# 函数

## 递归

一、函数调用

function isPalindrome( text ) {

if ( text.length <=1 ) return true

if( text.charAt(0) !== text.charAt(text.length -1) ) return false

return isPalindrome( text.substr( 1, text.length - 2 ) )

}

二、方法调用

var ninja = {

chirp: function(n) {

return n > 1 ? ninja.chirp(n - 1) + "-chirp" : "chirp";

}

};

var samurai = { chirp: ninja.chirp };

**优化**

return n > 1 ? this.chirp(n - 1) + "-chirp" : "chirp";

1. **内联命名函数**

上面的问题在于，其他对象引用时必须给方法命名为chirp，而用内联命令函数则不会有这个问题

var ninja = {

chirp: function signal(n) { //#1

return n > 1 ? signal(n - 1) + "-chirp" : "chirp";

}

};

内联函数的名称在函数外是不可用的。只在函数内部是可用的。如：

var a = function b() {

}

b -- 》 undefined

这也是为什么全局属性会被作为window的方法进行创建的原因，不使用window的属性，无法引用这些函数。

注意： arguments.callee() 也可以调用当前的函数，但是，es5规范中禁止使用这个属性。

## 函数存储与对象存储

问题：如何存储一组独立的函数？

关键： 可以将函数视为对象，在其上添加属性

函数属性存储优势

将状态和缓存信息存储在独立位置

利于代码组织

外部存储或缓存对象无需污染作用域

var store = {

nextId: 1, //#1

cache: {}, //#2

add: function(fn) { //#3

if (!fn.id) { //#3

fn.id = store.nextId++; //#3

return !!(store.cache[fn.id] = fn); //#3

} //#3

}

};

**自记忆函数**

用于缓存记忆昂贵的计算结果

function isPrime(value) {

if (!isPrime.answers) isPrime.answers = {}; //#1

if (isPrime.answers[value] != null) { //#2

return isPrime.answers[value]; //#2

} //#2

var prime = value != 1; // 1 can never be prime

for (var i = 2; i < value; i++) {

if (value % i == 0) {

prime = false;

break;

}

}

return isPrime.answers[value] = prime; //#3

}

但是也要权衡一下缺点

节省了计算性能，但是任何类型的缓存肯定会牺牲内存。

缓存是否应该与业务逻辑放在一起？（函数或方法应该只做一件事？）

缓存记忆DOM元素

function getElements( name ) {

if( !getElements.cache ) getElements.cache = {};

return getElements.cache[name] ||

document.getElementsByTagName(name)

}

类数组缓存

var elems = {

length: 0,

add: function(elem) {

Array.prototype.push.call(this, elem)

},

gather: function(id) {

this.add(document.getElementById(id))

}

}

## 利用参数进行函数重载

由于js函数没有重载的概念，可以借用参数模仿重载

注：函数的length属性，可以知道声明了多少形参

arguments.length属性，可以知道传入了多少实参

1. 更具实参个数进行判断进而重载

## 函数判定

如何判断一个对象是一个函数的实例，并且可以调用

1. typeof 的问题

一个非期望的函数结果返回

safari 返回nodeList为function

ie 。。。

1. Object.prototype.toString.call(fn) === ‘[object Function]’

返回一个对象内部的描述符

IE会将DOM元素的方法报告为object（测试情况不符合）

# 闭包

## 闭包如何工作

一个函数在创建时，允许该函数访问并操作该函数之外的变量所创建的作用域。

也就是说：函数调用时可以访问，该函数声明时的作用域里的变量和函数。

实际上是一种状态封装

在某一方面保护了，函数执行完毕后要被垃圾回收的变量和函数

## 闭包使用

### 访问私有变量

访问函数内部的变量

function Ninja() { //#1

var feints = 0; //#2

this.getFeints = function(){ //#3

return feints; //#3

}; //#3

this.feint = function(){ //#4

feints++; //#4

}; //#4

}

var ninja = new Ninja();

### 回调与计时器回调

### 绑定函数上下文

function bind(context,name){ //#1

return function(){ //#1

return context[name].apply(context,arguments); //#1

}; //#1

} //#1

var button = {

clicked: false,

click: function(){

this.clicked = true;

assert(button.clicked,"The button has been clicked");

console.log(this);

}

};

var elem = document.getElementById("test");

elem.addEventListener("click",bind(button,"click"),false); //#2

### 偏应用函数

也就是柯里化函数，

柯里化：在一个函数中首先填充几个参数（然后在返回一个新函数）的技术就是柯里化

偏应用方程返回一个含有预处理参数的新函数

### 隐藏调用方法

对于调用者，内部方法不可见。

Function.prototype.memoized = function(key){

this.\_values = this.\_values || {};

return this.\_values[key] !== undefined ?

this.\_values[key] :

this.\_values[key] = this.apply(this, arguments);

};

Function.prototype.memoize = function(){

var fn = this; //#1

return function(){ //#2

return fn.memoized.apply(fn, arguments);

};

};

var isPrime = (function(num) {

var prime = num != 1;

for (var i = 2; i < num; i++) {

if (num % i == 0) {

prime = false;

break;

}

}

return prime;

}).memoize();

assert(isPrime(17),"17 is prime"); //#3

函数包装

是一种封装函数逻辑的技巧，用于在单个步骤内重载创建新函数或继承函数。（在特定情况下重写函数功能）

一个最常见的使用场景是在编写跨浏览器代码，必须在某个特定浏览器下实现某个特定功能。

### 即时函数

1. 可以利用即时函数创建一个临时的作用域，用于储存数据状态。

var divs = document.getElementsByTagName("div"); //#1

for (var i = 0; i < divs.length; i++) {

divs[i].addEventListener("click", function() {

alert("divs #" + i + " was clicked."); //#2

}, false);

}

for (var i = 0; i < divs.length; i++) {

(function(i){

divs[i].addEventListener("click", function() {

alert("divs #" + i + " was clicked."); //#2

}, false);

})(i)

}

1. 利于即时函数，传递参数，防止外部变量污染

(function($){})(jQuery)

3.绑定简写的名称，或者其它即时执行的初始化动作

1. 类库包装

（function(){

var jQuery = window.jQuery = funtion(){

}

}）()

通常只暴露一个变量时，用下列方法更好。

var jQuery = (function(){

function jQuery(){

}

return jQuery

})()

# 原型与面向对象

原型表面上是对象的特性，实际上原型的本质是函数特性。

## 实例化

var t = new Test()

创建一个空对象 { },将函数中this指向这个对象，

这个对象拥有

**只读属性\_\_proto\_\_对象**（这个实例对象的原型）

\_\_proto\_\_对象拥有直接指向类的prototype对象

**实例属性**

由this实例化的属性

prototype对象

任何一个函数都默认有一个prototype的属性，这个prototype对象，默认有一个

constructor属性，指向这个函数本身。

可以对这个prototype操作：

如：

function A(){

}

1.在函数外部添加属性

A.prototype.a = function(){}

不会影响这个constructor

2.在函数外部重写prototype

A.prototype = {}

相当于

A.prototype = new Object()

原来的constructor被删除了

3.在函数内部重写prototype

function A(){

A.prototype={]

}

在实例化的时候，先初始化一个对象{}，这个{}的 \_\_proto\_\_已经绑定了默认的A.prototype,在运行A.prototype={]，共享的重写，\_\_proto\_\_不会发生任何变化。但是A.prototype确实被覆盖了

4.在函数内部添加属性

function A(){

A.prototype.get=function(){}

}

按共享引用的原则，\_\_proto\_\_也会添加这个属性

其他设置的原型属性

## 扩展HTML DOM的原型

现代浏览器中，所有DOM元素都继承于HTMLElement构造器，扩展其原型，可以扩展DOM 节点的能力

HTMLElement.prototype.remove = function() { //#1

if (this.parentNode)

this.parentNode.removeChild(this);

};

## 扩展原生对象上的一些问题

1. 错误的扩展

某些扩展会造成不可预计影响

1. 扩展数字

对于Number，在Number对象上扩展，存在不能处理字面量这种情况

Number.prototype.add = function(num){ //#1

return this + num;

};

var n = 5; //#2

assert(n.add(3) == 8,

"It works when the number is in a variable.");

assert((5).add(3) == 8, //#3

"Also works if a number is wrapped in parentheses.");

assert(5.add(3) == 8, "What about a simple literal?"); //#4

## 子类化原生对象

function myArr(){}

myArr.prototype = new Array();

而更好的方法是继承部分方法

(function(){

var methods = ['push', 'pop', 'shift', 'unshift']

for(let i =0; i<methods.length; i++){

((name) => {

MyArray.prototype[name] = function(){

return Array.prototype[ name ].apply(this, arguments)

}

})(methods[i])

}

})();

## 实例化相关应注意的问题

1. 直接调用构造函数

var name = ‘go’

funtion User(name){

this.name = name

}

User()

不能实例化，同时也会将name全局变量覆盖

## 原型相关方法

obj instanceof b 可以判断b是否在obj的原型链上

function User(){

if(!this intanceof arguments.callee){

return new User(...arguments)

}

}

但是注意callee即将被废弃

obj.hasOwnProperty(a) 可以判断a是否是ojb的实例属性（而不是原型属性）

# 正则表达式

优先用字面量形式来构建一个已知的正则

而用构造器方式来动态构建正则

函数序列化

/xyz/.test(function() { xyz; }) ? /\b\_super\b/ : /.\*/;

判断函数是否能够序列化（调用test会将函数序列化）

## 动态构建正则

动态构建正则即是通过new RegExp(str, ‘g’)构建

注意：

js中的转义

如果str是通过js计算得到，要注意对如 /s这种字符进行反转义。

## 提升正则的效率

1. 编译
2. 设定起始位置
3. 尝试匹配
4. 匹配失败，从下一位开始继续第三步
5. 最终结果
6. 编译

编译发生在正则表达式第一次被创建时。也就是说每次创建正则表达式（因此被编译），都会创建一个新的正则表达式对象。与其它原始类型不同，这些结果永远是独一无二的（/a/ ！== /a/）。

但是，注意，构造器创建的正则表达式，允许在运行时动态创建的字符串构建和编译一个表达式，这种情况下，正则表达式只会被编译一次，但是注意，每次调用的生成的正则表达式仍然是不同的对象（对构建大量重复的复杂表达式很有用）