JS 的递归遍历

作者：zhujie
原文链接：https://ld246.com/article/1588517121027
来源网站：链滴

JavaScript的递归遍历会经常遇到，适当的运用递归遍历，可以提高代码性质量。

1.某些时候递归能替换for循环

我们先看一下下面2个例子。

```
var arrList = [1,2,3,5,100,500,10000,10000,1000,10000002]
 //for循环测试
 function forTest(){
    console.time("for循环")
    for(let i in arrList){
        console.log(((arrList[i] + arrList[i]) * 5 - arrList[i])/arrList[i])
    }
    console.timeEnd("for循环")
 }
 //递归遍历测试
 function recursive() {
    console.time("递归遍历")
    const testFun = function (i) {
        console.log(((arrList[i] + arrList[i]) * 5 - arrList[i])/arrList[i])
        if(i == arrList.length - 1){
            return
        }
        i++
        testFun(i)
    }
    testFun(0)
    console.timeEnd("递归遍历")
 }
 forTest()
 recursive()
```

运行结果：

```
9
for循环: 15ms
9
9
9
9
9
9
9
9
9
9
递归遍历: 16.000244140625ms
```

可以看到，for循环去遍历一个数组和用递归遍历去遍历同一个数组得到的结果一样，耗时也几乎相同但是写法上有很大区别。

递归特点

每个递归都有一个结束递归的条件，上例中的:if(i == arrList.length - 1){ return }。每一个递归都会函数内部把函数本身调用一次，但是函数在每次运行的时候，都会发生一些变化，用来触发递归的结，上例中，testFun函数在内部调用了自己，并且每次调用i的值会+1，最终触发了结束条件，让递归

束。

2.使用递归，可以轻松实现多级遍历

我们先看下面的代码：

```
var data = [
 {
     name: "所有物品",
     children: [
         {
             name: "水果",
             children: [{name: "苹果", children: [{name: '青苹果'}, {name: '红苹果'}]}]
         },
         {
             name: '主食',
             children: [
                 {name: "米饭", children: [{name: '北方米饭'}, {name: '南方米饭'}]}
             ]
         },
         {
             name: '生活用品',
             children: [
                 {name: "电脑类", children: [{name: '联想电脑'}, {name: '苹果电脑'}]},
                 {name: "工具类", children: [{name: "锄头"}, {name: "锤子"}]},
                 {name: "生活用品", children: [{name: "洗发水"}, {name: "沐浴露"}]}
             ]
         }
  ]
}]
```

某些时候，坑逼后台让我们遍历上面的一个数组，最后在页面上显示没一级的最后一个数据。就是下的数据：

```
青苹果;红苹果;北方米饭;南方米饭;联想电脑;苹果电脑;锄头;锤子;洗发水;沐浴露;
```

我们先用遍历的方式来实现：

```
//普通遍历实现
var forFunction = function () {
    var str = ""
    data.forEach(function(row){
        row.children.forEach(function(row){
            row.children.forEach(function(row){
                row.children.forEach(function(row){
                    str += (row.name + ";")
                })
            })
        })
    })
    console.log(str)
}
forFunction()
//输出：青苹果;红苹果;北方米饭;南方米饭;联想电脑;苹果电脑;锄头;锤子;洗发水;沐浴露;
```

可以看到，前端累死累死写了4个forEach实现了产品要的效果，这时候如果再加点别的什么逻辑，就难弄了。这时候我们尝试用递归的思想实现：

```
//递归遍历实现
var recursiveFunction = function(){
    var str = ''
    const getStr = function(list){
        list.forEach(function(row){
            if(row.children){
                getStr(row.children)
            }else {
                str += row.name + ";"
            }
        })
    }
    getStr(data)
    console.log(str)
}
recursiveFunction()
//输出：青苹果;红苹果;北方米饭;南方米饭;联想电脑;苹果电脑;锄头;锤子;洗发水;沐浴露;
```

可以看到，递归的方式来实现的时候，我们只需要一个for循环，每次遍历接受到的数据，通过判断是还有children，没有就代表是最后一级了，有就继续把children这个list传给函数继续遍历，最后就得了我们想要的数据。

很明显，forEach的遍历的方式能实现多级的遍历，并且数据格式可以灵活一些，但是遍历的层级有，而且对于未知层级的情况下，就无从下手了。

递归遍历，理论上，只要内存够用，你能实现任意层级的遍历，但缺点也很明显，没一个层级里面需有固定的数据格式，否则无法遍历。

3.递归遍历轻松实现多个异步结果的依次输出

我们先看一下下面的需求，需要依次输出1,2,3,4,5，每个输出中间间隔1s。

我们先尝试下面的方式：

```
//常规实现
var forTest = function () {
   console.time("for时间")
    for (let i = 0; i < 5; i++) {
        setTimeout(function () {
            console.log('for循环输出：' + (i + 1))
            if(i+ 1 === 5){
                console.timeEnd("for时间")
            }
        }, 1000 * (i + 1))
    }
}
forTest()
//每隔一秒输出了1,2,3,4,5
```

上面我们用let当前作用域的特点实现了每隔1s输出，其实可以看出，是一起执行的，但是由于间隔时不一样，导致结果是每隔一秒输出的。

我们再用递归的方式实现：

```
//递归遍历实现
var recursiveTest = function(){
   console.time("递归时间")
    const test = function (i) {
        setTimeout(function () {
            i = i + 1
            console.log('递归输出：' + i)
            if(i < 5){
                test(i)
            }else {
                console.timeEnd("递归时间")
            }
        }, 1000)
    }
    test(0)
}
recursiveTest()
//每隔一秒输出1,2,3,4,5
```

这里利用递归实现就很好理解了，在setTimeout里面输出了之后，再开始下一个1s的输出。

最后我们看一下运行截图：

```
递归输出：1
for循环输出：1
for循环输出：2
递归输出：2
for循环输出：3
递归输出：3
for循环输出：4
递归输出：4
for循环输出：5
for时间：5004ms
递归输出：5
递归时间：5009ms
```

通过截图上的耗时来看：递归遍历需要的时间更长，但是更好理解一下，而且不依赖es6的环境。

4.递归遍历实现排序

```
var quickSort = function(arr) {
if (arr.length <= 1) {//如果数组长度小于等于1无需判断直接返回即可
    return arr;
}
var pivotIndex = Math.floor(arr.length / 2);//取基准点
var pivot = arr.splice(pivotIndex, 1)[0];//取基准点的值,splice(index,1)函数可以返回数组中被删除
那个数
var left = [];//存放比基准点小的数组
var right = [];//存放比基准点大的数组
for (var i = 0; i < arr.length; i++){ //遍历数组，进行判断分配
    if (arr[i] < pivot) {
        left.push(arr[i]);//比基准点小的放在左边数组
    } else {
        right.push(arr[i]);//比基准点大的放在右边数组
```

原文链接：JS 的递归遍历

```
    }
}
//递归执行以上操作,对左右两个数组进行操作，直到数组长度为<=1；
return quickSort(left).concat([pivot], quickSort(right));
};

var arr = [14, 50, 80, 7, 2, 2, 11];
console.log(quickSort(arr));
```

这是利用递归实现的快速排序，效率很高，有兴趣的同学可以试试普通的遍历实现，再进行比较。

5.总结

1.很多时候可以用递归代替循环，可以理解为递归是一种特殊的循环，但通常情况下不推荐这样做。

2.递归一般是在函数里面把函数自己给调用一遍，通过每次调用改变条件，来结束循环。

3.递归在数据格式一致，在数据层级未知的情况下，比普通的遍历更有优势。

4.递归在异步的时候，更容易理解，且更容易实现，因为可以在异步的回调里面，调用自己来实现每都能拿到异步的结果再进行其他操作。

5.递归实现的快速排序比普通遍历实现的排序效率更好。

所有的代码都能在GitHub下载：下载。