ES5和ES6

- 我们所说的 ES5 和 ES6 其实就是在 js 语法的发展过程中的一个版本而已
- 比如我们使用的微信
  - 最早的版本是没有支付功能的
  - 随着时间的流逝，后来出现了一个版本，这个版本里面有支付功能了
- ECMAScript 就是 js 的语法
  - 以前的版本没有某些功能
  - 在 ES5 这个版本的时候增加了一些功能
  - 在 ES6 这个版本的时候增加了一些功能
- 因为浏览器是浏览器厂商生产的
  - ECMAScript 发布了新的功能以后，浏览器厂商需要让自己的浏览器支持这些功能
  - 这个过程是需要时间的
  - 所以到现在，基本上大部分浏览器都可以比较完善的支持了
  - 只不过有些浏览器还是不能全部支持
  - 这就出现了兼容性问题
  - 所以我们写代码的时候就要考虑哪些方法是 ES5 或者 ES6 的，看看是不是浏览器都支持

ES5 增加的数组常用方法

数组方法之 forEach

- `forEach` 用于遍历数组，和 for 循环遍历数组一个道理
- 语法： `数组.forEach(function (item, index, arr) {})`

```
var arr = ['a', 'b', 'c']
// forEach 就是将数组循环遍历，数组中有多少项，那么这个函数就执行多少回
arr.forEach(function (item, index, arr) {
 // 在这个函数内部
 // item 就是数组中的每一项
 // index 就是每一项对应的索引
 // arr 就是原始数组
 console.log(item)
 console.log(index)
 console.log(arr)
})
```

  - 上面的代码就等价于

```
var arr = ['a', 'b', 'c']
```

```
var arr = ['a', 'b', 'c']
for (var i = 0; i < arr.length; i++) {
 fn(arr[i], i, arr)
}
function fn(item, index, arr) {
 console.log(item)
 console.log(index)
 console.log(arr)
}
```

数组方法之 map

- `map` 用于遍历数组，和 forEach 基本一致，只不过是有一个返回值
- 语法： `数组.map(function (item, index, arr) {})`
- 返回值： 一个新的数组

```
var arr = ['a', 'b', 'c']
// forEach 就是将数组循环遍历，数组中有多少项，那么这个函数就执行多少回
var newArr = arr.map(function (item, index, arr) {
 // 函数里面的三个参数和 forEach 一样
 // 我们可以在这里操作数组中的每一项，
 // return 操作后的每一项
 return item + '11'
})
console.log(newArr) // ["a11", "b11", "c11"]
```

- 返回值就是我们每次对数组的操作
- 等价于

```
var arr = ['a', 'b', 'c']
var newArr = []
for (var i = 0; i < arr.length; i++) {
 newArr.push(fn(arr[i], i, arr))
}
function fn(item, index, arr) {
 return item + '11'
}
console.log(newArr)
```

数组方法之 filter

- `filter` ： 是将数组遍历一遍，按照我们的要求把数数组中符合的内容过滤出来

- 语法： `数组.filter(function (item, index, arr) {})`
- 返回值： 根据我们的条件过滤出来的新数组

```
var arr = [1, 2, 3, 4, 5]
var newArr = arr.filter(function (item, index, arr) {
 // 函数内部的三个参数和 forEach 一样
 // 我们把我们的条件 return 出去
 return item > 2
})
console.log(newArr) // [3, 4, 5]
```

  - 新数组里面全都是大于 2 的数字
  - 等价于

```
var arr = [1, 2, 3, 4, 5]
var newArr = []
for (var i = 0; i < arr.length; i++) {
 if (fn(arr[i], i, arr)) {
 newArr.push(arr[i])
 }
}
function fn(item, index, arr) {
 return item > 2
}
console.log(newArr)
```

JSON 方法

- `json` 是一种特殊的字符串个是，本质是一个字符串

```
var jsonObj = '{ "name": "Jack", "age": 18, "gender": "男" }'
var jsonArr = '[{ "name": "Jack", "age": 18, "gender": "男" }, { "name": "Jack", "age": 18, "gender": "男" }, { "name": "Jack", "age": 18, "gender": "男" }]'
```

- 就是对象内部的 `key` 和 `value` 都用双引号包裹的字符串（必须是双引号）

JSON的两个方法

- 我们有两个方法可以使用 **JSON.parse**
-
- `json.stringify` 是将 js 的对象或者数组转换成为 json 格式的字符串

JSON.parse

- `JSON.parse` 是将 json 格式的字符串转换为 js 的对象或者数组

```
var jsonObj = '{ "name": "Jack", "age": 18, "gender": "男" }'
var jsonArr = '[{ "name": "Jack", "age": 18, "gender": "男" }, { "name": "Jack", "age": 18, "gender": "男" }, { "name": "Jack", "age": 18, "gender": "男" }]'

var obj = JSON.parse(jsonStr)
var arr = JSON.parse(jsonArr)

console.log(obj)
console.log(arr)
```

  - `obj` 就是我们 js 的对象
  - `arr` 就是我们 js 的数组

JSON.stringify

- `JSON.parse` 是将 json 格式的字符串转换为 js 的对象或者数组

```
var obj = {
 name: 'Jack',
 age: 18,
 gender: '男'
}
var arr = [
 {
 name: 'Jack',
 age: 18,
 gender: '男'
 },
 {
 name: 'Jack',
 age: 18,
 gender: '男'
 },
 {
 name: 'Jack',
 age: 18,
 gender: '男'
```

```
  }
]

var jsonObj = JSON.stringify(obj)
var jsonArr = JSON.stringify(arr)

console.log(jsonObj)
console.log(jsonArr)
```

- `jsonObj` 就是 json 格式的对象字符串
- `jsonArr` 就是 json 格式的数组字符串

this 关键字

- 每一个函数内部都有一个关键字是 `this`
- 可以让我们直接使用的
- 重点： **函数内部的 this 只和函数的调用方式有关系，和函数的定义方式没有关系**
- 函数内部的 this 指向谁，取决于函数的调用方式
  - 全局定义的函数直接调用， `this => window`

```
function fn() {
 console.log(this)
}
fn()
// 此时 this 指向 window
```

  - 对象内部的方法调用， `this => 调用者`

```
var obj = {
 fn: function () {
 console.log(this)
 }
}
obj.fn()
// 此时 this 指向 obj
```

  - 定时器的处理函数， `this => window`

```
setTimeout(function () {
 console.log(this)
```

```
}, 0)
// 此时定时器处理函数里面的 this 指向 window
```

- 事件处理函数，`this => 事件源`

```
div.onclick = function () {
 console.log(this)
}
// 当你点击 div 的时候，this 指向 div
```

- 自调用函数，`this => window`

```
(function () {
 console.log(this)
})()
// 此时 this 指向 window
```

call 和 apply 和 bind

- 刚才我们说过的都是函数的基本调用方式里面的 this 指向
- 我们还有三个可以忽略函数本身的 this 指向转而指向别的地方
- 这三个方法就是 call / apply / bind
- 是强行改变 this 指向的方法

call

- `call` 方法是附加在函数调用后面使用，可以忽略函数本身的 this 指向
- 语法：`函数名.call(要改变的 this 指向，要给函数传递的参数1，要给函数传递的参数2，...)`

```
var obj = { name: 'Jack' }
function fn(a, b) {
  console.log(this)
  console.log(a)
  console.log(b)
}
fn(1, 2)
fn.call(obj, 1, 2)
```

- `fn()` 的时候，函数内部的 this 指向 window
- `fn.call(obj, 1, 2)` 的时候，函数内部的 this 就指向了 obj 这个对象
- 使用 call 方法的时候
  - 会立即执行函数
  - 第一个参数是你要改变的函数内部的 this 指向

    - 第二个参数开始，依次是向函数传递参数

apply

- `apply` 方法是附加在函数调用后面使用，可以忽略函数本身的 this 指向
- 语法：`函数名.apply(要改变的 this 指向，[要给函数传递的参数1， 要给函数传递的参数2，...])`

```
var obj = { name: 'Jack' }
function fn(a, b) {
  console.log(this)
  console.log(a)
  console.log(b)
}
fn(1, 2)
fn.call(obj, [1, 2])
```

  - `fn()` 的时候，函数内部的 this 指向 window
  - `fn.apply(obj, [1, 2])` 的时候，函数内部的 this 就指向了 obj 这个对象
  - 使用 apply 方法的时候
    - 会立即执行函数
    - 第一个参数是你要改变的函数内部的 this 指向
    - 第二个参数是一个 **数组**，数组里面的每一项依次是向函数传递的参数

bind

- `bind` 方法是附加在函数调用后面使用，可以忽略函数本身的 this 指向
- 和 call / apply 有一些不一样，就是不会立即执行函数，而是返回一个已经改变了 this 指向的函数
- 语法：`var newFn = 函数名.bind(要改变的 this 指向); newFn(传递参数)`

```
var obj = { name: 'Jack' }
function fn(a, b) {
  console.log(this)
  console.log(a)
  console.log(b)
}
fn(1, 2)
var newFn = fn.bind(obj)
newFn(1, 2)
```

  - bind 调用的时候，不会执行 fn 这个函数，而是返回一个新的函数
  - 这个新的函数就是一个改变了 this 指向以后的 fn 函数

- `fn(1, 2)` 的时候 this 指向 window
- `newFn(1, 2)` 的时候执行的是一个和 fn 一摸一样的函数，只不过里面的 this 指向改成了 obj

ES6新增的内容

- 之前的都是 ES5 的内容
- 接下来我们聊一下 ES6 的内容

let 和 const 关键字

- 我们以前都是使用 `var` 关键字来声明变量的
- 在 ES6 的时候，多了两个关键字 `let` 和 `const`，也是用来声明变量的
- 只不过和 var 有一些区别

i. `let` 和 `const` 不允许重复声明变量

```
// 使用 var 的时候重复声明变量是没问题的，只不过就是后面会把前面覆盖掉
var num = 100
var num = 200
// 使用 let 重复声明变量的时候就会报错了
let num = 100
let num = 200 // 这里就会报错了
// 使用 const 重复声明变量的时候就会报错
const num = 100
const num = 200 // 这里就会报错了
```

ii. `let` 和 `const` 声明的变量不会在预解析的时候解析（也就是没有变量提升）

```
// 因为预解析（变量提升）的原因，在前面是有这个变量的，只不过没有赋值
console.log(num) // undefined
var num = 100
// 因为 let 不会进行预解析（变量提升），所以直接报错了
console.log(num) // undefined
let num = 100
// 因为 const 不会进行预解析（变量提升），所以直接报错了
console.log(num) // undefined
const num = 100
```

iii. `let` 和 `const` 声明的变量会被所有代码块限制作用范围

```
// var 声明的变量只有函数能限制其作用域，其他的不能限制
if (true) {
 var num = 100
}
console.log(num) // 100
// let 声明的变量，除了函数可以限制，所有的代码块都可以限制其作用域（if/while/for/...）
if (true) {
 let num = 100
 console.log(num) // 100
}
console.log(num) // 报错
// const 声明的变量，除了函数可以限制，所有的代码块都可以限制其作用域（if/while/for/...）
if (true) {
 const num = 100
 console.log(num) // 100
}
console.log(num) // 报错
```

- `let` 和 `const` 的区别

  i. `let` 声明的变量的值可以改变，`const` 声明的变量的值不可以改变

```
let num = 100
num = 200
console.log(num) // 200
const num = 100
num = 200 // 这里就会报错了，因为 const 声明的变量值不可以改变（我们也叫做常量）
```

  ii. `let` 声明的时候可以不赋值，`const` 声明的时候必须赋值

```
let num
num = 100
console.log(num) // 100
const num // 这里就会报错了，因为 const 声明的时候必须赋值
```

箭头函数

- 箭头函数是 ES6 里面一个简写函数的语法方式

- 重点： **箭头函数只能简写函数表达式，不能简写声明式函数**

```
function fn() {} // 不能简写
const fun = function () {} // 可以简写
const obj = {
 fn: function () {} // 可以简写
}
```

- 语法： （函数的行参）=> { 函数体内要执行的代码 }

```
const fn = function (a, b) {
 console.log(a)
 console.log(b)
}
// 可以使用箭头函数写成
const fun = (a, b) => {
 console.log(a)
 console.log(b)
}
const obj = {
 fn: function (a, b) {
 console.log(a)
 console.log(b)
 }
}
// 可以使用箭头函数写成
const obj2 = {
 fn: (a, b) => {
 console.log(a)
 console.log(b)
 }
}
```

箭头函数的特殊性

- 箭头函数内部没有 this，箭头函数的 this 是上下文的 this

```
// 在箭头函数定义的位置往上数，这一行是可以打印出 this 的
// 因为这里的 this 是 window
// 所以箭头函数内部的 this 就是 window
const obj = {
 fn: function () {
```

```
  console.log(this)
  },
  // 这个位置是箭头函数的上一行，但是不能打印出 this
  fun: () => {
  // 箭头函数内部的 this 是书写箭头函数的上一行一个可以打印出 this 的位置
  console.log(this)
  }
 }

 obj.fn()
 obj.fun()
```

- 按照我们之前的 this 指向来判断，两个都应该指向 obj
- 但是 fun 因为是箭头函数，所以 this 不指向 obj，而是指向 fun 的外层，就是 window

- 箭头函数内部没有 `arguments` 这个参数集合

```
const obj = {
 fn: function () {
 console.log(arguments)
 },
 fun: () => {
 console.log(arguments)
 }
}
obj.fn(1, 2, 3) // 会打印一个伪数组 [1, 2, 3]
obj.fun(1, 2, 3) // 会直接报错
```

- 函数的行参只有一个的时候可以不写 `()` 其余情况必须写

```
const obj = {
 fn: () => {
 console.log('没有参数，必须写小括号')
 },
 fn2: a => {
 console.log('一个行参，可以不写小括号')
 },
 fn3: (a, b) => {
 console.log('两个或两个以上参数，必须写小括号')
 }
}
```

- 函数体只有一行代码的时候，可以不写 `{}` ，并且会自动 return

```
const obj = {
 fn: a => {
 return a + 10
 },
 fun: a => a + 10
}

console.log(fn(10)) // 20
console.log(fun(10)) // 20
```

函数传递参数的时候的默认值

- 我们在定义函数的时候，有的时候需要一个默认值出现
- 就是当我不传递参数的时候，使用默认值，传递参数了就使用传递的参数

```
function fn(a) {
 a = a || 10
 console.log(a)
}
fn() // 不传递参数的时候，函数内部的 a 就是 10
fn(20) // 传递了参数 20 的时候，函数内部的 a 就是 20
```

  - 在 ES6 中我们可以直接把默认值写在函数的行参位置

```
function fn(a = 10) {
 console.log(a)
}
fn() // 不传递参数的时候，函数内部的 a 就是 10
fn(20) // 传递了参数 20 的时候，函数内部的 a 就是 20
```

  - 这个默认值的方式箭头函数也可以使用

```
const fn = (a = 10) => {
 console.log(a)
}
fn() // 不传递参数的时候，函数内部的 a 就是 10
fn(20) // 传递了参数 20 的时候，函数内部的 a 就是 20
```

  - 注意： **箭头函数如果你需要使用默认值的话，那么一个参数的时候也需要写 ()**

解构赋值

- 解构赋值，就是快速的从对象或者数组中取出成员的一个语法方式

解构对象

解构对象

- 快速的从对象中获取成员

```
// ES5 的方法向得到对象中的成员
const obj = {
 name: 'Jack',
 age: 18,
 gender: '男'
}

let name = obj.name
let age = obj.age
let gender = obj.gender
// 解构赋值的方式从对象中获取成员
const obj = {
 name: 'Jack',
 age: 18,
 gender: '男'
}

// 前面的 {} 表示我要从 obj 这个对象中获取成员了
// name age gender 都得是 obj 中有的成员
// obj 必须是一个对象
let { name, age, gender } = obj
```

解构数组

- 快速的从数组中获取成员

```
// ES5 的方式从数组中获取成员
const arr = ['Jack', 'Rose', 'Tom']
let a = arr[0]
let b = arr[1]
let c = arr[2]
// 使用解构赋值的方式从数组中获取成员
const arr = ['Jack', 'Rose', 'Tom']

// 前面的 [] 表示要从 arr 这个数组中获取成员了
// a b c 分别对应这数组中的索引 0 1 2
// arr 必须是一个数组
let [a, b, c] = arr
```

注意

- `{}` 是专门解构对象使用的
- `[]` 是专门解构数组使用的
- 不能混用

模版字符串

- ES5 中我们表示字符串的时候使用 `''` 或者 `""`
- 在 ES6 中，我们还有一个东西可以表示字符串，就是 ``（反引号）

```
let str = `hello world`
console.log(typeof str) // string
```

- 和单引号好友双引号的区别

i. 反引号可以换行书写

```
// 这个单引号或者双引号不能换行，换行就会报错了
let str = 'hello world'

// 下面这个就报错了
let str2 = 'hello
world'
let str = `
 hello
 world
`

console.log(str) // 是可以使用的
```

ii. 反引号可以直接在字符串里面拼接变量

```
// ES5 需要字符串拼接变量的时候
let num = 100
let str = 'hello' + num + 'world' + num
console.log(str) // hello100world100

// 直接写在字符串里面不好使
let str2 = 'hellonumworldnum'
console.log(str2) // hellonumworldnum
```

```
// 模版字符串拼接变量
let num = 100
let str = `hello${num}world${num}`
console.log(str) // hello100world100
```

- 在 `` 里面的 `${}` 就是用来书写变量的位置

展开运算符

- ES6 里面号新添加了一个运算符 `...` ，叫做展开运算符
- 作用是把数组展开

```
let arr = [1, 2, 3, 4, 5]
console.log(...arr) // 1 2 3 4 5
```

- 合并数组的时候可以使用

```
let arr = [1, 2, 3, 4]
let arr2 = [...arr, 5]
console.log(arr2)
```

- 也可以合并对象使用

```
let obj = {
 name: 'Jack',
 age: 18
}
let obj2 = {
 ...obj,
 gender: '男'
}
console.log(obj2)
```

- 在函数传递参数的时候也可以使用

```
let arr = [1, 2, 3]
function fn(a, b, c) {
 console.log(a)
 console.log(b)
 console.log(c)
}
fn(...arr)
// 等价于 fn(1, 2, 3)
```

Map 和 Set

- Map 和 Set 是 ES6 新增的两个数据类型
- 都是属于内置构造函数
- 使用 new 的方式来实例化使用

Set

- 使用方式就是和 new 连用

```
const s = new Set()
console.log(s)

/*
 Set(0) {}
 size: (...)
 __proto__: Set
 [[Entries]]: Array(0)
 length: 0
*/
```

- 就是一个数据集合
- 我们可以在 new 的时候直接向内部添加数据

```
// 实例化的时候直接添加数据要以数组的形式添加
const s = new Set([1, 2, 3, {}, function () {}, true, 'hwllo'])
console.log(s)

/*
 Set(7) {1, 2, 3, {…}, ƒ, …}
 size: (...)
 __proto__: Set
 [[Entries]]: Array(7)
 0: 1
 1: 2
 2: 3
 3: Object
 4: function () {}
```

```
4: function () {}
5: true
6: "hwllo"
length: 7
*/
```

- 看上去是一个类似数组的数据结构，但是不是，就是 **Set 数据结构**

常用方法和属性

- `size` ： 用来获取该数据结构中有多少数据的

```
const s = new Set([1, 2, 3, {}, function () {}, true, 'hwllo'])
console.log(s.size) // 7
```

  - 看上去是一个和数组数据类型差不多的数据结构，而且我们也看到了 length 属性
  - 但是不能使用，想要获取该数据类型中的成员数量，**需要使用 size 属性**

- `add` ：用来向该数据类型中追加数据

```
const s = new Set()
s.add(0)
s.add({})
s.add(function () {})
console.log(s.size) // 3
```

  - 这个方法就是向该数据类型中追加数据使用的

- `delete` ：是删除该数据结构中的某一个数据

```
const s = new Set()
s.add(0)
s.add({})
s.add(function () {})

s.delete(0)

console.log(s.size) // 2
```

- `clear` ： 清空数据结构中的所有数据

```
const s = new Set()
s.add(0)
s.add({})
s.add(function () {})
```

```
5.
6.  s.clear()
7.
8.  console.log(s.size) // 0
```

- `has` ： 查询数据解构中有没有某一个数据

```
1.  const s = new Set()
2.  s.add(0)
3.  s.add({})
4.  s.add(function () {})
5.
6.  console.log(s.has(0)) // true
```

- `forEach` ：用来遍历 Set 数据结构的方法

```
1.  const s = new Set()
2.  s.add(0)
3.  s.add({})
4.  s.add(function () {})
5.
6.  s.forEach(item => {
7.    console.log(item) // 0 {} function () {}
8.  })
```

- 方法介绍的差不多了，有一个问题出现了，那就是
- 我们的方法要么是添加，要么是删除，要么是查询，没有获取
- 因为要获取 Set 结构里面的数据需要借助一个 `...` 展开运算符
- 把他里面的东西都放到一个数组里面去，然后再获取

```
1.  const s = new Set([1, 2, 3, 4, 5, 6])
2.  const a = [...s]
3.  console.log(a) // (6) [1, 2, 3, 4, 5, 6]
4.
5.  console.log(a[0]) // 1
6.  console.log([...s][0]) // 1
```

- 又一个问题出现了，new 的时候需要以数组的形式传递
- 然后获取的时候又要转成数组的形式获取
- 那么我为什么不一开始就定义数组，要这个 Set 数据类型干什么

- 那么我为什么不一开始就定义数组，要这个 Set 数据类型干什么
- 这就不得不提到一个 Set 的特点
- **Set 不允许存储重复的数据**

```
const s = new Set([1, 2, 3])

s.add(4) // 此时 size 是 4
s.add(1) // 此时 size 是 4
s.add(2) // 此时 size 是 4
s.add(3) // 此时 size 是 4
```

Map

- 也是要和 new 连用
- 是一个数据集合，是一个很类似于 对象 的数据集合

```
const m = new Map()
console.log(m)

/*
 Map(0) {}
 size: (...)
 __proto__: Map
 [[Entries]]: Array(0)
 length: 0
*/
```

- 我们的对象中不管存储什么，key 一定是一个字符串类型
- 但是再 Map 里面，我们的 key 可以为任意数据类型
- 我们也管 Map 叫做 （值 = 值 的数据类型）

```
const m = new Map([[{}, {}], [function () {}, function () {}], [true, 1]])
console.log(m)

/*
 Map(3) {{…} => {…}, ƒ => ƒ, true => 1}
 size: (...)
 __proto__: Map
```

```
  [[Entries]]: Array(3)
  0: {Object => Object}
  key: {}
  value: {}
  1: {function () {} => function () {}}
  key: ƒ ()
  value: ƒ ()
  2: {true => 1}
  key: true
  value: 1
  length: 3

*/
```

常用方法和属性

- `size` ： 用来获取该数据类型中数据的个数

```
const m = new Map([[{}, {}], [function () {}, function () {}], [true, 1]])
console.log(m.size) // 3
```

- `delete` ：用来删除该数据集合中的某一个数据

```
const m = new Map([[{}, {}], [function () {}, function () {}], [true, 1]])
m.delete(true)

console.log(m.size) // 2
```

- `set` ：用来向该数据集合中添加数据使用

```
const m = new Map()
m.set({ name: 'Jack' }, { age: 18 })
console.log(m.size) // 1
```

- `get` ：用来获取该数据集合中的某一个数据

```
const m = new Map()

m.set({ name: 'Jack' }, { age: 18 })
m.set(true, function () {})
console.log(m.get(true)) // function () {}
```

- `clear` ：清除数据集合中的所有数据

```
const m = new Map()

m.set({ name: 'Jack' }, { age: 18 })
m.set(true, function () {})

m.clear()

console.log(m.size) // 0
```

- `has` ： 用来判断数据集合中是否存在某一个数据

```
const m = new Map()

m.set({ name: 'Jack' }, { age: 18 })
m.set(true, function () {})

console.log(m.has(true)) // true
```