JavaScript系列

常见的js数组去重方法

1. ES6的 new Set()

> ES6提供了新的数据结构Set。类似与数组Array的结构，但是成员的值是唯一。
> 可以利用 `new Set(array)` 来实现数组去重, 但是有弊端。对于 `{}` 无法进行去重判断，需要额外写方法判断。

2.利用for嵌套for，然后splice去重（ES5中最常用）

```
function unique(arr){
        for(var i=0; i<arr.length; i++){
            for(var j=i+1; j<arr.length; j++){
                if(arr[i]==arr[j]){
                  //第一个等同于第二个，splice方法删除第二个
                    arr.splice(j,1);
                    j--;
                }
            }
        }
return arr;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, '

console.log(unique(arr))
//[1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {…}, {…}]     //NaN和{}没有去重，两个null直接消失了
```

> 此方法太过于粗暴, 占用性能过高

3. 利用indexOf去重

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var array = [];
    for (var i = 0; i < arr.length; i++) {
        if (array.indexOf(arr[i]) === -1) {
            array.push(arr[i])
        }
    }
    return array;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, '
console.log(unique(arr))
   // [1, "true", true, 15, false, undefined, null, NaN, NaN, "NaN", 0, "a", {…}, {…}]  //NaN、{}没有去重
```

> 弊端: NaN无法判断,对象无法

4. sort()

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return;
    }
    arr = arr.sort()
    var arrry= [arr[0]];
    for (var i = 1; i < arr.length; i++) {
        if (arr[i] !== arr[i-1]) {
            arrry.push(arr[i]);
        }
    }
    return arrry;
}
     var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0,
        console.log(unique(arr))
// [0, 1, 15, "NaN", NaN, NaN, {…}, {…}, "a", false, null, true, "true", undefined]      //NaN、{}没有去重
```

> 由于它取决于具体实现，因此无法保证排序的时间和空间复杂性。

5. includes()

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var array =[];
    for(var i = 0; i < arr.length; i++) {
            if( !array.includes( arr[i]) ) {//includes 检测数组是否有某个值
                    array.push(arr[i]);
              }
    }
    return array
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, '
    console.log(unique(arr))
    //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}, {…}]     //{}没有去重
```

6. hasOwnProperty

```
function unique(arr) {
    var obj = {};
    return arr.filter(function(item, index, arr){
        return obj.hasOwnProperty(typeof item + item) ? false : (obj[typeof item + item] = true)
    })
}
    var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0,
        console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}]   //所有的都去重了
```

> 主要是利用对象的hasOwnProperty判断是否存在对象属性

7. filter

```
function unique(arr) {
  return arr.filter(function(item, index, arr) {
    //当前元素，在原始数组中的第一个索引==当前索引值，否则返回当前元素
    return arr.indexOf(item, 0) === index;
  });
}
    var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0,
        console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, "NaN", 0, "a", {…}, {…}]
```

> 每次indeOf只会返回第一个值的索引，后面的值索引对不上就-false

8. 递归去重

```
function unique(arr) {
        var array= arr;
        var len = array.length;

    array.sort(function(a,b){   //排序后更加方便去重
        return a - b;
    })

    function loop(index){
        if(index >= 1){
            if(array[index] === array[index-1]){
                array.splice(index,1);
            }
            loop(index - 1);    //递归loop，然后数组去重
        }
    }
    loop(len-1);
    return array;
}
 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0,
console.log(unique(arr))
//[1, "a", "true", true, 15, false, 1, {…}, null, NaN, NaN, "NaN", 0, "a", {…}, undefined]
```

> 和前面循环去重类似的套路

9. Map数据结构去重

```
function arrayNonRepeatfy(arr) {
  let map = new Map();
  let array = new Array();  // 数组用于返回结果
  for (let i = 0; i < arr.length; i++) {
    if(map.has(arr[i])) {  // 如果有该key值
      map.set(arr[i], true);
    } else {
      map.set(arr[i], false);   // 如果没有该key值
      array.push(arr[i]);
    }
  }
  return array ;
}
 var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0,
    console.log(unique(arr))
//[1, "a", "true", true, 15, false, 1, {…}, null, NaN, NaN, "NaN", 0, "a", {…}, undefined]``
```

> 还是遍历判断做值判断

10. reduce+includes

```
function unique(arr){
    return arr.reduce((prev,cur) => prev.includes(cur) ? prev : [...prev,cur],[]);
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, '
console.log(unique(arr));
// [1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}, {…}]
```

> 通过reduce初始化一个空数组，然后每次叠加的时候，通过includes进行范围查找

小结:

除了第一个使用的 Es6语法糖 , 剩下的大部分都是进行循环遍历进行比较.

JS判断数据类型的几种方法

- typeof 使用简单, 但是只适用于判断基础类型的数据
- instanceof 能判断引用类型， 不能检测出基本类型，且不能跨 iframe 使用
- constructor 基本能够判断所有类型，除了 null 和 undefined , 但是 constructor 容易被修改, 也不能跨 iframe 使用.
- toString 能判断所有类型，因此可将其封装成一个全能的 DataType() 判断所有数据类型

```
function DataType(tgt, type) {
    const dataType = Object.prototype.toString.call(tgt).replace(/\[object (\w+)\]/, "$1").toLowerCase();
    return type ? dataType === type : dataType;
}

DataType("young"); // "string"
DataType(20190214); // "number"
DataType(true); // "boolean"
DataType([], "array"); // true
DataType({}, "array"); // false
```

HTTP系列

浏览器存储的方式有哪些

类型	生命周期	存储大小	与服务器通信

类型	生命周期	存储大小	与服务器通信
cookie	一般由服务器生成， 可以设置过期时间	4K	每次都会携带在header， 对于请求性能影响
loca Storage	除非被清理，否则一直存在	5M	不参与
session Storage	页面关闭就清理	5M	不参与
indexedDB	除非被清理, 否则一直存在	无限	不参与

Tip

- cookie原本诞生之初用来储存的，而是用来与服务端通信的，如果需要存取需要字形封装api
- local Storage则自带getItem和setItem方法，使用方便。
- ① local Storage只能存字符串，存取JSON数据需要配合JSON.stringify()和 JSON.parse()
- ② 遇上禁用setItem的浏览器，需要使用try...catch捕获异常

输入URL发生什么

1. DNS域名解析
2. 浏览器与目标服务器建立一条TCP连接（三次握手）
3. 浏览器向服务器发送一条HTTP请求报文
4. 服务器返回给浏览器一条HTTP响应报文
5. 浏览器进行渲染
6. 关闭TCP连接(四次挥手)

浏览器系列

浏览器渲染的步骤

1. HTML解析出DOM Tree
2. CSS解析出Style Rules
3. 两者关联生成Render Tree
4. Layout(布局) 根据 Render Tree 计算每个节点的信息
5. Painting根据计算好的信息进行渲染整个页面

> 浏览器解析文档的过程中，如果遇到 script 标签，会立即解析脚本，停止解析文档（因为 JS 可能会改变 DOM 和 CSS,如果继续解析会造成浪费）。
>
> 如果是外部 script, 会等待脚本下载完成之后在继续解析文档。现在 script 标签增加了 defer 和 async 属性，脚本解析会将脚本中改变 DOM 和 css 的地方> 解析出来，追加到 DOM Tree 和 Style Rules 上

React

组件生命周期

当clock组件第一次被渲染到DOM中的时候，就为其设置一个计时器,这在React中被称为挂载（mount）.

当DOM中clock组件被删除的时候，应该清除计时器。这在React中称为“卸载（unmount）”.

这些叫做“生命周期方法”

一些重要的React生命周期方法

1. componentWillMount() – 在渲染之前执行，在客户端和服务器端都会执行。
2. componentDidMount() – 仅在第一次渲染后在客户端执行。
3. componentWillReceiveProps() – 当从父类接收到 props 并且在调用另一个渲染器之前调用。
4. shouldComponentUpdate() – 根据特定条件返回 true 或 false。如果你希望更新组件，请返回true 否则返回 false。默认情况下，它返回 false。
5. componentWillUpdate() – 在 DOM 中进行渲染之前调用。
6. componentDidUpdate() – 在渲染发生后立即调用。
7. componentWillUnmount() – 从 DOM 卸载组件后调用。用于清理内存空间。

官方文档生命周期说明

挂载

当组件实例被创建并插入DOM中，生命周期调用顺序如下：

- constructor
- static getDerivedStateFromProps()
- render()
- componentDidMount()

> **注意:**
>
> 下述生命周期方法即将过时，在新代码中应该避免使用它们：
>
> UNSAFE_componentWillMount()

更新

当组件的props或state发生变化时触发更新. 组件更新的生命周期调用顺序如下:

- static getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

> **注意:**
>
> 下述方法即将过时，在新代码中应该避免使用它们：
>
> - UNSAFE_componentWillUpdate()
>
> - UNSAFE_componentWillReceiveProps()

卸载

当组件从DOM中移除时会调用如下方法：

- componentWillUnmount()

错误处理

当渲染过程，生命周期，或子组件的构造函数中抛出错误时，会调用如下方法：

- static getDerivedStateFromError()
- componentDidCatch()

笔试综合算法题

setTimeout与promise

```
(function () {
  setTimeout(function () {
    console.log(1);
  }, 1000);
  setTimeout(function () {
    console.log(2);
  }, 0);
  new Promise(function (resolve) {
    console.log(3);
    setTimeout(function () {
      console.log(4);
      resolve();
    }, 0);
    console.log(5);
  }).then(function () {
    console.log(6);
  });
  console.log(7);
})();
//   输出 3572461
```

parseUrl处理

编写parseUrl函数， parseUrl("https://weibo.com/sportschannel?is_all=1");
期望得到结果：

```
{
    source: "https://weibo.com/sportschannel?is_all=1",
    protocol: "https",
    host: "weibo.com",
    port: "",
    query: "?is_all=1",
    params: {is_all: "1"},
    file: "sportschannel",
    hash: "",
    path: "/sportschannel",
    relative: "/sportschannel?is_all=1",
    segments: ["sportschannel"],
}
```

解题过程：

```
function parseUrl(url) {
    var a =  document.createElement('a');
    a.href = url;
    return {
        source: url,
        protocol: a.protocol.replace(':',''),
        host: a.hostname,
        port: a.port,
        query: a.search,
        params: (() => {
            var ret = {}, querys = [];
            var searchQuery = a.search.replace(/^\?/,'').split('&');
            for ( var i = 0;i < searchQuery.length; i++) {
                if (searchQuery[i]) {
                   querys = searchQuery[i].split('=');
                   ret[querys[0]] = querys[1];
                }
            }
            return ret;
        })(),
        file: (a.pathname.match(/\/([^\/?#]+)$/i) || [,''])[1],
        hash: a.hash.replace('#',''),
        path: a.pathname.replace(/^([^\/])/,'/$1'),
        relative: (a.href.match(/tps?:\/\/[^\/]+(.+)/) || [,''])[1],
        segments: a.pathname.replace(/^\//,'').split('/')
   };
}
```

字母异位词分组

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

```
输入: ["eat", "tea", "tan", "ate", "nat", "bat"],
输出:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

注意：

> 所有输入均为小写字母。
> 不考虑答案输出的顺序。

```
var groupAnagrams = function (strs) {
  var newStrs = strs.map((item) => {
    return item.split("").sort().join("");
  });
  var hash = {};
  for (var i = 0, len = newStrs.length; i < len; i++) {
    if (!hash[newStrs[i]]) {
      hash[newStrs[i]] = [];
      hash[newStrs[i]].push(i);
    } else {
      hash[newStrs[i]].push(i);
    }
  }
  var newArr = [];
  Object.keys(hash).forEach((item) => {
    var arrItem = [];
    for (var j = 0; j < hash[item].length; j++) {
      arrItem.push(strs[hash[item][j]]);
    }
    newArr.push(arrItem);
  });
  return newArr;
};
```