# \*\*\*\*\*\*\*\*ECMA简介\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***1. 它是一种由ECMA组织（前身为欧洲计算机制造商协会）制定和发布的脚本语言规范  
2. 而我们学的 JavaScript 是ECMA的实现, 但术语ECMAScript和JavaScript平时表达同一个意思  
3. JS包含三个部分：  
 1). ECMAScript（核心）  
 2). 扩展==>浏览器端  
 \* BOM（浏览器对象模型）  
 \* DOM（文档对象模型）  
 3). 扩展==>服务器端  
 \* Node  
4. ES的几个重要版本  
 \* ES5 : 09年发布  
 \* ES6(ES2015) : 15年发布, 也称为ECMA2015  
 \* ES7(ES2016) : 16年发布, 也称为ECMA2016 (变化不大)***

# \*\*\*\*\*\*\*\*ES5\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 1.严格模式

***1. 理解:  
 \* 除了正常运行模式(混杂模式)，ES5添加了第二种运行模式："严格模式"（strict mode）。  
 \* 顾名思义，这种模式使得Javascript在更严格的语法条件下运行  
2. 目的/作用  
 \* 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为  
 \* 消除代码运行的一些不安全之处，为代码的安全运行保驾护航  
 \* 为未来新版本的Javascript做好铺垫  
3. 使用  
 \* 在全局或函数的第一条语句定义为: 'use strict';  
 \* 如果浏览器不支持, 只解析为一条简单的语句, 没有任何其他影响  
4. 语法和行为改变:  
 \* 必须用var声明变量  
 \* 禁止自定义的函数中的this指向window  
 \* 创建eval作用域  
 \* 对象不能有重名的属性***

***//1.必须用var声明变量*function *demo*() {  
 *a* = 1 *//飘红*  
}  
*demo*()  
console.log(*a*)  
*//2.禁止自定义的函数中的this指向window*function *demo*() {  
 console.log(this) *//undefined*}  
*demo*()  
function *Person*(name,age) {  
 this.name = name  
 this.age = age  
}  
var *p* = new *Person*('kobe',18)  
console.log(*p*) *//Person {name: "kobe", age: 18}*  
*Person*('wade',18) *//报错 Cannot set property 'name' of undefined*  
  
*//3.eval作用域*var *a* = 1  
eval('var a = 2;console.log(a)') *//2*console.log(*a*) *//1*  
*//4.对象不能有重名的属性*var *obj* = {  
 name:"kobe", *//两个name飘红* name:'wade'}  
console.log(*obj*)**

# 2.JSON对象

***1. JSON.stringify(obj/arr)  
 \* js对象(数组)转换为json对象(数组)  
2. JSON.parse(json)  
 \* json对象(数组)转换为js对象(数组)  
3.使用JSON进行数据交互的优点  
 \* 1.数据相对小,更精简（相对于xml）  
 \* 关于xml格式的数据传送：  
 <xml>  
 <name>zhangsan</name>  
 <age>12</age>  
 </xml>  
 \* 如果是json数据：'{"name":"zhangsan","age":12}'  
 2.有完善的API直接把JSON对象转成js对象（反过来也可以）***

**var *jsonstr*** = **'{"name":"zhangsan","age":12}'  
var *obj*** = ***JSON***.parse(***jsonstr***)  
**console**.log(***obj***) ***//Object*var *obj2*** = {  
 **name**:**'小猪佩奇'**,  
 **age**:23  
}  
**var *arr*** = [1,2,**'atguigu'**]  
**var *str*** = ***JSON***.stringify(***obj2***)  
**var *arrStr*** = ***JSON***.stringify(***arr***)  
**console**.log(***str***) ***//{"name":"小猪佩奇","age":23} 字符串类型***  
**console**.log(***arrStr***) ***//[1,2,"atguigu"] 字符串类型***

# 3.Object扩展

***ES5给Object扩展了一些静态方法, 常用的2个:  
1. Object.create(prototype, descriptors)  
 \* 作用: 以指定对象为原型创建新的对象  
 \* 为新的对象指定新的属性, 并对属性进行描述  
 - value : 指定值  
 - writable : 标识当前属性值是否是可修改的, 默认为false  
 - configurable: 标识当前属性是否可以被删除 默认为false  
 - enumerable： 标识当前属性是否能用for in 枚举 默认为false***

**var *student* = {**

**school:'尚硅谷大学'  
 }var *stu* = Object.create(*student*,{  
 name:{  
 value:'张三',  
 writable:false,  
 configurable:false,  
 enumerable:true,  
 }  
})**

***2. Object.defineProperties(object, descriptors)  
 \* 作用: 为指定对象定义扩展多个属性  
 \* get ：用来获取当前属性值的回调函数  
 \* set ：修改当前属性值时触发的回调函数，并且实参即为修改后的值  
 \* 存取器：setter,getter一个用来存值，一个用来取值***

***对象本身的两个方法  
\* get 扩展属性名(){} 用来得到当前属性值的回调函数  
\* set 扩展属性名(){} 用来监视当前属性值变化的回调函数***

**var *person* = {  
 firstName:'小猪',  
 lastName:' 佩奇',  
 get fullName () {  
 return this.firstName + this.lastName  
 },  
 set fullName(value){  
 var name = value.split(' ')  
 this.firstName = name[0]  
 this.lastName = name[1]  
 }  
}  
*person*.firstName = '小狗2'  
*person*.lastName = '丹尼2'  
console.log(*person*.fullName) *//小狗2丹尼2***

# 4.Array扩展

***1. Array.prototype.indexOf(value) : 得到值在数组中的第一个下标  
2. Array.prototype.lastIndexOf(value) : 得到值在数组中的最后一个下标  
3. Array.prototype.forEach(function(item, index){}) : 遍历数组  
4. Array.prototype.map(function(item, index){}) : 遍历数组返回一个新的数组，返回加工之后的值  
5. Array.prototype.filter(function(item, index){}) : 遍历过滤出一个新的子数组， 返回条件为true的值***

***/\*  
 需求:  
 1. 输出第一个6的下标  
 2. 输出最后一个6的下标  
 3. 输出所有元素的值和下标  
 4. 根据arr产生一个新数组,要求新数组中的每个元素都比原来大10  
 5. 根据arr产生一个新数组,要求数组里的元素都是大于4的  
 \*/*var *arr* = [2,4,6,8,8,6,4,2,1,3,6]  
console.log(*arr*.indexOf(6)); *//2*  
console.log(*arr*.lastIndexOf(6)); *//10*  
*arr*.forEach(function (item,index,source) {  
 console.log(index,item,source)**

***//index 索引下标 item 每一项 source 原数组*  
})  
var *arr2* = *arr*.map(function (item,index,source) {  
 return item+10  
})  
console.log(*arr2*) *//(11) [12, 14, 16, 18, 18, 16, 14, 12, 11, 13, 16]*  
var *arr3* = *arr*.filter(function (item,index) {  
 return item>4  
})  
console.log(*arr3*) *//(5) [6, 8, 8, 6, 6]***

# 5.Function扩展

***1. Function.prototype.bind(obj) :  
 \* 作用: 将函数内的this绑定为obj, 并将函数返回，并没有进行调用  
2. 面试题: 区别bind()与call()和apply()?  
 \* 都能指定函数中的this  
 \* call()/apply()是立即调用函数  
 \* bind()是将函数返回***

# \*\*\*\*\*\*\*\*ES6\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 1.let关键字

***1. 作用:  
 \* 与var类似, 用于声明一个变量  
2. 特点:  
 \* 在块作用域内有效  
 \* 不能重复声明  
 \* 不会预处理, 不存在提升  
3. 应用:  
 \* 循环遍历加监听  
 \* 使用let取代var是趋势***

# 2.const关键字

***1. 作用:  
 \* 定义一个常量  
2. 特点:  
 \* 不能修改  
 \* 其它特点同let  
3. 应用:  
 \* 保存不用改变的数据***

# 3.变量的解构赋值

***1. 理解:  
 \* 从对象或数组中提取数据, 并赋值给变量(多个)  
2. 对象的解构赋值  
 let {n, a} = {n:'tom', a:12}  
3. 数组的解构赋值  
 let [a,b] = [1, 'hello'];  
4. 用途  
 \* 给多个形参赋值***

**let *person* = {  
 name:'张三',  
 age:18,  
 sex:'男',  
 address:'宏福科技园'  
}  
console.log(*person*.name,*person*.age,*person*.sex,*person*.address)**

***//张三 18 男 宏福科技园*  
let {name,age,sex,address} = *person*console.log(name,age,sex,address)*//张三 18 男 宏福科技园*  
let *arr* = [2,4,6]  
console.log(*arr*[0]) *//2*let [,,*t*] = *arr*console.log(*t*) *//6*  
function *demo*({age,sex}) {  
 console.log('@@@',age,sex) *//@@@ 18 男*  
}**

***demo*(*person*)**

# 4.模板字符串

***模板字符串 : 简化字符串的拼接  
 \* 模板字符串必须用 `` 包含  
 \* 变化的部分使用${xxx}定义***

**let *person* = {  
 name:'张三',  
 age:18,  
 sex:'男',  
 address:'宏福科技园'  
}  
let {name,age,sex,address} = *person*console.log('当前人的名字：'+'"'+name+'"','年龄为：'+age,'性别为:'+sex,address)  
*//当前人的名字："张三" 年龄为：18 性别为:男 宏福科技园*console.log(`当前人的名字："${name}",年龄为：${age}，性别为：${sex}`)**

***//当前人的名字："张三",年龄为：18，性别为：男***

# 5.简化的对象写法

***\* 省略同名的属性值  
\* 省略方法的function（注意是方法的function）***

***//之前的写法*let *point* = {  
 x:*x*,  
 y:*y*,  
 getPoint:function () {  
 return [*x*,*y*]  
 }  
}  
*//对象简写方式*let *ponit2* = {  
 *x*,  
 *y*,  
 getPoint () {  
 return [*x*,*y*]  
 }  
}**

# 6.箭头函数

***\* 箭头函数：  
 \* 作用: 定义匿名函数  
 \* 基本语法:  
 \* 没有参数: () => console.log('xxxx')  
 \* 一个参数: i => i+2  
 \* 大于一个参数: (i,j) => i+j  
 \* 关于函数体需要注意的：  
 \* 1.函数体不用大括号: 默认返回结果  
 \* 2.函数体如果有多个语句, 需要用{}包围，若有需要返回的内容，需要手动返回  
\* 使用场景: 多用来定义回调函数  
  
\* 箭头函数的特点：  
 1、简洁  
 2、箭头函数没有自己的this，箭头函数的this不是调用的时候决定的，而是在定义的时候，其所处的上下文对象就是它的this  
 3、扩展理解：箭头函数的this看外层的是否有函数，  
 如果有，外层函数的this就是内部箭头函数的this，  
 如果没有，则this是window。***

***//普通函数*function *demo*(a) {  
 console.log(a)  
}  
*demo*(1)  
*//箭头函数*let *demo2* = (a) => {  
 console.log(a)  
}  
*demo2*(1)**

***//1.箭头左边没有参数,不能省略()*let *demo3* = () => {  
 console.log('哈哈，我是demo3函数，我是用箭头函数定义的')  
}  
*demo3*()  
*//2.箭头左边只有一个参数，可以省略()*let *demo4* = a => {  
 console.log(a)  
}  
*demo4*(1)  
*//3.箭头左边有一个以上的参数，不能省略()***

**let *demo5* = (a,b,c) => {  
 console.log(a,b,c)  
}  
*demo5*(1,2,3)  
*//4.箭头右边如果只有一条函数体语句，{}可以省略，如果有多条语句，不能省略{}  
//但是要注意：如果箭头右侧省略了{},会自动返回代码执行的结果(函数的返回值)*let *demo6* = a => a+a  
let *d* = *demo6*(1)  
console.log(*d*); *//2*  
*//5.箭头函数没有自己的this，先找外层函数的this，最终找到window*let *test* = () => {  
 let *test2* = () =>{  
 console.log(this)  
 }  
 window.test2 = *test2*}  
*test*()  
test2()  
  
let *person* = {  
 name:'张三',  
 age:12,  
 getInfo : function() {  
 console.log('第一次输出的',this)*//person person person* let *demo* = () => {  
 console.log('第二次输出的',this)*//window person* console.log(this.name,this.age)  
 }  
 window.demo = *demo* }  
}  
*person*.getInfo()  
*demo*()  
  
let *btn1* = document.getElementById('btn1')  
*btn1*.onclick = () => {  
 console.log(this)  
}  
  
(()=>{  
 console.log(1)  
})()**

# **7.点点点运算符(三点运算符)**

***1. 用来取代arguments 但比 arguments 灵活  
2. 作为扩展运算符使用  
 let arr1 = [1,3,5];  
 let arr2 = [2,...arr1,6];  
 arr2.push(...arr1);***

**function *demo*(a,b,c) {  
 console.log(arguments)*//伪数组*}function *demo*(...test) {  
 console.log(test)*//真数组*}  
*demo*(1,2,3,4,5,6,'atguigu')**

**let *arr1* = [1,2,3,4,8,9,10 ]  
let *arr2* = [5,'haha',7]  
*arr1* = [1,2,3,4,...*arr2*,8,9,10 ]  
console.log(*arr1*); *//[1, 2, 3, 4, 5, "haha", 7, 8, 9, 10]  
arr2*.push(...*arr1*);  
console.log(*arr2*); *//[5, "haha", 7, 1, 2, 3, 4, 5, "haha", 7, 8, 9, 10]***

# 8.形参默认值

***1.当不传入参数的时候默认使用形参里的默认值  
function Point(x = 1,y = 2) {  
 this.x = x;  
 this.y = y;  
}***

# 9.Promise对象

***1. 理解:  
 \* Promise对象: 代表了未来某个将要发生的事件(通常是一个异步操作)  
 \* 有了promise对象, 可以将异步操作以同步的流程表达出来, 避免了层层嵌套的回调函数(俗称'回调地狱')  
 \* ES6的Promise是一个构造函数, 用来生成promise实例  
   
2. 使用promise基本步骤(共分为2步):  
 \* 创建promise对象  
 let promise = new Promise((resolve, reject) => {  
 //初始化promise状态为 pending  
 //执行异步操作  
 if(异步操作成功) {  
 resolve(value);//修改promise的状态为fullfilled  
 } else {  
 reject(errMsg);//修改promise的状态为rejected  
 }  
 })  
 \* 第一种：调用promise的then()  
 promise.then(function(  
 result => console.log(result),  
 errorMsg => alert(errorMsg)  
 ))  
 \* 第二种：调用promise的then()与.catch() -------- 用的不多  
 promise.then(function(  
 result => console.log(result),  
 errorMsg => alert(errorMsg)  
 ))  
3. promise实例对象的3个状态  
 \* pending: 初始化状态  
 \* fullfilled: 成功状态  
 \* rejected: 失败状态  
4. 应用:  
 \* 使用promise封装处理ajax请求***

**let *myPromise* = new Promise(function (resolve,reject) {  
 *//resolve()//将Promise的实例对象的状态，更改为“成功”  
 //reject()//将Promise的实例对象的状态，更改为“失败”* setTimeout(function () {  
 var data = 'atguigu'  
 console.log('异步任务，请求成功了')  
 *//resolve(data)* reject('任务失败了')  
 },1000)  
})**

***//第一种写法：  
myPromise* .then(function (data){  
 console.log(`成功了，数据为：${data}`)  
 },function (err) {  
 console.log(err)  
 })  
*//第二种写法：  
myPromise* .then((data)=>{  
 console.log(`成功了${data}`)  
 })  
 .catch((err)=>{  
 console.log(err)  
 })**

***//案例练习，需求：有两个异步任务，等第一个成功的执行了，之后再发送第二个  
 //模拟发送ajax请求* function *sendAjax*(id) {  
 return new Promise((resolve,reject)=>{  
 setTimeout(()=>{  
 resolve(`我是编号为：${id}的请求带回来的数据`)  
 },1000)  
 })  
 }  
*//第一次请求  
sendAjax*('01')  
 .then((data)=>{  
 console.log(data)  
 *//第二次请求* return *sendAjax*('02')  
 },(err)=>{  
 console.log(err)  
 })  
 .then((data)=>{  
 console.log(data)  
 },(err)=>{  
 console.log(err)  
 })**

***//回调地狱的写法如下：*setTimeout(function () {  
 var data = 'atguigu'  
 console.log('第一次异步任务，请求成功了')  
 setTimeout(function () {  
 console.log('第二次异步任务，请求成功了')  
 setTimeout(function () {  
 console.log('第三次异步任务，请求成功了')  
 },1000)  
 },1000)  
},1000)**

# 10.symbol

***前言：ES5中对象的属性名都是字符串，容易造成重名，污染环境  
Symbol：  
 概念：ES6中的添加了一种原始数据类型symbol  
 备注：已有的原始数据类型：String, Number, boolean, null, undefined, 对象（一般对象、函数对象、数组对象）  
 特点：  
 1、Symbol属性对应的值是唯一的，用于解决命名冲突问题（使用的极少）  
 2、Symbol值不能与其他数据进行计算，包括同字符串拼串  
 3、for in, for of遍历时不会遍历symbol属性。  
 使用：  
 1、调用Symbol函数得到symbol值  
 let symbol = Symbol();  
 let obj = {};  
 obj[symbol] = 'hello';  
 2、传参标识  
 let symbol = Symbol('one');  
 let symbol2 = Symbol('two');  
 console.log(symbol);// Symbol('one')  
 console.log(symbol2);// Symbol('two')  
 3、Symbol的iterator属性  
 \* 某些对象的原型对象上拥有Symbol.iterator属性，他指向该对象的默认遍历器方法***

let ***a*** = Symbol()  
 let ***b*** = Symbol()  
 **console**.log(***a***,***b***) Symbol() Symbol()  
 **console**.log(***a*** === ***b***) false

# 11.iterator遍历器

***概念：iterator是一种接口机制，为各种不同的数据结构提供统一的访问机制  
作用：  
 1、为各种数据结构，提供一个统一的、简便的访问接口；  
 2、使得数据结构的成员能够按某种次序排列  
 3、ES6创造了一种新的遍历命令:for of，并且for of底层依赖iterator实现的遍历  
工作原理：  
 - 创建一个指针对象(遍历器对象)，指向数据结构的起始位置。  
 - 第一次调用指针对象中的next方法，指针自动指向数据结构的第一个成员  
 - 接下来不断调用next方法，指针会一直往后移动，直到指向最后一个成员  
 - 每调用next方法返回的是一个包含value和done的对象，{value: 当前成员的值,done: 布尔值}  
 \* value表示当前成员的值，done对应的布尔值表示当前的数据的结构是否遍历结束。  
 \* 当遍历结束的时候返回的value值是undefined，done值为true  
原生具备iterator接口的数据(即：可以用for of遍历)  
 1、Array  
 2、arguments  
 3、String  
 ...***

***//模拟真实遍历器的遍历逻辑，手写一个遍历器*function *mockIterator*() {  
 let index = 0;  
 let that = this  
 *//console.log(this)* return {  
 next:function () {  
 return {  
 value:that[index++],  
 done:index-1<that.length ? false : true  
 }  
 }  
 }  
}  
let *arr* = [2,4,6,8]**

**Array.prototype[Symbol.iterator] = *mockIterator*for (let *item* of *arr*){  
 console.log(*item*)  
}**

## 11.2让对象支持for of遍历

**function *mockIterator*() {  
 let index = 0;  
 let that = this  
 *//console.log(this)* if(that instanceof Array){  
 return {  
 next:function () {  
 return {  
 value:that[index++],  
 done:index-1<that.length ? false : true  
 }  
 }  
 }  
 }  
 else{  
 return {  
 next:function () {  
 let keys = Object.keys(that)  
 return {  
 value:that[keys[index++]],  
 done:index-1<keys.length ? false : true  
 }  
 }  
 }  
 }  
}  
  
Object.prototype[Symbol.iterator] = *mockIterator*Array.prototype[Symbol.iterator] = *mockIterator*let *obj* = {  
 name:'kobe',  
 age:18  
}  
  
for (let *item* of *obj*){  
 console.log(*item*)  
}**

# 12.Generator函数（很不常用）

***概念：  
 1、ES6提供的解决异步编程的方案之一  
 2、Generator函数是一个状态机，内部封装了不同状态的数据，  
 3、用来生成遍历器对象  
 4、可暂停函数(惰性求值), yield可暂停，next方法可启动。每次返回的是yield后的表达式结果  
 特点：  
 1、function 与函数名之间有一个星号  
 2、内部用yield表达式来定义不同的状态  
 例如：  
 function\* generatorExample(){  
 let result = yield 'hello'; // 状态值为hello  
 yield 'generator'; // 状态值为generator  
 }  
 3、generator函数调用返回一个指针对象，不会执行函数内部逻辑  
 4、调用next方法函数内部逻辑开始执行，遇到yield表达式停止，返回{value: yield后的表达式结果/undefined, done: false/true}  
 5、再次调用next方法会从上一次停止时的yield处开始，直到最后  
 6、yield语句返回结果通常为undefined， 当调用next方法时传参内容会作为启动时yield语句的返回值。***

# 13.async函数(ES7)

***async函数(源自ES2016)  
概念： 真正意义上去解决异步回调的问题，同步流程表达异步操作  
本质： Generator的语法糖  
语法：  
 async function foo(){  
 await 异步操作;  
 await 异步操作；  
 }  
特点：  
 1、不需要像Generator去调用next方法，遇到await等待，当前的异步操作完成就往下执行  
 2、返回的总是Promise对象，可以用then方法进行下一步操作  
 3、async取代Generator函数的星号\*，await取代Generator的yield  
 4、语意上更为明确，使用简单，经临床验证，暂时没有任何副作用***

***//案例练习，需求：有两个异步任务，等第一个成功的执行了，之后再发送第二个  
//模拟发送ajax请求*function *sendAjax*(id) {  
 return new Promise((resolve,reject)=>{  
 setTimeout(()=>{  
 resolve(`我是编号为：${id}的请求带回来的数据`)  
 },1000)  
 })  
}(async()=>{  
 console.log(await *sendAjax*('01'));  
 console.log(await *sendAjax*('02'));  
 console.log(await *sendAjax*('03'));  
 console.log(await *sendAjax*('04'));  
 console.log(await *sendAjax*('05'));  
})()**

# 14.class

***1. 通过 class 定义类/实现类的继承  
2. 在类中通过constructor定义构造方法  
3. 通过new来创建类的实例  
4. 通过 extends 来实现类的继承  
5. 通过super调用父类的构造方法  
6. 重写从父类中继承的一般方法***

**class Person {  
 constructor(name,age){  
 *//属性全部定义在constructor中* this.name = name  
 this.age = age  
 }  
 speak(){  
 return this.name + this.age  
 }  
}  
*//继承*class Child extends Person{  
 constructor(name,age,sex){  
 super(name,age)  
 this.sex = sex  
 }  
 study(){  
 return '我正在学习'  
 }  
 speak(){  
 return this.name + this.age + this.sex  
 }  
}**

**let *p* = new Person('kobe',18)  
console.log(*p*)  
let *c* = new Child('wade',19,'男')  
console.log(*c*.speak())**

# \*\*\*\*\*\*\*\*其它\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 1.字符串扩展

***1. includes(str) : 判断是否包含指定的字符串  
2. startsWith(str) : 判断是否以指定字符串开头  
3. endsWith(str) : 判断是否以指定字符串结尾  
4. repeat(count) : 重复指定次数***

# 2.数值扩展

***1. 二进制与八进制数值表示法: 二进制用 0b, 八进制用 0o  
2. Number.isFinite(i) : 判断是否是有限大的数  
3. Number.isNaN(i) : 判断是否是NaN  
4. Number.isInteger(i) : 判断是否是整数  
5. Number.parseInt(str) : 将字符串转换为对应的数值  
6. Math.trunc(i) : 直接去除小数部分***

# 3.数组扩展

***1. Array.from(v) : 将伪数组对象或可遍历对象转换为真数组  
2. Array.of(v1, v2, v3) : 将一系列值转换成数组  
3. find(function(value, index){return true}) : 找出第一个满足条件返回true的元素  
4. findIndex(function(value, index){return true}) : 找出第一个满足条件返回true的元素下标***

# 4.对象扩展

***1. Object.is(v1, v2)  
 \* 判断2个数据是否完全相等  
2. Object.assign(target, source1, source2..)  
 \* 将源对象的属性复制到目标对象上  
3. 直接操作 \_\_proto\_\_ 属性  
 let obj2 = {};  
 obj2.\_\_proto\_\_ = obj1;***

**console.log(NaN === NaN)*//false*console.log(Object.is(0,-0))*//false*console.log(Object.is(NaN,NaN))*//true***

# 5.深度克隆

***1、数据类型：  
 \* 数据类型分为：  
 -基本的数据类型(String, Number, boolean, Null, Undefined, Symbol)  
 特点： 存储的是该对象的实际数据  
 -对象数据类型  
 特点： 存储的是该对象在栈中引用，真实的数据存放在堆内存里  
   
2、复制数据  
 - 基本数据类型存放的就是实际的数据，可直接复制  
 let number2 = 2;  
 let number1 = number2;  
 - 克隆数据：对象/数组  
 1、区别： 浅拷贝/深度拷贝  
 判断： 拷贝是否产生了新的数据还是拷贝的是数据的引用  
 知识点：对象数据存放的是对象在栈内存的引用，直接复制的是对象的引用  
 let obj = {username: 'kobe'}  
 let obj1 = obj; // obj1 复制了obj在栈内存的引用  
 2、常用的拷贝技术  
 1). arr.concat(): 数组浅拷贝  
 2). arr.slice(): 数组浅拷贝  
 3). JSON.parse(JSON.stringify(arr/obj)): 数组或对象深拷贝, 但不能处理函数数据,函数会变成null  
 4). 浅拷贝包含函数数据的对象/数组  
 5). 深拷贝包含函数数据的对象/数组  
  
 3、扩展：如何做到精准判断数据类型？  
 用：Object.prototype.toString.call(xx)，可以获得类似 [object Type] 的字符串。  
 且注意：这种方式判断NaN，返回的是[object Number]  
   
 Object.prototype.toString.call(1) //[object Number]  
 Object.prototype.toString.call('1') //[object String]  
 Object.prototype.toString.call(undefined) //[object Undefined]  
 Object.prototype.toString.call(true)// [object Boolean]  
 Object.prototype.toString.call(console.log)//[object Function]  
 Object.prototype.toString.call(Symbol())//[object Symbol]  
 Object.prototype.toString.call(b) //报错  
 Object.prototype.toString.call([]) //[object Array]  
 Object.prototype.toString.call({}) //[object Object]  
 Object.prototype.toString.call(NaN)//[object Number]-------注意这里***

**let *a* = 1  
 let *b* = *a  
 a* = 5  
 console.log(*b*)  
   
 let *obj* = {  
 name:'kobe',  
 age:18  
 }  
   
 let *obj2* = *obj  
 obj*.name = 'wade'  
 console.log(*obj2*)  
  
*//1.通过arr.concat(): 数组浅拷贝*let *arr* = [2,4,6,8,{name:'kobe'}]  
let *arr2* = []  
*arr2* = *arr2*.concat(*arr*)  
*arr*[4].name = 'haha'  
console.log(*arr2*);  
  
*//2.通过arr.slice(): 数组浅拷贝*let *arr* = [2,4,6,8,{name:'kobe'}]  
let *arr2* = []  
*arr2* = *arr*.slice(0)  
*arr*[4].name = 'haha'  
console.log(*arr2*);  
  
*//3.通过JSON.parse进行深度复制(缺陷：会把函数对象搞丢)*let *arr1* = [2,4,6,8,{name:'kobe'},function demo(){console.log(1)}]  
let *arr2* = *JSON*.parse(*JSON*.stringify(*arr1*))  
*arr1*[4].name = 'wade'  
console.log(*arr1*);  
console.log(*arr2*);  
  
*//手写一个深度复制* let *obj* = {name:'kobe',age:18}  
 console.log(*obj*.toString());  
   
 let *arr* = [2,4,6,8]  
 console.log(*arr*.toString());  
   
 let *demo* = Object.prototype.toString.call(*arr*)  
 console.log(*demo*);  
  
console.log(Object.prototype.toString.call(1)); *//[object Number]*console.log(Object.prototype.toString.call('1')); *//[object String]*console.log(Object.prototype.toString.call(undefined)); *//[object Undefined]*console.log(Object.prototype.toString.call(true));*// [object Boolean]*console.log(Object.prototype.toString.call(console.log));*//[object Function]*console.log(Object.prototype.toString.call(Symbol()));*//[object Symbol]*console.log(Object.prototype.toString.call(null));*//[object Null]  
 //Object.prototype.toString.call(b) //报错*console.log(Object.prototype.toString.call([])); *//[object Array]*console.log(Object.prototype.toString.call({})); *//[object Object]*console.log(Object.prototype.toString.call(NaN));*//[object Number]-------注意这里  
  
//专门用于判断数据类型*function *getType*(target) {  
 return Object.prototype.toString.call(target).slice(8,-1)  
}  
  
function *deepClone*(target) {  
 let result = null  
 if(*getType*(target) === 'Object'){  
 result = {}  
 }  
 else if(*getType*(target) === 'Array'){  
 result = []  
 }  
 else{  
 return target  
 }  
 for (let key in target){  
 let item = target[key]  
 if(*getType*(item) === 'Object'||'Array'){  
 result[key] = *deepClone*(item)  
 }else{  
 result[key] = item  
 }  
 }  
 return result  
}  
  
let *obj* = {  
 name:'kobe',  
 age:18,  
 sex:{  
 option1:'男',  
 option2:'女',  
 option3:'不男不女',  
 },  
 getInfo(){  
 return this.name + this.age  
 }  
}  
  
let *obj2* = *deepClone*(*obj*)  
  
*obj*.sex.option3 = '人妖'  
*obj*.age = 19  
console.log(*obj*)  
console.log(*obj2*)  
console.log(*obj*.getInfo())  
console.log(*obj2*.getInfo())**

# 6.Set和Map数据结构

***1. Set容器 : 无序不可重复的多个value的集合体  
 \* Set()  
 \* Set(array)  
 \* add(value)  
 \* delete(value)  
 \* has(value)  
 \* clear()  
 \* size  
2. Map容器 : 无序的 key不重复的多个key-value的集合体  
 \* Map()  
 \* Map(array)  
 \* set(key, value)//添加  
 \* get(key)  
 \* delete(key)  
 \* has(key)  
 \* clear()  
 \* size***

***//关于set容器的使用：  
//1.set ------->新增*console.log(*set*);  
*set*.add('abc');  
console.log(*set*, *set*.size);  
*//2.delete(value)---->删除所有指定value的元素  
set*.delete(2);  
console.log(*set*);  
*//3.has(value)---->是否拥有指定数值* console.log(*set*.has(1));*//true* console.log(*set*.has(2));*//false  
//4.clear()  
set*.clear();*//清空set容器里的所有数据*console.log(*set*);  
  
*//关于map容器的使用：  
//1.创建一个map容器并将数据放入map容器（注意是二维数组的形式传参）*let *map* = new Map([['name', 'kobe'],['age', '18']]);  
console.log(*map*)  
*//set------>向map容器里增加一个key-value  
map*.set('sex', '男');  
console.log(*map*);  
*//get------->从容器中获取指定key对应的value*console.log(*map*.get('age'));*//age  
//delete(key)------>从容器中删除指定  
map*.delete('性别');  
console.log(*map*);  
console.log(*map*.has('name'));*//false*console.log(*map*.has('age'));*//true  
map*.clear();*//清空*console.log(*map*);*//Map(0) {}***

# \*\*\*\*\*\*\*\*ES7\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***1. 指数运算符(幂): \*\*  
2. Array.prototype.includes(value) : 判断数组中是否包含指定value***

**let *arr* = [1,4,2,3,4,5,7,8,6,7]  
console.log((3 \*\* 3));  
console.log(*arr*.includes(3));**

# \*\*\*\*\*\*\*\*ES8\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Object.values() 将对象中所有属性的值，提取出来作为一个数组  
Object.entries() 将对象转化为数组***

**const *person* = {name: 'jack', age: 18};  
const *values* = Object.values(*person*);  
const *keys* = Object.keys(*person*);  
*console*.log(*keys*, *values*);  
const *array* = Object.entries(*person*);  
*console*.log(*array*);**

# \*\*\*\*\*\*\*\*ES9\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**const *promise2* = new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve('数据1');  
 }, 2000)  
})  
const *promise3* = new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve('数据2');  
 }, 1000)  
})  
const *promise4* = new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve('数据3');  
 }, 3000)  
})  
const *arr* = [*promise2*, *promise3*, *promise4*];  
*// 在forEach遍历中 async await没有作用  
// 而在for、for in、for of才会起作用  
arr*.forEach(async (item) => {  
 const result = await item;  
 *console*.log(result);  
})**

***// 默认是初始化状态promise对象  
 const promise = new Promise((resolve, reject) => {  
 setTimeout(() => {  
 reject('定时器执行玩了');  
 }, 1000)  
 })  
  
 promise  
 .then((res) => {  
 console.log(res);  
 })  
 .catch((err) => {  
 console.log(err);  
 })  
 .finally((res) => {  
 // finally 最终的。不管是成功还是失败，都会触发的回调函数  
 console.log(res);  
 })  
 // 默认是成功状态promise对象  
 // const promise1 = Promise.resolve('resolved');  
 const promise1 = Promise.reject('rejected');  
 promise1  
 .then((res) => {  
 console.log(res);  
 })  
 .catch((err) => {  
 console.log(err);  
 })***

**const *promise2* = new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve('数据1');  
 }, 2000)  
})  
  
const *promise3* = new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 reject('数据2');  
 }, 1000)  
})  
  
const *promise4* = new Promise((resolve, reject) => {  
 *setTimeout*(() => {  
 resolve('数据3');  
 }, 3000)  
})  
  
Promise.all([*promise2*, *promise3*, *promise4*])  
 .then((res) => {  
 *// 等所有promise对象的状态变成成功的状态时，才会触发  
 console*.log(res);  
 })  
 .catch((err) => {  
 *// 一旦有一个promise对象的状态是失败的状态时，就会触发  
 console*.log(err);  
 })**

# \*\*\*\*\*\*\*\*ES10\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**const *arr* = [[1], 2, [[3, 4], [[5]], 6]];  
*// 数组的扁平化处理：将一个多维数组转化成1维数组  
console*.log(*arr*.flat(Infinity));**