**事件：**

**文档或浏览器窗口中发生的一些特定的交互瞬间**

事件处理程序：

1、HTML事件处理程序

缺点就是html代码和js的耦合性太强

2、DOM0级事件处理程序

元素.onclick = function(){}

3、DOM2级事件处理程序

标准：**addEventListener("click",fn1,false)** 第三个参数：是否在捕获阶段调用事件处理程序，默认false

　　　非标准**：attachEvent("on"+事件名称,事件函数)**

　　　　　　　ie不支持捕获，所以只会在冒泡阶段调用事件处理程序

**两种事件流：**

**dom0级和html事件处理程序只有冒泡**

1、**冒泡**

最具体的元素先接收事件，然后依次往外传递，一直传递到document

阻止事件冒泡

2、**捕获**

最不具体的元素先接收事件，然后依次往里传递，一直传递最具体的元素

由外到内

　　　　addEventListener第三个参数是是否在捕获阶段执行函数

**低版本的ie不支持捕获，ie提出的是冒泡流，网景提出的是捕获流**

attachEvent()没有捕获事件对象

**对象的格式：两种访问方法:.和[]**

btn.onclick = function(){}

　　　btn[**"onclick"**] = function(){} (**括号里面如果是变量不加引号**)

　　　　　　中扩号的方式优点：可以写变量、访问一些不合法的名称

遍历对象 for in

**事件取消**

1、DOM0级

**元素.onclick = null**

2、DOM 2级

**标准：removeEventListener()** 与添加事件传入参数相同，所以**不能移除匿名函数的事件**

**非标准：detachEvent()**

事件流：

DOM是树形结构，当标签元素产生一个事件的时候，该事件会在元素节点与根节点之间的路径传播，路径所经过的节点都会收到

该事件，这个传播的过程就叫做dom事件流。

**IE提出的是冒泡流，而网景Netscape提出的是捕获流**

目前低版本的IE浏览器还是只能支持冒泡流，所以为了能够兼容更多的浏览器，建议大家使用冒泡流。

一个完整的JS事件流是从document开始，最后回到document的一个过程,有的浏览器一直冒泡到window

dom2级事件规范要求从document开始，但是有的浏览器是从window开始的

　　dom2级事件流包括3个阶段:事件捕获阶段、处于目标阶段、事件冒泡阶段，首先发生的是事件捕获

**取消冒泡 （e)为事件对象**

oDiv.onclick = function(e){

**var ev = e || window.event;//兼容的写法**

**（var ev = e?e:window.event;）**

for(var i in ev){

console.log(i+":"+ev[i]);

}

**ev.stopPropagation();//阻止冒泡**

**ev.cancelBubble = true;//兼容低版本的ie**

if(ev.stopPropagation){

ev.stopPropagation();

}else{

ev.cancelBubble = true;

}

}

**取消默认事件**

function fn(e){

var ev = e || window.event;//兼容的写法

（var ev = e?e:window.event;）

if(e.preventDefault){

e.preventDefault();//支持标准浏览器

}else{

e.returnValue = false;//兼容低版本的ie

}

}

**事件委托**

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

lis[i].onclick = function(e){

var ev = e || window.event;

**ev.target触发该事件的最具体的元素** 不支持非标准的浏览器 **非标准的使用ev.srcElement**

console.log(ev.target);

**var ele = ev.target || ev.srcElement;//获取触发该事件的最具体的元素**

alert(ele.innerHTML);

}

}

**采用事件委托（原理：冒泡） 把事件委托给父级或者祖父等等**

ul.onclick = function(e){

**this和事件对象里面的currentTarget 在事件中指的是绑定该事件的元素 this和target可能相等**

**this一定和currentTarget相等**

var ev = e || event;

var obj = ev.target || ev.srcElement;

**tagName是target的属性 大写的**

**var tagName = obj.tagName;//大写的**

if(tagName=="LI"){

alert(obj.innerHTML);

}

}

**事件类型**

**UI事件**

onload 加载事件

onscroll 滚动条事件

error 错误事件

**鼠标事件**

onmouseover 鼠标滑过事件

onmouseout 鼠标离开事件

onclick 单击事件

ondblclick 双击事件

onmouseup 鼠标抬起事件

onmousedown 鼠标按下事件

onmousemove 鼠标移动事件

onmousewheel 鼠标滚轮事件

**键盘事件**

onkeydown 键盘按下事件

onkeyup 键盘抬起事件

onkeypress 键盘按下又抬起(按下字母事件)

**文本事件**

oninput 文本事件（在输入框里只要输入字符就会触发）

**焦点事件**

onfocus 获取焦点事件

onblur 失去焦点事件

**表单事件**

onsubmit 提交事件

onreset 重置事件

type 事件对象的类型

currentTarget 事件对象的源元素

target 点击的元素

**常用事件**

**onload 当页面所有的都加载完再执行函数里面的东西 通常情况下：window、body、img**

**页面中只能有一个window.onload**

window.onload = function(){

var div = document.getElementsByTagName("div")[0];

console.log(div);

}

**onerror js脚本出错的时候触发**

**onresize**

**鼠标事件**

document.**onmouseup** = function(){

console.log("鼠标抬起");

}

**onmouseover** 鼠标滑过

**onmousedown** 鼠标按下

**onmouseout** 鼠标移出

**onmousemove** 鼠标移动

**onmouseenter** 鼠标滑过 （后）

**onmouseleave** 鼠标移出 （后）

**事件对象属性**

document.onmousemove = function(e){

var ev = e || event;

**console.log(ev.type);//事件类型**

**clientX clientY 鼠标到可视区、视口 的距离**

**pageX、pageY 鼠标到页面的距离 会算上滚动条的距离**

}

document.getElementsByTagName("div")[0].onmousemove = function(e){

var ev = e || event;

**console.log(ev.offsetX+","+ev.offsetY)//鼠标相对于触发该事件的元素的距离**

}

offsetLeft

元素的属性：

clientWidth\clientHeight 元素的width+padding

offsetWidth\offsetHeight 元素的width+padding+border

offsetLeft、offsetTop 元素距离有定位父级的距离，如果父级都没有定位，就相当于文档的距离

**事件对象：**

**clientX\clientY 鼠标相对于可视区 视口的距离**

**pageX\pageY、 鼠标相对页面的距离 （clientX+scroolLeft\clientY+scrollTop）**

**offsetX\offsetY 鼠标相对于触发该事件的元素的距离**

**焦点事件**

**onfocus 获取焦点**

ele.focus();//自动获取焦点

**onblur 失去焦点**

**键盘事件**

**按键抬起**

document.**onkeyup** = function(){

}

**按下 字符键 按下会触发 按住不动会连续触发**

document.**onkeypress** = function(){

}

**按下 任意键都会触发 按住不动会连续触发**

document.**onkeydown** = function(e){

var ev = e || window.event;

console.log(**ev.keyCode**);

**enter 13 空格键 32 左上右下37/38/39/40**

**0-9 48**

**a-z 65**

}

**文本事件**

var oInput = document.getElementsByTagName("input")[0];

**当value发生变化的时候会触发，通过脚本改变值不会触发 不支持低版本的ie**

oInput.**oninput** = function(){

console.log(1234);

}

**ie特有的 属性改变就会触发**

oInput.**onpropertychange** = function(){

console.log(3455);

}

**当value发生改变 并且失去焦点 才会去触发**

oInput.**onchange** = function(){

console.log("change触发了");

}

**正则**表达式

console.log(/[sd]/g==/[sd]/g);//false 正则不等于正则

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，

这个“规则字符串”用来表达对字符串的一种过滤逻辑

通过正则表达式，从字符串中获取我们想要的特定部分

创建方式

1、字面量创建

**reg = /正则/修饰符**  修饰符可以省略

2、实例化方式

**reg = new RegExp("正则","修饰符")**

两种方式区别：**第二种可以使用变量**,**并且\需要使用\\**

第二种创建方式 1、区别 多加\ 2、可以使用变量

var reg1 = new RegExp("\\d","g");

var temp = "\\d{2}";

console.log(reg1.test("23wse")); //true

reg2 = new RegExp(temp,"g");

console.log(reg2.test("23wse")); //true

修饰符：

**i忽略大小写**

**修饰符m 查找多行**

**g全局**，被应用于所有字符串，而非在发现第一个匹配项时立即停止

**常用方法：**

**正则表达式的常用方法**

正则下面的方法：exec、test

字符串下面的方法：search、match、replace、split

**test(要检测的字符串)正则的方法 检测字符串是否匹配我们设定的模式 ，有返回值boolean 如果匹配了就返回true false**

**reg.exec(字符串)** 正则的方法 **exec() 返回符合匹配模式的字符结果 是一个数组**，如果没找到匹配项 返回**null**，返回的数组第一个元素是匹配结果第一项，第二个元素开始就是符合匹配的子匹配项 元字符：有特殊功能的字符叫做元字符

**正则的属性lastIndex 上一次查找文本后的第一个字符位置**

**\d 表示数字 指的是0-9的数字**

**\D表示非数字**

**\w 表示字母数字、下划线**

**[abc]表示字符集合，匹配包含的任意字符,相当于a或b或c**

**|表示 a|b|c表示a或b或c**

**[a-c]表示范围，例如\d就等价于[0-9],[a-z]匹配a到z的小写字母**

**.除了\n之外的任意一个字符**

**\s空格或空白**

**\S非空格**

**^ 开始**

**$ 结束**

**[a-zA-Z] 判断大小写字母**

**[a-zA-Z0-9\_]判断大小写数字、下划线 \w**

()分组 (1). 在被修饰匹配次数的时候，括号中的表达式可以作为整体被修饰

(2). 取匹配结果的时候，括号中的表达式匹配到的内容可以被单独得到

**\转义**

**\b用来设置单词界限。**匹配单词（由字母数字或下划线组成的）的开始或结束(匹配开始时，单词之前不能有\w;匹配结束时，单词之后不能有\w

写在表达式某个字符或字符串之前，表示此字符或字符串之前不能有\w对应的字符；写在字符后面，表示此字符之后不能有\w对应的字符。 所以\b可以只有一个放在字符前面或后面，也可以有2个（表示字符前后都不能有\w对应的字符）。

\1重复

**[^a]排除除了a**

量词：

**{n} 匹配n个字符**

**{n,m} 匹配n到m次**

**{n,} 匹配至少n个字符**

**+ 至少1个**

**? 0到1个**

**\* 0到多个**

**字符串常用的方法：**

**match()** **把符合匹配模式的字符取出来 返回的是数组，没有匹配项也是返回null**

**trim 前后空格去掉**

**match**方法是字符串的方法，会把符合匹配模式的字符取出来，组成一个新的数组，如果匹配模式没有g，

\* 则返回的数组第一个元素是对应的匹配项，第二个元素开始就是子匹配项(第一个小括号)

\* **如果匹配模式是g，则返回所有的匹配项，不返回子匹配项**

**exec**是正则的方法 返回的结果是数组，没有匹配项返回null，如果匹配模式是g ，那么它再次查找的时候会在上一次查找的基础之上去查找

**对于exec方法，即使在模式中设置了全局标志g，他每次也只会返回一个匹配项**。

\* **如果匹配模式不是g ，始终从字符串 0 去查找**

**replace** 第二个参数是函数的时候，这个函数第一个参数指的是 匹配项 第二个参数开始指的是第一个小括号里面的东西 第三个参数 第二个小括号....第99个小括号

这个函数有返回值，返回值就是替换的文本

reg = /(-)(\d)/g;

str = "-2-34";//2-3-4

console.log(str.replace(reg,"**$2$1**"));//$1指的是第一个小括号的内容 。。。$99

**\b 单词边界**，不占位置，什么样的是单词边界(除了\w)

**search()**第一个与regexp相匹配的子串的起始位置，无结果，返回-1，不执行全局匹配，它将忽略标志 g

**split**

var reg = /a/g;

console.log("desadae".search(reg));//返回符合条件的字符的索引、下标 3

var str="dwe,deee;frrr,gdfgd";

console.log(str.split(/[,;]/)); //["dwe", "deee", "frrr", "gdfgd"]

**. ? （转义）**

**通过：\ 这个符号来进行转义**

**需要转义的符号 ： [ ] { } ( ) . ? \* + ^ $ \ / |**

str="dee.deee";

reg=/\./;

console.log(reg.test(str)); //true

**匹配汉字**

var reg = /[**\u4e00-\u9fa5**]/;

console.log(reg.test("的333")); //true

**数据类型**

基本数据类型：

number、string、boolean、null、undefined

引用数据类型 object

只能给引用数据类型的值动态添加属性

存放的位置：

**引用类型值在内存中占据空间大小不固定，因此被保存在堆内存中。**

**基本类型值在内存中占据固定大小空间，因此被保存在栈内存中。**

**var n = 2;**

**console.log(fn(n)); //3**

**console.log(n); //2**

**//基本数据类型函数传参传递的是具体的值**

**function fn(num){**

**num++;**

**return num;**

**}**

**//**

**var array = [2,3,4];//是引用数据类型函数传参传递的地址值**

**//var array1 = new Array();**

**console.log(fn2(array)); //2,3,4,5**

**console.log(array); //2,3,4,5**

**function fn2(arr){**

**arr.push(5);**

**return arr;**

**}**

复制变量的值：基本类型是复制具体的值,引用类型复制的是地址值或者称作指针

传递参数：按值传递

基本类型的参数传递和基本类型复制变量一样

引用类型的参数传递和引用类型复制变量一样，传递的是地址值

检测类型：

typeof 检测变量是字符串、数值、布尔值、还是undefined

console.log(typeof [1,23]);//object

console.log(typeof function(){});//function

console.log(typeof {});//object

console.log(typeof typeof null);//string

typeof用来检测基本数据类型的

Instanceof

检测引用数据类型 被检测的对象 instanceof 构造函数 返回值是Boolean

Object RegExp Date Array Function

console.log([123,4] instanceof Array);// true

console.log([123,4] instanceof Object);//true 所有的对象都是Object的实例

**instanceof 检测类型**

function fn(){}

console.log(fn instanceof Array) false

用来检测引用数据类型

2、执行环境与作用域

执行环境：定义了变量和函数有权访问的其他数据，决定了他们各自的行为，每个环境都有一个与之关联的变量对象，环境中定义的所有变量和函数都保存在这个对象中

**全局执行环境是最外围的一个执行环境，它的对象就是window**

某个执行环境里面的代码执行完毕后，保存在其中的所有的变量和函数也会随之销毁，全局执行环境直到程序退出

每个函数都有自己的执行环境

代码在环境中执行时会创建一个作用域链确保对变量和函数的有序访问。

**作用域链查找规则是从内到外。**

**延长作用域链**：

**延长作用域链原理是 在作用域链的前端临时增加一个变量对象，在代码执行后该变量对象会被立即移除。**

**with语句、try catch语句、eval**

**没有块级作用域,只有全局作用域和函数作用域**

1. **函数提升**:把声明和值都提升并且优先级高于变量提升，只有函数声明的方式定义的函数才 能提升 函数表达式声明的函数不能提升，会报错
2. **变量提升**

将变量声明及函数声明提升至当前作用域的顶端，然后进行接下来的处理

**函数提升的优先级高于变量提升**

5、垃圾收集：

局部变量：离开执行环境时，自动销毁

全局变量：手动解除引用或者关闭程序

自动垃圾回收机制

**垃圾收集方式：**

**1、标记清除**

当变量进入环境时，做上标记进入环境，离开环境时标记离开环境

给当前不使用的值加上标记

**2、引用计数**

**"use strict"**

//严格模式下编写代码，需要js的规范去编写代码

// 如果没有var 就相当于全局变量，但是在严格模式下会报错

**递归**

函数调用函数自身,最终要有一个停止的条件

用来求阶乘的

**arguments.callee正在执行的函数**

**严格模式下("use strict")不能访问arguments.callee**

可以使用：

var factorial = function f(num){

if(num<=1){

return 1;

}else{

return num\*f(num-1);

}

}

function fn(n){

if(n<=1){

return 1;

}

return n\* fn(n-1);=======return n\* arguments.callee(n-1);

（**严格模式下("use strict")不能**）

//4\*3\*2\*fn(1)

}

console.log(fn(4)); //24

var tem = fn;

fn = null;//

console.log(tem(1));//1

console.log(tem(2));//报错

**遍历对象**

var obj = {

name:"zhangsan",

age:20,

classes:{

name:"js",

time:"afternoon"

}

}

for(var i in obj){

if(typeof obj[i]=="object"){

for(var j in obj[i]){

console.log(j+"+"+obj[i][j]); //name+js time+afternoon

}

}else{

console.log(i+"+"+obj[i]); //name+zhangsan age+20

}

}

**自执行函数**

模仿块级作用域

(function(){

if(true){

var b = 4;

console.log(b) //4

}

})();

(function(){

console.log(123); //123

})();

**函数自执行：**

**(function(){})();**

**(function(){}());**

**!function(){}(); ==> (!还可以是+ - ~）**

**闭包：**

**函数里面套函数，并且把这个函数返回回去 这就形成了一个闭包**

有权访问另一个函数作用域中的变量的函数，函数嵌套函数,子函数要使用到父函数局部变量

什么是闭包？

特点：

1）函数嵌套函数2）内部函数可以访问外部函数的变量3）参数和变量不会被回收。

闭包的用途就是可以读取函数内部的变量，让这些变量的值始终保持在内存中

闭包会影响浏览器的反应速度境地用户体验或造成浏览器无响应

闭包有可能造成内存泄漏

function fn(){

var a = 1;

function fn1(){

//fn1 > fn > window

return a;

}

return fn1;

}

console.log(fn()()); //1

**删除对象的属性**

delete obj.属性

var obj = {

name:"zhangsan",

age:20

}

delete obj.age;//删除属性

console.log(obj); // name:"zhangsan"

**this**

当前函数执行的上下文，this是在函数调用的时候确定的，不是在定义的时候确定的，谁调用它就指的谁

1、**事件绑定：this指的是绑定该事件的元素**

2、函数调用

函数名()

**自执行函数**

**setInterval**

**setTimeOut**

**this指向window**

3、方法调用

this指的是拥有这个方法的对象

**如果是多个对象调用的话指的是最近的对象**

4、构造函数的this（后面讲）

**箭头函数的this**

箭头函数会捕获其所在上下文的this作为自己的this。也就是说，箭头函数内部与其外部的this是保持一致的。

function fn(a,b){

return a+b;

}

console.log(fn(3,4)) //7

var aa=(a,b)=>{

console.log(this); //window

return a+b;

}

console.log(aa(3,4)); //7

var obj = {

a:2,

fn:function(){

setTimeout(()=>{

console.log(this); //obj

})

}

}

obj.fn();

**严格模式下**

"use strict"

function fn(){

console.log(this);

}

fn();//严格模式下 指的是undefined

**函数的方法**

apply()、call()、bind()方法

**call、apply 是函数的方法,执行者两个方法：可以修改this的指向，并且会执行该函数**

第一个参数就是指的this指向**(null\undefined this就是指的是window)**，从第二个参数往后就是该函数原有的参数

当函数被.call()或者.apply()调用时，执行的是隐式调用。

**格式：函数.call(this指向, 'value1', 'value2')**

**格式：函数.apply(this指向, ['value1', 'value2'])**

作用是调用函数，并改变函数中的this指向。

.call()接受一组参数，例如myFunction.call(thisValue, 'value1', 'value2')

.apply()接受的一组参数必须是一个类似数组的对象，例如myFunction.apply(thisValue, ['value1', 'value2'])

**console.log(Math.max.apply(null,[2,5,1,6,2]));//6**

**console.log(Math.min.apply(null,[2,5,1,6,2]));//1**

**bind是函数的方法，可以修改this指向，但是不会执行函数**

**格式：函数.bind(this指向)（'value1', 'value2'）===》**

**函数.bind(this指向，'value1', 'value2')（）**

bind方法传递给调用函数的参数逐个列出。bind方法与call、apply最大的不同就是

　　前者返回一个绑定上下文的函数，而后两者是直接执行了函数。

**类数组**

类数组：就是对象，属性名称是对应的下标，也有length属性

类数组和数组区别：

一个是对象，一个是数组；类数组不能调用数组的方法

常见的类数组对象：

arguments、getElementsByClassName(className)、getElementsByTagName(tagName)等

**类数组转数组：**

1、Array.prototype.slice.call(类数组) （es5）

2、Array.from(类数组) （es6）

3、[...类数组]（es6）

4、[].slice.call(类数组)

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

console.log(oDivs instanceof Array);//false

//检测oDivs是否为数组 返回值为bloorean

**如何把类数组转换成数组**

var arr=Array.prototype.slice.apply(oDivs);

var arr1 = [...oDivs];

var arr2 = Array.from(oDiv);

函数式编程

好处：

A、语义更加清晰

B、可复用性更高

C、可维护性更好

D、作用域局限，副作用少

**高阶函数：**

某个函数，如果它的参数是函数，那么这个函数就叫做高阶函数，同时它的参数叫做回调函数

ie9下不支持

1. **封装forEach方法**

**对数组的每一项运行给定函数，并且没有返回值**

var arr = [3,4];

//数组的方法 forEach 用来遍历数组，它的参数就是函数

//**item 指的是当前遍历的元素 index 当前遍历的下标 array 该数组**

arr.forEach(function(item,index,array){

console.log(item+":"+index+":"+array); //3:0:3,4 4:1:3,4

})

1. **封装map方法**

**可以根据给定项修改数组的每一个值并且返回一个新数组**

**map 首先会去遍历数组，在遍历数组的时候会执行函数，这个函数会有返回值，把返回值重新组合成一个新的数组**

var arr = ["a","b","c","d"];

var Rarr=arr.map(function(item,index,array){

return item+(index+1);

});

console.log(Rarr); //["a1", "b2", "c3", "d4"]

1. **封装filter方法**

**用来过滤数组，并且不会修改原数组**

数组的方法，返回一个新的数组，遍历数组，遍历的过程中执行回调函数(过滤的规则),

**这个回调函数有返回值（boolean类型）**,最终会把返回值是true的元素重新组合成一个新的数组

var arr = [5,3,6,7];

var arr1=arr.filter(function(item,index,array){

/\*if(item%2!=0){

return true;

}else return false;\*/

return item%2!=0;

})

console.log(arr1); //[5, 3, 7]

1. **封装some方法**

**有一个符合条件就返回true**

1. **封装every方法**

**every 返回boolean类型 判断数组是不是每个元素都符合某个条件 如果符合就返回true**

var arr = [3,5,4,6];

var arr1=arr.every(function(item,index,array){

return item>5;

});

console.log(arr1); //false

**浅拷贝、深拷贝**

**1、面向对象的编程**

我们把这种有对象，对象下面有属性和方法的这种叫做面向对象。我们把这种形式的叫做面向对象编程。

对象的组成：

**对象由两部分组成：属性（对象里面的变量）**

**方法（对象里面的函数）**

面向对象编程是目前流行的设计开发技术，它的特征主要是封装、多态、继承

2、创建对象：

1、**工厂模式**

把创建对象的过程、细节给封装起来了

对象有很多种类型 Object（Array、Function、Date、RegExp...）

缺点：把对象的类型就明确定义成Object

**function createPerson(name,age,sex){**

**var obj = new Object(); //var obj={}**

**obj.name = name;**

**obj.age = age;**

**obj.sex = sex;**

**return obj;**

**}**

**var p1 = createPerson("aw",22,"nan");**

2、**构造函数**

只要通过**new**操作符调用的函数，就可以把他当做构造函数，

Object Array Date RegExp 构造函数通常首字母大写（就是为了其他的函数区分开），

构造函数的返回值就是this，只不过不用写

**构造函数的缺点 每个实例都会重新把方法创建一遍**

**function Animal(a,b,leg){**

**this.name = a;**

**this.color = b;**

**//this.leg = leg; //如果没传leg undefined**

**if(leg){**

**this.leg = leg;//如果没传leg 这行代码不执行 就会去原型对像上面找**

**}**

**this.run = function(){**

**console.log("会跑");**

**}**

**}**

**Animal.prototype.leg = 4;**

**var cat = new Animal("小花","黑白",3);//cat 就是构造函数的实例**

**var dog = new Animal("小黑","黑色");**

**console.log(cat.run===dog.run);//false**

**console.log(cat.leg);//3**

**查找某个对象的属性 先看自己有没有，如果没有就去原型对象上找**

**console.log(dog.leg);//undefined**

3、**原型模式：**

**每个函数都有一个属性prototype,**这个属性对应的值是一个对象，我们叫做**原型对象**，

**原型对象的作用：该对象的属性和方法是被所有的实例共享的，**换句话就是所有的实例都能使用原型对象的属性和方法

**function Person(name,age,sex){**

**this.name = name;**

**this.age = age;**

**this.sex = sex;**

**this.city = "bj";**

**}**

**Person.prototype.city = "shanghai";**

**Person.prototype.say = function(){**

**console.log("会说话");**

**}**

**var p1 = new Person("aa",20,"nan");**

**var p2= new Person("aa22",20,"nan");**

**console.log(p1.city);//bj**

**console.log(p2.city);//bj**

**p1.say();**

**p2.say();**

**console.log(p1.say===p2.say);//true**

**原型对象有一个constructor属性指向构造函数**

**function Person(name,age,sex){**

**this.name = name;**

**this.age = age;**

**this.sex = sex;**

**this.city = "bj";**

**}**

**console.log(Person.prototype.constructor===Person);//true**

**var p1 = new Person("aa",22,"nan");**

**console.log(p1.constructor===Person);//true**

**\_\_proto\_\_**

**每个对象都有这个属性，实例的\_\_proto\_\_指向的是原型对象(该属性对应的值就是原型对象)**

**function A(){**

**}**

**var a1 = new A();**

**console.log(A.prototype.constructor===A);//true**

**console.log(a1.\_\_proto\_\_===A.prototype);//true**

**var A = new Function()**

**//A是一个函数 A是Function的一个实例**

**console.log(A.\_\_proto\_\_===Function.prototype);//true**

**//Array 构造函数 是Function的一个实例**

**判断是否是数组**

**var arr = [34,56];//new Array();**

**console.log(arr instanceof Array);//true**

**console.log(Array.prototype.constructor === Array);**

**console.log(arr.constructor===Array);//判断是否是数组 //true**

**原型字面量的写法**

**function Person(name,age,sex){**

**this.name = name;**

**this.age = age;**

**this.sex = sex;**

**this.city = "bj";**

**}**

**Person.prototype = {**

**constructor:Person,**

**say:function(){**

**console.log("deee");**

**},**

**eat:function(){**

**console.log("dddss");**

**}**

**}**

**//new Object()**

**console.log(Person.prototype.constructor===Object); //false**

**如果没有写constructor:Person为true；**

构造函数与原型组合模式是用创建面向对象的常用方式

**系统自带构造函数Array、Date、Object、RegExp**

console.log(Array.prototype.constructor===Array);//true

var a1 = [];//new Array()

console.log(a1.constructor===Array );//true

function A() {

}

**A是一个函数 A是Function的一个实例**

**var A = new Function()**

console.log(Function.prototype.constructor===Function);//true

console.log(A.constructor===Function);//true

Array 构造函数 是Function的一个实例

console.log(Array.constructor===Function);//true

**继承**：子类可以继承父类的一些功能，但是子类不会影响父类

0、\_\_proto\_\_（[[prototype]]）

每个对象都有\_\_proto\_\_这个属性,实例对象的\_\_proto\_\_指向的是原型对象

1、**原型链继承**

**每个构造函数都有一个指向原型对象的属性(prototype)，每个原型对象也有一个指向构造函数的一个属性(constructor),**

**每个实例都有一个指向原型对象的属性(\_\_proto\_\_)**

如果我把某个类型(假如A)原型对象指向另外一个类型(假如是B)的实例，

该A的原型对象是不是也有一个属性指向另外一个类型的(B)原型对象,如果我再把B的原型对象指向另外一个类型(假如是C)，上面的结论仍然成立，

层层递进，形成了实例与原型对象的一个链条，就被称作原型链

**构造函数A.prototype = new B();**

function A(){

}

function B(){

}

A.prototype = new B();

console.log(A.prototype.constructor==B)//TRUE

console.log(A.prototype.\_\_proto\_\_==B.prototype)//TRUE

console.log(B.prototype.\_\_proto\_\_==Object.prototype)//TRUE

//A继承B B继承Object

console.log(Object.prototype.\_\_proto\_\_);//null

注意：

使用原型链的方式进行继承，子类型定义自己的方法或者子类型重写父类型的方法，注意书写的位置，

要放在原型链继承的那条语句下面

**当对象访问属性或方法时，先从自身查找，找不到就去构造函数的原型对象上找，如果找不到，就去原型对象的原型对**

**象上找，直到 Object.prototype.\_\_proto\_\_ == null. 如果没找到，返回 undefined。**

**原型链继承的问题：**

**父类型的引用类型的属性会被子类型所有的实例共享**

**原型链继承一般不会单独使用**

2、**构造函数继承**

**在子类型的构造函数里面调用父类型的构造函数，使用call或者apply**

但是构造函数继承解决原型链引用数据类型的问题，**但是它还有作为构造函数自己的问题**

**构造函数继承一般也不会单独使用**

**function B(name,age,famliy){**

**this.name = name;**

**this.age = age;**

**this.famliy = famliy;**

**this.friends = ["zhangsan","lisi","wangwu"];**

**}**

**B.prototype.say = function(){**

**console.log("能说话");**

**}**

**function A(name,age,famliy){**

**(第一种方法) B.call(this,name,age,famliy);**

**(第二种)B.apply(this,[name,age,famliy]);**

**(第三种)B.apply(this,arguments); this指s1**

**}**

**s1 = new A("aa",21,"张");**

**B.prototype.say = function(){**

**console.log("能说话");**

**}**

**s1.say(); //报错**

**通常使用原型链和构造函数结合起来进行继承**

**function Father(name,age){**

**this.name = name;**

**this.age = age;**

**this.friends = ["zhangsan","lisi","wangwu"];**

**}**

**Father.prototype.say = function(){**

**console.log("能说话"); //能说话**

**}**

**function Son(name,age){**

**Father.call(this,name.age);//this指s1**

**}**

**Son.prototype = new Father();**

**var s1 = new Son("heihei",20);**

**console.log(s1.friends);//["zhangsan", "lisi", "wangwu"]**

**s1.say();**

**系统自带继承关系**

**在javascript 中，继承就是靠原型链来实现的。**

Array 继承了Object Function 继承了Object Date 继承了Object RegExp 继承了Object

Array.prototype=new Object()

Array.prototype.\_\_proto\_\_===Object.prototype

Function.prototype.\_\_proto\_\_===Object.prototype

Date.prototype.\_\_proto\_\_===Object.prototype

Object.prototype.\_\_proto\_\_===null

**确定原型和实例的关系：**

**实例 instanceof 构造函数**

**原型对象.isPrototypeOf(实例)**

function A(){}

var a1 = new A();

console.log(a1.\_\_proto\_\_===A.prototype);//true

console.log(a1 instanceof A);//true

console.log(A.prototype.isPrototypeOf(a1));//true

**判断是否有某个属性**

**in 先看自己有吗？再看原型对象有吗？就去父亲上找。。。Object**

function A(name){

this.name = name;

}

A.prototype.city = "shanghai";

var a = new A("zhangsan");

console.log("name" in a);//true

console.log("city" in a);//true

**hasOwnProperty(属性名称) 检测某个对象是否有自己的属性**

**console.log(a.hasOwnProperty("name"));//true**

**console.log(a.hasOwnProperty("city"));//false**

**Json**

json 一种数据格式

1、 字符串、数字、boolean

2、数组和对象

**json格式的对象属性名称必须是双引号，属性值如果是字符串也必须是双引号**

**同步：阻塞 同一时间段只能有一个进程执行，上面的代码没执行完，后面的代码必须等前面的代码执行再执行**

**异步：非阻塞 上面的代码没执行完，后面的代码不需要等可以立刻执行**

**ajax：优点：局部刷新 用户体验较好**

**缺点： 1、破坏浏览器的后退功能**

**2、搜索引擎支持较弱**

**json常用的方法**

1、**JSON.stringify()** 把对象转换成字符串

**var obj = {**

**"name":"zhangsan",**

**"age":20**

**}**

console.log(JSON.stringify(obj));//方便我们去查看对象里面的具体内容

console.log(typeof JSON.stringify(obj));//string

**var o = {**

**name:"zhangsan",**

**sex:'女'**

**}**

console.log(JSON.stringify(o)); //{"name":"zhangsan","sex":"女"}’

//把任意对象转换成字符串，其实就是把属性方法显示出来

2、**JSON.parse(str)** 把JSON格式的**字符串转**换成json格式的**对象**

JSON.parse解析字符串的时候，字符串必须是**JSON格式的对象**，里面的属性名称必须是双引号

**var s = '{"name":"aa","age":20}';**

**var o1 = JSON.parse(s);**

**for(var i in o1){**

**console.log(i+","+o1[i]); //name,aa age,20**

**}**

**JSON.stringify(obj);//对象转字符串**

**JSON.parse(str);//字符串转对象 字符串必须是JSON格式的对象**

**eval()**系统自带的函数，参数是字符串，会把字符串里面的内容作为js代码去执行

eval("alert(123)");

var s = '{name:"zhangsan",age:20}';

var o=**eval("("+s+")")**;如果是数组不需要加小括号

console.log(o);

**字符串转换成对象有两种方式：**

1、JSON.parse 字符串必须满足json的格式

2、eval() 对数据格式要求不严格，但是转换的时候需要加小括号

如果是数组不需要加小括号

**ajax**

ajax 异步的JavaScript和XML XML(和json很像)

同步：同一时间段，只有一个进程执行

异步：同一时间段，有多个进程执行

ajax就是用来获取数据的

局部刷新页面

setTimeout(function(){

console.log(3333);

},5000);

console.log(444); 先打印444

**ajax的使用过程**

1、**创建对象**

var xhr = window.XMLHttpRequest?new XMLHttpRequest():new ActiveXObject("Microsoft.XMLHTTP");

2、**创建一个http链接，设置一些相关的信息**

**xhr.open();**

open方法有三个参数：

1、method

get(GET)或者post(POST)

2、url

请求的地址

3、是否异步

true|false 默认是true

3、**提交，发送请求**

**xhr.send()**

send方法里面的参数是一个字符串，这个参数表示的是向后台传递的参数

**如果设置的请求是get 那么send方法里面就直接写null**

如果是post请求，并且要向后台传递一些参数，那么就把这些参数以字符串的形式写进来

2、**readystate： ajax的状态值**

ajax对象有一个属性 readystate ajax的状态值

0 还未初始化 还没调用open方法

1 已经调用了open方法，还没调用send();

2 调用了send()

3 后台只是发送了一部分数据或者后台正在解析数据

4 已经获取了所有的数据，可以进行操作了

1. **ajax对象提供了一个事件onreadystatechange，这个事件是readystate发生变化的时候会触发这个事件**

4、**status:http的状态码**

200 表示的是成功 404 请求的资源没找到 500 服务器内部错误

302 表示浏览器会产生两次http请求

1. **xhr.responseText 请求回来以后后台返回来的响应数据，这是一个字符**

6、**get和post的区别**

1、get传输的数据大小有限制，最多一般不超过2Kb,post基本无限制

2、传递参数的方式不同

get：直接把参数拼接到请求地址的后面http://www.weibo.sina.com?name=zhangsan&age=20&sex=male&tel=13456765678;

post: **xhr.setRequestHeader("content-type","application/x-www-form-urlencoded")**;

xhr.send("name=lisi&age=20&tel=13456787656");

3、get请求不安全

**get传参**

var name=ipts[0].value;

password=ipts[1].value;

var xhr=window.XMLHttpRequest?new XMLHttpRequest:new ActiveXObject("Microsoft.XMLHTTP");

**xhr.open('get','http://www.baidu.com?name='+name+'&password='+password,true);**

xhr.send(null);

xhr.onreadystatechange=function(){

if(xhr.readyState==4&&xhr.status==200){

console.log(xhr.responseText)

}

}

**Post传参**

var xhr=window.XMLHttpRequest?new XMLHttpRequest:new

ActiveXObject(Microsoft.XMLHTTP);

xhr.open('post','http://www.baidu.com',true);

**xhr.setRequestHeader('content-type','application/x-www-form-urlencoded');**

**xhr.send('name='+name+'&password='+password);**

xhr.onreadystatechange=function(){

if(xhr.readyState&&xhr.status){

console.log(xhr.responseText)

}

}

**Promise**

es6 用来解决异步的 **resolve 表示成功 reject 表示失败** pending 进行中

构造函数promise的参数是一个函数，这个函数的参数也是函数，resolve表示成功的函数，reject表示失败的函数

**封装一个异步加载图片，返回promise对象**

var p1=new Promise(function(resolve,reject){

var img=document.createElement('img');

img.src='images/img1.jpg';

img.onload=function(){

resolve(img)

}

img.onerror=function(){

reject(img.src)

}

});

p1.then(function(img){

document.body.appendChild(img);

},function(error){

alert(error)

})

**encodeURI和encodeURIComponent**

这两个方法都会是以utf-8的方式对字符串进行编码，两者的区别：encodeURI只对中文进行编码，

**encodeURIComponent除了中文还会对一些特殊符号(如：/、？、=、&等等)编码**

**ajax跨域**

ajax不允许跨域访问资源，什么情况是跨域呢？

1）**协议不同**

http/https

2）**端口不同**

http://sina.com:8080 去访问 http://sina.com:9999

3）**域名不同**

http://sina.com 去访问 http://hao123.com

报错信息：

No 'Access-Control-Allow-Origin' 表示需要跨域

**解决跨域的方式**：

**jsonp 是解决跨域的方式**

jsonp原理：动态创建script标签，把请求地址放到script标签的src里面，并且告诉他一个回调函数的名字

这个回调函数是在我们本地定义的，但是回调函数实在服务端调用的

**promise es6新增的**

es5：**用回调函数(callback)解决异步 、事件监听**

用来解决异步的

Promise构造函数

如何使用Promise

创建一个Promise的实例

构造函数Promise的参数是一个函数(fn),fn有两个参数(resolve,reject)，这两个参数分别对应成功的函数和失败的函数

Promise的实例有三种状态：1 成功 resolve 2 失败 reject 3 进行中 pending

promise的实例有一个**then**方法，该方法有两个参数，第一个参数是成功对应的函数，第二个参数是失败对应的函数；

第二个参数可以省略不写,可以写到catch里面

**p1.then(function(){}).catch(funtion(){})**

**图片的异步加载（预加载）**

异步加载图片 先把图片加载出来 再把图片绑定到dom元素上面

异步加载图片封装

**function loadImg(url){**

**return new Promise(function(resolve,reject){**

**var img=new Image()**

**img.src=url**

**img.onload=function(){**

**resolve(img)**

**}**

**img.onerror=function(){**

**reject("图片没找到或格式错误")**

**}**

**})**

**}**

**loadImg("baby.jpg").then(function(data){**

**document.body.appendChild(data)**

**},function(msg){**

**console.log(msg)**

**})**

**img1 = document.creatElement("img");====img=new Image()**

**Promise的两个方法**

Promise.all([p1,p1]) **all方法**接收一个由多个Promise组成的数组作为参数，返回一个新的Promise实例

**当数组里面的所有的实例都是成功的状态，那么这个新的实例它的状态才是成功**,

并且返回的数据是这多个实例返回的数据组合成的数组

失败的状态：**有一个实例失败，那么这个新实例的状态就是失败**，并且新实例返回的数据是最先失败的那个实例的数据

**race**

和all方法类似，接收一个由多个Promise组成的数组作为参数，返回一个新的Promise实例

**哪一个实例最先执行完，该实例对应的状态就是新的实例的状态**，并且返回的数据也是新的实例的数据

**PromiseStatus：**用于记录Promise对象的三种状态，这三中状态分别是：

**pending**：待定状态，Promise对象刚被初始化的状态

**fulfilled**：完成状态，承诺被完成了的状态

**rejected**：拒绝状态，承诺完成失败的状态