【前端面试】一文搞懂reduce的用法和使用...

前言

JavaScript 中的数组操作方法有好几十种，有的很简单，有的稍显复杂。而今天我们单独拿出来讲的便是 reduce 方法，为什么单独把 reduce 拿出来说呢？因为它算是 JS 数组方法里面较为复杂的一个了，而且很多面试官非常喜欢让面试者手动实现 reduce，所以我们有必要好好学一学这个数组方法，本篇文章主要是以学习 reduce 为主，后续会推出手写 reduce 的文章。

1.基本概念

想要学习一个 API 或者新的知识，我们通常是先去看看官网，有了大概的了解，我们才能深入下去学习，学习 reduce 一样，我们同样先去看看官网对它的解释。

官方解释：

> reduce() 方法对数组中的每个元素按序执行一个由您提供的 reducer 函数，每一次运行 reducer 会将先前元素的计算结果作为参数传入，最后将其结果汇总为单个返回值。
>
> 第一次执行回调函数时，不存在“上一次的计算结果”。如果需要回调函数从数组索引为 0 的元素开始执行，则需要传递初始值。否则，数组索引为 0 的元素将被作为初始值 initialValue，迭代器将从第二个元素开始执行（索引为 1 而不是 0）。

从上面我们可以看出，官网的解释也蛮多的，说明这个 API 不是那么的简单。大家可能正对于第一段话还比较好理解，无非就是 reduce 方法接收一个回调函数，每一次回调函数的计算结果将被当作下一个回调函数的执行参数，这一点就很类似于我们熟知的 map 等方法了。

第二段话大家可能就没有那么容易理解了，我们可以拿 map 来举例，我们都知道 map 循环数组的方式如下：

```
arr.map((item) => {})
```

map 接收一个回调函数，回调函数的参数就是每一个数组元素，我们这里的 reduce 方法也是接收的一个回调函数，但是回调函数的参数不再是数组元素了，而是上一次执行回调函数的执行结果或者说是即使算结果。这里我们就有疑问，第一次执行回调函数是没有上一次的执行结果

的，那么它的参数是多少呢？这就是官方第二段内容解释的：第一次执行回调函数，参数应该是初始值，初始值我们可以设置，如果我们没有设置，那么初始值就是数组第一个元素。

语法：

上面一大段话大家看起来可能比较头痛，我们还是来看代码吧，更好理解。

```
reduce((previousValue, currentValue, currentIndex, array) => {}, initialValue)
// 简洁语法
reduce(callbackFn, initialValue)
```

reduce 的语法非常好理解，reduce 方法接收两个参数：回调函数和初始值。这里所说得到初始值就是前面我们说的回调函数第一次执行时的参数。

参数详解：

- callbackFn：回调函数
  - previousValue：上一次回调函数的返回值，第一次调用回调函数时，如果指定的初始值 initialValue，那么该参数值就是 initialValue，否则就是数组元素的第一个。
  - currentValue：数组中正在处理的元素。在第一次调用时，若指定了初始值 initialValue，那么当前处理的元素就是数组的第一个元素，否则的话就是第二个元素。
  - currentIndex：数组中正在处理的元素的索引。若指定了初始值 initialValue，则起始索引号为 0，否则从索引 1 起始。
  - array：用于遍历的数组。
- initialValue：初始值，也就是第一次调用回调函数时参数 previousValue，是否指定初始值将会影响其它几个参数的值。

从参数详解可以看出，initialValue 初始值很重要，因为它是否出现极大的影响了其它参数的值。

返回值：

reduce 方法的返回值就是遍历所有数组执行回调函数后的返回值。

2.基本使用

reduce 的基本语法和概念我们了解了，那么我们一起来看看它最重要的使用方式。这里我们需要分为两种情况，一种是有初始值 initialValue，一种是没有初始值 initialValue，因为它们两种的情况还是蛮大的。

2.1 有 initialValue

回顾一下：有初始值 initialValue，则回调函数第一次执行时的 previousValue 为初始值，currentValue 为数组元素的第一个，currentIndex 为 0。

代码如下：

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
const total = arr.reduce((previousValue, currentValue) => {
  return previousValue + currentValue; // 100 + 1 + 2 + 3 + 4 + 5 + 6 + 7 +
}, 100);
console.log(total); // 136
```

上段代码中我们使用 reduce 方法循环了数组 arr，并且设置了初始值 100，第一次执行回调函数时，previousValue 为 100，currentValue 为 1，返回结果为 101。101 作为第二次循环的 previousValue 传入回调函数，返回 101+2=103，依次类推……。

上段代码中我们少些了两个参数：currentIndex,和 array。我们把这两个参数带上再来看看什么结果。

代码如下：

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
const total = arr.reduce((previousValue, currentValue, currentIndex, array)
  console.log(previousValue, currentValue, currentIndex, array)
  return previousValue + currentValue; // 100 + 1 + 2 + 3 + 4 + 5 + 6 + 7 +
}, 100);
console.log(total); // 136
```

输出结果：

```
100 1 0 ▶(8) [1, 2, 3, 4, 5, 6, 7, 8]
101 2 1 ▶(8) [1, 2, 3, 4, 5, 6, 7, 8]
103 3 2 ▶(8) [1, 2, 3, 4, 5, 6, 7, 8]
106 4 3 ▶(8) [1, 2, 3, 4, 5, 6, 7, 8]
110 5 4 ▶(8) [1, 2, 3, 4, 5, 6, 7, 8]
115 6 5 ▶(8) [1, 2, 3, 4, 5, 6, 7, 8]
121 7 6 ▶(8) [1, 2, 3, 4, 5, 6, 7, 8]
128 8 7 ▶(8) [1, 2, 3, 4, 5, 6, 7, 8]
136
```

2.2 没有 initialValue

接下来我们再来尝试下没有初始值 initialValue 时，执行结果是什么？我们采用同样的代码，这样更好分析结果。

代码如下：

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
const total = arr.reduce((previousValue, currentValue, currentIndex, array)
  console.log(previousValue, currentValue, currentIndex, array)
  return previousValue + currentValue; // 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8
});
console.log(total); // 36
```

输出结果：

```
1 2 1 ▶ (8) [1, 2, 3, 4, 5, 6, 7, 8]
3 3 2 ▶ (8) [1, 2, 3, 4, 5, 6, 7, 8]
6 4 3 ▶ (8) [1, 2, 3, 4, 5, 6, 7, 8]
10 5 4 ▶ (8) [1, 2, 3, 4, 5, 6, 7, 8]
15 6 5 ▶ (8) [1, 2, 3, 4, 5, 6, 7, 8]
21 7 6 ▶ (8) [1, 2, 3, 4, 5, 6, 7, 8]
28 8 7 ▶ (8) [1, 2, 3, 4, 5, 6, 7, 8]
36
```

从上面的执行结果可以看出，没有初始值时，第一次执行回调函数时 previousValue 为 1，即数组的第一个元素，currentValue 为 2，即数组第二个元素，currentIndex 为 1。所以依次执行回调函数，实际上就是在累加数组中的元素。

3.使用场景

reduce 的使用场景是非常宽泛的，因为它很强大。当然，很多小伙伴可能会有这样一个疑问：使用 for 循环或者 forEach 循环能完成的功能，为什么要使用 reduce 呢？事实确实如此，有很多场景都有多种解决方案，我们选用我们最为熟悉、最为简单或者性能最好的方式就行了。

3.1 数组求和

在我们的基本使用中，其实就体现了数组求和的场景，这里我们简单温习一遍。

代码如下：

```
let arr1 = [1, 2, 3, 4, 5, 6, 7, 8];
const total1 = arr1.reduce((previousValue, currentValue) => {
  return previousValue + currentValue;
}, 0);
console.log(total1); // 36
```

3.2 数组对象求和

这和数组求和非常的类似，只不过我们的数组结稍微变得复杂了一点。

代码如下：

```
let arr2 = [{ x: 1 }, { x: 2 }, { x: 3 }];
const total2 = arr2.reduce((previousValue, currentValue) => {
  return previousValue + currentValue.x;
}, 0);
console.log(total2); // 6
```

举这个例子的目的就是让大家重点理解 previousValue 和 currentValue 的关系。

3.3 二维数组转为一维数组

这种场景也可以称作为数组的扁平化，只不过这儿只针对于二维数组的扁平化。

代码如下：

```
let arr3 = [[0, 1], [2, 3], [4, 5]];
const flattened = arr3.reduce((previousValue, currentValue) => {
  return previousValue.concat(currentValue);
}, [])
console.log(flattened); // [0, 1, 2, 3, 4, 5]
```

代码中我们将初始值设置为了[]，然后利用 concat 将数组中的每一项与初始值拼接，得到一个新的数组。

3.4 计算数组元素出现次数

这种场景通常出现在算法题当中，借助 reduce 我们简单实现它。

代码如下：

```
let names = ['小猪课堂', '张三', '李四', '王五', '小猪课堂']
let countedNames = names.reduce(function (allNames, name) {
  // 判断当前数组元素是否出现过
  if (name in allNames) {
    allNames[name]++
  } else {
    allNames[name] = 1
  }
  return allNames
}, {})
console.log(countedNames); // {小猪课堂: 2, 张三: 1, 李四: 1, 王五: 1}
```

上段代码中我们主要学习的是实现思路，我们通过键值对的形式巧妙的将出现的次数存储下来。

总结

reduce 的使用场景远不止上面几种，我们这里只是举例了一些简单场景，实际上 reduce 还可以在很多复杂的场景下使用，这都得益于它的初始值概念，以及回调函数 previousValue 得概念。至于更多使用场景，还需要大家自行探索自行总结了。