ES6+ CHEAT SHEET

github.com/Hengyu-DU

ECMAScript 2015-2021

知识点		涉及语法	功能	备注
变量	let, const	1 只在声明所在的块级作用域有效 2 不存在变量提升 3 暂时性死区（变量只能在声明后使用） 4 不允许重复声明		作用：防止在变量声明前就使用这个变量
	const	声明一个只读的常量， 改变常量的值会报错	1 对简单数据类型而言：值保存在变量指向的内存地址中，不会改动。 2 对复杂数据类型（对象、数组）而言：变量指向的内存地址**保存的是一个指针**，指向的数据结构本身是可以改变的，const无法控制。	
		Object.freeze(obj)	冻结对象	常规模式下，obj.prop = 123 不会起作用 严格模式下，上面这行会报错
	块级作用域	1 基本使用	外层作用域不影响内层作用域，内层作用域可以声明外层已声明的同名作用域	块级作用域的出现使得[ES5]立即执行匿名函数（IIFE）不在必要了。
		2 函数声明	块级作用域中的函数声明： 浏览器ES6环境中：当var处理 其它环境中：当let处理	应该尽量避免在块级作用域内声明函数
	ES6声明变量的6种方法	1 var 2 function 2 let 4 const 5 import 6 class	let b = 1 window.b **// undefined**	var\function声明的全局变量是顶层对象的属性；let\const\class声明的全局变量**不属于顶层对象的属性**
解构赋值	数组的解构 事实：只要某种数据结构具有Iterator接口，就都可以采用数组形式的解构赋值。	let [a, [[b], c]] = [1, [[2], 3]]	1 模式匹配	完美对应
		let [x, y] = [1, 2, 3]	2 不完全解构	非完美对应
		let [x, y='b'] = ['a'] **// x='a' y='b'**	3 默认值	生效条件：对象的属性值严格等于undefined
	对象的解构	let {a,b} = {a:1,b:2} **// a=1 b=2**	1 变量与属性同名	对象的属性的次序可以打乱
		let {a:foo} = {a:1,b:2} **//foo = 1**	2 变量与属性不同名	真正被赋值的是后者，不是前者
		let {x, y=5} = {x:1} **// x=1, y=5**	3 默认值	生效条件：对象的属性值严格等于undefined
		let arr = [1,2,3] let {0:a, [arr.length-1]:b} = arr **a // 1 b // 3**	* 对数组进行对象属性的解构	因为数组本质上是一个特殊的对象 [arr.length-1]的写法属于“属性名表达式”
	字符串的赋值解构	let [a,b,c,d,e] = '12345' let [...arr] = '12345'	**arr // ['1','2','3','4','5']** **a//1 b//2 c//3 d//4 e//5**	
	函数参数的赋值解构	[1,undefined,3].map((x='yes')=>x) // [1,'yes',3]		
字符串		.codePointAt()	正确处理4个字节储存的字符，返回一个字符的码点	[ES5]的charCodeAt()无法正确处理4个字节的字符
		String.fromCodePoint(编码)	从码点返回字符	
		.includes('',index)	表示是否包含，返回布尔值	index表示开始搜索位置
		.startsWith('',index)	表示是否以它开始，返回布尔值	index表示开始搜索位置
		.endsWith('',index)	表示是否以它结束，返回布尔值	index表示前n个字符
		.repeat(n)	表示将原数组重复n次,返回一个新字符串	小数会被取整、负数会报错、NaN等同于0
		.padStart(n,'')	用于头部补全	n表示最小长度
		.padEnd(n,'')	用于尾部补全	n表示最小长度
正则表达式	语法	var reg = new RegExp('xyz','i')	RegExp构造函数使用一	[ES5]
		var reg = new RegExp(/xyz/i)	RegExp构造函数使用二	[ES5]
		let reg = new RegExp(/xyz/,'i')	**ES6允许的新的传参方式**	
		u修饰符	处理4个字节的UTF-16编码	
		y修饰符（粘连修饰符sticky）	全局匹配，但每次都从上一次匹配成功的下一个位置开始匹配，且暗含(^)头部匹配。	单独一个y修饰符对match方法只返回第一个匹配，必须与g修饰符连用才能返回所有匹配。
		.sticky	表示是否设置了y，返回布尔值	
		.flags	返回所有修饰符	
	方法	**RegExp.prototype.exec(str)**	在有g或y修饰符的情况下，从上次匹配结束的地方开始，返回匹配到的元素。每调用一次exec，该正则都会记录一次lastIndex，当匹配不到时，lastIndex自动为0	
		RegExp.prototype.test(str)	返回布尔值，表示是否匹配	
		String.prototype.replace(regexp,xx)	替换，第二个参数可以是字符串，也可以是函数	
		String.prototype.split(regexp)	以正则匹配到的字符作为分隔符，拆分字符串	
		String.prototype.search(regexp)	返回字符串中第一个匹配到正则的字符的位置，找不到就返回-1	
		String.prototype.match(regexp)	返回所有匹配到的字符串，如果用了g，则返回数组，如果没用g，则返回第一个匹配成功的	
	ES2020	String.prototype.matchAll(regexp)	一次性取出所有匹配。不过，它返回的是一个遍历器（Iterator），而不是数组。	
数值	Number	0b / 0B	二进制数值	如果要转换为10进制，用Number('')
		0o / 0O	八进制数值	
		Number.isFinite()	判断一个数值是否有限	[ES5]的isFinite()和isNaN()先调用Number()把参数转为数值，再进行判断。ES6新方法只对数值有效
		Number.isNaN()	判断一个数值是否为NaN	
		Number.parseInt()	转为整数	目的是减少全局性方法
		Number.parseFloat()	转为小数	
		Number.isInteger()	用来判断一个数是否为整数	3和3.0被视为同一个值
		Number.EPSILON	一个极小的常量，用于为浮点数计算设置一个误差范围，只要小于这个数，我们就可以视结果为正确。	
		Number.MAX_SAFE_INTEGER	常量，最大安全整数	JavaScript能够准确表示-2的53次方和2的53次方之间（不含端点）的整数，超过这个范围就无法精确表示。
		Number.MIN_SAFE_INTEGER	常量，最小安全整数	
		Number.isSafeInteger()	判断是否落在安全整数范围内	

ES6+ CHEAT SHEET

github.com/Hengyu-DU

ECMAScript 2015-2021

知识点		涉及语法	功能	备注
数值	Math对象 17个	Math.trunc(n)	去除小数部分，返回整数部分	对于非数值，会先调用Number()转为数值
		Math.sign(n)	判断一个数是正(+1)负(-1)、还是零(0)、还是其它(NaN)	
		Math.cbrt(n)	计算一个数的立方根	
		Math.hypot(n,n,n…)	返回所有参数的平方和的平方根	Math.hypot(3,4) // 5
		Math.expm1(x)	返回e的x方-1	
		Math.log1p(x)	返回ln(1+x)	
		Math.log10(x)	返回以10为底的x的对数	
		Math.log2(x)	返回以2为底的x的对数	
		Math.clz32()	Math.imul()	Math.fround()
		Math.sinh() Math.cosh() Math.tanh() Math.asinh() Math.acosh() Math.atanh() 6个双曲函数方法		
	BigInt (ES2020)	1n 2n 3n	使用后缀n来表示	
		BigInt()	生成BigInt类型	必须有参数，转换规则与Number()一致
运算符	指数运算符 (ES2016)	**	2 ** 2 // 4 a **= 3 // 等同于 a = a*a*a	内部实现和Math.pow(n,指数）不同，对特别大的运算结果，两者会有细微差异。
	链判断运算符 (ES2020)	let a = info?.body?.user?.name || 'default'	在链式调用的时候判断，左侧的对象是否为null或undefined。如果是的，就不再往下运算，而是返回undefined。	
		iterator.return?.()	iterator.return如果有定义，就会调用该方法，否则iterator.return直接返回undefined，不再执行?.后面的部分。	
	Null 判断运算符 (ES2020)	const a = obj.prop ?? 'hello'	?? 它的行为类似 || ，但是**只有运算符左侧的值为null或undefined时**，才会返回右侧的值。	[ES5]的||运算符常被用来指定默认值，开发者希望只要属性的值为null或undefined，默认值就会生效，但是属性的值如果为**空字符串或false或0**，默认值也会生效。为了避免这种情况，**ES2020** 引入了新的Null判断运算符。
		const animationDuration = response.settings?.animationDuration ?? 300;	如果response.settings是null或undefined，或者response.settings.animationDuration是null或undefined，就会返回默认值300。这一行代码包括了两级属性的判断。	这个运算符的一个目的，就是跟链判断运算符?.配合使用，为null或undefined的值设置默认值。
	逻辑赋值运算符 (ES2021)	x ||= y	x || (x = y)	先进行逻辑运算，然后根据运算结果，再视情况进行赋值运算。 用途：为变量设定默认值 user.id ||= 1
		x &&= y	x && (x = y)	
		x ??= y	x ?? (x = y)	
函数	函数参数的默认值	1 ES6允许为函数参数设置默认值		
		2 参数变量是默认声明的，所以不能用let或const再次声明		
		3 使用参数默认值时，函数不能有同名参数		
		4 参数默认值是惰性求值，即每次调用函数时都会重新计算默认值表达式		
		5 参数默认值可以和解构赋值一起用	function m1({x=0,y=0} = {}){…}	参数的默认值是空对象，但设置了解构赋值的默认值
			function m2({x,y} = {x:0,y:0}){…}	参数的默认值是有具体属性的对象，但解构赋值没有设置默认值
		6 通常情况下，**定义了默认值的参数应该放在所有参数的最后一位**，因为这样方便看出哪些参数可以省略。如果是非尾部的参数设置默认值，实际上这个参数是不能省略的，除非显式输入undefined		
		7 length属性返回不含默认值的参数个数	(function(a){}).length // 1 (function(a,b,c=5){}).length // 2	如果**非尾部参数**设置了默认值，那么length属性不会计入默认值参数后面的参数的个数
		8 作用域：一旦设置了参数默认值，在函数进行声明初始化时会形成一个单独的作用域。详见P108-P111		
	rest参数	...变量名	function add(...values){…}	rest参数只能放在参数末尾
	严格模式 (ES2016)	函数参数如果使用了默认值、解构赋值或扩展运算符，就不能在内部显式设置为严格模式，否则会报错。原因是函数执行时先执行参数、再执行函数体，而只有进入函数体才能确定是否开启了严格模式，这样不合理。		规避限制： 方式一： 设置全局性的严格模式 方式二： 把函数包在一个无参数的立即执行函数中，为该立即执行函数开启严格模式。
	name属性	function foo(){} foo.name // "foo"	返回函数的函数名	1 new Function() 创造的函数： 返回 'anonymous' 2 bind()方法创造的函数： 返回 'bound 原函数名'
	箭头函数	1 箭头函数的代码块部分多于一句的时候，要用大括号括起来，并使用return返回。	var foo1 = id => id var foo2 = id => { return id } var foo3 = id => ({id:id,name:'a'})	如果直接返回一个对象，必须在对象外面加上括号，否则会被当作是代码块
		2 箭头函数不可以当作构造函数，不可以使用arguments对象，不可以使用yield命令		
		3 箭头函数根本没有自己的this，而是引用外层的this，因此也无法用call,bind,apply去改变this的指向		
	尾调用优化	某个函数的最后一步就是调用另一个函数，这样可只保留内层函数的调用帧，使每次执行时调用帧只有一项，从而大大节省内存。（尾调用优化只在严格模式下开启？）		
	柯里化	将多参数的函数转换成单参数的形式。		

ES6+ CHEAT SHEET

github.com/Hengyu-DU

ECMAScript 2015-2021

知识点		涉及语法	功能	备注
数组	扩展运算符	`[...arr1,...arr2,...arr3]`	合并数组	
		`const [a,...b] = [1,2,3,4,5]`	解构赋值	扩展运算符只能放在最后一位
		`[...'yes'] // ['y','e','s']`	字符串转数组	额外好处：能正确识别32位的Unicode字符

		`function f(a,b,c){}`
		`var arr = [1,2,3]`
		`f(...arr)`
			替代数组的apply方法	

	数组对象的方法	`Array.from(*)`	将**类似数组的对象**和**可遍历对象**转为数组	伪数组、字符串、Set、Map （所谓类似数组的对象，本质特征是必须有**length属性**）
		`Array.from(*[,函数,this])`	第二个参数可以传入一个方法，对每个元素进行加工处理，返回新的元素，类似于map	
		`Array.of(n,n,n...)`	将一组数值转为数组，弥补Array()或new Array()的行为差异	
	数组实例的方法	`.copyWithin(target[,start,end])`	将数组内部指定位置成员复制到目标位置	start默认为0，end默认为数组长度
		`.find(fn)`	用于找到第一个符合条件(fn)的数组成员	找到就返回该成员，没有返回undefined
		`.findIndex(fn)`	用于找到第一个符合条件(fn)的数组成员	找到就返回成员位置，没有则返回-1。 以上两个方法都可以发现NaN，弥补了indexOf()的不足
		`.fill(n [,start,end])`	用给定值n对数组内指定位置进行填充	
		`.includes(n [,index])`	表示是否包含给定的值	可以识别NaN，并且比indexOf()更直观!
		`.keys()`	遍历键名	返回一个遍历器对象，可用for..of循环遍历
		`.values()`	遍历键值	
		`.entries()`	遍历键值对	
对象	属性名表示	`var foo = 'bar'` `var obj = { foo } // {foo:'bar'}`	1 简洁表示法	
		`obj['f'+'o'+'o'] = 123`	2 属性名表达式	不能和简洁表示法一起用

	name属性	返回对象方法的函数名	`const a = {`
			`  sayHi() {} }`
			`a.sayHi.name // 'sayHi'`
				1 如果使用了getter和setter:
				返回 'get 方法名' 或 'set 方法名'
				2 如果方法名是一个Symbol值:
				返回 Symbol值的描述

	严格比较	`Object.is(n1,n2)`	比较两个值是否严格相等	与===的不同在于，+0不等于-0，NaN等于自身
	属性的遍历		**自身的属性** / **继承的属性** / **Symbol属性**	
		`for...in`	仅可枚举的 / 仅可枚举的 / ✕	
		`Object.keys(obj)`	仅可枚举的 / ✕ / ✕	
		`Object.getOwnPropertyNames(obj)`	可枚举的 + 不可枚举的 / ✕ / ✕	
		`Object.getOwnPropertySymbols(obj)`	✕ / ✕ / 所有Symbol属性	
		`Reflect.ownKeys(obj)`	可枚举的 + 不可枚举的 / ✕ / 所有Symbol属性	
	原型对象相关方法	`obj.__proto__ = 原型对象`	指定原型对象	__proto__不是一个正式对外的API，从兼容性角度，最好不要使用它
		`Object.setPrototypeOf(对象，原型对象)`	设置原型对象	es6正式推荐的
		`Object.getPrototypeOf(对象)`	读取原型对象	es6正式推荐的
	遍历键值对	`Object.keys(obj)`	遍历键名	仅含自身的可枚举的属性
		`Object.values(obj)`	遍历键值	
		`Object.entries(obj)`	遍历键值对	
	复制对象	`Object.assign(target,source1,source2)`	把源对象(source)的所有**可枚举属性**复制到目标对象(target)	1 如果有同名属性，后面的会覆盖前面的 2 字符串的包装对象会产生可枚举的实义属性 3 Object.assign方法是**浅复制**，不是深复制 4 可以处理数组，但会把数组作为对象来处理，即属性名为各自的index序号。

	操作对象的相关方法	`Object.create(原型对象[,属性])`	创建一个对象，指定原型和属性	[ES5]
				第二个参数添加的对象属性如果不显式声明，默认是不可枚举的。
				显式声明:
				`let obj = Object.create({},{p:`
				`{ value: 10, enumerable: true}})`
				`Object.values(obj) // [10]`

		`Object.defineProperty(对象，属性名,`
		`{ enumerable:true,`
		`value:属性值 } )`
			显式定义一个对象的属性	[ES5]

		`Object.getOwnPropertyDescriptor(对象,属性名)`	获取某属性的描述对象:
			`Object { value:123,`
			`writable:true,`
			`enumerable:true,`
			`configurable:true }`
				当enumerable为false时:
				for…in
				Object.keys()
				JSON.stringify()
				均无法枚举该属性

		`Object.getOwnPropertyDescriptors(对象)`	返回指定对象自身的所有属性（非继承属性）的描述对象	**ES2017**引入，该方法解决了Object.assign()无法正确复制get属性和set属性的问题

			浅合并:
			`const shallowMerge = (target, source) => Object.defineProperties(`
			`target,`
			`Object.getOwnPropertyDescriptors(source) )`	

			浅克隆:
			`const shallowClone = (obj) => Object.create(`
			`Object.getPrototypeOf(obj),`
			`Object.getOwnPropertyDescriptors(obj) )`	

ES6+ CHEAT SHEET

github.com/Hengyu-DU

ECMAScript 2015-2021

知识点		涉及语法	功能	备注
对象	扩展运算符	let {x,y,...z} = {x:1,y:2,a:3,b:4} z // {a:3,b:4} let aClone = {...a} 相当于: let aClone = Object.assign({},a) let ab = {...a,...b} 相当于: let ab = Object.assign({},a,b)	扩展运算符与解构赋值 （取出参数对象的所有可枚举属性，并复制到当前对象中）	解构赋值是**浅复制**，且不复制原型对象 let obj = {a:{b:1}} let {...x} = obj obj.a.b = 2 x.a.b //2 如果想要完整复制对象原型的属性: let clone = Object.assign(Object.create(Object.getPrototypeOf(obj)),obj)
		自定义属性放在扩展运算符前面 （后面的同名属性会覆盖前面的）	let obj = {...a, x:1, y:2} let obj = {...a, ...{x:1, y:2}} let x=1,y=2, obj = {...a, x, y} let obj = Object.assign({}, a, {x:1,y:2})	
		自定义属性放在扩展运算符后面 （相当于设置了新对象的**默认属性值**）	let obj = {x:1, y:2, ...a} let obj = Object.assign({}, {x:1,y:2}, a) let obj = Object.assign({x:1,y:2}, a)	
symbol	基本使用	let mySymbol = Symbol('描述')	Symbol函数前不能用new，因为生成的Symbol是一个原始类型的值，不是对象，类似于字符串	
	Symbol作为属性名	var a = {} a[mySymbol] = 'hello'	写法一	a[mySymbol] // 'hello' 1. 使用Symbol值定义和读取属性时，Symbol值必须放在方括号中 2. Symbol作为属性名时，只有Object.getOwnPropertySymbols方法和Reflect.ownKeys方法可以返回Symbol键名，利用这个特性可以为对象定义一些非私有但又只希望用于内部的方法

		var a = {
		[mySymbol] : 'hello'
		}
			写法二	

		Object.defineProperty(a, mySymbol, { value:'hello' })	写法三	
	方法	Symbol.for('')	返回一个symbol值且登记在全局，下次调用时先在全局中搜索，有的话仍会返回这个值	
		Symbol.keyFor()	返回一个已经登记了的symbol类型值的key	
	内置的Symbol值（11个）	Symbol.hasInstance	Symbol.replace	Symbol.toPrimitive
		Symbol.isConcatSpreadable	Symbol.search	Symbol.toStringTag
		Symbol.species	Symbol.split	Symbol.unscopables
		Symbol.match	Symbol.iterator	
Set	初始化	const s = new Set(参数)	参数可以是：数组和任何有iterator接口的数据结构	
	属性	Set.prototype.constructor	构造函数，默认是Set函数	
		Set.prototype.size	返回Set实例的成员总数	
	操作方法	add(value)	添加某个值，返回Set结构本身	
		delete(value)	删除某个值，返回一个布尔值，表示是否删除成功	
		has(value)	返回一个布尔值，表示参数是否是Set的成员	
		clear()	清除所有成员，无返回值	
	遍历方法	keys() values() entries()		
		forEach()		
		[...set].map() [...set].filter()	利用扩展运算符把Set转为数组后，也可使用map和filter	
	数组去重	let newArr = [...new Set(arr)]		
		let newArr = Array.from(new Set(arr))		
	并集、交集、差集	并集	交集	差集
		new Set([...a, ...b])	new Set([...a].filter(x => b.has(x)))	new Set([...a].filter(x => !b.has(x)))
WeakSet	和Set的区别	1 WeakSet的成员只能是对象，而不能是其它类型的值 2 WeakSet中的对象都是弱引用，WeakSet不可遍历，没有size属性		
	初始化	let a = new WeakSet(参数)	参数应该是一个对象，或一个数组，成员都为对象	
	操作方法	add(value)		
		delete(value)		
		has(value)		
Map	初始化	let a = new Map(参数)	参数可以是任何具有Iterator接口，且每个成员都是一个双元素数组的数据结构，包括数组、Set、Map	
	属性和方法	Map.prototype.size	返回Map实例的成员总数	
		set(key,value)	设置key所对应的键值，返回整个Map结构（因此可以采用链式写法）	如果key已经有值，则键值会被更新，否则就新生成该键。
		get(key)		
		has(key)		
		delete(key)		
		clear()		
	遍历方法	keys() values() entries()	结合… 可以实现键、值、键值快速转为数组	
		forEach(function(**value,key**,map){...})		
		[...map].map() [...map].filter()	利用扩展运算符把map转为数组后，也可使用map和filter，后面函数的形参是**[k,v]**	
WeakMap	和Map的区别	1 WeakMap只接收对象作为键名 2 WeakMap中的键名所引用的对象都是弱引用，不可遍历，没有size属性		
	初始化	let a = new WeakMap(参数)		
	操作方法	get(key)		
		has(key)		
		has(key)		
		delete(key)		