result = Array.prototype.map.call(arrayLike, function(s){

return s.toUpperCase() ;

});

undefined: JavaScript无法无法提供具体的值

//未赋值的变量的初始值

var x ;

x ; //undefined

//访问对象中不存在的属性

var obj = {} ;

obj.x ; // undefined

//一个函数体结局使用未带参数的return语句或未使用return语句都会产生返回值undefined

function g() {}

g()；

选项对象

function Alert(parent, message, opts){

opts = opts || {} ;

//假定0是有效值，所以要测试undefined

this.width = opts.width===undefined ? 320 : opts.width ;

this.height = opts.height===undefined ? 240 : opts.height ;

//假定空字符串是无效值，用逻辑或

this.title = opts.title || "Alert" ;

//使用双重否定模式将参数转换为布尔值

this.modal = !! opts.modal ;

}

一般Javascript库和框架都会提供一个extend函数，该函数接收一个target对象和一个source对象，并将source的属性复制到target对象中。

function extend(target, source){

if (source){

for (var key in source){

var val = source[key] ;

if (typeof val !== "undefined") {

target[key] = val ;

}

}

}

return target ;

}

function Alert(parent, message, opts){

opts = extend({

width: 320,

height: 240,

title: "Alert",

modal: false

}, opts) ;

extend(this, opts) ;

}

无状态API和有状态API

无状态API提供的函数或方法，只取决于输入，而与程序的状态改变无关。

有状态API如Web的Canvas库，它提供了绘制形状和图片到其平面的用户界面元素方法。

使用结构类型来实现多态性，从而避免使用继承

编译类型语言使用类和继承来结构化程序。

动态语言可以使用结构类型（或叫鸭子类型）来设计对象的接口，任何对象只要具有预期的结构就属于该类型，从而它不需要编写任何显示的声明，一个调用某个对象方法的函数能够与任何实现了相同接口的对象一起工作。

function Wiki(format){

this.format = format ;

}

Wiki.prototype.displayPage = function(source){

var page = this.format(source) ;

var title = page.getTitle() ;

var author = page.getAuthor() ;

var output = page.toHTML() ;

//...

};

//不需要使用子类化继承，只要类实现相同的接口

Wiki.format.MEDIAWIKI = function(source){

// extract contents from source

//...

return {

getTitle: function() {} ,

getAuthor: function() {},

toHTML: function() {}

};

} ;

var app = new Wiki(Wiki.formats.MEDIAWIKI) ;

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

Javascript无法实现重载（函数重名，但参数不同）

因为在调用函数时，所提供的参数数量不一定要和函数的参数数据一致，如果调用函数时比它拥有的参数少些，那么任何你未提供的参数的值都是undefined.若调用函数的参数比它有的多了，那么多出来的会被忽略。