## 1 缓存

### 1.1 缓存简介

缓存：cache

缓存的作用，就是将一些常用的数据，存储起来，提供使用，提升性

CDN Content Delivery Network

数据库 高并发

非关系型数据库（内存型数据库） MongoDB Redis

网站静态页面缓存机制

将网页静态化，存储在服务器端

### 1.2 斐波那契数列性能问题

//f(n) = f(n-1) + f(n - 2)

var count = 0;

function fib(n){

count ++;

if(n <= 2){

return 1;

}

return fib(n - 1) + fib(n - 2);

}

fib(5);

console.log(count);

我们发现，随着计算数值越大，计算量增长速度远远超过我们的预期。

使用缓存优化：

定义一个缓存数组，存储已经计算出来的斐波那契数

1.计算的步骤

1.先从cache数组中去取想要获取的数字

2.如果获取到了，直接使用

3.如果没有获取到，就去计算，计算完之后，把计算结果存入cache，然后将结果返回

var cache = [];

function fib(n){

//1.从cache中获取数据

if(cache[n] !== undefined){

//如果缓存中有 直接返回

return cache[n];

}

//如果缓存中没有 就计算

if(n <= 2){

//把计算结果存入数组

cache[n] = 1;

return 1;

}

var temp = fib(n - 1) + fib(n - 2);

//把计算结果存入数组

cache[n] = temp;

return temp;

}

console.log(fib(6));

这样创建的缓存暴露了出来，不安全，可以使用闭包方式：

var count =0 ;

function createFib(){

var cache = [];

function fib(n){

count ++;

//1.从cache中获取数据

if(cache[n] !== undefined){

//如果缓存中有 直接返回

return cache[n];

}

//如果缓存中没有 就计算

if(n <= 2){

//把计算结果存入数组

cache[n] = 1;

return 1;

}

var temp = fib(n - 1) + fib(n - 2);

//把计算结果存入数组

cache[n] = temp;

return temp;

}

return fib;

}

var fn = createFib();

fn(6);

console.log(count);

jQuery源码分析：

function createCache() {

var keys = [];

function cache( key, value ) {

// 使用(key + " ") 是为了避免和原生（本地）的原型中的属性冲突

if ( keys.push( key + " " ) > 3 ) {

// 只保留最新存入的数据

delete cache[ keys.shift() ];

}

// 1 给 cache 赋值

// 2 把值返回

return (cache[ key + " " ] = value);

}

return cache;

}

var typeCache = createCache();

typeCache("monitor");

console.log(typeCache["monitor" + " "]);

typeCache("monitor","张学友");

console.log(typeCache["monitor1" + " "]);

typeCache("monitor","刘德华");

console.log(typeCache["monitor2" + " "]);

typeCache("monitor3","彭于晏");

console.log(typeCache["monitor3 "]);

// console.log(typeCache["monitor "]);

### 1.3 jQuery缓存处理

jQuery的查找也是从缓存中进行查找，

//eleCache

//typeCache

//classCache

//eventCache

function createCache(){

//cache对象中以键值对的形式存储我们的缓存数据

var cache = {};

//index数组中该存储键，这个键是有顺序，可以方便我们做超出容量的处理

var index = [];

return function (key, value) {

//如果传了值，就说名是设置值

if(value !== undefined){

//将数据存入cache对象，做缓存

cache[key] = value;

//将键存入index数组中，以和cache中的数据进行对应

index.push(key);

//判断缓存中的数据数量是不是超出了限制

if(index.length >= 50){

//如果超出了限制

//删除掉最早存储缓存的数据

//最早存入缓存的数据的键是在index数组的第一位

//使用数组的shift方法可以获取并删除掉数组的第一个元素

var tempKey = index.shift();

//获取到最早加入缓存的这个数据的键，可以使用它将数据从缓存各种删除

delete cache[tempKey];

}

}

//如果没有传值，只穿了键，那就是获取值

// else{

// return cache[key];

// }

return cache[key];

}

}

var eleCache = createCache();

eleCache("name","高金彪");

console.log(eleCache("name"));

var typeCche = createCache();

### 1.4斐波那契数列最终版

// 创建缓存容器

// function createCache(){

// var cache = {};

// return function (key, value) {

// //如果传了值，就说名是设置值

// if(value !== undefined){

// cache[key] = value;

// return cache[key];

// }

// //如果没有传值，只穿了键，那就是获取值

// else{

// return cache[key];

// }

// }

// }

var count =0 ;

function createFib(){

var fibCache = createCache();

function fib(n){

count ++;

//1.从cache中获取数据

if(fibCache(n) !== undefined){

//如果缓存中有 直接返回

return fibCache(n) ;

}

//如果缓存中没有 就计算

if(n <= 2){

//把计算结果存入数组

fibCache(n , 1) ;

return 1;

}

var temp = fib(n - 1) + fib(n - 2);

//把计算结果存入数组

fibCache(n, temp) ;

return temp;

}

return fib;

}

## 2 沙箱

//沙箱

//与外界隔绝的一个环境，外界无法修改该环境内任何信息，沙箱内的东西单独属于一个世界

//360沙箱模式

//将软件和操作系统进行隔离，以达到安全的目的

//苹果手的app使用的就是沙箱模式去运行

//隔离app的空间，每个app独立运行

//JS中的沙箱模式

//沙箱模式的基本模型

// (function(){

// var a = 123;

// })();

(function(){

//在沙箱中将所有变量的定义放在最上方

//中间就放一些逻辑代码

//最后，如果需要，就给外界暴露一些成员（通过window）

var sum = 0;

for(var i = 1; i<=100;i++){

sum+=i;

}

console.log(sum);

})();

//为什么要使用立即执行函数表达式（IIFE）

//因为IIFE不会再外界暴露任何的全局变量，但是又可以形成一个封闭的空间

//刚好可以实现沙箱模式

//jQuery当中的沙箱模式

//思考：直接使用上述沙箱模式，则完全独立了，jQuery就无法提供给外界一些可以调用的信息。

//如果需要在外界暴露一些属性或者方法，就可以将这些属性和方法

//加到window全局对象上去

//但是这window全局对象不可以直接引用，因为直接引用会破坏沙箱原则

//所以我们选择使用传参的形式将 window对象 传入沙箱内

//此时沙箱内使用window对象的时候，不会再去全局搜索window对象

//而使用的就是沙箱内部定义的形参

(function(win){

var Obj= {

getEle:function () {

}

}

win.Obj= win.$ = Obj;

})(window)

如果不传入，直接在内部使用全局的window对象，那么就会造成沙箱的污染。

使用传入的window，等于使用的是沙箱自己的window

//沙箱模式一般应用在书写第三方框架

//或者为第三方框架书写插件

//或者书写功能独立的一些组件

//沙箱模式的优势

//1.沙箱模式使用的是IIFE，不会再外界暴露任何的全局变量，也就不会造成全局变量污染

//2.沙箱中的所有数据，都是和外界完全隔离的，外界无法对其进行修改，也就保证了代码的安全性

//js中沙箱模式的实现原理就是

//函数可以构建作用域！上级作用域不能直接访问下级作用域中的数据

## 3 函数调用模式

//1.函数模式

//this指向window全局对象

//2.方法模式

//this指向调用这个方法的对象

//3.构造函数模式

//this 使用new创建出来的对象

//4.上下文模式

function test(){

console.log(this);

}

test();

var obj1 = {

test:function(){

console.log(this);

}

}

obj1.test();

function Person(){

console.log(this);

}

var obj =new Person();

var age = 38;

var obj = {

age: 18,

getAge: function(){

console.log(this.age);

}

};

// console.log(obj.getAge()); //18 调用者是obj

var getAge = obj.getAge;

getAge(); //38 调用者是window 可以理解为这里的getAge纯粹是一个函数，函数体和obj内的getAge一样。

注意：很多时候不能用node环境测试，因为Node环境没有window

var age = 38;

var obj = {

age: 18,

getAge: function() {

console.log(this.age);

function foo() {

console.log(this.age); //

}

foo();

}

};

obj.getAge();

其实就是判断foo是谁调用，foo()是直接调用模式，调用对象是window，自然答案是38

## 4 构造函数

//构造函数调用模式的特征

//1.构造函数的首字母要大写

//2.一般情况下和new关键字一起使用

//3.构造函数中的this指定而是new关键字创建出来的对象

//4.默认的返回new创建出来的这个对象

function Person(){

}

var p = new Person();

//构造函数的返回值：

//默认返回new创建创建出来的对，若是值类型的数据，没有影响

//若是对象类型，则返回这个对象，不会返回原来创建出来的对象

//1.工厂模式的构造函数

function Person(name,age){

var o = {

name:name,

age:age,

sayHello:function(){

}

}

return o;

}

var p = Person("张三", 18);

console.log(p);

//简单工厂模式的构造函数 创建出来的对象 跟该构造函数无关

//简单工厂模式的构造函数，实际的调用模式是 函数模式

//2.寄生式构造函数

function Person(name,age){

var o = {

name:name,

age:age,

sayHello:function(){

}

}

return o;

}

var p = new Person();

## 5 上下文调用模式apply call

//上下文

//字面意思：上面的文字，下面的文字

//JS中的上下文

//context 执行环境的意思

//this

//在上下文调用模式中，可以修改this的值，也就是可以修改函数的调用方式

var name = "lisi";

function sayHi(){

console.log(this.name);

}

sayHi();

var obj = {

name:"zs"

}

//这里sayHi输出了 lisi 因为是函数调用模式，调用者是window

//如果我们想让函数输出结果是 obj对象 内的内容，那么如何做呢？

sayHi.apply(obj); //这里通过apply函数的参数，修改了this的指向

sayHi.call(obj); //用法同上

//使用如下两个方法，可以修改函数调用上下文，也就是this的值

//apply

//api文档中的语法语句中 [] 代表括起来的东西可有可无

//函数.apply(对象, 函数需要参数列表，是一个数组)

//call

//函数.call(对象,arg1,arg2,arg3...argn)

带参数情况：

var name = "莱昂纳多·自强·郭";

function sayHello(a, b) {

console.log(this.name + "吃了"+ (a \* b) + "个馒头");

}

// sayHello(); //

var obj = {

name:"尼古拉斯·电饭·锅"

}

var arr= []

arr.push();

arr.push();

sayHello.apply(obj,arr); //

function test(a , b ,c){

}

sayHello.call(obj, 1, 2);

// sayHello.call(obj); //

//左值 右值

// function test(){

// this = 1;

// }

//call和apply的区别

//1.第一个参数都是要把this修改成的对象

//2.当函数需要参数的时候，那么apply是用数组进行参数的传递

//3.而call是使用单个的参数进行传递

//call用于确定了函数的形参有多少个的时候使用

//apply用于函数的形参个数不确定的情况

总结：

apply方法第一个参数，代表this指向的对象，传入了谁，就修改this为谁

apply方法第二个参数是一个数组，调用时，会将数组中每个元素取出作为形参，挨个传给函数。

apply和call的第一个参数都为null时，表示为函数调用模式，即this指向window

部分案例：

//案例：求一个数组中的最大值

var arr = [9, 1, 4, 10, 7, 22, 8];

//Math.max

//Math.max(1,2,34,5);

var max = Math.max.apply(null, arr);

console.log(max);

//案例：将传入的参数打印，参数之间用-相互连接

function foo() {

//return arguments.join("-");

//伪数组不具有join方法，所以这个时候就要考虑去借用一下数组的join方法

//var str = Array.prototype.join.apply(arguments,["-"]);

var str = [].join.apply(arguments,["-"]);

return str;

}

var str = foo(1, 3, "abc", "ffff", 99) // 1-3-abc-ffff-99

console.log(str);

var arr = [1,2,3,4];

console.log(arr.join("-"));

//案例：给页面上所有的 div 和 p 标签添加背景色

window.onload = function () {

var divs = document.getElementsByTagName("div");

var ps = document.getElementsByTagName("p");

var arr = [];

//little tip: push方法可以传多个参数

//arr.push(1,2,3,4,4,5)

arr.push.apply(arr,divs);

arr.push.apply(arr,ps);

//如果使用arr.push()直接把divs传进来

//那么相当于在arr中的第一个元素中存储了一个divs数组

//但是我们需要把divs中的每一个元素单独的存入arr中

//所以需要调用push方法的如下形式 push(1,2,4,4,5)

//要实现这个形式的调用，就用到了apply方法的第二个参数的特性

//在调用的时候，会第二个参数的数组，拆成每一个元素以（a,b,c,d,e,f,g） 传入函数

//相当于 arr.push(divs[0],divs[1],divs[..])

// arr.push(divs)

for (var k = 0; k < arr.length; k++) {

var ele = arr[k];

ele.style.backgroundColor = "yellow";

}

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

// var div = divs[i];

// div.style.backgroundColor = "#ccc";

// }

//

// for (var j = 0; j < ps.length; j++) {

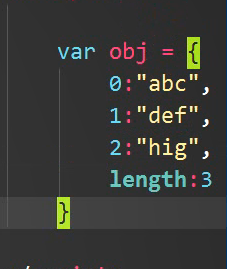
// var p = ps[j];

// p.style.backgroundColor = "#ccc";

// }

}

伪数组：



如图所示的伪数组，我们无法使用obj.0获取属性，也无法像数组那样obj[0]来获取元素，可以使用apply等方式。

var obj = {

0: "a",

1: "b",

2: "c",

length: 3

}

// var arr = [].concat(1,2,3); //会产生数组[1,2,3]

var arr = [].concat.apply([],obj); //[].push(obj[0],obj[1],obj[3]);

console.log(arr);

注意：apply出入的参数为null或者undefined的时，this会只想window



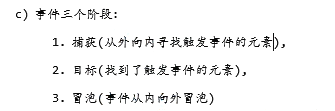
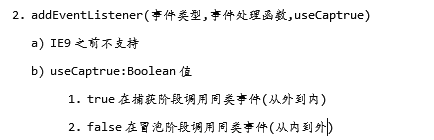
## 6 注册事件

//注册事件的方式

//1. ele.on事件类型 = function(){}

//2. addEventListener(事件类型,事件处理函数,useCapture) 第三个参数默认是false，冒泡阶段执行

//3. attachEvent(事件类型，事件处理函数)



兼容性问题：

//通用的

// div.onclick = huidiao;

//ie9以上才支持

// div.addEventListener("click",huidiao)

//ie9（不包括9）以下支持的

// div.attachEvent("onclick",function(){

// huidiao.call(div, window.event);

// })

初步封装：

//1.在注册事件的时候，判断浏览器的注册事件的方式，然后直接使用该方式进行注册事件

//复用性太差

//2.将注册事件的代码封装到一个函数中

//每次调用该函数，都会进行浏览器能力检测

//3.在函数中返回函数，让外部函数只执行一次，判断也就只会执行一次

//使用函数内创建的函数返回给外界，就可以重复使用该函数，进行事件的注册

function registeEvent(target, type, handler){

if(target.addEventListener){

target.addEventListener(type,handler)

}else if(target.attachEvent){

target.attachEvent("on"+type,handler)

}else{

target["on"+type] = handler;

}

}

最终封装：

//封装成函数，问题是每次都会判断

//使用更好的方式，避免了多次的判断

//这里存在问题就是

//我们使用统一的方式，进行事件注册

//1、注册的事件的处理函数中的，this指向不一致

//使用addEventListener的方式注册的点击事件的回调函数中的this 指向target

//但是使用attachEvent的方式注册点击事件的回调函数中的this 指向window

//2、3种注册事件的方式中，回调函数内获取事件对象的方式也是不一致的

//要让他们统一，

//在第二种的事件注册方式（attachEvent）中，手动给handler传递window.event

function createEventRegister(){

if(window.addEventListener){

return function(target, type, handler){

// this ---> window

target.addEventListener(type,handler)

}

}else if(window.attachEvent){

return function(target, type, handler) {

target.attachEvent("on" + type, function(){

handler.call(target, window.event);

})

}

}else{

return function(target, type, handler) {

target["on" + type] = handler;

}

}

}

var registeEvent = createEventRegister();

window.onload =function () {

var div = document.getElementById("div1");

registeEvent(div,"click",function(e){

console.log(e);

console.log(this);

//this---->该事件的触发对象

alert("太阳天空照，花儿对我笑，小鸟说：完了")

})

}

// div.addEventListener("click",function(e){

// this

// e.screenX;

// })

//

// div.attachEvent("onclick".function(){

//// this--->window

// window.event.screenX

// })