ECMAScript部分

**es5**

**一、浏览器简介**

**主流浏览器 内核**

IE trident

Chrome（谷歌）webkit/blink

firefox Gecko

Opera Presto

Safari webkit

JS是解释性语言，单线程的

**二、基础语法**

**（1）变量命名规则**

1.首字母是英文，\_，$开头

2.组成是英文，\_，$开头，数字

3.不可以用系统的关键字命名

**（2）数据类型**

原始值：Number String Boolean undefind null

引用值：array object function…

原始值存储在stack栈中，引用值存储在heap堆，引用值的heap地址值存储在stack栈

**（3）基础运算**

/除

0/0====NaN

1/0====Infinity无穷

a++先打印再++

++a先++再打印

字符串的比较是比较asc码

NaN不等于任何东西包括自身

undefined==null

undefined null NaN “” 0 false 转化成boolean值是**false**，其他都是**true**

&& 第一个是假返回第一个

|| 第一个是真返回第一个

！ 转化成布尔值再取反

**（4）条件语句**

**1**--for（）{

}

例子：打印所有能被3，5，7整除的100以内发数

document.write('能被3，5，7整除的100以内的数：')

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

if(i%3==0 || i%5==0 || i%7==0){

document.write(i);

document.write(' ');

}

}

**2**--while（条件判断）{

输出

循环体

}

例子：逢七过游戏

var i=1;

while(i<100){

if(i%7==0 || i%10==7){

document.write(i+" ");

}

i++;

}

**3**---switch(条件){

case 判断条件：

执行体

break/continue

}

break终止

continue是跳过终止本次，进行下一次

switch case break 没有break天生往下漏的执行

**typeof()**

typeof返回六种值 string boolean number object undefined function

typeof返回的值是一个string类型的值

typeof(a)a没有定义，返回undefined，只有这里没有定义也不会报错

**（5）显式类型转换**

**1、**Number()转化成数值型

“123”转化123

“-123”转化-123

null转化0

false转化0

true转化1

undefined转化NaN

“abc”转化NaN

**“123a”转化NaN**

**2、**ParseInt(demo,radix) radix(进制2-36，还加0进制)

把demo作为radix进制数转化为十进制，默认值是10进制数

“123a”>123 “123a1”>123

**3、** ParseFloat()转化成浮点型，保留一位

**4、**String()转化成字符串 加空字符串就能转化成字符串

**5、**Boolean()转化成布尔值

undefined null NaN “” 0 false转化为false

**6、**xxx.toString(radix)

null和undefined不能用这个方法

把10进制的xxx转化为radix进制的数

**7、**toFixed（n）保留n位小数,四舍五入

**（6）隐式类型转化**

**isNaN()**判断是否为NaN返回布尔值

把目标值放入Number()再和NaN比较

**++ -- + - （正负）**

转化成本Number 然后计算

**+**

加两侧有字符串就变成字符串

**- \* / %**

都是转化成Number

**&& || !**

**< > =**

字符串和数字比，字符串Number()再比较，

字符串与字符串比较，比较unico编码

连续比较先比较前面，得出布尔值，再与后面比较

**== !=**

{}=={}-----false两个对象是两个heap房间

**===**

不发生类型转化绝对等于

**，**

逗号操作符，看一眼前面，看一眼后面的，返回前面的

**（7）函数**

耦合（重复）低效

要讲究高内聚 低耦合

函数简化代码

取名字 第一个单词的首字母小写 后面的单词首字母大写

function sum(a,b) {

arguments存实参列表

arguments[1，2];

}

sum(1,2);

**eval(“console.log(a)”)能执行字符串**

**（8）递归**

**1.找规律**

**2.找出口**

一个函数实现斐波那契函数1 1 2 3 5 8......

function fibonacci(n){

var firstNum=1;

var secondNum=1;

var thirdNum;

if(n>2){

for(var i=1;i<=n-2;i++){

thirdNum=firstNum+secondNum;

firstNum=secondNum;

secondNum=thirdNum;

}

console.log(thirdNum)

}

else{

console.log("1");

}

}

function fibo(n) {

if (n == 1 || n == 2) {

return 1;

}

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

}

**（9）立即执行函数**

只想要执行一次，初始化功能函数

加个括号var name=(function x (形参){return}(实参))

(function a() {

console.log("123");

}())

只有表达式才能被执行符号执行，函数声明不能

**（10）预编译**

语法分析->预编译->解释执行

函数声明，整体提升

变量声明，声明提升，赋值不提升

imply global 暗示全局变量,如何变量未经声明 变量归window所有

window就是全局的域,变量声明了也归window所有

预编译在执行前一刻

1.创建AO对象就是活跃对象 Activation Object 执行期上下文

2.找形参和变量声明

3.将实参和形参相统一

4.在函数体fn里找函数声明function x(){}的样式，值赋予函数体，

function fn(a) {

console.log(a);//预编译结果fna

var a = 123;

console.log(a);//重新赋值123

function a() {}

console.log(a);//赋值123

var b = function () {}

console.log(b);//赋值fn

function d() {};

console.log(d);//预编译结果fnd

}

fn(1);

AO{

a:undefined,-->1,-->function a(){},-->解释执行123,

b:undefined,

d:function d(){},

}

**全局也有预编译**

全局生成GO Global Object===window

if（）括号里里不能有函数声明

var x=1;

if(function f(){}){

x+=typeof(f);

}

console.log(x);

**（11）闭包**

但凡内部的函数被保存到外部，一定形成了闭包

函数不执行只定义 那不会去读函数里面的

闭包会导致作用域链不释放，占空间，造成内存泄漏

function text() {

var arr = [ ];

for (var i = 0; i < 10; i++) {

arr[i] = function () {

document.write(i + " ");//开始没有执行，不随外面的i变化

}

}

return arr;保存出去了

}

var maArr = text();

for (var j = 0; j < 10; j++) {

maArr[j]();//开始执行各个function的时候才会去让函数体内的i找对应的i，也就是共用的text里的AO里的i值，刚好自循环到10停止

}

function text() {

var arr = [ ];

for (var i = 0; i < 10; i++) {

(function (j) {

arr[j] = function () {

document.write(j + " ")

}

}(i));

}

return arr;

}

var maArr = text();

for (var j = 0; j < 10; j++) {

maArr[j]();

}

首先fori循环把i=0实参传给形参j，使j=0，arr[0]被保存到arr[]里传出，先不执行后面的函数，依稀把十个i都传入，arr[]里

就有了十个值，执行最里面的函数，把j=0传入到，执行arr[0]里的函数，j现在这里立即执行函数里等于0，所以输出0，依次

**闭包的私有化属性（构造函数）**

function Deng(name, wife) {

var prepareWife = "xiaozhang"闭包的私有化变量

this.name = name;

this.wife = wife;

this.divoce = function () {

this.wife = prepareWife;

}

this.changePrepareWife = function (ta) {

preparewife = ta;

}

this.sayPrepareWife = function () {

console.log(prepareWife);

}

}

**（12）对象**

有属性有方法

var mrDeng = {

name: "MrDeng",

age: 40,

sex: "male",

health: 100,

smoke: function () {

console.log("i am smoking! cool!")

this.health--;

},

drink: function () {

console.log("i am drinking!")

this.health++;

}

}

**增**

mrDeng.wife = "xiaoliu";

**查**

mrDeng.health;

**改**

mrDeng.age = 41;

**删**

delete mrDeng.age;

对象的创建方法

1.**var xxx={}**对象字面量/对象直接量

2.构造函数

1）.系统自带的构造函数 object（）

**var obj = new Object()**;

2）.自定义

3**.var obj=Object.create(obj)**

构造函数和函数造型上一样

**构造函数首字母大写，函数是第一个不大写**

function Car(color) {

this.color = color;

this.name = "BMW";

this.height = "1400";

this.lang = "4900";

this.weight = 1000;

}

var car = **new** Car("red");

var car1 = **new** Car();

console.log(car);

function Student(name, age, sex) {

var this={};隐式

this.name = name;

this.age = age;

this.sex = sex;

this.grade = 2020;

ruturn this;隐式

}

var zhang = **new** Student("zhang", 11, "male");

题目：

1、运行test(); new test();的结果分别是什么

var a=5;

function test(){

a=0;

console.log(a);

console.log(this.a);//this 指向window

var a;

console.log(a)

}

test();

new test();会隐式的再函数体里写入一个this{ }

var x = 1, y = z = 0;

function add(n) {

return n = n + 1;

}

y = add(x);

function add(n) {

return n = n + 3;

}

z = add(x);

x=1

y=4

z=4

函数有一个预编译都是过程

能否打印出（1，2，3，4，5）

function foo(x) {

console.log(arguments);

return x;

}

foo(1, 2, 3, 4, 5);

function foo(x) {

console.log(arguments);

return x;

} (1, 2, 3, 4, 5);

把括号当成运算,上下分离

(function foo(x) {

console.log(arguments);

}(1, 2, 3, 4, 5));

function foo() {

bar.apply(undefined, arguments);

}

function bar(x) {

console.log(arguments);

}

foo(1, 2, 3, 4, 5);

写一个方法，求一个字符串长度。（提示：字符串有一个方法charCodeAt（）；一个中文占两个字节，一个英文占一个字节。str.charCodeAt(n)表示str字符串的第n个字节的ASCII码数,大于255则占两个字节）

function stri(str) {

var count = str.length;

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

if (str.charCodeAt(i) > 255) {

count++;

}

}

console.log(count);

}

var str = String(window.prompt("请输入"));

stri(str);

**三、内置函数**

**（1）Math对象**

**abs（x）；**

**cell（x）；向上舍入**

**floor（x）；向下舍入**

**max（x，y,…）；取最大**

**min（）；取最小**

**pow（x，y）；x的y次方**

**random（）随机数（0，1）**

**round（x）四舍五入**

**（2）String对象**

**str.charAt(n)第n个字符**

**str.conchat(str1)拼接数组**

**str.replace(x,y)字符串里的x换成y替换一次**

**str.** **substring(a,b)截取字符串，从第a位截取到第b位**

**str.** **substr(a,b) 从第a位截取b位**

**str.slice(a,b) 从第a位截取到第b位**

**str.indexOf(‘a’,3)搜索字符串“a”从第3位往后出现位置的索引，检索不到返回-1**

**str.charCodeAt(index)** **方法可返回指定位置的字符的 Unicode 编码。这个返回值是 0 - 65535 之间的整数。**

**（3）Date对象**

**Date的属性：constructor和prototype**

**Date的方法：**

**1.var date=new Date( );date记录此时此刻的时间**

**2.date.getDate();返回本月的第几天（1-31）**

**3.date.getDay();返回本周的第几天（0-6）**

**4.date.getMonth();返回本年的第几月（0-11）**

**5.date.getFullYear();返回四位数的年份**

**6.date.getHours();返回现在的小时数（0-23）**

**7.date.getMinutes();返回现在的分（0-59）**

**8.date.getSeconds();返回现在的秒（0-59）**

**9.date.getTime();返回从1970年1月1日0（计算机纪元年）时刻起至今的毫秒数可以验证程序的运行效率**

**封装一个函数准确打印何年何月何日何时何分何秒**

function nowTime(){

var date=new Date();

var year=date.getFullYear();

var month=date.getMonth()+1;

var day=date.getDate();

var hours=date.getHours();

var minutes=date.getMinutes();

var seconds=date.getSeconds();

console.log("现在是北京时间"+year+"年"+month+"月"+day+"日"+hours+"时"+minutes+"分"+seconds+"秒")

}

**四、**

**（1）包装类**

原始值不能有属性和方法

var str = "abc";

str += 1;

var te = typeof (str);

if (te.length == 6) {

te.s = "111111";

}

console.log(te.s);结果undefined

**（2）原型**

每一个对象都有\_\_proto\_\_属性，指向原型，指向可以被修改

自身上没有的属性可以向原型上的\_\_proto\_\_上找

A.prototype就是对象的原型，就是构造函数的祖先，是一个空对象

**(prototype属性一定是在构造函数上的，普通的对象obj{ }在obj.constructor就是该对象的构造函数)**

**var ObjNO2={ };**

**ObjNO2.constructor.prototype.lastName="hu";**

A.constructor查看对象构造函数

A.prototype.name = "AA"

function A() {

var this ={\_\_proto\_\_:A.prototype}

}

var a = new A();

console.log(a.name);

Person.prototype.name = "sunny";

function Person() {

}

var person = new Person();

Person.prototype = {

name: "cherry",

}

console.log(person.name);

**（3）原型链**

链接点就是prototype

Object.prototype是所有原型的最底层，是终端

var=obj{}也是有原型的

绝大多数对象的最终都会继承自Object.protoype

var obj=Object.create(null),就没原型，自己添加都不能

原型是一个隐式的内部属性

**call/apply**

改变this指向

call

function Person(name, age) {

this.name = name;//obj.name

this.age = age;

}

var person = new Person("zhang", 23);

var obj = {

}

**Person();与Person.call();是一样的（）里没有加东西是一样的**

Person.call(obj, "jing", 21);

function People(name, age) {

this.name = name;

this.age = age;

}

借用 别人的方法

function Student(name, age, sex, tel, grade) {

Person.call(this, name, age);

Person.apply(this,[name,age])

this.sex = sex;

this.tel = tel;

this.grade = grade;

}

var student = new Student("hu", 25, true, "131", 2015);

**apply与call的传参列表不同**

**apply后面必须一个数组**

**（4）继承**

由于原型链过多的继承了一些没用的东西，所以要建立新的继承关系

1.call/apply

2.共享原型

Father.prototype.lastName = "hu"

function Father() {

}

function Son() {

}

function inherit(Targer, Origin) {

Targer.prototype = Origin.prototype;

}

inherit(Son, Father);//传入的数字不要传错顺序

var son = new Son;

3.圣杯模式

Father.prototype.lastName = "hu"

function Father() {

}

function Son() {

}

function inherit(Targer,Origin){

function F(){}

F.prototype=Origin.prototype;

Targer.prototype=new F();

Targer.prototype.constuctor=Targer;

Targer.prototype.uber=Origin.prototype;//真继承自

}

inherit(Son,Father);

var son=new Son();

4.雅虎圣杯模型

Father.prototype.lastName = "hu";

function Father() {

}

function Son() {

}

var inherit = (function () {

var F = function () { };//成为下面的函数的私有化

return function (Targer, Origin) {

F.prototype = Origin.prototype;

Targer.prototype = new F();//切记构造函数的new

Targer.prototype.constuctor = Targer;

Targer.prototype.uber = Origin.prototype;

}

}());

inherit(Son,Father);

var son = new Son();

var father = new Father();

**（5）命名空间**

var name = "456";

var putWords = (function () {

var name = "123";

function words() {

console.log(name);

}

return function () {

words();

}

}())

putWords();

外面的name和里面的name名字一样 但并不影响

**（6）连续调用**

var continuous = {

a: function () {

console.log("A");

return this;

},

b: function () {

console.log("B")

return this;

},

c: function () {

console.log("C")

return this;

}

}

continuous.a( ).b( ).c( ) 相当于是continuous调用a，this指向continuous

**（7）避免变量名的污染**

var arr = [1, 2, 3, 10, 5]

Array.prototype.myUnshift = function () {

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

this[j + arguments.length] = this[j]

}

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

this[i] = arguments[i]

}

}

arr.sort(function(a,b){

return b-a

})

var name = "bcd";

var init = (function () {

var name = "abc";

function callname() {

console.log(name);

}

return callname;

}())

init();

**a.name---->a["name"]访问属性两种方式**

var deng = {

wife1: { name: "xiaozhang" },

wife2: { name: "xiaoxu" },

wife3: { name: "xiaowang" },

wife4: { name: "xiaoyu" },

saywife: function (num) {

return this["wife" + num];

}

}

**（8）对象的枚举**

var array = [1, 2, 3, 4, 5, 6];

**遍历数组的过程**

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

console.log(array[i]);

}

**遍历对象的过程（forin循环）**

var obj = {

name: "hufei",

age: 25,

sex: true,

}

for (var objName in obj) {

console.log(objName)

console.log(obj[objName] + ":" + typeof (obj**[objName]**));

但访问obj的属性obj.objName系统会转换成obj["objName"]转化成了字符串，不在是变量，所以写法如上

}

**（9）对象的相关方法**

**1、hasOwnProperty()**

**obj.hasOwnProperty(prop)**判断是否是自身属性

**2、in**

**in操作** “height” in obj 返回布尔值，in只能判断能不能访问到这个属性，父级也可以

**3、instanceof**

A对象 **instanceof** B构造函数(必须是构造函数)

A是不是B构造函数构造出来的

看A对象的原型链上有没有B的原型

**（10）区别数组和对象的方法**

**1**查看构造函数

[ ].constructor-------Array()

{ }.constructor-------Object()

**2** instanceof

[ ].instanceof Array-----true

{ }.instanceof Array-----false

**3** toString方法

Object.prototype.toString.call([ ])------[object Array]

Object.prototype.toString.call({ })------[object Object]

Object.prototype.toString.call(123)------[object Number]

**（11）深度克隆（不考虑function）**

1.遍历对象，是否是原始值 typeof()是否是object

2.判断是数组还是对象

3.建立相应的数组或对象

function deepClone(origin, target) {

var target = target || { };

toStr = Object.prototype.toString

arrStr = "[object Array]"

for (var prop in origin) {

if (origin.hasOwnProperty(prop)) {

if (origin[prop] != "null" && typeof (origin[prop]) == "object") {

if (toStr.call(origin[prop]) == arrStr) {

target[prop] = [ ];

} else {

target[prop] = { };

}

deepClone(origin[prop], target[prop]);

}

else {

target[prop] = origin[prop];

}

}

}

return target;

}

**（12）封装判断数值类型type**

1.原始值 还是 引用值

2.区分引用值

function type(target) {

var template = {

"[object Array]": "array",

"[object Object]": "object",

"[object Number]": "number-object",

"[object Boolean]": "boolean-object",

"[object String]": "string-object"

}

if(target===null){

return "null";

}

else if(typeof(target)=="object"){

var str=Object.prototype.toString.call(target);

return template[str];

}

else{

return typeof(target);

}

}

**（13）this指向**

1.函数预编译过程this----->window

2.全局作用域也是指向window

3.call/apply可以改变函数运行时的this指向

4.obj.functionName();functionName()里的this指向obj

**1..**

function text(c){

var a=123;

function b(){}

}

AO{

argument:[1],

this:window,

a:undefined,

b:function(){},

c:1

}

text(1);//函数执行前有一个预编译

不new执行预编译

如果new text（1）;this指向text

**2..**

全局打印this 也是指向window

**3..**

function A(b,c){

this.b=b;

this.c=c

}//this指向A

function D(b,c,e){

A.call(this,b,c)//A.apply(this,[b,c])两者一样

this.e=e

}

var d=new D(1,2,3)

console.log(d.a);

**4..**

var obj = {

a: function () {

console.log(this.name)

},

name: "ab"

}

obj.a();//ab

不调用a方法，this走预编译指向window

**练习题**

var name="222";

var a={

name:"111",

say:function(){

console.log(this.name);

}

}

var fun=a.say;就是把函数拿到全局去执行

fun();222

a.say();111

var b={

name:"333",

say:function(fun){

fun();这里是一个预编译环节指向window,这个方法的执行不是a也不是b所以在全局

}

}

b.say(a.say);222

b.say=a.say;

b.say();333

**（14）callee和caller**

**arguments.callee**

**callee指向函数的引用**

function text(){

console.log(arguments.callee)-----打印的东西就是text构造函数

}

text();

var foo = "b";

function print() {

this.foo = "a"

console.log(foo);

}

print();a

new print();b

访问的是foo 不是this.foo，函数体里面没有foo，上函数体外找

function p(){

this.foo="a"

console.log(this.foo)a 这样才指向a

}

new p();

p();

上面两个都指向a

运行text(); new text();结果分别是什么

var a = 5;

function text() {

a = 0;

console.log(a)

console.log(this.a)

var a;

console.log(a);

}

text(); 0 5 0

new text();0 undfinde 0

求以下代码的执行结果

function print() {

var marty = {

name: "marty",

printName: function () {

console.log(this.name)

}

}

var test1 = { name: "test1" };

var test2 = { name: "test2" };

var test3 = { name: "test3" };

test3.printName = marty.printName;

var printName2 = marty.printName.bind({ name: 123 })

marty.printName.call(test1);---test1//call和apply改变了this指向

marty.printName.apply(test2);---test2

marty.printName();---marty//marty调用了printName,所以this指向marty

printName2();---123

test3.printName();---test3//test3调用这个函数

}

print();

**（bind的用法**

this.x = 9; 在浏览器中，this 指向全局的 "window" 对象

var module = {

x: 81,

getX: function() { return this.x; }

};

module.getX(); 81

var retrieveX = module.getX;

retrieveX();

返回 9 - 因为函数是在全局作用域中调用的

创建一个新函数，把 'this' 绑定到 module 对象

可能会将全局变量 x 与 module 的属性 x 混淆

var boundGetX = retrieveX.bind(module);

boundGetX(); /81

**）**

var bar={a:"002"};

function print(){

bar.a="a";

Object.prototype.b="b";

return function inner(){

console.log(bar.a);

console.log(bar.b);

}

}

print()();---ab

**arguments.callee**

**arguments.callee就是指向函数体本身**

var num=(function(n){

if(n==1){

return 1;

}

return n\*arguments.callee(n-1)

}(100))

**caller**

caller指在什么环境下被调用

function A(){

B();

}

function B(){

console.log(B.caller)

}

A();

打印出的结果就是B在A的环境下被调用，结果指向A这函数体

**（15）三目运算符**

**条件判断 ？ 是 ： 否**

var mun= 1>0 ? 2+2 : 1+1

**（16）数组**

定义

1>var arr=[ ]

2>var arr=new Array() **()括号里可以传值，如果只穿一个值，表示数组的长度**

**数组的方法**

**改变原数组的方法**

**1.arr.push(a),**往数组的末尾加a，可以同时加多个

**封装mypush方法**

Array.prototype.mypush=function(){

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

this[this.length]=arguments[i];

}

return this.length

}

**2.arr.pop();**剪切最后一位，你需要传参数，调用一次剪切当前的最后一位一次

**mypop方法**

Array.prototype.mypop=function(){

var a=this[this.length-1];

delete this[this.length-1];

this.length-=1;

return a;

}

**3.arr.unshift(n);**往数组的前边加n，可以加多个

Array.prototype.myunshift = function (num) {

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

this[this.length-i]=this[this.length-i-1]

}

this[0]=num;

return this.length

}

**4.arr.shift(n);**在前面的减去第一位n

**5.arr.reverse();**数组内的每一个的位置逆转123---321

**6.arr.splice(index，homany，item1，item2);**

第一位：从第几位开始，第一位是第0号。（如果是负数-1表示从倒数第一位开始截取）

第二位：向后删除多少长度，0则不删除

第三位：在切口处添加新的数据。

返回被删除的数组

**7.arr.sort();**按字符表升序

arr.sort().reverse();按字符表降序

**冒泡排序**

arr.sort(function(a,b){

if(a>b){

return正数

}

else{

return 负数

}

});

1.必须写两个形参，上述的a b 两个两个进行比较，比较就会变成数值型，然后进行排列

2.看返回值，当返回值是负数的时候，那么前面的数a排在前面；当返回值是正数的时候，后面b的数排在前面；为0不动,也是升序的排列方法

第一步

arr.sort(function(a,b){

if(a-b>0){

return正数

}

else if(a-b<0){

return 负数

}

});

第二步

arr.sort(function(a,b){

if(a-b>0){

return a-b

}

else if(a-b<0){

return a-b

}

});

第三步结论

arr.sort(function(a,b){

return a-b

});

就是升序

arr.sort(function(a,b){

return b-a

});

就是降序

衍生出乱序使用Math.random()随机产生一个0到1之间的一个数

arr.sort(function(a,b){

return (Math.random-0.5)

});

**冒泡排序的封装**

**Array.prototype.mySort= function() {**

**for (var j = 0; j < this.length; j++) {**

**for (var i = 0; i < this.length - j; i++) {**

**if (this[i] < this[i - 1]) {**

**var arrayTemp = this[i];**

**this[i] = this[i - 1];**

**this[i - 1] = arrayTemp;**

**}**

**}**

**}**

**}**

**选择排序的封装**

**Array.prototype.mySortTwo = function () {**

**for (var j = 0; j < this.length; j++) {**

**var min = j;**

**for (i = j+1; i < this.length; i++) {**

**if (this[i] < this[min]) {**

**min = i;**

**}**

**}**

**var arrayTemp = this[j];**

**this[j] = this[min];**

**this[min] = arrayTemp;**

**}**

**}**

**插入排序的封装**

**Array.prototype.mySortThree = function () {**

**for (var j = 1; j < this.length; j++) {**

**for (var i = j - 1; i >= 0; i--) {**

**if (this[i + 1] < this[i]) {**

**var arrayTemp = this[i];**

**this[i] = this[i + 1];**

**this[i + 1] = arrayTemp;**

**}**

**}**

**}**

**}**

**不改变原数组的方法**

**8.arr.concat(arr1);**把arr和arr1并在一起，arr在前。

**9.arr.toString();**把数组变成字符串

**10.slice();**截取

里面没有参数 整个截取（类数组）

里面一个参数 从第几位开始截取到最后一位

里面两个参数 从第几位开始，截取到第几位

**11.join(n);**用n连接数组里的每一位，散列方式连接

**12.split(n)**;用n去断开形成数组，与join互逆

**13.数组去重**

Array.prototype.unique=function(){

var temp={ };

var arr=[ ];

var len=this.length;

for(var i=0;i<len;i++){

if(!temp[this[i]]){

temp[this[i]]=true;

arr.push(this[i])

}

}

return arr;

}

**14多维数组**

a[0][0][0]

**类数组**

arguments 就是一个类数组

var obj={

“0”:1,

“1”:2,

“length”:2，

push:Array.prototype.push

}

类数组的组成：1.属性名要为索引位（数字）属性

2必须要有length

3.最好有push

**（18）错误的捕捉**

try{ }

catch(error){ }

1在try里发生的错误，就不会执行错误之后的代码

2 没有错那正常执行

3 如果try里有错误，catch会把错误信息封装到error里提供给使用者操作

操作的方法：error.name error.message

用注释也可以调试错误

**（19）错误的类型**

error.name有六种

1.EvallError:evall的使用与定义不一致

2.RangeError:数值越界

3.TypeError: 操作数组型错误，（数组方法操作对象方法）

4.ReferenceError:非法或不能识别的引用数值

5.SyntaxError:发生语法解析错误（比如写了中文）

6.UREError:URI处理函数使用不当（引用地址）

**（20）es5严格模式**

基于es3.0的方法，es5.0的新增方法

冲突部分就是es5.0的严格模式

**启用方法**

**“use strict”**

写在第一行就是全局使用严格模式

写在函数体里面就是部分严格

向后代浏览器兼容

**（21）严格模式的内容**

**1.不允许使用with**

with(obj)

obj成了最顶端的域

改变作用域链了，所以弃用

**2.argument.callee不能用**

**3.text.caller不能用**

**4.变量赋值前必须声明**

**5.局部的this必须被赋值，预编译的this指向undefined**

**6.拒绝重复属性和参数**