ES6~ES12（三）

王红元 coderwhy

Set的基本使用

- 在ES6之前，我们存储数据的结构主要有两种：数组、对象。
  - 在ES6中新增了另外两种数据结构：Set、Map，以及它们的另外形式WeakSet、WeakMap。
- Set是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**。
  - 创建Set我们需要通过Set构造函数（暂时没有字面量创建的方式）：
- 我们可以发现Set中存放的元素是不会重复的，那么Set有一个非常常用的功能就是给数组去重。

```
const set1 = new Set()
set1.add(10)
set1.add(14)
set1.add(16)
console.log(set1) // Set(3) { 10, 14, 16 }

const set2 = new Set([11, 15, 18, 11])
console.log(set2) // Set(3) { 11, 15, 18 }
```

```
const arr = [10, 20, 10, 44, 78, 44]
const set = new Set(arr)
const newArray1 = [...set]
const newArray2 = Array.from(set)
console.log(newArray1, newArray2)
```

Set的常见方法

- Set常见的属性：
  - size：返回Set中元素的个数；
- Set常用的方法：
  - add(value)：添加某个元素，返回Set对象本身；
  - delete(value)：从set中删除和这个值相等的元素，返回boolean类型；
  - has(value)：判断set中是否存在某个元素，返回boolean类型；
  - clear()：清空set中所有的元素，没有返回值；
  - forEach(callback, [, thisArg])：通过forEach遍历set；
- 另外Set是支持for of的遍历的。

WeakSet使用

- 和Set类似的另外一个数据结构称之为WeakSet，也是内部元素不能重复的数据结构。
- **那么和Set有什么区别呢？**
  - 区别一：WeakSet中只能存放对象类型，不能存放基本数据类型；
  - 区别二：WeakSet对元素的引用是弱引用，如果没有其他引用对某个对象进行引用，那么GC可以对该对象进行回收；

```
const wset = new WeakSet()

// TypeError: Invalid value used in weak set
wset.add(10)
```

- WeakSet常见的方法：
  - 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；

WeakSet的应用

- 注意：WeakSet不能遍历
  - 因为WeakSet只是对对象的弱引用，如果我们遍历获取到其中的元素，那么有可能造成对象不能正常的销毁。
  - 所以存储到WeakSet中的对象是没办法获取的；
- 那么这个东西有什么用呢？
  - 事实上这个问题并不好回答，我们来使用一个Stack Overflow上的答案；

```
const pwset = new WeakSet()
class Person {
  constructor() {
    pwset.add(this)
  }
  running() {
    if(!pwset.has(this)) throw new Error("不能通过其他对象调用running方法")
    console.log("running", this)
  }
}
```

Map的基本使用

- 另外一个新增的数据结构是Map，用于存储映射关系。
- 但是我们可能会想，在之前我们可以使用对象来存储映射关系，他们有什么区别呢？
  - 事实上我们对象存储映射关系只能用字符串（ES6新增了Symbol）作为属性名（key）；
  - 某些情况下我们可能希望通过其他类型作为key，比如对象，这个时候会自动将对象转成字符串来作为key；
- 那么我们就可以使用Map：

```
const obj1 = { name: "why" }
const obj2 = { age: 18 }

const map = new Map()
map.set(obj1, "abc")
map.set(obj2, "cba")
console.log(map.get(obj1))
console.log(map.get(obj2))
```

```
const map = new Map([
  [obj1, "abc"],
  [obj2, "cba"],
  [obj1, "nba"]
])
console.log(map.get(obj1)) // nba
console.log(map.get(obj2)) // cba
```

Map的常用方法

- Map常见的属性：
  - size：返回Map中元素的个数；
- Map常见的方法：
  - set(key, value)：在Map中添加key、value，并且返回整个Map对象；
  - get(key)：根据key获取Map中的value；
  - has(key)：判断是否包括某一个key，返回Boolean类型；
  - delete(key)：根据key删除一个键值对，返回Boolean类型；
  - clear()：清空所有的元素；
  - forEach(callback, [, thisArg])：通过forEach遍历Map；
- Map也可以通过for of进行遍历。

WeakMap的使用

- 和Map类型相似的另外一个数据结构称之为WeakMap，也是以键值对的形式存在的。
- 那么和Map有什么区别呢？
  - 区别一：WeakMap的key只能使用对象，不接受其他的类型作为key；
  - 区别二：WeakMap的key对对象想的引用是弱引用，如果没有其他引用引用这个对象，那么GC可以回收该对象；

```
const weakMap = new WeakMap()
// Invalid value used as weak map key
weakMap.set(1, "abc")
// Invalid value used as weak map key
weakMap.set("aaa", "cba")
```

- WeakMap常见的方法有四个：
  - set(key, value)：在Map中添加key、value，并且返回整个Map对象；
  - get(key)：根据key获取Map中的value；
  - has(key)：判断是否包括某一个key，返回Boolean类型；
  - delete(key)：根据key删除一个键值对，返回Boolean类型；

WeakMap的应用

- 注意：WeakMap也是不能遍历的
  - 因为没有forEach方法，也不支持通过for of的方式进行遍历；
- 那么我们的WeakMap有什么作用呢？

```
// WeakMap({key(对象): value}): key是一个对象，弱引用
const targetMap = new WeakMap();
function getDep(target, key) {
  // 1.根据对象(target)取出对应的Map对象
  let depsMap = targetMap.get(target);
  if (!depsMap) {
    depsMap = new Map();
    targetMap.set(target, depsMap);
  }

  // 2.取出具体的dep对象
  let dep = depsMap.get(key);
  if (!dep) {
    dep = new Dep();
    depsMap.set(key, dep);
  }
  return dep;
}
```

ES7 - Array Includes

- 在ES7之前，如果我们想判断一个数组中是否包含某个元素，需要通过 indexOf 获取结果，并且判断是否为 -1。
- 在ES7中，我们可以通过includes来判断一个数组中是否包含一个指定的元素，根据情况，如果包含则返回 true，否则返回false。

> arr.includes(valueToFind[, fromIndex])

```
if (names.includes("why")) {
  console.log("包含why")
}

if (names.includes("why", 4)) {
  console.log("包含why")
}

console.log(names.indexOf(NaN)) // -1
console.log(names.includes(NaN)) // true
```

ES7 –指数(乘方) exponentiation运算符

- 在ES7之前，计算数字的乘方需要通过 Math.pow 方法来完成。
- 在ES7中，增加了 ** 运算符，可以对数字来计算乘方。

```
const result1 = Math.pow(3, 3)
const result2 = 3 ** 3

console.log(result1, result2)
```

ES8 Object values

- 之前我们可以通过 Object.keys 获取一个对象所有的key，在ES8中提供了 Object.values 来获取所有的value值：

```
const obj = {
  name: "why",
  age: 18,
  height: 1.88
}

console.log(Object.values(obj)) // [ 'why', 18, 1.88 ]

// 如果传入一个字符串
console.log(Object.values("abc")) // [ 'a', 'b', 'c' ]
```

ES8 Object entries

- 通过Object.entries 可以获取到一个数组，数组中会存放可枚举属性的键值对数组。

```
const obj = {
  name: "why",
  age: 18,
  height: 1.88
}

console.log(Object.entries(obj)) // [ [ 'name', 'why' ], [ 'age', 18 ], [ 'height', 1.88 ] ]
for (const entry of Object.entries(obj)) {
  const [key, value] = entry
  console.log(key, value)
}

// 如果是一个数组
console.log(Object.entries(["abc", "cba", "nba"])) // [ [ '0', 'abc' ], [ '1', 'cba' ], [ '2', 'nba' ] ]

// 如果是一个字符串
console.log(Object.entries("abc")) // [ [ '0', 'a' ], [ '1', 'b' ], [ '2', 'c' ] ]
```

ES8 - String Padding

- 某些字符串我们需要对其进行前后的填充，来实现某种格式化效果，ES8中增加了 padStart 和 padEnd 方法，分别是对字符串的首尾进行填充的。

```
const message = "Hello World"

console.log(message.padStart(15, "a")) // aaaaHello World
console.log(message.padEnd(15, "b")) // Hello Worldbbbb
```

- 我们简单具一个应用场景：比如需要对身份证、银行卡的前面位数进行隐藏：

```
const cardNumber = "3242523524256245223879"
const lastFourNumber = cardNumber.slice(-4)
const finalCardNumber = lastFourNumber.padStart(cardNumber.length, "*")
console.log(finalCardNumber) // ******************3879
```

ES8 - Trailing Commas

- 在ES8中，我们允许在函数定义和调用时多加一个逗号：

```
function foo(a, b,) {
  console.log(a, b)
}

foo(10, 20,)
```

ES8 - Object Descriptors

- ES8中增加了另一个对对象的操作是 Object.getOwnPropertyDescriptors ，这个在之前已经讲过了，这里不再重复。

ES9新增知识点

- Async iterators：后续迭代器讲解
- Object spread operators：前面讲过了
- Promise finally：后续讲Promise讲解

ES10 - flat flatMap

- flat() 方法会按照一个可指定的深度递归遍历数组，并将所有元素与遍历到的子数组中的元素合并为一个新数组返回。
- flatMap() 方法首先使用映射函数映射每个元素，然后将结果压缩成一个新数组。
  - 注意一：flatMap是先进行map操作，再做flat的操作；
  - 注意二：flatMap中的flat相当于深度为1；

```
const nums = [10, 20, [5, 8], [[2, 3], [9, 22]], 100]

const newNums1 = nums.flat(1)
const newNums2 = nums.flat(2)

// [ 10, 20, 5, 8, [ 2, 3 ], [ 9, 22 ], 100 ]
console.log(newNums1)
// [ 10, 20, 5, 8, 2, 3, 9, 22, 100 ]
console.log(newNums2)
```

```
const messages = ["Hello World", "你好啊 李银河", "my name is why"]

const newMessages = messages.flatMap(item => {
  return item.split(" ")
})
console.log(newMessages)
```

ES10 - Object fromEntries

- 在前面，我们可以通过 Object.entries 将一个对象转换成 entries，那么如果我们有一个entries了，如何将其转换成对象呢？
  - ES10提供了 Object.formEntries来完成转换：
- 那么这个方法有什么应用场景呢？

```
const obj = {
  name: "why",
  age: 18,
  height: 1.88
}

const entries = Object.entries(obj)
console.log(entries)

const info = Object.fromEntries(entries)
console.log(info)
```

```
const paramsString = 'name=why&age=18&height=1.88'
const searchParams = new URLSearchParams(paramsString)
for (const param of searchParams) {
  console.log(param)
}
const searchObj = Object.fromEntries(searchParams)
console.log(searchObj)
```

ES10 - trimStart trimEnd

- 去除一个字符串首尾的空格，我们可以通过trim方法，如果单独去除前面或者后面呢？
  - ES10中给我们提供了trimStart和trimEnd；

```
const message = "   Hello World   "
console.log(message.trim())
console.log(message.trimStart())
console.log(message.trimEnd())
```

ES10 其他知识点

- Symbol description：已经讲过了
- Optional catch binding：后面讲解try cach讲解

ES11 - BigInt

- 在早期的JavaScript中，我们不能正确的表示过大的数字：
  - 大于MAX_SAFE_INTEGER的数值，表示的可能是不正确的。

```
const maxInt = Number.MAX_SAFE_INTEGER
console.log(maxInt)

// 大于MAX_SAFE_INTEGER值的一些数值,无法正确的表示
console.log(maxInt + 1) // 9007199254740992
console.log(maxInt + 2) // 9007199254740992
```

- 那么ES11中，引入了新的数据类型BigInt，用于表示大的整数：
  - BitInt的表示方法是在数值的后面加上n

```
const bigInt = 9007199254740991n
console.log(bigInt + 1n)
console.log(bigInt + 2n)
```

ES11 - Nullish Coalescing Operator

- ES11，Nullish Coalescing Operator增加了空值合并操作符：

```
const foo = ""

const result1 = foo || "默认值"
const result2 = foo ?? "默认值"
console.log(result1) // 默认值
console.log(result2) // ""
```

ES11 - Optional Chaining

■ 可选链也是ES11中新增一个特性，主要作用是让我们的代码在进行null和undefined判断时更加清晰和简洁：

```
const obj = {
  friend: {
    girlFriend: {
      name: "lucy"
    }
  }
}

if (obj.friend && obj.friend.girlFriend) {
  console.log(obj.friend.girlFriend.name)
}

// 可选链的方式
console.log(obj.friend?.girlFriend?.name)
```

ES11 - Global This

- 在之前我们希望获取JavaScript环境的全局对象，不同的环境获取的方式是不一样的
  - 比如在浏览器中可以通过this、window来获取；
  - 比如在Node中我们需要通过global来获取；
- 那么在ES11中对获取全局对象进行了统一的规范：globalThis

```
console.log(globalThis)
console.log(this) // 浏览器上
console.log(global) // Node中
```

ES11 - for..in标准化

- 在ES11之前，虽然很多浏览器支持for...in来遍历对象类型，但是并没有被ECMA标准化。
- 在ES11中，对其进行了标准化，for...in是用于遍历对象的key的：

```
const obj = {
  name: "why",
  age: 18,
  height: 1.88
}

for (const key in obj) {
  console.log(key)
}
```

ES11 其他知识点

- **Dynamic Import**：后续ES Module模块化中讲解。
- **Promise.allSettled**：后续讲Promise的时候讲解。
- **import meta**：后续ES Module模块化中讲解。

ES12 - FinalizationRegistry

- FinalizationRegistry 对象可以让你在对象被垃圾回收时请求一个回调。
  - FinalizationRegistry 提供了这样的一种方法：当一个在注册表中注册的对象被回收时，请求在某个时间点上调用一个清理回调。（清理回调有时被称为 finalizer ）；
  - 你可以通过调用register方法，注册任何你想要清理回调的对象，传入该对象和所含的值;

```
let obj = { name: "why" }

const registry = new FinalizationRegistry(value => {
  console.log("对象被销毁了", value)
})

registry.register(obj, "obj")

obj = null
```

ES12 - WeakRefs

- 如果我们默认将一个对象赋值给另外一个引用，那么这个引用是一个强引用：
  - 如果我们希望是一个弱引用的话，可以使用WeakRef；

```
let obj = { name: "why" }
let info = new WeakRef(obj)
```

ES12 - logical assignment operators

```
// 1.逻辑或运算符
let message = ""
// message = message || "hello world"
message ||= "Hello World"
console.log(message)

let obj = {
  name: "why"
}

// 2.逻辑与操作符
// obj = obj && obj.foo()
obj &&= obj.name
console.log(obj)

// 3.逻辑空运算符
let foo = null
foo ??= "默认值"
console.log(foo)
```

ES12其他知识点

- Numeric Separator：讲过了；
- String.replaceAll：字符串替换；