Symbol | Map | Set | Const

Symbol

1、用于创建一个独一无二的标识符
2 、主要用途是作为对象属性的键，而这些属性是独一无二的，可以防止属性名的冲突

可以在创建Symbol值的时候传入一个描述description：在MDN中，对其释义是可用于调试但不是访问 symbol 本身

let sym1 = Symbol('小余');//括号内是对应的描述标记
console.log(sym1.description);//小余
1 、Es10 以后新增的
2 、 ES10 以前使用Symbol.prototype.toString()。自己实现的，优先级大于Object.prototype.toString()

1 、Symbol本身都具备一个唯一标识，每次调用Symbol() 时，由 JavaScript 引擎生成。
2 、这个标识是隐蔽的，开发者无法访问
2 、虽然标识无法访问，但描述是可以的，通过对应的实例属性进行获取

获取 key
1 、使用Symbol作为key的属性名的话，在遍历/Object.keys等情况中，是获取不到这些Symbol值的
2 、使用getOwnPropertyNames方法只能获取字符串形式的属性名
3 、对于Symbol类型的属性名，有相似的方法getOwnPropertySymbols去进行获取

实例方法
const sym = Symbol("example");
sym === sym[Symbol.toPrimitive](); // true
toString
valueOf
[Symbol.toPrimitive]() - 用于将Symbol对象转为symbol值

静态方法
Symbol.for与Symbol.keyFor
全局符号注册表，在MDN中，称呼为Symbol注册表，for通常代表循环遍历的含义，在这里遍历内容指的就是遍历该注册表，从而进行查询对应内容

for是查询Symbol本身，keyFor查询的是Symbol的注释，也就是description内容

```
// 创建一个全局 Symbol
var globalSym = Symbol.for('foo')
console.log(Symbol.keyFor(globalSym)) // "foo"
var globalSym1 = Symbol.for()
console.log(Symbol.keyFor(globalSym1)) // "undefined"

var localSym = Symbol()
console.log(Symbol.keyFor(localSym)) // undefined，localSym是普通使用Symobl创建，而非放入Symbol注册表中
var localSym1 = Symbol('bar')
console.log(Symbol.keyFor(localSym1))
```

Symbol.for(key) 方法会根据给定的键 key，来从运行时的 symbol 注册表中找到对应的 symbol，如果找到了，则返回它，否则，新建一个与该键关联的 symbol，并放入全局 symbol 注册表中

for方法和keyFor方法的出现，主要是为了解决全局状态管理的问题，在复杂的应用或多个合作组件间需要共享某些全局状态或标识时，能够得以实现

Map

正常情况下，对象作为key，但这是不可以的，这个时候键值对会自动将对象转成字符串来作为key

```
//对象作为key会被转化为字符串
const obj1= {name:"小余"}
const obj2= {name:"coderwhy"}

const info = {
  [obj1]:'aaa',
  [obj2]:'bbb',
}
// 会被覆盖
console.log(info);//{ '[object Object]': 'bbb' }
```

```
const originalMap = new Map([
  ['key1', 'value1'],
  ['key2', 'value2']
]);
//旧Map到新Map的转换
const newMap = new Map(originalMap);
```

key(键)只能够是对象

set、get、has、delete

不能遍历

WeakMap

```
// 使用 Map 的情况
const map = new Map();
const element = document.querySelector('.my-element');

// 存储一些与元素相关的数据
map.set(element, {
  clickCount: 0,
  lastClicked: null,
  customData: 'some data'
});

// 当元素被删除时
element.remove(); // 元素从 DOM 中删除

// 尝试重新获取元素
const newElement = document.querySelector('.my-element');
console.log(newElement); // 输出: null

// 但是我们可以通过原始的 element 引用从 Map 中获取数据
console.log(map.get(element)); // 仍然可以获取到数据
// 输出: { clickCount: 0, lastClicked: null, customData: 'some data' }
```

如果是 WakMap，就没有了，因为已经被回收

const let

变量环境 词法环境

在 ECMAScript 的最新规范中，每个执行上下文包括两个环境：
变量环境 (VE)：主要用于存储由 var 声明的变量和函数声明。
词法环境 (LE)：用于存储由 let 和 const 声明的变量，以及包含块级作用域的其他信息

window是浏览器实现的，在V8引擎中是没有window的，这也是我们在node环境下打印window会报错
如今实际的var声明是放在variable_中的，这是一个VariableMap类型，也是一种用C++实现的hashMap结构，且C++也是需要分配内存来实现存储的，所以内容依旧是存储在内存当中的
在目前的情况下，虽然使用var仍可以在window中找到，但这是为了兼容性所做出的牺牲取舍，正常情况下，这是不应该存在的
var在变量存储位置进行存放时，会同步在window进行操作，但也说明了在ES6中，window和variable_已经不是同一个对象了

作用域提升

let/const被划分为两个阶段，从该内容所暴露出来的信息，可以确定创建阶段是不变的
关键的改变在于访问阶段：词法绑定被求值之前(赋值过程之前)，无法访问
这么看来，在执行上下文的词法环境创建出来的时候，变量事实上已经被创建了，只是这个变量是不能被访问的

事实上维基百科并没有对作用域提升有严格的概念解释，那么我们自己从字面量上理解
作用域提升：在声明变量的作用域中，如果这个变量可以在声明之前被访问，那么我们可以称之为作用域提升
在这里，它虽然被创建出来了，但是不能被访问，我认为不能称之为作用域提升
所以我的观点是let、const没有进行作用域提升，但是会在执行上下文创建阶段被创建出来

块级作用域

ES5中，JavaScript只会形成两个作用域：全局作用域和函数作用域

1、内层能够链式访问外层

2、外层无法访问内存
3、这不是绝对的，比如说我在块级作用域中书写一个函数，在外层能否访问
根据规范，块级作用域对function声明的类型是有效的，所以外层是无法访问内层函数的
4、但实际上可以访问，因为不同的浏览器有不同的实现方式，大部分浏览器为了兼容以前的代码，放开该层限制，让function不具备块级作用域
5、可以配合let声明来实现具备块级作用域

```
{
  let foo = function(){

console.log('xiaoyu');
  }
}

foo()//不管是浏览器还是node环境，都无法访问
```

for循环

1、for循环中，一般采用let，而非const的原因主要是在这的索引一般是递增或者以某种规律进行改变
2、采用const，则无法正常的进行变化
3、其他的场景下，也有采用const进行声明的形式，例如for of遍历，该遍历的使用目标为可迭代对象，这一点我们后面还会进行说明
4、从目前的角度来说，for of是直接遍历集合的元素而不是索引的方法，所以可以不使用let声明一个变量来记录索引，对于不想要改变的数据此时就可以使用const来进行声明，直接访问元素最直观的就是简化迭代

ES6引入统一的迭代协议：可迭代协议和迭代器协议

暂时性死区

在变量声明和初始化之间存在一个时间段，在这段时间里，变量虽然已经被声明，但还不能被访问或使用

```
//暂时性死区
function foo(){
   console.log("小余");
   console.log("测试");
   // console.log(bar); 无法访问

   let bar = "bar"
   let good = "好的"
}
```

Set

一个新增的数据结构，可以用来保存数据，类似于数组，但是和数组的区别是元素不能重复

检查一个元素是否存在于一个大数组中通常需要遍历整个数组(线性)，这是一个时间复杂度为O(n)的操作。而Set内部结构（通常实现为哈希表）做到查找操作的平均时间复杂度接近O(1)，从而在频繁检查元素存在性的场景下性能大大提高

WeakSet

WeakSet中只能存放对象类型，不能存放基本数据类型

不能遍历，遍历相当于强引用

WeakSet对元素的引用是弱引用
弱引用指的是对对象的引用不足以阻止该对象被垃圾回收机制回收。在
WeakSet中，对象的存在不会影响到它们的生命周期
如果没有其他引用存在，这些对象仍然可以被GC回收

add(value)：添加某个元素，返回WeakSet对象本身
delete(value)：从WeakSet中删除和这个值相等的元素，返回boolean类型
has(value)：判断WeakSet中是否存在某个元素，返回boolean类型