02

OPEN ORIENTED

凹 凸 实 验 室

ES6(二)

暖暖

目录

02

目录

02

Symbol

Symbol

02

六种数据类型：

undefined、null、布尔值、
字符串、数值、对象

原始数据类型，第七种数据类型

```
let s = Symbol();
typeof s     // "symbol"
```

Symbol

- 独一无二的值：Symbol函数的参数只是表示对当前Symbol值的描述，即使参数相同也是不相等的。

```
let s1 = Symbol('foo');
let s2 = Symbol('foo');

s1 === s2   // false
```

Symbol

- Symbol值不能与其他类型的值进行运算，会报错。
- Symbol值可以显式转为字符串。
- Symbol值可以转为布尔值
- Symbol值不能转为数值。

```
let s3 = Symbol('foo');

`your symbol is ${s3}` // TypeError
s3.toString()  // "Symbol(foo)"
!!s3     // true
++s3    // TypeError
```

Symbol

- Symbol值作为对象属性名时，不能用点运算符。

```
let s4 = Symbol();
let a = {};

a.s4 = 'Hello!';
a[s4]    // undefined
a['s4']  // "Hello!"

a[s4] = 'Hi!';  // 用中括号
```

Symbol

- Symbol值作为名称的属性，不会被常规方法遍历得到。可以利用这个特性，为对象定义一些非私有的、但又希望只用于内部的方法。

```
let s5 = Symbol('foo');
let b = {};

b.x = 'Hello!';
b[s5] = 'Hi!';

Object.keys(b) // ["x"]
Object.values(b)  // ["Hello!"]

Object.getOwnPropertyNames(b) // ["x"]
Object.getOwnPropertySymbols(b)  // [Symbol(foo)]

Reflect.ownKeys(b) // ["xx", Symbol(foo)]
```

Symbol

02

	Symbol()	Symbol.for()	Symbol.keyFor()
作用	// 等同于 var a = 1,b = 2,c = 3; var [a, b, c] = [1, 2, 3];	在全局环境中供搜索，供于重新使用同一个Symbol值	返回一个已登记的Symbol类型值的key。未登记的Symbol值，返回undefined。

备注	let s1 = Symbol('foo');
	let s2 = Symbol('foo');
	
	s1 === s2 // false
		let s6 = Symbol.for("fooo");
		let s7 = Symbol.for("fooo");
		
		s6 === s7 // true
		
		Symbol.keyFor(s6) // "fooo"
		Symbol.keyFor(s6) // "fooo"	

Symbol

- 十一个内置的Symbol值，指向语言内部使用的方法

Symbol.hasInstance
Symbol.isConcatSpreadable
Symbol.species
Symbol.match
Symbol.replace
Symbol.search
Symbol.split
Symbol.iterator
Symbol.toPrimitive
Symbol.toStringTag
Symbol.unscopables

Symbol

- 十一个内置的Symbol值，指向语言内部使用的方法

Symbol.isConcatSpreadable

```
let obj = {length: 2, 0: 'c', 1: 'd'};
['a', 'b'].concat(obj, 'e') // ['a', 'b', obj, 'e']

obj[Symbol.isConcatSpreadable] = true;
['a', 'b'].concat(obj, 'e') // ['a', 'b', 'c', 'd', 'e']
```

目录

02

Set和Map数据结构

Set和Map数据结构

	Set
作用	Set本身是一个构造函数，用来生成Set数据结构，**类似于数组，但是成员的值都是唯一的。**
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。 • 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。
遍历操作	• keys()：返回一个键名的遍历器 （键名和键值是同一个值） • values()：返回一个键值的遍历器 （默认遍历器生成函数） • entries()：返回一个键值对的遍历器 • forEach()：使用回调函数遍历每个成员

Set和Map数据结构

	Set
作用	Set本身是一个构造函数，用来生成Set数据结构，**类似于数组，但是成员的值都是唯一的。**
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。 • 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。
遍历操作	• keys()：返回一个键名的遍历器 （键名和键值是同一个值） • values()：返回一个键值的遍历器 （默认遍历器生成函数） • entries()：返回一个键值对的遍历器 • forEach()：使用回调函数遍历每个成员

```
// 成员的值都是唯一
let set = new Set([1, 2, 3, 4, 4])
set // Set(4) {1, 2, 3, 4}

// 转成数组
[...set]  // [1, 2, 3, 4]
Array.from(new Set([1, 2, 3, 4, 4]));
// [1, 2, 3, 4]
```

Set和Map数据结构

	Map
作用	构造函数，**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。**
属性和方法	• size属性返回Map结构的成员总数。 • set(key,value)，返回的是Map本身，因此可以采用链式写法。 • get(key) 如果找不到key，返回undefined。 • has(key) 返回一个布尔值，表示某个键是否在Map数据结构中。 • delete(key) 删除某个键，返回true。如果删除失败，返回false。 • clear() 清除所有成员，没有返回值。
遍历操作	• 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（默认遍历器接口）。 • forEach()：遍历Map的所有成员。
注意	Map 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键。

Set和Map数据结构

	Map
作用	构造函数，**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。**
属性和方法	• size属性返回Map结构的成员总数。 • set(key,value)，返回的是Map本身，因此可以采用链式写法。 • get(key) 如果找不到key，返回undefined。 • has(key) 返回一个布尔值，表示某个键是否在Map数据结构中。 • delete(key) 删除某个键，返回true。如果删除失败，返回false。 • clear() 清除所有成员，没有返回值。
遍历操作	• 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（默认遍历器接口）。 • forEach()：遍历Map的所有成员。
注意	Map 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键。

```
// 任何具有 Iterator 接口的数据结构
// 都可以当作Map构造函数的参数
var map = new Map([["name", "张三"],
["title", “Author"]]);
// Map(2) {"name" => "张三", "title" =>
"Author"}

map.size // 2
map.has("name") // true
map.get("name") // “张三"

[...map.values()]  // ["张三", "Author"]
```

Set和Map数据结构

	Map
作用	构造函数，**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。**
属性和方法	• size属性返回Map结构的成员总数。 • set(key,value)，返回的是Map本身，因此可以采用链式写法。 • get(key) 如果找不到key，返回undefined。 • has(key) 返回一个布尔值，表示某个键是否在Map数据结构中。 • delete(key) 删除某个键，返回true。如果删除失败，返回false。 • clear() 清除所有成员，没有返回值。
遍历操作	• 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（默认遍历器接口）。 • forEach()：遍历Map的所有成员。
注意	Map 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键。

```
// Map 的键实际上是跟内存地址绑定
const map = new Map();

map.set(['a'], 555);
map.get(['a']) // undefined
```

Set和Map数据结构

	WeakSet
作用	WeakSet结构**与Set类似，也是不重复的值的集合**
与Set有两个区别	• WeakSet的**成员只能是对象**。 • 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。WeakSet的一个用处，是储存DOM节点，而不用担心这些节点从文档移除时，会引发内存泄漏。
方法	• add(value)：添加一个新成员。 • delete(value)：清除指定成员。 • has(value)：返回一个布尔值，表示某个值是否在WeakSet实例之中。

栗子	const ws = new WeakSet();
	ws.add(1) // 报错
	ws.add([1,2]) // WeakSet {[1, 2]}
	ws.size() // TypeError 没有size属性，没有办法遍历它的成员

Set和Map数据结构

	WeakSet	WeakMap
作用	WeakSet结构**与Set类似，也是不重复的值的集合**	与Map结构基本类似
与Set有两个区别	• WeakSet的**成员只能是对象**。 • 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。WeakSet的一个用处，是储存DOM节点，而不用担心这些节点从文档移除时，会引发内存泄漏。	• 唯一的区别是它**只接受对象作为键名（null除外）**。 • WeakMap **弱引用的只是键名，而不是键值**。键名所指向的对象，不计入垃圾回收机制。WeakMap 应用的典型场合就是 DOM 节点作为键名，监听函数是该键名的键值；一旦 DOM 对象消失，跟它绑定的监听函数也会自动消失。
方法	• add(value)：添加一个新成员。 • delete(value)：清除指定成员。 • has(value)：返回一个布尔值，表示某个值是否在WeakSet实例之中。	• get()、set()、has()、delete()。 • 无遍历操作、size属性、clear()方法。

栗子	const ws = new WeakSet();
	ws.add(1) // 报错
	ws.add([1,2]) // WeakSet {[1, 2]}
	ws.size() // TypeError 没有size属性，没有办法遍历它的成员
		const listener = new WeakMap();
		listener.set(element1, handler1);
		element1.addEventListener('click',
		listener.get(element1), false);

目录

02

Proxy

Proxy

02

Proxy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，因此可以对外界的访问进行过滤和改写。

用法：**var proxy = new Proxy(target, handler)**
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。

Proxy

Proxy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，因此可以对外界的访问进行过滤和改写。

用法：**var proxy = new Proxy(target, handler)**

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。

```
// 拦截对象属性的读取
var target = {time: 1,name: 2,_private: 3};
var proxy = new Proxy(target, {
  get (target, property) {
    return `${property} is ${target[property]}`;
  }
});

proxy.time // "time is 1"
proxy._private // "_private is 3"
```

Proxy

Proxy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，因此可以对外界的访问进行过滤和改写。

用法：**var proxy = new Proxy(target, handler)**

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。

```
var target = {time: 1,name: 2,_private: 3};
var proxy = new Proxy(target, {
  has(target,property) {
     if (property[0] === '_') {
      return false;
    }
    return property in target;
  }
});

'time' in proxy  // true
'_private' in proxy  // false
```

Proxy

虽然 Proxy 可以代理针对目标对象的访问，但它不是目标对象的透明代理，即不做任何拦截的情况下，也无法保证与目标对象的行为一致。

```
// 在 Proxy 代理的情况下，
// 目标对象内部的this关键字指向 Proxy 代理。
const target = new Date();
const handler = {};
const proxy = new Proxy(target, handler);

proxy.getDate();
// Uncaught TypeError: this is not a Date
object.
```

Proxy

02

- **get**(target, propKey [, receiver]) **拦截对象属性的读取**。如果一个属性不可配置（configurable）和不可写（writable），则该属性不能被代理，通过 Proxy 对象访问该属性会报错。
- **set**(target, propKey, value, receiver) **拦截对象属性的设置**，返回一个布尔值。如利用set方法，还可以数据绑定，即每当对象发生变化时，会自动更新DOM；
- **has**(target, propKey) **拦截propKey in proxy的操作**，返回一个布尔值。如果原对象不可配置或者禁止扩展，使用has拦截会报错。
- **deleteProperty**(target, propKey) 拦截delete proxy[propKey]的操作，返回一个布尔值。
- enumerate(target) 拦截for (var x in proxy)，返回一个遍历器。**废弃！**
- **ownKeys**(target) 拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)，返回一个数组。该方法返回对象所有自身的属性，而Object.keys()仅返回对象可遍历的属性。
- **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。

Proxy

02

- **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。只有当Object.isExtensible(proxy)为false（即不可扩展）时，proxy.preventExtensions才能返回true，否则会报错。
- **getPrototypeOf**(target) 拦截Object.getPrototypeOf(proxy)以及其他一些操作（__proto__、isPrototypeOf()、instanceof等），返回一个对象。
- **isExtensible**(target) 拦截Object.isExtensible(proxy)，返回一个布尔值。
- **setPrototypeOf**(target, photo) 拦截Object.setPrototypeOf(proxy, proto)，返回一个布尔值。
- **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, proxy) 拦截Proxy实例作为构造函数调用的操作，比如new proxy(...args)。如果construct方法返回的不是对象，就会抛出错误。

Proxy

Proxy.revocable方法返回一个可取消的Proxy实例。

```
let target = {};
let handler = {};

// 该对象的proxy属性是Proxy实例。
// revoke属性是一个函数，可以取消Proxy实例。
let {proxy, revoke} = Proxy.revocable(target, handler);

proxy.foo = 123;
proxy.foo // 123

// 执行revoke。
revoke();
// 再访问Proxy实例，就会抛出一个错误。
proxy.foo // TypeError: Revoked
```

目录

02

Reflect

Reflect

02

特点：

- 明显属于语言内部的方法，放到Reflect对象上。
- 修改某些Object方法的返回结果，让其变得更合理。
- 让Object操作都变成函数行为。
- Reflect对象的方法与Proxy对象的方法一一对应。不管Proxy怎么修改默认行为，你总可以在Reflect上获取默认行为。

Reflect

特点：

- 明显属于语言内部的方法，放到Reflect对象上。
- 修改某些Object方法的返回结果，让其变得更合理。
- 让Object操作都变成函数行为。
- Reflect对象的方法与Proxy对象的方法一一对应。不管Proxy怎么修改默认行为，你总可以在Reflect上获取默认行为。

Object.defineProperty(obj, name, desc)在无法定义属性时，会抛出一个错误。

=》

Reflect.defineProperty(obj, name, desc)则会返回false。

类似的还有
Reflect.set()、
Reflect.preventExtensions()

Reflect

特点：

- 明显属于语言内部的方法，放到Reflect对象上。
- 修改某些Object方法的返回结果，让其变得更合理。
- 让Object操作都变成函数行为。
- Reflect对象的方法与Proxy对象的方法一一对应。不管Proxy怎么修改默认行为，你总可以在Reflect上获取默认行为。

name in obj和delete obj[name]

=》

Reflect.has(obj, name)和
Reflect.deleteProperty(obj, name)
让它们变成了函数行为。

Reflect

特点：

- 明显属于语言内部的方法，放到Reflect对象上。
- 修改某些Object方法的返回结果，让其变得更合理。
- 让Object操作都变成函数行为。
- Reflect对象的方法与Proxy对象的方法一一对应。不管Proxy怎么修改默认行为，你总可以在Reflect上获取默认行为。

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.enumerate(target)废弃！
Reflect.isExtensible(target)
Reflect.preventExtensions(target)
Reflect.getOwnPropertyDescriptor(target, name)
Reflect.getPrototypeOf(target)
Reflect.setPrototypeOf(target, prototype)

Reflect

02

使用 Proxy 实现观察者模式

使用 Proxy 实现观察者模式

使用 Proxy 写一个观察者模式的最简单实现，即实现**observable**和**observe**这两个函数。

思路：observable函数返回一个原始对象的 Proxy 代理，拦截赋值操作，触发观察者的各个函数。

Reflect

使用 Proxy 写一个观察者模式的最简单实现，即实现**observable**和**observe**这两个函数。

思路：observable函数返回一个原始对象的 Proxy 代理，拦截赋值操作，触发观察者的各个函数。

```
const queuedObservers = new Set();
// 所有观察者函数都放进Set集合里
const observe = fn => queuedObservers.add(fn);
// 返回一个原始对象的 Proxy 代理
const observable = obj => new Proxy(obj, {set});

// Proxy 代理的handler，  拦截赋值操作
function set(target, key, value, receiver) {

  // 触发原始对象的赋值操作
  const result = Reflect.set(target, key, value, receiver);
  // 循环Set集合，  触发观察者
  queuedObservers.forEach(observer => observer());

  return result;
}
```

Reflect


```
const queuedObservers = new Set();
// 所有观察者函数都放进Set集合里
const observe = fn => queuedObservers.add(fn);
// 返回一个原始对象的 Proxy 代理
const observable = obj => new Proxy(obj, {set});

// Proxy 代理的handler， 拦截赋值操作
function set(target, key, value, receiver) {

  // 触发原始对象的赋值操作
  const result = Reflect.set(target, key, value, receiver);
  // 循环Set集合， 触发观察者
  queuedObservers.forEach(observer => observer());

  return result;
}
```

```
// person是观察目标
const person = observable({
  name: '张三',
  age: 20
});

function print() {
  console.log(`${person.name}, ${person.age}`)
}
// print是观察者函数， 放进Set集合里
observe(print);

person.name = '李四';
// 输出
// 李四, 20
```

参考文献

http://es6.ruanyifeng.com/

02

OPEN ORIENTED

凹 凸 实 验 室

THANKS FOR LISTENING

王彩暖