setInterval的用法与setTimeout完全一致，区别仅仅在于setInterval指定某个任务每隔一段时间就执行一次，也就是无限次的定时执行

let 是在代码块内有效，var 是在全局范围内有效:

const 声明一个只读的常量，一旦声明，常量的值就不能改变。

let 只能声明一次 var 可以声明多次:

let 不存在变量提升，var 会变量提升:

const 声明一个只读变量，声明之后不允许改变。意味着，一但声明必须初始化，否则会报错。

解构赋值是对赋值运算符的扩展

1.数组模型的解构（Array）:

基本   let [a, b, c] = [1, 2, 3]; //a=1 b=2 c=3

可嵌套  let [a, [[b], c]] = [1, [[2], 3]]; //a=1 b=2 c=3

可忽略  let [a, , b] = [1, 2, 3]; //a=1 b=3

不完全解构  let [a = 1, b] = []; // a = 1, b = undefined

剩余运算符   let [a, ...b] = [1, 2, 3]; //a = 1 //b = [2, 3]

字符串等    let [a, b, c, d, e] = 'hello'; // a = 'h' // b = 'e' // c = 'l' // d = 'l' // e = 'o'

解构默认值   let [a = 2] = [undefined]; // a = 2

            let [a = 3, b = a] = []; // a = 3, b = 3 let [a = 3, b = a] = [1]; // a = 1, b = 1 let [a = 3, b = a] = [1, 2]; // a = 1, b = 2

            a 与 b 匹配结果为 undefined ，触发默认值：a = 3; b = a =3

            a 正常解构赋值，匹配结果：a = 1，b 匹配结果 undefined ，触发默认值：b = a =1

            a 与 b 正常解构赋值，匹配结果：a = 1，b = 2

2.对象模型的解构（Object）

基本: let { foo, bar } = { foo: 'aaa', bar: 'bbb' }; // foo = 'aaa' // bar = 'bbb'

let { baz : foo } = { baz : 'ddd' }; // foo = 'ddd'

可嵌套可忽略: let obj = {p: ['hello', {y: 'world'}] };

let {p: [x, { y }] } = obj; // x = 'hello' // y = 'world'

let obj = {p: ['hello', {y: 'world'}] }; let {p: [x, { }] } = obj; // x = 'hello'

不完全解构: let obj = {p: [{y: 'world'}] };

let {p: [{ y }, x ] } = obj; // x = undefined // y = 'world'

剩余运算符: let {a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40}; // a = 10 // b = 20 // rest = {c: 30, d: 40}

解构默认值  let {a = 10, b = 5} = {a: 3}; // a = 3; b = 5;

let {a: aa = 10, b: bb = 5} = {a: 3}; // aa = 3; bb = 5;

ES6 引入了一种新的原始数据类型 Symbol ，表示独一无二的值，最大的用法是用来定义对象的唯一属性名。

ES6 数据类型除了 Number 、 String 、 Boolean 、 Object、 null 和 undefined ，还新增了 Symbol 。

基本用法:Symbol 函数栈不能用 new 命令，因为 Symbol 是原始数据类型，不是对象

let sy = Symbol("KK"); console.log(sy); // Symbol(KK)    typeof(sy);// "symbol"      // 相同参数 Symbol() 返回的值不相等 let sy1 = Symbol("kk"); sy === sy1; // false

使用场景:

1.作为属性名

let sy = Symbol("key1"); // 写法1 let syObject = {}; syObject[sy] = "kk"; console.log(syObject); // {Symbol(key1): "kk"} // 写法2 let syObject = { [sy]: "kk" }; console.log(syObject); // {Symbol(key1): "kk"} // 写法3 let syObject = {}; Object.defineProperty(syObject, sy, {value: "kk"}); console.log(syObject); // {Symbol(key1): "kk"}

Symbol 作为对象属性名时不能用.运算符，要用方括号 let syObject = {}; syObject[sy] = "kk"; syObject[sy]; // "kk" syObject.sy; // undefined

注意点!!!

Symbol 值作为属性名时，该属性是公有属性不是私有属性，可以在类的外部访问。但是不会出现在 for...in 、 for...of 的循环中，也不会被 Object.keys() 、 Object.getOwnPropertyNames() 返回。如果要读取到一个对象的 Symbol 属性，可以通过 Object.getOwnPropertySymbols() 和 Reflect.ownKeys() 取到。

let syObject = {}; syObject[sy] = "kk"; console.log(syObject); for (let i in syObject) { console.log(i); } // 无输出 Object.keys(syObject); // [] Object.getOwnPropertySymbols(syObject); // [Symbol(key1)] Reflect.ownKeys(syObject); // [Symbol(key1)](这里有点麻烦我没有很理解,大概看了一下)

2.定义常量

是用字符串不能保证常量是独特的，这样会引起一些问题,但是使用 Symbol 定义常量，这样就可以保证这一组常量的值都不相等

Symbol 的值是唯一的，所以不会出现相同值得常量，即可以保证 switch 按照代码预想的方式执行。

3.Symbol.for()

Symbol.for() 类似单例模式，首先会在全局搜索被登记的 Symbol 中是否有该字符串参数作为名称的 Symbol 值，如果有即返回该 Symbol 值，若没有则新建并返回一个以该字符串参数为名称的 Symbol 值，并登记在全局环境中供搜索。

let yellow = Symbol("Yellow");

let yellow1 = Symbol.for("Yellow"); yellow === yellow1; // false

let yellow2 = Symbol.for("Yellow"); yellow1 === yellow2; // true

4.Symbol.keyFor()

Symbol.keyFor() 返回一个已登记的 Symbol 类型值的 key ，用来检测该字符串参数作为名称的 Symbol 值是否已被登记。

let yellow1 = Symbol.for("Yellow"); Symbol.keyFor(yellow1); // "Yellow"

ES6 Map 与 Set ===============(不是很理解)

Map 对象保存键值对。任何值(对象或者原始值) 都可以作为一个键或一个值。

Map 中的 key:key 是字符串,key 是对象,key 是函数,key 是 NaN

Map 的迭代:

for...of,

forEach()

var myMap = new Map(); myMap.set(0, "zero"); myMap.set(1, "one"); // 将会显示两个 logs。 一个是 "0 = zero" 另一个是 "1 = one" myMap.forEach(function(value, key) { console.log(key + " = " + value); }, myMap)

Set 对象

Set 对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用。

Set 对象存储的值总是唯一的，所以需要判断两个值是否恒等。有几个特殊值需要特殊对待：

+0 与 -0 在存储判断唯一性的时候是恒等的，所以不重复；

undefined 与 undefined 是恒等的，所以不重复；

NaN 与 NaN 是不恒等的，但是在 Set 中只能存一个，不重复。

Set 对象作用

数组去重

var mySet = new Set([1, 2, 3, 4, 4]); [...mySet]; // [1, 2, 3, 4]

并集

var a = new Set([1, 2, 3]); var b = new Set([4, 3, 2]); var union = new Set([...a, ...b]); // {1, 2, 3, 4}

交集

var a = new Set([1, 2, 3]); var b = new Set([4, 3, 2]); var intersect = new Set([...a].filter(x => b.has(x))); // {2, 3}

差集

var a = new Set([1, 2, 3]); var b = new Set([4, 3, 2]); var difference = new Set([...a].filter(x => !b.has(x))); // {1}

类型转换

Array

// Array 转 Set var mySet = new Set(["value1", "value2", "value3"]); // 用...操作符，将 Set 转 Array var myArray = [...mySet]; String // String 转 Set var mySet = new Set('hello');  // Set {'h', 'e', 'l', 'l', 'o'} // 注：Set 中 toString 方法是不能将 Set 转换成 String

Proxy 与 Reflect 是 ES6 为了操作对象引入的 API   ===============(不是很理解)

Proxy 可以对目标对象的读取、函数调用等操作进行拦截，然后进行操作处理。它不直接操作对象，而是像代理模式，通过对象的代理对象进行操作，在进行这些操作时，可以添加一些需要的额外操作。

Reflect 可以用于获取目标对象的行为，它与 Object 类似，但是更易读，为操作对象提供了一种更优雅的方式。它的方法与 Proxy 是对应的。

ES6 字符串

ES6 之前判断字符串是否包含子串，用 indexOf 方法，ES6 新增了子串的识别方法。

includes()：返回布尔值，判断是否找到参数字符串。

startsWith()：返回布尔值，判断参数字符串是否在原字符串的头部。

endsWith()：返回布尔值，判断参数字符串是否在原字符串的尾部。

ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。Set本身是一个构造函数，用来生成 Set 数据结构。

//官网例子亲测正确，这个方法可以实现一位数组去重(1)

const s = new Set();

[2, 3, 5, 4, 5, 2, 2].forEach(x => s.add(x));

for (let i of s) {

  console.log(i);}

//这个方法实现一位数组去重好像更棒棒哒(2)

const set = new Set([1, 2, 3, 4, 4]);[...set] //[ 1, 2, 3, 4, 5 ]

总结一下：

// 去除数组的重复成员

[...new Set(array)]

上面的方法也可以用于，去除字符串里面的重复字符。

[...new Set('ababbc')].join('')

// "abc"

1.向 Set 加入值的时候，不会发生类型转换，所以5和"5"是两个不同的值，主要的区别是NaN等于自身，而精确相等运算符认为NaN不等于自身。

2.两个对象总是不相等的。

let set = new Set();

set.add({});set.size // 1

set.add({});set.size // 2

Set 实例的属性和方法：

\* Set.prototype.constructor：构造函数，默认就是Set函数。

\* Set.prototype.size：返回Set实例的成员总数。

Set 实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。

\* 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，表示删除是否成功。

\* has(value)：返回一个布尔值，表示该值是否为Set的成员。

\* clear()：清除所有成员，没有返回值。

const items = new Set([1, 2, 3, 4, 5]);

const array = Array.from(items);

这就提供了去除数组重复成员的另一种方法。

function dedupe(array) {

  return Array.from(new Set(array));}

dedupe([1, 1, 2, 3]) // [1, 2, 3]

Set 结构的实例有四个遍历方法，可以用于遍历成员。

\* keys()：返回键名的遍历器

\* values()：返回键值的遍历器

\* entries()：返回键值对的遍历器

\* forEach()：使用回调函数遍历每个成员

Set 结构的键名就是键值（两者是同一个值），因此第一个参数与第二个参数的值永远都是一样的。另外，forEach方法还可以有第二个参数，表示绑定处理函数内部的this对象。

扩展运算符（...）内部使用for...of循环，所以也可以用于 Set 结构

let arr = [3, 5, 2, 2, 5, 5];let unique = [...new Set(arr)];

// [3, 5, 2]

因此使用 Set 可以很容易地实现并集（Union）、交集（Intersect）和差集（Difference）。

let a = new Set([1, 2, 3]);let b = new Set([4, 3, 2]);

// 并集

let union = new Set([...a, ...b]);

// Set {1, 2, 3, 4}

// 交集

let intersect = new Set([...a].filter(x => b.has(x)));

// set {2, 3}

// 差集

let difference = new Set([...a].filter(x => !b.has(x)));

// Set {1}

WeakSet 结构与 Set 类似，也是不重复的值的集合。但是，它与 Set 有两个区别。首先，WeakSet 的成员只能是对象，而不能是其他类型的值。其次，WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakSet 之中。

WeakSet 结构有以下三个方法。

\* WeakSet.prototype.add(value)：向 WeakSet 实例添加一个新成员。

\* WeakSet.prototype.delete(value)：清除 WeakSet 实例的指定成员。

\* WeakSet.prototype.has(value)：返回一个布尔值，表示某个值是否在 WeakSet 实例之中。

ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。

proxy

Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。

var proxy = new Proxy(target, handler);

Proxy 对象的所有用法，都是上面这种形式，不同的只是handler参数的写法。其中，new Proxy()表示生成一个Proxy实例，target参数表示所要拦截的目标对象，handler参数也是一个对象，用来定制拦截行为。

（这个感觉内容有点多，没看太明白，项目中用到的拦截器也是比较简单的，后面再仔细研究一下这个）

Reflect对象与Proxy对象一样，也是 ES6 为了操作对象而提供的新 API

Reflect对象一共有 13 个静态方法。

\* Reflect.apply(target, thisArg, args)

\* Reflect.construct(target, args)

\* Reflect.get(target, name, receiver)

\* Reflect.set(target, name, value, receiver)

\* Reflect.defineProperty(target, name, desc)

\* Reflect.deleteProperty(target, name)

\* Reflect.has(target, name)

\* Reflect.ownKeys(target)

\* Reflect.isExtensible(target)

\* Reflect.preventExtensions(target)

\* Reflect.getOwnPropertyDescriptor(target, name)

\* Reflect.getPrototypeOf(target)

\* Reflect.setPrototypeOf(target, prototype)

上面这些方法的作用，大部分与Object对象的同名方法的作用都是相同的，而且它与Proxy对象的方法是一一对应的

Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

Promise对象有以下两个特点：（1）对象的状态不受外界影响 （2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。

这个里面太多内容了 (土拨鼠叫)