昵称：ming1025
园龄：5年4个月
粉丝：27
关注：2
+加关注

< 2021年10月 >

日	一	二	三	四	五	六
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签
iOS学习笔记(41)
Welcome-to-Swift(24)
iOS学习笔记、Swift基础部分(8)
软件设计原则(8)
iOS(8)
iOS大神班笔记(3)
MySQL的安装与配置——详细教程(1)
el-select中使用el-tooltip时，在下拉滚动时整个网页会出现滚动条，并抖动(1)
JS树结构操作:查找、遍历、筛选、树结构和列表结构相互转换(1)
js树结构查找节点(1)
更多

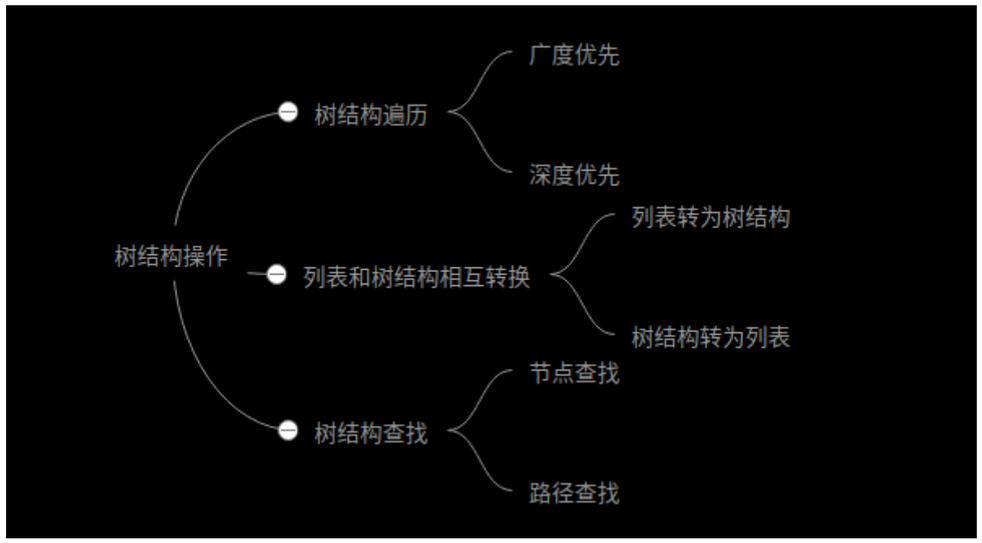
随笔分类
js算法(10)
医疗器械行业知识(2)

随笔档案

ming1025

JS树结构操作:查找、遍历、筛选、树结构和列表结构相互转换

经常有同学问树结构的相关操作，在这里总结一下JS树形结构一些操作的实现思路，并给出了简洁易懂的代码实现。
本文内容结构大概如下：



一、遍历树结构

1. 树结构介绍

JS中树结构一般是类似于这样的结构：

```
let tree = [
  {
    id: '1',
    title: '节点1',
    children: [
      {
        id: '1-1',
        title: '节点1-1'
      },
```

2020年12月(2)
2020年11月(2)
2020年10月(17)
2020年9月(5)
2020年6月(30)
2018年11月(1)
2017年9月(1)
2017年8月(3)
2017年4月(10)
2017年3月(2)
2016年12月(19)
2016年11月(76)

链接

公用的宏

阅读排行榜

1. vue-cli（vue脚手架）超详细教程(146817)
2. iOS开发--字典(NSDictionary)和JSON字符串(NSString)之间互转(97929)
3. 纯CSS画的基本图形（矩形、圆形、三角形、多边形、爱心、八卦等）(31700)
4. Swift UI控件详细介绍(上)（15922)
5. js中的promise详解(11988)

评论排行榜

1. vue-cli（vue脚手架）超详细教程(5)
2. 纯CSS画的基本图形（矩形、圆形、三角形、多边形、爱心、八卦等）(3)
3. Vue组件的渲染更新原理解析(2)
4. vue中ajax请求放在哪个生命周期中(1)
5. iOS开发--字典(NSDictionary)和JSON字符串(NSString)之间互转(1)

推荐排行榜

1. vue-cli（vue脚手架）超详细教程(23)
2. 纯CSS画的基本图形（矩形、圆形、三角形、多边形、爱心、八卦等）(8)
3. js中的promise详解(3)
4. iOS开发--字典(NSDictionary)和JSON字符串(NSString)之间互转(2)
5. MySQL的安装与配置——详细教程(1)

最新评论

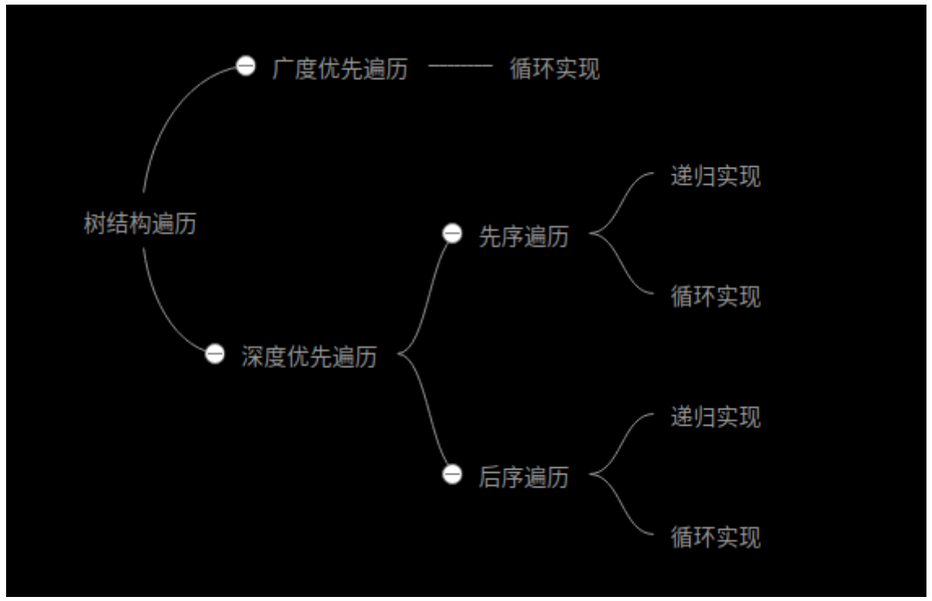
1. Re:Vue组件的渲染更新原理解析

```
    {
      id: '1-2',
      title: '节点1-2'
    }
  ]
},
{
  id: '2',
  title: '节点2',
  children: [
    {
      id: '2-1',
      title: '节点2-1'
    }
  ]
}
]
```

为了更通用，可以用存储了树根节点的列表表示一个树形结构，每个节点的children属性（如果有）是一颗子树，如果没有children属性或者children长度为0，则表示该节点为叶子节点。

2. 树结构遍历方法介绍

树结构的常用场景之一就是遍历，而遍历又分为广度优先遍历、深度优先遍历。其中深度优先遍历是可递归的，而广度优先遍历是非递归的，通常用循环来实现。深度优先遍历又分为先序遍历、后序遍历，二叉树还有中序遍历，实现方法可以是递归，也可以是循环。



广度优先和深度优先的概念很简单，区别如下：

- 深度优先，访问完一颗子树再去访问后面的子树，而访问子树的时候，先访问根再访问根的子树，称为先序遍历；先访问子树再访问根，称为后序遍历。
- 广度优先，即访问树结构的第n+1层前必须先访问完第n层

3. 广度优先遍历的实现

广度优先的思路是，维护一个队列，队列的初始值为树结构根节点组成的列表，重复执行以下步骤直到队列为空：

- 取出队列中的第一个元素，进行访问相关操作，然后将其后代元素（如果有）全部追加到队列最后。

下面是代码实现，类似于数组的forEach遍历，我们将数组的访问操作交给调用者自定义，即一个回调函数：

```
// 广度优先
function treeForeach (tree, func) {
  let node, list = [...tree]
  while (node = list.shift()) {
```

@zhouyaxue 应该是指在render的同时触发了data的getter，从而进行依赖的收集。...

--陈呵呵

2. Re:Vue组件的渲染更新原理解析

您好，我想问一下，

这里面的 touch 指的什么意思呀?

--zhouyaxue

3. Re:vue-cli（vue脚手架）超详细教程

赞

--黄昏见证使徒

4. Re:vue中ajax请求放在哪个生命周期中

真的是这样吗

--小宁同学

5. Re:vue-cli（vue脚手架）超详细教程

赞

--MrTangerine

```
    func(node)
    node.children && list.push(...node.children)
  }
}
```

很简单吧，~,~
用上述数据测试一下看看：

```
treeForeach(tree, node => { console.log(node.title) })
```

输出，可以看到第一层所有元素都在第二层元素前输出：

```
> 节点1
> 节点2
> 节点1-1
> 节点1-2
> 节点2-1
```

4. 深度优先遍历的递归实现

先序遍历，三五行代码，太简单，不过多描述了：

```
function treeForeach (tree, func) {
  tree.forEach(data => {
    func(data)
    data.children && treeForeach(data.children, func) // 遍历子树
  })
}
```

后序遍历，与先序遍历思想一致，代码也及其相似，只不过调换一下节点遍历和子树遍历的顺序：

```
function treeForeach (tree, func) {
  tree.forEach(data => {
    data.children && treeForeach(data.children, func) // 遍历子树
    func(data)
  })
}
```

测试：

```
treeForeach(tree, node => { console.log(node.title) })
```

输出：

```
// 先序遍历
> 节点1
> 节点1-1
> 节点1-2
> 节点2
> 节点2-1

// 后序遍历
> 节点1-1
> 节点1-2
> 节点1
> 节点2-1
> 节点2
```

5. 深度优先循环实现

先序遍历与广度优先循环实现类似，要维护一个队列，不同的是子节点不追加到队列最后，而是加到队列最前面：

```
function treeForeach (tree, func) {
  let node, list = [...tree]
  while (node = list.shift()) {
    func(node)
    node.children && list.unshift(...node.children)
  }
}
```

后序遍历就略微复杂一点，我们需要不断将子树扩展到根节点前面去，（艰难地）执行列表遍历，遍历到某个节点如果它没有子节点或者它的子节点已经扩展到它前面了，则执行访问操作，否则扩展子节点到当前节点前面：

```
function treeForeach (tree, func) {
  let node, list = [...tree], i =  0
  while (node = list[i]) {
    let childCount = node.children ? node.children.length : 0
    if (!childCount || node.children[childCount - 1] === list[i - 1]) {
      func(node)
      i++
    } else {
      list.splice(i, 0, ...node.children)
    }
  }
}
```

二、列表和树结构相互转换

1. 列表转为树

列表结构通常是在节点信息中给定了父级元素的id，然后通过这个依赖关系将列表转换为树形结构，列表结构是类似于：

```
let list = [
  {
    id: '1',
    title: '节点1',
    parentId: '',
  },
  {
    id: '1-1',
    title: '节点1-1',
    parentId: '1'
  },
  {
    id: '1-2',
    title: '节点1-2',
          parentId: '1'
  },
  {
    id: '2',
    title: '节点2',
    parentId: ''
  },
  {
    id: '2-1',
    title: '节点2-1',
        parentId: '2'
```

```
  }
]
```

列表结构转为树结构，就是把所有非根节点放到对应父节点的chilren数组中，然后把根节点提取出来：

```
function listToTree (list) {
  let info = list.reduce((map, node) => (map[node.id] = node, node.children = [], map), {})
  return list.filter(node => {
    info[node.parentId] && info[node.parentId].children.push(node)
    return !node.parentId
  })
}
```

这里首先通过info建立了id=>node的映射，因为对象取值的时间复杂度是O(1)，这样在接下来的找寻父元素就不需要再去遍历一次list了，因为遍历寻找父元素时间复杂度是O(n)，并且是在循环中遍历，则总体时间复杂度会变成O(n^2)，而上述实现的总体复杂度是O(n)。

2. 树结构转列表结构

有了遍历树结构的经验，树结构转为列表结构就很简单了。不过有时候，我们希望转出来的列表按照目录展示一样的顺序放到一个列表里的，并且包含层级信息。使用先序遍历将树结构转为列表结构是合适的，直接上代码:

```
//递归实现
function treeToList (tree, result = [], level = 0) {
  tree.forEach(node => {
    result.push(node)
    node.level = level + 1
    node.children && treeToList(node.children, result, level + 1)
  })
  return result
}

// 循环实现
function treeToList (tree) {
  let node, result = tree.map(node => (node.level = 1, node))
  for (let i = 0; i < result.length; i++) {
    if (!result[i].children) continue
    let list = result[i].children.map(node => (node.level = result[i].level + 1, node))
    result.splice(i+1, 0, ...list)
  }
  return result
}
```

三、树结构筛选

树结构过滤即保留某些符合条件的节点，剪裁掉其它节点。一个节点是否保留在过滤后的树结构中，取决于它以及后代节点中是否有符合条件的节点。可以传入一个函数描述符合条件的节点:

```
function treeFilter (tree, func) {
  // 使用map复制一下节点，避免修改到原树
  return tree.map(node => ({ ...node })).filter(node => {
    node.children = node.children && treeFilter(node.children, func)
    return func(node) || (node.children && node.children.length)
  })
}
```

四、树结构查找

1. 查找节点

查找节点其实就是一个遍历的过程，遍历到满足条件的节点则返回，遍历完成未找到则返回null。类似数组的find方法，传入一个函数用于判断节点是否符合条件，代码如下：

```
function treeFind (tree, func) {
  for (const data of tree) {
    if (func(data)) return data
    if (data.children) {
      const res = treeFind(data.children, func)
      if (res) return res
    }
  }
  return null
}
```

2. 查找节点路径

略微复杂一点，因为不知道符合条件的节点在哪个子树，要用到回溯法的思想。查找路径要使用先序遍历，维护一个队列存储路径上每个节点的id，假设节点就在当前分支，如果当前分支查不到，则回溯。

```
function treeFindPath (tree, func, path = []) {
  if (!tree) return []
  for (const data of tree) {
    path.push(data.id)
    if (func(data)) return path
    if (data.children) {
      const findChildren = treeFindPath(data.children, func, path)
      if (findChildren.length) return findChildren
    }
    path.pop()
  }
  return []
}
```

用上面的树结构测试：

```
let result = treeFindPath(tree, node => node.id === '2-1')
console.log(result)
```

输出：

```
["2","2-1"]
```

3. 查找多条节点路径

思路与查找节点路径相似，不过代码却更加简单：

```
function treeFindPath (tree, func, path = [], result = []) {
  for (const data of tree) {
    path.push(data.id)
    func(data) && result.push([...path])
    data.children && treeFindPath(data.children, func, path, result)
    path.pop()
  }
  return result
}
```

五、结语

对于树结构的操作，其实递归是最基础，也是最容易理解的。递归本身就是循环的思想，所以可以用循环来改写递归。熟练掌握了树结构的查找、遍历，应对日常需求应该是绰绰有余啦。

如果我的内容能对你有所帮助，我就很开心啦！

标签: JS树结构操作:查找、遍历、筛选、树结构和列表结构相互转换

好文要顶 关注我 收藏该文

ming1025
关注 - 2
粉丝 - 27
+加关注

0 推荐 0 反对

« 上一篇： js树结构查找节点
» 下一篇： 禁用h5页面中长按图片弹出的弹层

posted on 2020-09-16 10:05 ming1025 阅读(2834) 评论(0) 编辑 收藏 举报

刷新评论 刷新页面 返回顶部

登录后才能查看或发表评论，立即 登录 或者 逛逛 博客园首页

【推荐】并行超算云面向博客园粉丝推出“免费算力限时申领”特别活动
【推荐】跨平台组态\工控\仿真\CAD 50万行C++源码全开放免费下载！
【推荐】华为全面助力青少年编程教育普及，花瓣少儿编程正式上线

编辑推荐：
· 技术管理进阶——管人还是管事?
· 以终为始：如何让你的开发符合预期
· 五个维度打造研发管理体系
· 不会SQL也能做数据分析？浅谈语义解析领域的机会与挑战
· Spring IoC Container 原理解析

最新新闻：
· 大脑并行处理语音和其它声音（2021-10-22 22:41）
· 黑客把你家网线作“天线”，读取电磁信号就能偷走数据（2021-10-22 22:38）
· 股价屡创新高 科技股“老大哥”微软业绩能否续写辉煌？（2021-10-22 22:12）
· 特斯拉股价创历史新高 市值突破9000亿美元（2021-10-22 21:50）
· Apple Watch Series7的大屏幕，能带给我们多少「大不同」？（2021-10-22 21:22）
» 更多新闻...