ES6 知识点及常考面试题

本章节我们将来学习 ES6 部分的内容。

var、let 及 const 区别

> 涉及面试题：什么是提升？什么是暂时性死区？var、let 及 const 区别？

对于这个问题，我们应该先来了解提升（hoisting）这个概念。

```
console.log(a) // undefined
var a = 1
```

从上述代码中我们可以发现，虽然变量还没有被声明，但是我们却可以使用这个未被声明的变量，这种情况就叫做提升，并且提升的是声明。

对于这种情况，我们可以把代码这样来看

```
var a
console.log(a) // undefined
a = 1
```

接下来我们再来看一个例子

```
var a = 10
var a
console.log(a)
```

对于这个例子，如果你认为打印的值为 `undefined` 那么就错了，答案应该是 `10`，对于这种情况，我们这样来看代码

```
var a
var a
```

```
a = 10
console.log(a)
```

到这里为止，我们已经了解了 `var` 声明的变量会发生提升的情况，其实不仅变量会提升函数也会被提升。

```
console.log(a) // ƒ a() {}
function a() {}
var a = 1
```

对于上述代码，打印结果会是 `ƒ a() {}`，即使变量声明在函数之后，这也说明了函数会被提升，并且优先于变量提升。

说完了这些，想必大家也知道 `var` 存在的问题了，使用 `var` 声明的变量会被提升到作用域的顶部，接下来我们再来看 `let` 和 `const` 。

我们先来看一个例子：

```
var a = 1
let b = 1
const c = 1
console.log(window.b) // undefined
console.log(window. c) // undefined

function test(){
  console.log(a)
  let a
}
test()
```

首先在全局作用域下使用 `let` 和 `const` 声明变量，变量并不会被挂载到 `window` 上，这一点就和 `var` 声明有了区别。

再者当我们在声明 `a` 之前如果使用了 `a`，就会出现报错的情况

```

```

你可能会认为这里也出现了提升的情况，但是因为某些原因导致不能访问。

首先报错的原因是因为存在暂时性死区，我们不能在声明前就使用变量，这也是 `let` 和 `const` 优于 `var` 的一点。然后这里你认为的提升和 `var` 的提升是有区别的，虽然变量在编译的环节中被告知在这块作用域中可以访问，但是访问是受限制的。

那么到这里，想必大家也都明白 `var`、`let` 及 `const` 区别了，不知道你是否会有这么一个疑问，为什么要存在提升这个事情呢，其实提升存在的根本原因就是为了解决函数间互相调用的情况

```
function test1() {
    test2()
}
function test2() {
    test1()
}
test1()
```

假如不存在提升这个情况，那么就实现不了上述的代码，因为不可能存在 `test1` 在 `test2` 前面然后 `test2` 又在 `test1` 前面。

那么最后我们总结下这小节的内容：

- 函数提升优先于变量提升，函数提升会把整个函数挪到作用域顶部，变量提升只会把声明挪到作用域顶部
- `var` 存在提升，我们能在声明之前使用。`let`、`const` 因为暂时性死区的原因，不能在声明前使用
- `var` 在全局作用域下声明变量会导致变量挂载在 `window` 上，其他两者不会
- `let` 和 `const` 作用基本一致，但是后者声明的变量不能再次赋值

原型继承和 Class 继承

> 涉及面试题：原型如何实现继承？Class 如何实现继承？Class 本质是什么？

首先先来讲下 `class`，其实在 JS 中并不存在类，`class` 只是语法糖，本质还是函数。

```
class Person {}
Person instanceof Function // true
```

在上一章节中我们讲解了原型的知识点，在这一小节中我们将会分别使用原型和 `class` 的方式来实现继承。

组合继承

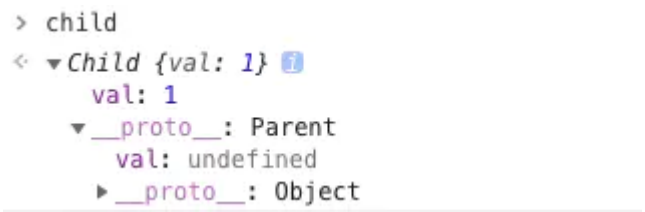
组合继承是最常用的继承方式，

```
function Parent(value) {
  this.val = value
```

```
}
Parent.prototype.getValue = function() {
  console.log(this.val)
}
function Child(value) {
  Parent.call(this, value)
}
Child.prototype = new Parent()

const child = new Child(1)

child.getValue() // 1
child instanceof Parent // true
```

以上继承的方式核心是在子类的构造函数中通过 `Parent.call(this)` 继承父类的属性，然后改变子类的原型为 `new Parent()` 来继承父类的函数。

这种继承方式优点在于构造函数可以传参，不会与父类引用属性共享，可以复用父类的函数，但是也存在一个缺点就是在继承父类函数的时候调用了父类构造函数，导致子类的原型上多了不需要的父类属性，存在内存上的浪费。



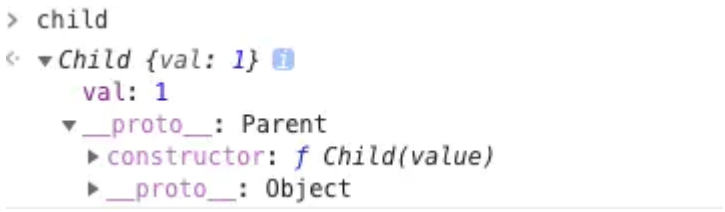
寄生组合继承

这种继承方式对组合继承进行了优化，组合继承缺点在于继承父类函数时调用了构造函数，我们只需要优化掉这点就行了。

```
function Parent(value) {
  this.val = value
}
Parent.prototype.getValue = function() {
  console.log(this.val)
}

function Child(value) {
  Parent.call(this, value)
}
Child.prototype = Object.create(Parent.prototype, {
  constructor: {
    value: Child,
    enumerable: false,
    writable: true,
    configurable: true
```

```
  }
})

const child = new Child(1)

child.getValue() // 1
child instanceof Parent // true
```

以上继承实现的核心就是将父类的原型赋值给了子类，并且将构造函数设置为子类，这样既解决了无用的父类属性问题，还能正确的找到子类的构造函数。



Class 继承

以上两种继承方式都是通过原型去解决的，在 ES6 中，我们可以使用 `class` 去实现继承，并且实现起来很简单

```
class Parent {
  constructor(value) {
    this.val = value
  }
  getValue() {
    console.log(this.val)
  }
}
class Child extends Parent {
  constructor(value) {
    super(value)
  }
}
let child = new Child(1)
child.getValue() // 1
child instanceof Parent // true
```

`class` 实现继承的核心在于使用 `extends` 表明继承自哪个父类，并且在子类构造函数中必须调用 `super`，因为这段代码可以看成 `Parent.call(this, value)`。

当然了，之前也说了在 JS 中并不存在类，`class` 的本质就是函数。

模块化

> 涉及面试题：为什么要使用模块化？都有哪几种方式可以实现模块化，各有什么特点？

使用一个技术肯定是有原因的，那么使用模块化可以给我们带来以下好处

- 解决命名冲突
- 提供复用性
- 提高代码可维护性

立即执行函数

在早期，使用立即执行函数实现模块化是常见的手段，通过函数作用域解决了命名冲突、污染全局作用域的问题

```
(function(globalVariable){
   globalVariable.test = function() {}
   // ... 声明各种变量、函数都不会污染全局作用域
})(globalVariable)
```

AMD 和 CMD

鉴于目前这两种实现方式已经很少见到，所以不再对具体特性细聊，只需要了解这两者是如何使用的。

```
// AMD
define(['./a', './b'], function(a, b) {
  // 加载模块完毕可以使用
  a.do()
  b.do()
})
// CMD
define(function(require, exports, module) {
  // 加载模块
  // 可以把 require 写在函数体的任意地方实现延迟加载
  var a = require('./a')
  a.doSomething()
})
```

CommonJS

CommonJS 最早是 Node 在使用，目前也仍然广泛使用，比如在 Webpack 中你就能见到它，当然目前在 Node 中的模块管理已经和 CommonJS 有一些区别了。

```
// a.js
module.exports = {
    a: 1
}
// or
exports.a = 1

// b.js
var module = require('./a.js')
module.a // -> log 1
```

因为 CommonJS 还是会使用到的，所以这里会对一些疑难点进行解析

先说 `require` 吧

```
var module = require('./a.js')
module.a
// 这里其实就是包装了一层立即执行函数，这样就不会污染全局变量了，
// 重要的是 module 这里，module 是 Node 独有的一个变量
module.exports = {
    a: 1
}
// module 基本实现
var module = {
  id: 'xxxx', // 我总得知道怎么去找到他吧
  exports: {} // exports 就是个空对象
}
// 这个是为什么 exports 和 module.exports 用法相似的原因
var exports = module.exports
var load = function (module) {
    // 导出的东西
    var a = 1
    module.exports = a
    return module.exports
};
// 然后当我 require 的时候去找到独特的
// id，然后将要使用的东西用立即执行函数包装下，over
```

另外虽然 `exports` 和 `module.exports` 用法相似，但是不能对 `exports` 直接赋值。因为 `var exports = module.exports` 这句代码表明了 `exports` 和 `module.exports` 享有相同地址，通过改变对象的属性值会对两者都起效，但是如果直接对 `exports` 赋值就会导致两者不再指向同一个内存地址，修改并不会对 `module.exports` 起效。

ES Module

ES Module 是原生实现的模块化方案，与 CommonJS 有以下几个区别

- CommonJS 支持动态导入，也就是 `require(${path}/xx.js)`，后者目前不支持，但是已有提案
- CommonJS 是同步导入，因为用于服务端，文件都在本地，同步导入即使卡住主线程影响也不大。而后者是异步导入，因为用于浏览器，需要下载文件，如果也采用同步导入会对渲染有很大影响
- CommonJS 在导出时都是值拷贝，就算导出的值变了，导入的值也不会改变，所以如果想更新值，必须重新导入一次。但是 ES Module 采用实时绑定的方式，导入导出的值都指向同一个内存地址，所以导入值会跟随导出值变化
- ES Module 会编译成 `require/exports` 来执行的

```
// 引入模块 API
import XXX from './a.js'
import { XXX } from './a.js'
// 导出模块 API
export function a() {}
export default function() {}
```

Proxy

> 涉及面试题：Proxy 可以实现什么功能?

如果你平时有关注 Vue 的进展的话，可能已经知道了在 Vue3.0 中将会通过 `Proxy` 来替换原本的 `Object.defineProperty` 来实现数据响应式。 Proxy 是 ES6 中新增的功能，它可以用来自定义对象中的操作。

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

`target` 代表需要添加代理的对象， `handler` 用来自定义对象中的操作，比如可以用来自定义 `set` 或者 `get` 函数。

接下来我们通过 `Proxy` 来实现一个数据响应式

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

```
  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 // 监听到属性a改变
p.a // 'a' = 2
```

在上述代码中，我们通过自定义 `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。

> 更新：评论中有同学对于 Proxy 无需一层层递归为每个属性添加代理有疑问，以下是实现代码。

```
get(target, property, receiver) {
    getLogger(target, property)
    // 这句判断代码是新增的
    if (typeof target[property] === 'object' && target[property] !== null) {
        return new Proxy(target[property], handler);
    } else {
        return Reflect.get(target, property);
    }
}
```

map, filter, reduce

> 涉及面试题：map, filter, reduce 各自有什么作用?

`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放入到新的数组中。

```
[1, 2, 3].map(v => v + 1) // -> [2, 3, 4]
```

另外 `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

```
['1','2','3'].map(parseInt)
```

- 第一轮遍历 `parseInt('1', 0) -> 1`
- 第二轮遍历 `parseInt('2', 1) -> NaN`
- 第三轮遍历 `parseInt('3', 2) -> NaN`

`filter` 的作用也是生成一个新数组，在遍历数组的时候将返回值为 `true` 的元素放入新数组，我们可以利用这个函数删除一些不需要的元素

```
let array = [1, 2, 4, 6]
let newArray = array.filter(item => item !== 6)
console.log(newArray) // [1, 2, 4]
```

和 `map` 一样， `filter` 的回调函数也接受三个参数，用处也相同。

最后我们来讲解 `reduce` 这块的内容，同时也是最难理解的一块内容。 `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`

想必通过以上的解析大家应该明白 `reduce` 是如何通过回调函数将所有元素最终转换为一个值的，当然 `reduce` 还可以实现很多功能，接下来我们就通过 `reduce` 来实现 `map` 函数

```
const arr = [1, 2, 3]
const mapArray = arr.map(value => value * 2)
const reduceArray = arr.reduce((acc, current) => {
  acc.push(current * 2)
  return acc
}, [])
console.log(mapArray, reduceArray) // [2, 4, 6]
```

如果你对这个实现还有困惑的话，可以根据上一步的解析步骤来分析过程。