1.ES6新增语法

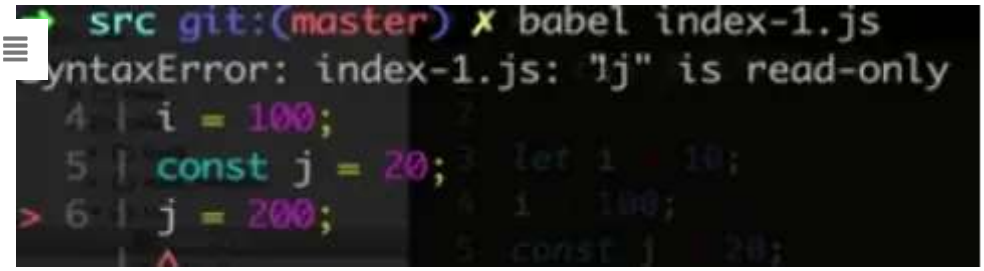
一、声明及数据赋值的方式

1. let、const

- let和var类似，声明和定义变量（但**let**又能解决**块级作用域/闭包**的问题）
- const，定义常量，不允许重新赋值
- 示例【babel转es5】

```
let i = 10;
i = 100 ;
const j = 20;
//j = 200;
```


```
~~~
let i = 10;
i = 100 ;
const j = 20;
j = 200;
~~~
```



1-1. var、let 及 const 区别?

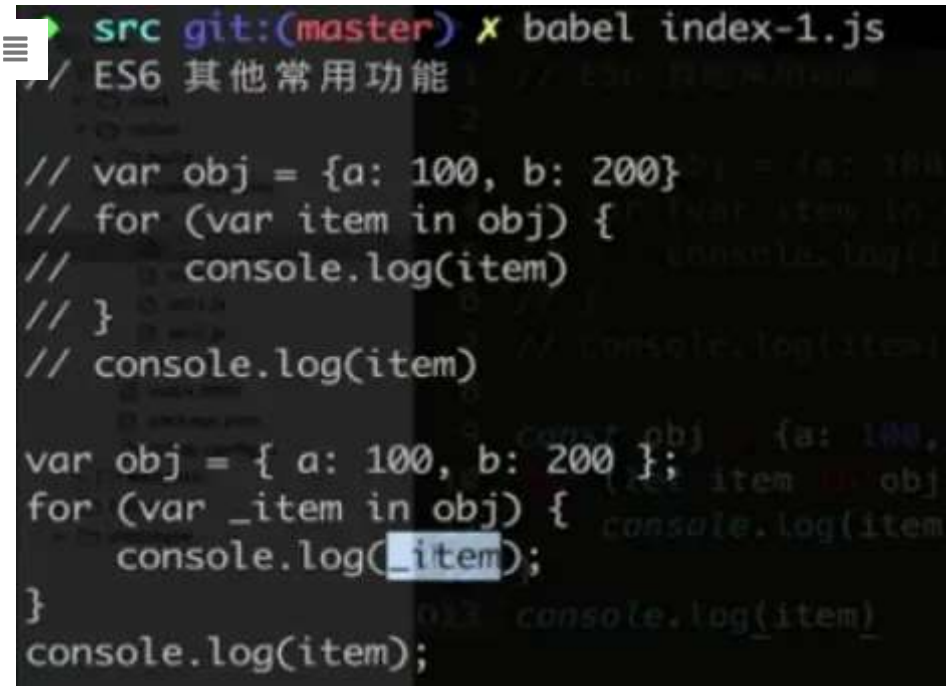
- **全局**申明的var变量会**挂载**在window上，而let和const不会
- **var**声明变量存在**变量提升**，let和const没有变量提升??
- **let、const**的作用范围是**块级作用域**，而var的作用范围是**函数作用域**
- 同一作用域下let和const**不能声明同名变量/重复声明**，而var可以
- 同一作用域下在let和const声明前使用会存在**暂时性死区**(变量声明之前不允许操作)
- **const**
  - 一旦声明**必须立刻赋值**,不能使用null占位
  - 声明后**不能再修改**
  - 如果声明的是**复杂的引用类型**，可以修改其属性（此时一定要谨慎使用）

1-2. let的块级作用域

- **let**彻底解决了普通JS**无块级作用域**的坑
- 避免了变量外露和变量污染

```
// var obj = {a:100, b:200}
// for (var item in obj){
//     console.log(item)
// }
// console.log(item) //'b'  外部可以访问到

var obj = {a:100, b:200}
for (let item in obj){   //let实现了块级作用域（重新定义了_item）
    console.log(item)
}
console.log(item)  //undefined  外部不可以访问到
```



2. Symbol

- JS的第六种值类型的数据类型【Number、String、Boolean、Null、Undefined】
- **引入背景**：**对象的属性名**容易易产生生命名冲突，为保证键名的唯一性，故es6引入**Symbol**这种新的原始数据类型，确保创建的每个变量量都是**独**一无二的

2-1. **特点**：

- **不能new。**Symbol类型的数据是类似**字符串**的数据类型，由于Symbol函数返回的值是原始类型的数据，**不能是对象**，故Symbol函数前**不能使用new**命令，否则会报错

```
> a = new String('a');
< ▶ String {"a"}
> b = new Number(1);
< ▶ Number {1}
> c = new Boolean(true);
< ▶ Boolean {true}
> d = new Symbol('name');
✖ ▶ Uncaught TypeError: Symbol is not a constructor
      at new Symbol (<anonymous>)
      at <anonymous>:1:5
> d = Symbol('name');
< Symbol(name)
```

- **可选参数。**由于控制台输出不同的Symbol变量时都是Symbol()，故为了区分，可在创建Symbol变量时**传入参数**进行区分。如：

```
// 这里的a1，a2的作用可以说是为了备注，为了在输出Symbol变量时能够区分不同变量。
let x = Symbol('a1') //Symbol(a1)
```

```
let y = Symbol('a2') //Symbol(a2)
```

2-2. 用法

- 定义对象的**唯一属性**名

```
// 在对象里用Symbol作为属性名的三种写法,首先定义
let name = Symbol()

// 第一种方式：借助数组读取name变量，此时不能用点运算符,点运算符默认后面的参数是字
let a = {}
a[name] = 'Nick'
//{Symbol(): "Nick"}

// 第二种方式：构造时声明
let a = {
    [name]: 'Nick'
}
//{Symbol(): "Nick"}

// 第三种 Object.defineProperty
let a = {}
Object.defineProperty(a, name, { value: 'Nick' });
//{Symbol(): "Nick"}
```

- 定义**常量**

```
// 定义字符串常量
const name = Symbol("Nick");
//Symbol(Nick)
```

3. 解构赋值

- 解构赋值可以理解为**赋值操作**的**语法糖**，它是针对**数组或者对象**进行模式匹配，然后对其中对变量进行赋值。代码书写上言简意赅，语义明确，也方便了**对象数据的读取操作**。
- 一行代码把一个对象或者数组中的多个属性/元素拆解出来
  - ES6中只要**某种数据**有**Iterator接口**（也就是可以循环迭代），都可以进行数组的解构赋值。

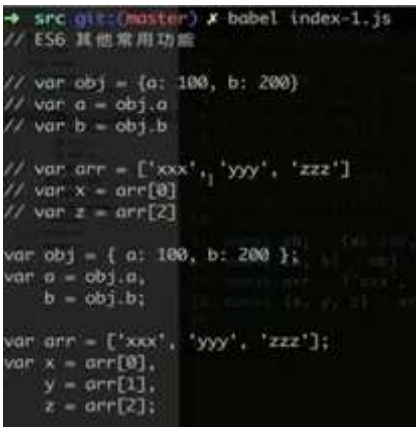
```
// var obj = {a:100, b:200}
// var a = obj.a
// var b = obj.b
// var arr = ['xxx','yyy','zzz']
```

```
// var x = arr[0]
// var z = arr[2]

const obj = {a:100,b:200} //从对象obj中把a和b两个属性解构出来
const{a,b} = obj  //再赋值给a和b两个变量
const arr = ['xxx','yyy','zzz']
const [x,y,z] = arr

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

- 数组解构
- 对象解构
- 字符串解构
- 布尔值解构
- 函数参数解构
- 数值解构



二、新增内置对象及已有对象的扩展

1. 字符串扩展的API及模板字符串

- 扩展的API

方法	描述
includes(string, position)	判断字符串中是否包含指定字符串，返回值是布尔值
startsWith(string, position)	判断字符串的开头是否包含指定字符串，返回值是布尔值
endsWith(string, position)	判断字符串的尾部是否包含指定字符串，返回值是布尔值
repeat(n)	repeat() 方法返回一个新字符串，表示将院字符串重复n 次。
字符串补全	第一个参数是补全后的字符串长度，第二个参数是用于补全的字符串
padStart(length, str)	用于头部补全
padEnd(length, str)	用于尾部补全

1-1. 加强对Unicode支持

- 加强了对Unicode的支持

  在ES5中我们知道JavaScript 允许采用\uxxxx形式表示一个字符，其中xxxx表示字符的 Unicode 码点。这种表示法只限于码点在\u0000~\uFFFF之间的字符。超出这个范围的字符，必须用两个双字节的形式表示，但是ES5却无法正确的识别这个有两个字节组成的字符。ES6中，JavaScript增加了对超出\u0000~\uFFFF

  Unicode范围的字符支持。

  ES6的方案：将超过两个字节的组成的字符的码点放在一对花括号里就可以正确的识别。

1-2. 模板字符串

- 通过**反引号``的方式**可以在多行中定义完整字符串，可以用来定义html模板;
- 并且可以**使用${}方式**引入外部变量
- 易读性大大提升，代码也更简洁

```
//  var name = "zhangsan" ,age = 20, html ='';
//  html += '<div>';
//  html += '  <p>'+name+'</p>';
//  html += '  <p>'+age+'</p>';
//  html += '<div>';
```

```
const name = "zhangsan" ,age = 20;
const html = `
            <div>
                <p>${name}</p>
                <p>${age}</p>
            </div>
        `;   //注意必须使用反引号
```

模板字符串注意事项

- 在模板字符串**内部**如需使用**反引号**，反引号前要用**反斜杠 \ 转**义
- 使用模板字符串表示多行字符串时，所有的**空格和缩进都会被保留**在输出之中
- 模板字符串串中引入变量量，要用**${变量量名}这**样的形式引入才可以
- 大括号中的值不是字符串时，将按照一般的规则**转为字符串。**比比如，大括号中是一个对象，将默认调用对象的**toString方法**
- 模板字符串串中的${.....} 大大括号内部可以放入任意的 **JavaScript 表达式**，可以进行运算、可以引用对象属性、可以调用函数、可以甚至还能嵌套，甚至还能调用自己本身

2. 数组扩展的API及扩展运算符

介：介绍ES6和ES7提供的新的数组方法及扩展运算符的使用

- 扩展运算符的使用
  - 复制数组
  - 分割数组
  - 将数组转化成参数传递给函数
- 新增的常用方法
  - fill
  - find
  - findIndex
  - Includes
  - flat
  - filter

2-1. 扩展运算符

用途：复制数组、分割数组、将数组转化为参数传递给函数

```
{
    // 扩展运算符的使用 ...

    // 复制数组的操作
    const list = [1, 2, 3, 4, 5]
    let list2 = [...list]
    console.log(list2)  //[ 1, 2, 3, 4, 5 ]
    list2.push(6)
    console.log(list2)  //[ 1, 2, 3, 4, 5, 6 ]
    console.log(list)  //[ 1, 2, 3, 4, 5 ]

    // 分割数组
    const totalList = [1, 'a', 'b', 'c']
    let [, ...strList] = totalList
    console.log(strList)  //[ 'a', 'b', 'c' ]

    // 将数组转化为参数传递给函数
    function add(x, y) {
        return x + y
    }
    let addList = [1, 2]
    console.log(add(...addList)); //3
}
```

2-2. 新增API

1. **ary.fill(val,start,end)**

   fill(填充值，开始填充位置，停止填充位置[不包含])

2. **ary.find(item=>item.id===1)**
3. **ary.findIndex(item=>item.id===1)**
4. **ary.includes(val) ary.indexOf(val)**

```
//和indexOf在简便性、精确性有所区别，但是根据判断存在和查找索引的情境来区分使用
[1, 2, NaN].includes(NaN)     // true
[1, 2, NaN].indexOf(NaN)  // -1

[1, 2, NaN].includes(2)     // true
[1, 2, NaN].indexOf(2)  // 1
```

5. **ary.flat(N) 扁平化展开层数N**

```
// flat展开数组的操作
const list = [1, 2, 3, ['2nd', 4, 5, 6, ['3rd', 7, 8]]]
let flatList = [].concat(...list)
console.log(flatList)  //[ 1, 2, 3, '2nd', 4, 5, 6, [ '3rd', 7, 8 ] ]

// 默认只展开第一层数组
let flatList2 = list.flat(2)
console.log('flat', flatList2)  //[1, 2, 3, "2nd", 4, 5, 6, "3rd", 7, 8]
let flatList3 = list1.flat(Infinity) //一直展开到无法展开为止
```

2-3. map和reduce

- **map(fn,context)** 第一个参数一般是回调函数，第二个参数不常用（this指向）
- reduce 对数组中的每个元素进行一次回调，升序执行然后将回调值汇总一个返回值
  - @params ***cb(acc, currentValue, currentIndex, Array), initalValue*** @params ***acc***：回调返回值 @params ***currentValue***：当前值 @params currentIndex：当前索引 @params Array：当前数组 @params ***initalValue***：可选初始值，存在时替换acc

2-3-1. map

map 作用是生成一个新数组，遍历原数组，将每个元素拿出来做一些变换然后返回一个新数组，原数组不发生改变。

- map 的**回调函数**接受三个参数，分别是当前**索引元素，索引，原数组**

```
var arr = [1,2,3];
var arr2 = arr.map(item => item + 1)
```

```
arr   //[ 1, 2, 3 ]
arr2  // [ 2, 3, 4 ]
```

```
['1','2','3'].map(parseInt)
// -> [ 1, NaN, NaN ]
```

- 第一个 parseInt('1', 0) -> 1
- 第二个 parseInt('2', 1) -> NaN
- 第三个 parseInt('3', 2) -> NaN
- **parseInt第二个参数只能是 0，2，8，10，16**

2-3-2. reduce

- **reduce** 可以将**数组中的元素通过回调函数最终转换为一个值。**
- 如果我们想实现一个功能将函数里的元素全部相加得到一个值，可能会这样写代码：

```
const arr = [1, 2, 3]
let total = 0
for (let i = 0; i < arr.length; i++) {
  total += arr[i]
}
console.log(total) //6
```

- 但是如果我们使用 reduce 的话就可以
  将遍历部分的代码优化为一行代码

```
const arr = [1, 2, 3]
const sum = arr.reduce((acc, current) => acc + current, 0)
console.log(sum)
```

- 对于reduce来说，它接受两个参数，分别是回调函数和初始值，接下来我们来分解上述代码中 reduce 的过程：
  - 首先初始值为 0，该值会在执行**第一次回调函数**时作为**第一个参数**传入
  - 回调函数接受四个参数，分别为**累计值、当前元素、当前索引、原数组**，后三者想必大家都可以明白作用，这里着重分析第一个参数
  - 在一次执行回调函数时，**当前值和初始值**相加得出结果 1，该结果会在**第二次执行回调函数**时当做**第一个参数**传入
  - 所以在第二次执行回调函数时，相加的值就分别是 1 和 2，以此类推，循环结束后得到结果 6

3. 对象的新特性和新增方法

1. 新方法

3-1-1. 扩展运算符的使用

- 复制对象
- 给对象设置默认值
- 合并对象

```
{
    // 一、对象中扩展运算符的使用

    // 1.复制对象
    const obj = { name: 'Nick', video: 'es6' }
    const initObj = { color: 'red' }
    let videoObj = { ...obj }
    console.log(videoObj) //{ name: 'Nick', video: 'es6' }

    // 2.设置对象默认值(在后面替换掉不想要的属性name)
    let obj2 = { ...obj, name: 'Jack' }
    console.log(obj2) //{ name: 'Jack', video: 'es6' }
    let obj21 = {  name: 'Jack',...obj }  //相当于是赋值操作，会覆盖
    console.log(obj21) //{ name: 'Nick', video: 'es6' }

    // 3.合并对象
    let obj3 = { ...obj, ...initObj }
    console.log(obj3) //{ name: 'Nick', video: 'es6', color: 'red' }

    // 坑点（只能复制一层,浅拷贝）
    // 1. 在设置对象默认值时，...的位置不同也会有所改变
    // 2. 简单类型的时候，使用扩展运算符是没问题的，
        // 但是如果扩展运算符展开对象以后，还是一个对象的话，我们复制的只是一个指
}
```

3-1-2. **属性和对象方法初始化**的简写

```
{
    // 二、属性和对象方法初始化的简写
    let name = '小明'
    let age = 18
    let es5Obj = {
        name: name,
        age: age,
        sayHello: function () {
            console.log('this is es5Obj')
        }
    }
```

```
    // es6中属性名(同名时)、方法都可以简写[一种语法糖]
    let es6Obj = {
        name,
        age,
        sayHello() {
            console.log('this is es6Obj')
        }
    }

    console.log('es5', es5Obj) //es5 { name: '小明', age: 18, sayHello: [Functi
    console.log('es6', es6Obj) //es6 { name: '小明', age: 18, sayHello: [Functi
    es5Obj.sayHello()  //this is es5Obj
    es6Obj.sayHello()  //this is es6Obj
}
```

3-1-3. 可计算的属性名

```
{
    //三 、可计算的属性名
    let key = 'name'
    let es5Obj = {}
    es5Obj[key] = '小明'  //对象属性是一个变量，用中括号
    let es6Obj = {
        [key]: '小红'  //初始化时就传入对象属性
    }
    console.log(es5Obj, es6Obj)
    //{ name: '小明' } { name: '小红' }
}
```

3-1-4. 新增方法

- **Object.is(NaN,NaN)** 【除了该场景，其他场景等同于===】
- **Object.assign(des,src)** 【浅拷贝/合并对象】
- **Object.keys**【可以实现深拷贝 for (const key of Object.keys(json))】
- Object.values
- Object.entries 【返回数组[key,value]】

3-2. Map和WeakMap

3-2-1. **Map**

- JavaScript中的对象，实质就是键值对的集合(Hash结构)。 但是在对象里却**只能用字符串作为键名**。在一些特殊的场景里就满足不了我们的需求了，正因为此，Map这**一类似对象的数据结构**提出了，它是 JavaScript中的一种**更完善的Hash结构**。

Map对象

- 用于保存键值对，任何值(对象或者原始值)都可以作为一个键或一个值。
- 使用介绍

```
// 通过构造函数创建一个Map
var m = new Map();
```

- 内置API

属性/方法	作用	例子
size	返回键值对的数量	m.size
clear()	清除所有键值对	m.clear()
has(key)	判断键值对中是否有指定的键名，返回值是布尔值	m.has(key)
get(key)	获取指定键名的键值对，如不存在则返回undefined	m.get(key)
set(key, value)	添加键值对，如键名已存在，则更新键值对	m.set(key, value)
delete(key)	删除指定键名的键值对	m.delete(key)

  - 遍历器生成函数
    - keys
    - values
    - entries
  - 遍历方法
    - forEach

3-2-2. WeakMap

1. **键名必须是对象**，不接受其他类型的数据作为键名
2. 因为键名所指的对象不触发垃圾回收机制
3. **没有clear， 没有size， 无法遍**历

3-3. Set和WeakSet

3-3-1. **Set**

- Set是ES6给开发者提供的**一种类似数组的数据结构**，可以理解为值的集合。
- 它和数组的最大的区别就在于: **它的值不会有重复项。**

```
// 创建
let set = new Set();
let set2 = new Set([1,2,3])

// 添加元素
set.add(1)
```

- 特点
  - 成员值唯一
- 属性及方法

属性/方法	作用	例子
size	返回成员个数	s.size
clear()	清除所有成员	s.clear()
has(value)	判断键值对中是否有指定的值，返回值是布尔值	s.has(key)
delete(value)	删除指定值	s.delete(key)

- 用途

```
// 去重
let arr = [1,2,2,3,4,4,4];
let s = new Set(arr);
//结果: Set {1,2,3,4}

let newArr = Array.from(s);
//结果: [1,2,3,4],完成去重
```

3-3-2. **WeakSet**

- 数组成员**必须是对象**
- WeakSet结构也**提供了add( ) 方法，delete( ) 方法，has( )方法**给开发者使用，作用与用法跟Set结构完全一致。
- WeakSet 结构**不可遍历**。因为它的成员都是对象的弱引用，随时被回收机制回收，成员消失。所以WeakSet 结构不会有keys( )，values( )，entries( )，forEach( )等方法和size属性。

3-4. Map、Set与Array及Object间的区别

- 增删改查（map最简单）
- 类型转换
  - map和对象间的转换
    - entries(obj)和fromEntries(map)

- 数组和set
  - newSet(array)和Array.from(set)

3-5. Set、Map、WeakSet 和 WeakMap 的区别?

这些集合对象在存储数据时会使用到键，包括可迭代的 `Map` 和 `Set` ，支持按照插入顺序来迭代元素。

3-5-1. Set

- 表示有没有，成员的值都是唯一的，没有重复的值
- 可以接受一个数组（或可迭代的数据结构）作为参数
- 注：两个对象总是不相等的
- 属性：
  - 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。
- 方法：
  - add(value)：添加某个值，返回 Set 结构本身。
    - `s.add(1).add(2).add(2)` ；
  - 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。
- 遍历方法
  - 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

3-5-2. WeakSet

WeakSet 结构与 Set 类似，也是不重复的值的集合。但与 Set 有几个区别：

- WeakSet 的成员**只能是对象**，而不能是其他类型的值
- WeakSet 中的对象都是弱引用
  - 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
  - **垃圾回收机制依赖引用计数，如果一个值的引用次数不为0，垃圾回收机制就不会释放这块内存。**结束使用该值之后，有时会忘记取消引用，导致内存无法释放，进而可能会引发内存泄漏。WeakSet 里面的引用，都不计入垃圾回收机制，所以就不存在这个问题。因此，

WeakSet 适合临时存放一组对象，以及存放跟对象绑定的信息。**只要这些对象在外部消失它在 WeakSet 里面的引用就会自动消失。**

- WeakSet 不可遍历
  - 由于 WeakSet 内部有多少个成员，取决于垃圾回收机制有没有运行，运行前后很可能成员个数是不一样的，而垃圾回收机制何时运行是不可预测的
- WeakSet 结构中没有clear方法。

3-5-3. Map

- 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Map的键。**
- 遍历方法
- Map 结构生提供三个遍历器生成函数和一个遍历方法。
  - keys()：返回键名的遍历器。
  - values()：返回键值的遍历器。
  - entries()：返回所有成员的遍历器。
  - forEach()：遍历 Map 的所有成员。

3-5-4. WeakMap

- WeakMap的设计目的在于: 有时我们想在某个对象上面存放一些数据，但是这会形成对于这个对象的引用，而一旦不再需要这两个对象，我们就必须手动删除这个引用，否则垃圾回收机制就不会释放被引用对象占用的内存。
- 基本上，如果你要往对象上添加数据，又不想干扰垃圾回收机制，就可以使用 WeakMap。
- 一个典型应用**场景**是，在网页的 DOM 元素上添加数据，就可以使用WeakMap结构。当该 DOM 元素被清除，其所对应的WeakMap记录就会自动被移除。

4. Proxy和Reflect

4-1. **Proxy**

- 正如Proxy的英译"代理"所示，Proxy是ES6为了操作对象引入的API。它不直接作用在对象上，而是作为一种媒介，如果需要**操作对象**的话，需要**经过这个媒介的同意。Vue3.0** 中将会通过 Proxy 来替换原来的**Object.defineProperty** 来实现数据响应式。
- 使用方式

```
/* @params
** target: 用用Proxy包装的目目标对象
** handler: 一个对象，对代理对象进行拦截操作的函数，如set、get
```

```
*/
let p = new Proxy(target, handler)
```

- 常用方法（见代码）
  - get (target, key)
  - set(target, key, value)
  - has(target, key)
  - deleteProperty(target, key)
  - ownKeys(target)

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 obj = new Proxy({}, {
  get: function (target, key, receiver) {
    console.log(`getting ${key}!`);
    return Reflect.get(target, key, receiver);
  },
  set: function (target, key, value, receiver) {
    console.log(`setting ${key}!`);
    return Reflect.set(target, key, value, receiver);
  }
});
```

Proxy 支持的拦截操作一览，一共 13 种。

- get(target, propKey, receiver)
  - 拦截对象属性的读取，比如proxy.foo和proxy['foo']。
- set(target, propKey, value, receiver)
  - 拦截对象属性的设置，比如proxy.foo = v或proxy['foo'] = v，返回一个布尔值。
- has(target, propKey)
  - 拦截propKey in proxy的操作，返回一个布尔值。
- deleteProperty(target, propKey)
  - 拦截delete proxy[propKey]的操作，返回一个布尔值。
- ownKeys(target)
  - 拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)、for...in循环，返回一个数组。该方法返回目标对象所有自身的属性的属

性名，而Object.keys()的返回结果仅包括目标对象自身的可遍历属性。

- getOwnPropertyDescriptor(target, propKey)
  - 拦截Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。
- defineProperty(target, propKey, propDesc)
  - 拦截Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。
- preventExtensions(target)
  - 拦截Object.preventExtensions(proxy)，返回一个布尔值。
- getPrototypeOf(target)
  - 拦截Object.getPrototypeOf(proxy)，返回一个对象。
- isExtensible(target)
  - 拦截Object.isExtensible(proxy)，返回一个布尔值。
- setPrototypeOf(target, proto)
  - 拦截Object.setPrototypeOf(proxy, proto)，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。
- apply(target, object, args)
  - 拦截 Proxy 实例作为函数调用的操作，比如proxy(...args)、proxy.call(object, ...args)、proxy.apply(...)。
- construct(target, args)
  - 拦截 Proxy 实例作为构造函数调用的操作，比如new proxy(...args)。

4-2. **Reflect**

- 与Proxy相同，ES6引入Reflect也是用来**操作对象**的，它将对象里一些明显属于语言内部的方法移植到Reflect对象上，它对某些方法的返回结果进行了修改，使其更合理，并且使用函数的方式实现了Object的命令式操作
- 使用方法
- 常见用法 `Reflect.has(obj, name)` 是 **name in obj** 指令的函数化，用用于判断对象中是否有某一一个属性， 返回值为**布尔值。**

4-3. **使用Proxy与Reflect实现简单的双向数据绑定**

- 获取dom对象
- 设置代理对象
- 配置代理选项
- 添加事件
- 实现双向数据绑定

Proxy实现简单响应】

- Proxy 可以用来自定义对象中的操作。

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    set(target, property, value, receiver) {
      setBind(value, property)
      return Reflect.set(target, property, value)
    },
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver)
    }
  }
  return new Proxy(obj, handler)
}

let obj = { a: 1 }
let p = onWatch(
  obj,
  (v, property) => {
    console.log(`监听到属性${property}改变为${v}`)
  },
  (target, property) => {
    console.log(`'${property}' = ${target[property]}`)
  }
)
p.a = 2 // 控制台输出：监听到属性a改变
p.a // 'a' = 2
```

- 自定义 **set 和 get** 函数的方式，在原本的逻辑中插入了我们的函数逻辑，实现了在对象任何属性进行**读写时发出通知**。
- 当然这是简单版的响应式实现，如果需要实现一个 Vue 中的响应式，需要我们在 get 中收集依赖，在 set 派发更新，之所以 Vue3.0 要使用 Proxy 替换原本的 API 原因在于 **Proxy 无需一层层递归为每个属性添加代理**，一次即可完成以上操作，性能上更好，并且原本的实现有一些数据更新不能监听到，但是 Proxy 可以完美监听到任何方式的数据改变，唯一缺陷可能就是**浏览器的兼容性不好了**。

ES7

1. ES7求幂运算符（**）

```
Math.pow(3, 2) === 3 ** 2    // 9
```

上一篇：三、ES6

下一篇：1.let