18 ES2015+考点集中解析

更新时间：2020-05-26 15:01:59

> 天才就是这样，终身努力，便成天才。——门捷列夫

本节我们将通过小节形式的知识点串讲，对 ES6 中剩余的面试高频考点做逐一排查和梳理，意在帮助大家查漏补缺。在理解难度上，本节比上节更小，但知识点具备一定的琐碎度，因此同样不容忽视。

对象与数组的解构

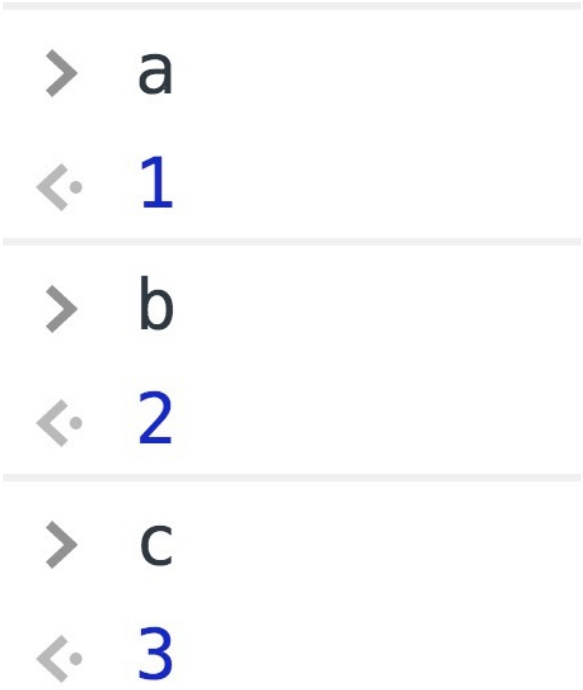
解构是 ES6 提供给我们的一种新的提取数据的模式，这种模式能够帮助我们从对象或数组里有针对性地拿到我们想要的数值。

数组的解构

在解构数组时，我们是以元素的位置为匹配条件来提取我们想要的数据的。举个例子：

```
const [a, b, c] = [1, 2, 3]
```

把这段代码丢进控制台里跑一跑，你会发现a、b、c分别被赋予了数组第0、1、2个索引位的值：

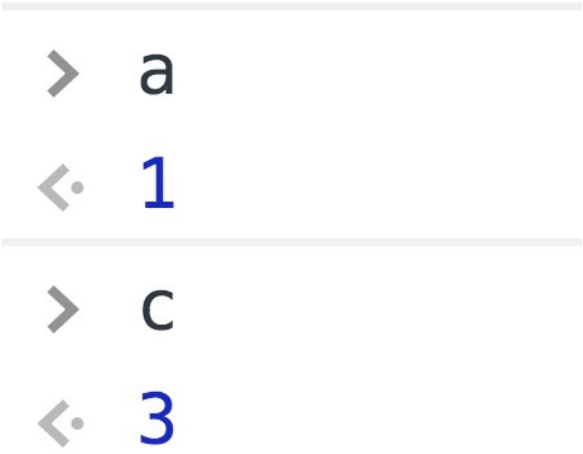


“一个萝卜一个坑”，数组里的0、1、2索引位的元素值，精准地被映射到了左侧的第0、1、2个变量里去，这就是数组解构的工作模式。

我们还可以通过给左侧变量数组设置空占位的方式，实现对数组中某几个元素的精准提取：

```
const [a,,c] = [1,2,3]
```

通过把中间位留空，我们可以顺利地把数组第一位和最后一位的值赋给 a、c 两个变量：



对象的解构

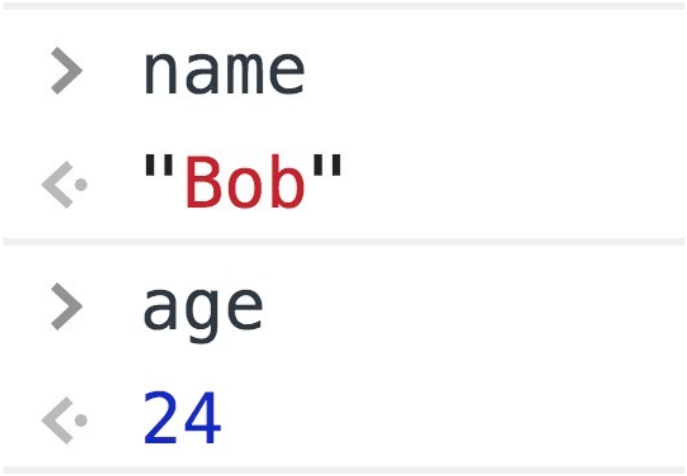
对象解构比数组结构稍微复杂一些，也更显强大。在解构对象时，我们是以属性的名称为匹配条件，来提取我们想要的数据的。举个例子，我们现在定义一个对象：

```
const stu = {
  name: 'Bob',
  age: 24
}
```

假如我们想要解构它的两个自有属性，就可以这么干：

```
const { name, age } = stu
```

如此一来，我们就得到了 name 和 age 两个和 stu 平级的变量：



注意，对象解构严格以属性名作为定位依据，所以说咱们就算调换了 name 和 age 的位置，结果也是没差的：

```
const { age, name } = stu
```

考点点拨：如何提取高度嵌套的对象里的指定属性？

有时我们会遇到一些嵌套程度非常深的对象：

```
const school = {
  classes: {
    stu: {
      name: 'Bob',
      age: 24,
    }
  }
}
```

像此处的 name 这个变量，嵌套了足足四层，此时如果我们仍然尝试老方法来提取它：

```
const { name } = school
```

显然是不奏效的，因为 school 这个对象本身是没有 name 这个属性的，name 位于 school 对象的“儿子的儿子”对象里面。要想把 name 提取出来，一种比较笨的方法是逐层解构：

```
const { classes } = school
const { stu } = classes
const { name } = stu

name // 'Bob'
```

但是还有一种更标准的做法，我们可以用一行代码来解决这个问题：

```
const { classes: { stu: { name } }} = school

name // 'Bob'
```

没错，我们可以在解构出来的变量名右侧，通过冒号+{目标属性名}这种形式，进一步解构它，一直解构到我们拿到目标数据为止。

考点点拨： 解构同时重命名

如果你接到的需求是给 name 起个新名字，采取 属性名：新变量名 这种形式就好：

```
const { classes: { stu: { name: newName } }} = school

newName // 'Bob'
```

一言难尽的“...”

"..."相信大家已经非常熟悉了，它是 ES6 中一个应用非常广泛的运算符——扩展运算符。这个符号需要引起大家的警惕，因为不管你用得再多，你可能都很难说是真正地了解它，因为它不止一面，不信你瞅瞅：

扩展运算

这是大家最熟悉的一面了。扩展运算在对象和数组中有着不同的表现：

对象扩展运算

对象中的扩展运算符(...)用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中。举个例子：

```
const me = {
  name: 'xiuyan',
  age: 24
}

const meCopy = { ...me }

meCopy // {name: "xiuyan", age: 24}
```

这里的 ...me 其实就等价于下面这种写法：

```
Object.assign({}, me)
```

数组扩展运算

在数组中，扩展运算可以将一个数组转为用逗号分隔的参数序列。举个例子：

```
console.log(...['haha', 'hehe', 'xixi']) // haha hehe xixi
```

再举个例子：

```
function mutiple(x, y) {
  return x*y
}

const arr = [2, 3]
mutiple(...arr) // 6
```

考点点拨：合并两个数组

```
const arr1 = [1, 2, 3, 4]
const arr2 = [5, 6, 7, 8]
```

问如何把两个数组合并到一个数组里去。这套题可能有很多种解题姿势，但是最好看的一种姿势莫过于用我们的扩展符：

```
const newArr = [...arr1, ...arr2]
```

rest 参数

上面我们为大家梳理了各位相对更为熟悉的一些扩展符的操作。但是扩展符还有它的另一面，当它被用在函数形参上时，**它还可以把一个分离的参数序列整合成一个数组**：

```
function mutiple(...args) {
  let result = 1;
  for (var val of args) {
    result *= val;
  }
  return result;
}

mutiple(1, 2, 3, 4) // 24
```

在这个例子里，我们传入 mutiple 的是四个分离的参数，但是如果你在 mutiple 函数里尝试输出 args 的值，你会发现它是一个数组：

```
function mutiple(...args) {
  console.log(args)
}

mutiple(1, 2, 3, 4) // [1, 2, 3, 4]
```

这就是 … 运算符的又一层威力了，它可以把函数的多个入参收敛进一个数组里。这一点经常被我们用于获取函数的多余参数，或者像上面这样处理函数参数个数不确定的情况。

类数组的转换

这里我要敲下黑板哈，类数组是本节的重中之重。如果你记性很差，学完了这满满一节只能记住一个考点，那我求你记住类数组。别问为啥，问就是会考！
先来看看啥是类数组对象，ECMA-262对它的定义是：

1. 它必须是一个对象
2. 它有 length 属性

按照这个标准，只要有 length 属性的对象就是类数组对象：

```
const book = {
  name: 'how to read a book',
  age: 10,
  length: 300
} // 这是一个类数组对象
```

注意，这里很多同学可能还看过另一个版本的解释：

> 类数组对象是指拥有一个 length 属性和若干索引属性的对象

按照这个标准，好像必须得这样才是类数组对象：

```
const book = {
  0: 'how to read a book',
  1: 10,
  length: 2
}
```

其实通常，类似数组的对象也有一些具有整数索引名的属性，不过这并非定义的要求。因此上面两个 book，都可以认为是类数组的对象。
不过大家也不必纠结太多，在类数组的考察上死扣定义的面试官几乎没有。类数组这里需要给大家点拨的一个高频考点是**类数组的转换**。

考点点拨： 如何把类数组对象转换为真正的数组？
命题形式：大概率以编码题形式出现。
真题解析：将如下的类数组对象转换为数组，你能想到哪些方法？

```
const arrayLike = {0: 'Bob', 1: 'Lucy', 2: 'Daisy', length: 3 }
```

这道题期望大家能答出下面 3 种思路：

- Array原型上的slice方法—— 这个方法如果不传参数的话会返回原数组的一个拷贝，因此可以用此方法转换类数组到数组：

```
const arr = Array.prototype.slice.call(arrayLike);
```

- Array.from方法——这是 ES6 新增的一个数组方法，专门用来把类数组转为数组：

```
const arr = Array.from(arrayLike);
```

- 扩展运算符——"..."也可以把类数组对象转换为数组，前提是这个类数组对象上部署了遍历器接口。在这个例子中，arrayLike 没有部署遍历器接口，所以这条路走不通。但一些对象，比如函数内部的 arguments 变量（它也是类数组对象），就满足条件，可以用这种方法来转换：

```
function demo() {
  console.log('转换后的 arguments 对象：', [...arguments])
}

demo(1, 2, 3, 4) // 转换后的 arguments 对象：[1, 2, 3, 4]
```

模板语法与字符串处理

ES6 中的字符串也是一个不可忽视的考点，因为 ES6 针对字符串做了不少突破性的改造。其中最激动人心的，要数“模板语法”了。

模板语法

大家知道，在 ES6 问世以前，我们拼接字符串是一件挺麻烦的事情：

```
var name = 'xiuyan'
var career = 'coder'
var hobby = ['coding', 'writing']

var finalString = 'my name is ' + name + ', I work as a ' + career + ', I love ' + hobby[0] + ' and ' + hobby[1]
```

仅仅几个变量，我们写了这么多加号，还要时刻小心里面的空格和标点符号有没有跟错地方。但是有了模板字符串，拼接难度直线下降：

```
var name = 'xiuyan'
var career = 'coder'
var hobby = ['coding', 'writing']

var finalString = `my name is ${name}, I work as a ${career} I love ${hobby[0]} and ${hobby[1]}`
```

我们发现字符串不仅好拼了，也更易读了，代码整体的质量都变高了。这就是模板字符串的第一个福利——允许我们用${}这样简单的方式嵌入变量。但这还不是问题的关键，模板字符串的关键优势有两个：

- 在模板字符串中，空格、缩进、换行都会被保留
- 模板字符串完全支持“运算”式的表达式，你可以在${}里完成一些计算

基于第一点，我们可以在模板字符串里无障碍地直接写 html 代码：

```
let list = `
 <ul>
  <li>列表项1</li>
  <li>列表项2</li>
 </ul>
`;

console.log(message); // 顺利输出，不存在报错
```

基于第二点，我们完全可以把一些简单的计算和调用丢进 ${} 来做，非常方便：

```
function add(a, b) {
  const finalString = `${a} + ${b} = ${a+b}`
  console.log(finalString)
}

add(1, 2) // 输出 '1 + 2 = 3'
```

更强的方法

除了模板语法外， ES6中还新增了一系列的字符串方法用于提升我们的开发效率，大家日后做实际编码题的时候可能会用到。这里我们提取其中常用的部分，总结如下：

- **存在性判定**：在过去，当我们想判断一个字符/字符串是否在某字符串中时，只能用 indexOf > -1 来做。现在 ES6 提供了三个方法：includes、startsWith、endsWith，它们都会返回一个布尔值来告诉你是否存在。
    - **includes**：判断字符串与子串的包含关系：

```
const son = 'haha'
const father = 'xixi haha hehe'

father.includes(son) // true
```

- **startsWith**：判断字符串是否以某个/某串字符开头：

```
const father = 'xixi haha hehe'

father.startsWith('haha') // false
father.startsWith('xixi') // true
```

- **endsWith**：判断字符串是否以某个/某串字符结尾：

```
const father = 'xixi haha hehe'

father.endsWith('hehe') // true
```

- 自动重复： 我们可以使用 repeat 方法 来使同一个字符串输出多次（被连续复制多次）：

```
const sourceCode = 'repeat for 3 times;'
const repeated = sourceCode.repeat(3)

console.log(repeated) // repeat for 3 times;repeat for 3 times;repeat for 3 times;
```

}

17 变量提升与暂时性死区

19 DOM基本功