从dedup说起之JS数组去重

在JavaScript中，数组去重是一个基本的操作，方法众多：遍历去重到Set、Map去重、hashTable、Lodash Uniq，数组中是否存在对象、函数，每个去重方法的表现各有差异，本文将以此作为切入点深入源码进行分析。

1. **定义重复**

在JS中，对于原始值而言，我们很容易想到1和1是相等的，'1'和'1'也是相等的。1和'1'是不相等的。那么对于如下情况呢？

1. NaN，NaN不是独立的数据类型，是数字类型，但是NaN不等于任何一个数字，也不等于任何一个其他的NaN，例如a=NaN，b=NaN，a== b不成立。但是这显然是不符合我们的判断是重复的需求的。好在js中有isNaN来弥补这个不足：isNaN(a) && isNaN(b)。
2. 对象和函数

初始化一个对象时，对象名放在桟中，它指向堆中存放的内容。所以两个对象直接对比是否相等是永远不等的，{a:1} == {a:1}永远不成立。函数也是类似的。要判断两个对象是否显式相等，我们可以采用遍历递归的方式来进行判断。

1. Null

Null并不是一个单独的特定类型，当对象的属性赋值为null时，表示该属性是空。Null的类型是object。Null虽然是对象，但是两个null是相等的

本文讨论ref去重，在这里不再赘述显式相等。有兴趣可以参看对象显式判等【代码1】

1. **数组去重**
2. **遍历（indexOf和includes）**

遍历是最容易想到也比较直观的方法：

代码如下：

function uniq1(arr) {

var newArr = [];

arr.forEach(function(item){

if(!newArr.includes(item)){

newArr.push(item);

}

});

return newArr ;

}

之所以使用includes而不使用indexOf方法，是因为对于NaN的判定，indexOf是不符合我们的去重规定的：

var arr = [1, 2, NaN];

arr.indexOf(NaN); // -1

arr.includes(NaN); // true

评价：includes是ES6的新语法，因此使用有局限性。另外不管是使用indexOf还是includes，本质上都是一个双重循环的方法，时间复杂度比较高，性能一般。

1. **hashTable方法**

简介一下hashTable方法：HashTable算法有一个规则，约束键与存储位置的关系。此方法就是利用键值对的对应关系，计算出一个hash值存为key，value存储将要判断的数据（数字或字符串）。因此通过使用HashTable结构记录已有的元素hash是一种典型以空间换时间的算法，其查找键值对的速度非常快。另外HashTable在JavaScript中实现较为简单，详见代码。

function uniq2(array) {

var newArr = []

var current\_temp = {}

for(var i = 0 ; i < array.length ; i++){

if(!current\_temp[array[i]]){

newArr.push(array[i])

current\_temp[array[i]] = true

}

}

return newArr

}

评价：需要注意的是，hashTable存放的key是不能区分对象和函数的，Key都将被转换为string类型，对象存储的key都是’[object object]’，函数存储的都是’[function function]’。

hashTable快于原生的Set方法，hashTable本身不存在对于Boolean、Number、String、NaN、undefined、null等类型的判断，一律会转为String，因此需要额外添加判断语句，本例hashTable方法还有许多不足。

1. **改进的hashTable方法**

如果像上文所说，hashTable方法是不完善的，我们可以通过手动添加条件，将hashTable方法完善起来。除了下面的方法可以完善hashTable，注：还有一种序列化Key的方法可以完善hashTable方法，使用JSON.stringify()进行序列化 ，将typeof arr[i] + JSON.stringify(arr[i])存储为key值从而保证每个元素的独一性，但是时间复杂度与双重循环一样，性能较差，并且不能判定是否是function。

function uniq3 (arr) {

      var tmp\_map = {

            string: {},

            number: {},

      };

      var has\_undefined = false

      var has\_true = false

      var has\_false = false

      var returnList = []

      var noneHashableList = []

      var len = arr.length;

      for (var i = 0; i < len; i++) {

            var value = arr[i];

            var type = typeof value;

            if (type === 'string' || type === 'number') {

                  if (!tmp\_map[type][value]) {

                        returnList.push(value);

                        tmp\_map[type][value] = true;

                  }

            } else if (value === undefined) {

                  if (has\_undefined) continue;

                  has\_undefined = true;

                  returnList.push(undefined);

            } else if (value === true) {

                  if (has\_true) continue;

                  has\_true = true;

                  returnList.push(true);

            } else if (value === false) {

                  if (has\_false) continue;

                  has\_false = false;

                  returnList.push(has\_false);

            } else if (!noneHashableList.includes[value]) {

                  noneHashableList.push(value);

                  returnList.push(value);

            }

      }

      return returnList;

}

总结：经过完善后的hashTable方法，增加了多次判断，因此性能会有所影响。但是在只有number和String类型的情况下性能应该还是很快。

**4.Map方法**

总结上述方法，因为key类型的限制，使得hashTable方法存在局限性，那么有没有对key类型没有限制的对象呢？答案就是利用Es6的Map方法：Map是一种新的数据类型，可以把它想象成key类型没有限制的对象。此外，它的存取使用单独的get()、set()、has()接口。使用原生的Map方法时，但是object和func作为Map元素的key 时， 会执行toString方法，导致速度变慢。

function uniq4(array) {

const seen = new Map()

array.filter((a) => !seen.has(a) && seen.set(a, 1))

return array;

}

**5.Set方法**

利用Es6的Set方法，Set将数组转换为一个不含有重复元素的对象，然后再使用Array.from方法转换为数组。因为 Set 中的值总是唯一的，所以需要判断两个值是否相等。判断相等的算法与严格相等（===操作符）不同。具体来说，对于 Set ， +0 （+0 严格相等于-0）和-0是不同的值。尽管在最新的 ECMAScript 6规范中这点已被更改。从Gecko 29.0和 [recent nightly Chrome](https://code.google.com/p/v8/issues/detail?id=3069)开始，Set 视 +0 和 -0 为相同的值。另外，NaN和undefined都可以被存储在Set 中， NaN之间被视为相同的值（尽管 NaN !== NaN）。综上所述，Set方式可以说是最全面的数组去重的方法。

function uniq5(array) {

return Array.from(new Set(array));

}

1. **裸写的Map方法**

不同的浏览器对于方法是否包裹在函数内部有针对性的优化，例如谷歌浏览器。

代码如下：

//裸写的Map方法

const seen = new Map()

array.filter((a) => !seen.has(a) && seen.set(a, 1))

//裸写的Set方法

arr = [...new Set(arr2)];

1. **uniq方法**

使用lodash库的uniq方法，Lodash用来操作对象和集合。查看uniq的Github可以发现:[https://github.com/lodash/lodash/blob/76ab9cd539feba8ae923372c19ab27d312078ee5/uniq.js](https://github.com/lodash/lodash/blob/76ab9cd539feba8ae923372c19ab27d312078ee)，uniq是分了情况来判断是否相等的，另外该方法对于NaN类型是支持的。该方法在数组内数据量小于200的时候使用的是直接递归判断（outer），当数据量大于200的时候，使用的是原生的Map方法，详见lodash代码：[https://github.com/lodash/lodash/blob/2f281c68b01f7a10c910cf4f67f55f514f7b1081/.internal/baseUniq.js#9](https://github.com/lodash/lodash/blob/2f281c68b01f7a10c910cf4f67f55f514f7b1081/.internal/baseUniq.js" \l "9)。

function uniq6(array) {

return \_.uniq(array);

}

1. **测试**
2. 测试用例

Github测试地址：<https://github.corp.ebay.com/pengcheng/LearnJavaScript/blob/master/arratTest.html>

示例：

测试数组：[true, false, false, 1, "1", "1", 0, 0, "0", "0", undefined, undefined, null, null, () => (false),() => (false),Array(0), Array(0), /a/, /a/]

预期结果：[true, false, 1, "1", 0, "0", undefined, null, () => (false),() => (false), Array(0), Array(0), /a/, /a/]

测试用例代码：

//数字  
 for (var i = 0; i < 900000; i++) {  
 arr1.push(parseInt(Math.random() \* 10) + 1);  
 }  
 //字符串  
 for (var i = 0; i < 20000; i++) {  
 stringArr.push((parseInt(Math.random() \* 10) + 1).toString())  
 }  
 for (var i = 0; i < 20000; i++) {  
 arr1.push(stringArr[i]);  
 }  
 //对象  
 for (var i = 0; i < 20000; i++) {  
 objArr.push({a: (parseInt(Math.random() \* 10) + 1)})  
 }  
 for (var i = 0; i < 20000; i++) {  
 arr1.push(objArr[i]);  
 }  
 //函数  
 for (var i = 0; i < 20000; i++) {  
 var a = () => (parseInt(Math.random() \* 10) + 1)  
 funArr.push(a)  
 }  
 for (var i = 0; i < 20000; i++) {  
 arr1.push(funArr[i]);  
 }  
 //undefined  
 for (var i = 0; i < 10000; i++) {  
 funArr.push(undefined)  
 }  
 for (var i = 0; i < 20000; i++) {  
 arr1.push(funArr[i]);  
 }  
 //null  
 for (var i = 0; i < 5000; i++) {  
 funArr.push(null)  
 }  
 for (var i = 0; i < 5000; i++) {  
 arr1.push(funArr[i]);  
 }  
 //true、false  
 for (var i = 0; i < 5000; i++) {  
 arr1.push(parseInt(i%2 === 0 ? true : false);  
 }

实际测试数据量：

90万数字、2万字符串、2万对象、2万函数、1万undefined、5千null、5千true和false，合计100万

1. 测试结果

测试环境：Win7 64位 8G内存 Core i7 5320 数据量（100万） Chrome浏览器

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 遍历（includes） | hashTable（改进） | Map | Map without function | Set | Set without function | Loda  Uniq |
| Number only | 128ms | 10ms | 61ms | 59ms | 66ms | 52ms | 51ms |
| Number & String | 146ms | 10ms | 66ms | 66ms | 85ms | 98ms | 95ms |
| Number & String & Object & Function | 19772ms | 85ms | 104ms | 91ms | 93ms | 75ms | 75ms |

测试环境：Win7 64位 8G内存 Core i7 5320 数据量（100万） Firefox浏览器

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 遍历（includes） | hashTable（改进） | Map | Map without function | Set | Set without function | Loda  Uniq |
| Number only | 39ms | 7ms | 55ms | 53ms | 28ms | 28ms | 33ms |
| Number & String | 39ms | 9ms | 52ms | 50ms | 33ms | 32ms | 33ms |
| Number & String & Object & Function | 16774ms | 87ms | 95ms | 91ms | 66ms | 60ms | 62ms |

1. 测试总结

只有数字和字符串的类型的测试中，Chrome和Firefox浏览器hashTable方法都是最快的，甚至远远快于原生的set方法。而在Firefox中，对ES6的原生方法includes支持是最快的，set方法和includes方法很接近。对于includes方法，Firefox的底层实现是直接取对象而不是通过指针读取数据，所以它的includes要快于Chrome。

对于，个人判断是对于if的执行语句Firefox是同步执行的，而chrome是异步执行。

在含有对象和函数的类型中，改进的HashTable要慢很多。Chrome和Firefox都是lodash的uniq方法较为出色，该方法性能也很稳定。

对于原生的includes方法，一旦数组中含有对象会慢很多。

此外，因为不同的浏览器对裸露函数和包裹函数的执行进行过各自的优化，可以看出Chrome浏览器在这方面要优于Firefox，同时可以看到，包裹在Function里面的执行方法与裸露的方法优化是根据不同的方法进行的优化。

【代码1】

**深度递归判断对象是否显式相等**

deepEquals = (obj1, obj2) => {  
 if (obj1 === obj2 || (isNaN(obj1) && isNaN(obj2))) return true;  
 if (typeof obj1 !== typeof obj2 || obj1 === null || obj2 === null) {  
 return false;  
 } else if (typeof obj1 === 'object') {  
 const keys1 = Object.keys(obj1);  
 const keys2 = Object.keys(obj2);  
 if (keys1.length !== keys2.length) return false;  
  
 for (let i = keys1.length - 1; i >= 0; i -= 1) {  
 const key = keys1[i];  
 if (!deepEquals(obj1[key], obj2[key])) return false;  
 }  
 return true;  
 }  
 return false;  
};