本章重点考点介绍

新年快乐~

简单说一说

1. 算法：解决一个问题、一系列问题的一个具体的步骤。
2. 数据结构：实际问题的抽象；把实际问题抽象成为数据结构。（青蛙跳井、接雨水）
3. 讲解顺序：数据结构、排序、双指针、动规（最近常考）、回溯（最近常考，尤其排列组合的题）、贪心、二分。
4. 链表 - React 源码：effect、nextEffect、lastEffect、updateQueue、hook、memoryStack。
5. 非常容易考：DOMTOJSON、JSONTODOM、数组打平。

实现一个 LRU 缓存

概念：LRU -- 最近最少使用（least recently used）。
应用：vue - keep-alive 缓存。
实现：LRUCache，缓存，有一个大小 capacity=2 `const lru = new LRUCache(capacity);`

lru.put(1, 1) // { 1 => 1 }
lru.put(2, 2) // { 1 => 1, 2 => 2 }
lru.get(1) // { 2 => 2, 1 => 1 }
// 1

lru.put(3, 3) // { 1 => 1, 3 => 3 }
lru.get(2) // { 1 => 1, 3 => 3 }
// -1

lru.put(4, 4) // { 3 => 3, 4 => 4 }
lru.get(1) // { 3 => 3, 4 => 4 }
// -1

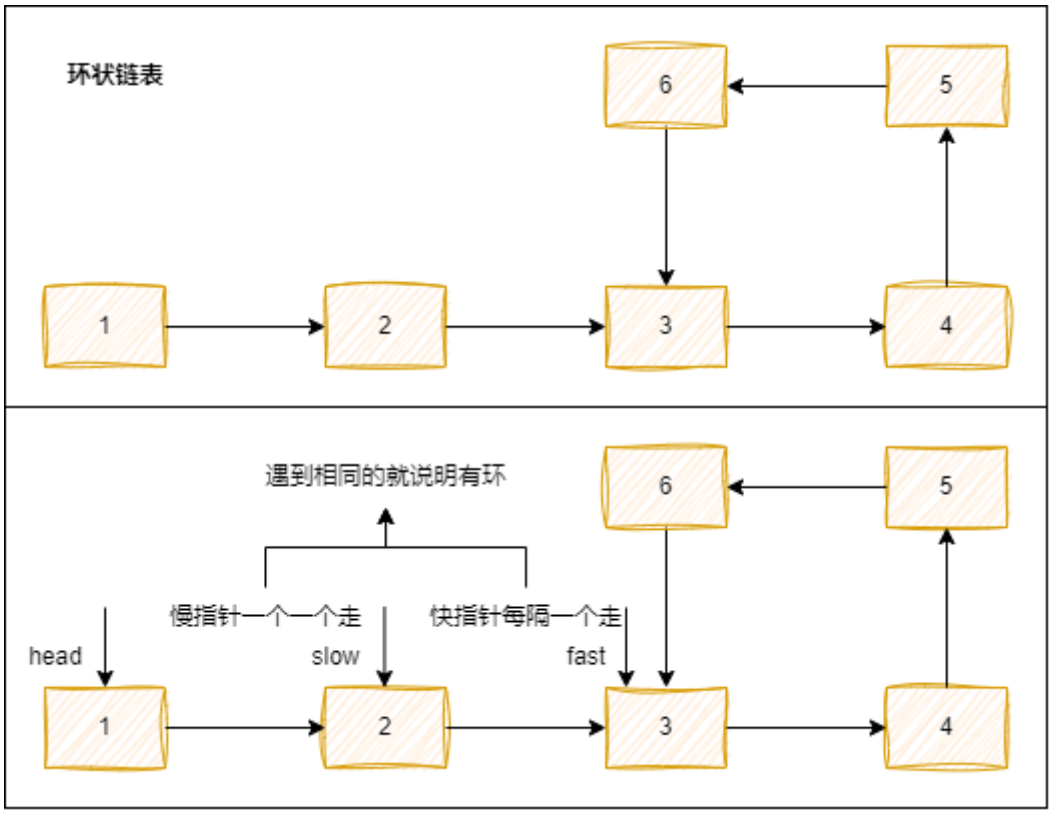
1_lru.js

```
// 函数的 get 和 put 必须以 O(1)的时间复杂度运行。
// get , 我是Hash, Map
// ES6 迭代器 iterator.

const LRUCache = function (capacity) {
  this.cacheQueue = new Map()
  this.capacity = capacity
}

LRUCache.prototype.get = function (key) {
  if (this.cacheQueue.has(key)) {
    // 如果我找到了，我是不是这个 key 对应的 value， 要提升新鲜度。
    const result = this.cacheQueue.get(key)
    this.cacheQueue.delete(key)
    this.cacheQueue.set(key, result)
    console.log(16, this.cacheQueue)
    return result
  }
  console.log(19, this.cacheQueue)
  return -1
}

LRUCache.prototype.put = function (key, value) {
  if (this.cacheQueue.has(key)) {
    this.cacheQueue.delete(key)
  }

  if (this.cacheQueue.size >= this.capacity) {
    // 删除 map 的第一个元素，即最长未使用的。
    this.cacheQueue.set(key, value)
    this.cacheQueue.delete(this.cacheQueue.keys().next().value)
  } else {
    this.cacheQueue.set(key, value)
  }
  console.log(35, this.cacheQueue)
}

// 测试用例:
const lru = new LRUCache(2)
lru.put(1, 1) // { 1 => 1 }
lru.put(2, 2) // { 1 => 1, 2 => 2 }
console.log(lru.get(1)) // { 2 => 2, 1 => 1 }
// 1
lru.put(3, 3) // { 1 => 1, 3 => 3 }
console.log(lru.get(2)) // { 1 => 1, 3 => 3 }
// -1
lru.put(4, 4) // { 3 => 3, 4 => 4 }
console.log(lru.get(1)) // { 3 => 3, 4 => 4 }
```

```
// -1

// 输出:
// 35 Map(1) { 1 => 1 }
// 35 Map(2) { 1 => 1, 2 => 2 }
// 16 Map(2) { 2 => 2, 1 => 1 }
// 1
// 35 Map(2) { 1 => 1, 3 => 3 }
// 19 Map(2) { 1 => 1, 3 => 3 }
// -1
// 35 Map(2) { 3 => 3, 4 => 4 }
// 19 Map(2) { 3 => 3, 4 => 4 }
// -1
```

求环状链表（简单，常见）

思路：快指针每隔一个走，慢指针一个一个走，快慢相遇有环。

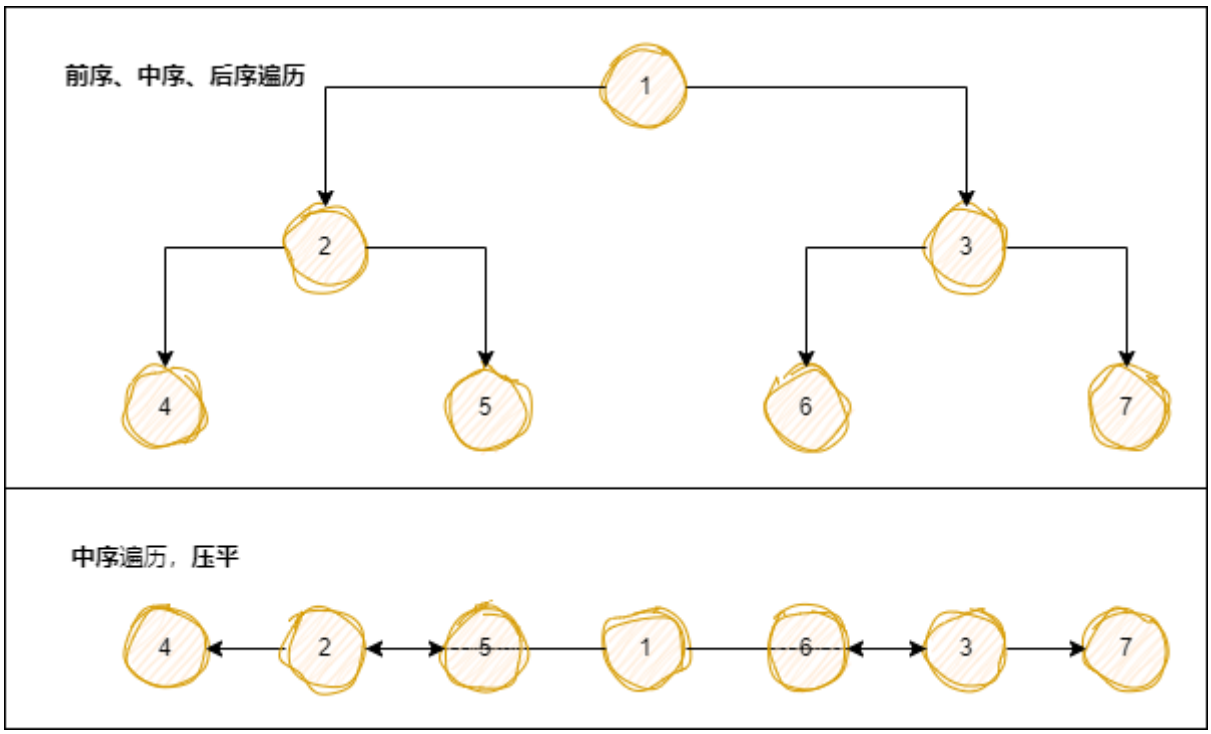


141.环形链表.js

```
const hasCycle = function (head) {
  // 思路：快指针每隔一个走，慢指针一个一个走，快慢相遇有环。
  let fast = (slow = head)
  while (fast && fast.next) {
    fast = fast.next.next
    slow = slow.next
    if (fast === slow) {
      return true
    }
  }
  return false
}
```

二叉树的前序、中序、后序遍历

思路：
前序，根在前；中序，根在中；后序，根在后。
前中后序的前中后就是根的位置。



2_tree.js

```
const treeRoot = {
  val: 1,
  left: {
    val: 2,
    left: {
      val: 4
    },
    right: {
      val: 5
    }
  },
  right: {
    val: 3,
    left: {
      val: 6
    },
    right: {
      val: 7
    }
  }
}

const preOrder = function (node) {
  if (node) {
    console.log(node.val)
    preOrder(node.left)
    preOrder(node.right)
  }
}

const inOrder = function (node) {
  if (node) {
    inOrder(node.left)
    console.log(node.val)
    inOrder(node.right)
  }
}

const postOrder = function (node) {
  if (node) {
    postOrder(node.left)
    postOrder(node.right)
    console.log(node.val)
  }
}

console.log(preOrder(treeRoot)) // 前序: 1 2 4 5 3 6 7
```

```
console.log(inOrder(treeRoot)) // 中序: 4 2 5 1 6 3 7
console.log(postOrder(treeRoot)) // 后序: 4 5 2 6 7 3 1
```

144.二叉树的前序遍历.js

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var preorderTraversal = function (root) {
  // 回溯算法思想
  let res = []
  const preOrder = function (node) {
    if (node == null) return
    res.push(node.val)
    preOrder(node.left)
    preOrder(node.right)
  }
  preOrder(root)
  return res
}
```

94.二叉树的中序遍历.js

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var inorderTraversal = function (root) {
  // 回溯算法思想
  let res = []
  const inOrder = function (node) {
    if (node == null) return
    inOrder(node.left)
    res.push(node.val)
    inOrder(node.right)
  }
  inOrder(root)
  return res
}
```

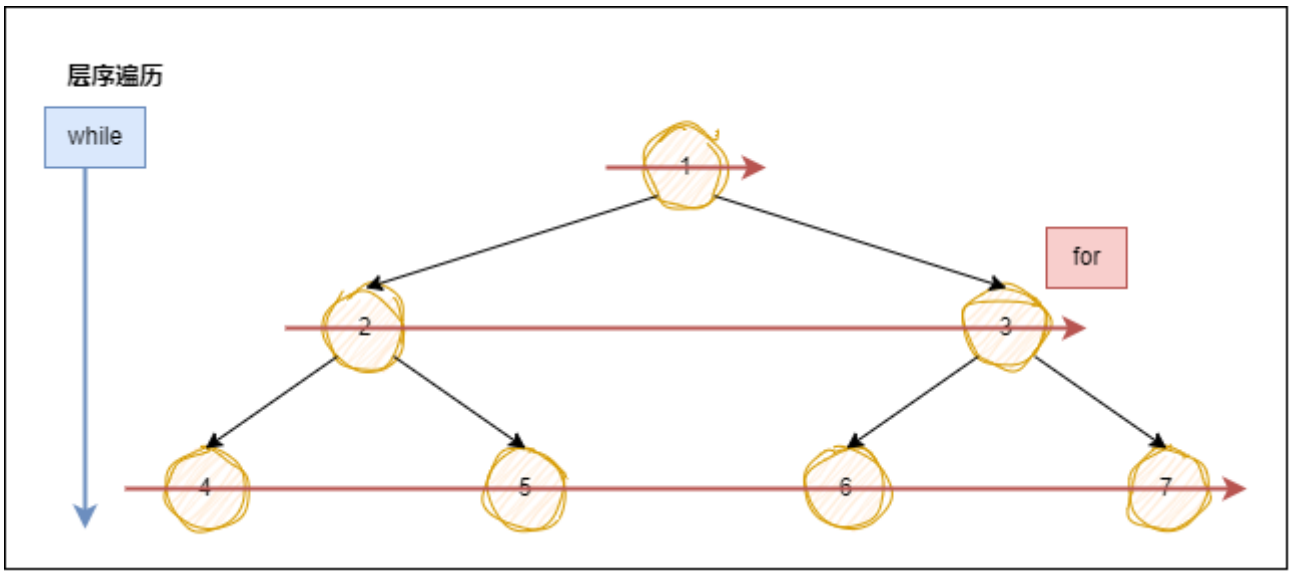
145.二叉树的后序遍历.js

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var postorderTraversal = function (root) {
  // 回溯算法思想
  let res = []
  const postOrder = function (node) {
    if (node == null) return
    postOrder(node.left)
    postOrder(node.right)
    res.push(node.val)
  }
  postOrder(root)
  return res
}
```

树的层序遍历

思路：

1. for 循环时，依次记录上层每个节点值且出队（记录过所以出队表示记录完成），并且在依次记录上层节点的过程中（在前面这个过程中），把下层节点（每个上层节点的左右节点）记录到队列 queue 里。
2. 这样 result 记录上层值，queue 记录下层节点；一直在 while 队列的长度 & 然后 for 循环队列；当队列 queue 为空的时候，也就是层序遍历完成之时。



102.二叉树的层序遍历

> (⭐)

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var levelOrder = function (root) {
  if (!root) return [] // 注意
  let queue = [root]
  let result = []
  while (queue.length) {
    const len = queue.length // 注意
    let level = []
    for (let i = 0; i < len; i++) {
      let node = queue.shift()
      level.push(node.val)
      node.left && queue.push(node.left)
      node.right && queue.push(node.right)
    }
    result.push(level)
  }
  return result
}
```

测试用例:

```
const root = {
  val: 3,
  left: {
    val: 9
  },
  right: {
    val: 20,
    left: {
      val: 15
    },
    right: {
      val: 7
    }
  }
}
console.log(levelOrder(root))
console.log(levelOrder({ val: 1 }))
console.log(levelOrder({}))
```

另一种方式书写：

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var levelOrder = function (root) {
  if (!root) return []
  let queue = [root]
  let result = []

  // 开始循环
  while (queue.length) {
    let tmpQueue = []
    let tmpResult = []
    let len = queue.length
    for (let i = 0; i < len; i++) {
      let node = queue.shift()
      tmpResult.push(node.val)
      node.left && tmpQueue.push(node.left)
      node.right && tmpQueue.push(node.right)
    }
    // 循环完毕后，
    result.push(tmpResult)
    tmpResult = []
    queue = tmpQueue
  }

  return result
}
```

获取二叉树的层级

法 I：二叉树的层级，我可以直接 return 上面层序遍历的 length.

法 II：

104.二叉树的最大深度

```
var maxDepth = function (root) {
  if (!root) return 0
  return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1
}
```

实现 类数组转数组

3_arr.js

```
const arrayLike = document.querySelectorAll('div')

// 1.扩展运算符
const arr = [...arrayLike]
// 2.prototype
Array.prototype.slice.call(arrayLike)
Array.prototype.concat.apply([], arrayLike)
Array.apply(null, arrayLike) // Array 是构造函数
// 3.JS 权威指南 7.1.5 Array.from() P146
Array.from(arrayLike)

// ------------------------------------------

function test() {
  // console.log(typeof arguments)
  let args = arguments
  const c = Array.apply(null, args)
  console.log(c, args)
}
test(1, 2, 3, 4)
```

实现 DOM 转 JSON（经典）大厂概率高（简单，常考）

关键点：在于正向遍历

F12 -> sources -> Snippets -> New Snippet -> dom2json

```
const dom = document.getElementById('wrapper')

function dom2json(dom) {
  let obj = {}
  obj.name = dom.tagName // dom.nodeName 范围更大
  obj.children = []
  dom.childNodes.forEach((child) => obj.children.push(dom2json(child)))

  return obj
}

dom2json(dom)
```

实现 JSON 转 DOM（经典）大厂概率高（考的少一点，可

能遇到）

json2dom.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>json2dom</title>
  </head>

  <body>
    <div id="root"></div>
  </body>
  <script>
    window.onload = function () {
      console.log('on load')
      const json = {
        tag: 'div',
        attrs: {
          id: 'app',
          className: 'app'
        },
        children: [
          {
            tag: 'ul',
            children: [
              { tag: 'li', children: ['list 1'] },
              { tag: 'li', children: ['list 2'] },
              { tag: 'li', children: ['list 3'] },
              { tag: 'li', children: ['list 4'] },
              { tag: 'li', children: ['list 5'] }
            ]
          }
        ]
      }

      function json2dom(vnode) {
        if (typeof vnode === 'string' || typeof vnode === 'number') {
          return document.createTextNode(String(vnode))
        } else {
          const __dom = document.createElement(vnode.tag)
          if (vnode.attrs) {
            Object.entries(vnode.attrs).forEach(([key, value]) => {
              if (key === 'className') {
                __dom[key] = value
              } else {
```

```
                __dom.setAttribute(key, value)
              }
            })
          }
          vnode.children.forEach((child) => __dom.appendChild(json2dom(child)))
          return __dom
        }
      }

      const app = json2dom(json)
      const root = document.getElementById('root')
      root.appendChild(app)
    }
  </script>
  <style>
    .app {
      background-color: yellowgreen;
    }
  </style>
</html>
```

实现 树转数组（经典）大厂概率高

treeToList

> 4_treeAndList.js

```
const root = [
  {
    id: 1,
    text: '根节点',
    // parentId: 0,
    children: [
      {
        id: 2,
        text: '一级节点1'
        // parentId: 1
      },
      {
        id: 3,
        text: '一级节点2',
        // parentId: 1,
        children: [
          {
            id: 5,
            text: '二级节点2-1'
            // parentId: 3
          },
          {
            id: 6,
            text: '二级节点2-2'
            // parentId: 3
          },
          {
            id: 7,
            text: '二级节点2-3'
            // parentId: 3
          }
        ]
      },
      {
        id: 4,
        text: '一级节点3'
        // parentId: 1
      }
    ]
  }
]

function treeToList(root) {
  let res = []

  const dfs = function (data, parentId) {
    data.forEach((item) => {
      if (item.children) {
        dfs(item.children, item.id)
        delete item.children
```

```
      }
      item.parentId = parentId
      res.push(item)
    })
  }

  dfs(root, 0)

  return res
}

console.log(treeToList(root))
```

实现 数组转树（经典）大厂概率高（难，链表相关）

listToTree

> 4_treeAndList.js

```
const list = [
  { id: 2, text: '一级节点1', parentId: 1 },
  { id: 5, text: '二级节点2-1', parentId: 3 },
  { id: 6, text: '二级节点2-2', parentId: 3 },
  { id: 7, text: '二级节点2-3', parentId: 3 },
  { id: 3, text: '一级节点2', parentId: 1 },
  { id: 4, text: '一级节点3', parentId: 1 },
  { id: 1, text: '根节点', parentId: 0 }
]

function listToTree(data) {
  let deps = {}
  let result = []

  // 依赖收集一遍。
  data.forEach((item) => {
    deps[item.id] = item
  })

  // 设置孩子
  for (let i in deps) {
    // 不是根节点
    if (deps[i].parentId !== 0) {
      // 项item的parentId对应到的deps无children，即子节点的父节点没有孩子
      if (!deps[deps[i].parentId].children) {
        deps[deps[i].parentId].children = [] // 给父节点弄children做准备
      }
      deps[deps[i].parentId].children.push(deps[i]) //  给父节点的children推子节点
    } else {
      result.push(deps[i]) // 根节点放到结果里
    }
  }

  return result
}

console.log(JSON.stringify(listToTree(list)))
```

实现 数组打平（经典）大厂概率高（简单）

flattenArray

> 5_flatten.js

```
// 数组打平
const arr = [1, 2, 3, [4, 5, [6, 7, 8], 9, 20]]

function flattenArray(arr) {
  // if (!arr.length) return
  if (!Array.isArray(arr)) return
  return arr.reduce(
    (pre, cur) =>
      Array.isArray(cur) ? [...pre, ...flattenArray(cur)] : [...pre, cur],
    []
  )
}

console.log(flattenArray(arr))
```

实现 对象打平（经典）大厂概率高

flattenObject

> 5_flatten.js

```
// 对象打平
const obj = {
  a: {
    b: {
      c: 1,
      d: 2,
      e: 3
    }
  }
}
// -->
// {
//   'a.b.c': 1,
//   'a.b.d': 2,
//   'a.b.e': 3
// }

function flattenObject(obj) {
  if (typeof obj !== 'object' || obj == null) return
  // let cost = 0 // 开销
  let res = {}
  const dfs = function (cur, prefix) {
    // for (let k in cur) {
    //   // cost++
    //   if (typeof cur[k] === 'object' && cur[k] != null) {
    //     dfs(cur[k], `${prefix}${prefix ? '.' : ''}${k}`)
    //   } else {
    //     res[`${prefix}.${k}`] = cur[k]
    //   }
    // }
    if (typeof cur === 'object' && cur != null) {
      for (let k in cur) {
        // cost++
        dfs(cur[k], `${prefix}${prefix ? '.' : ''}${k}`)
      }
    } else {
      res[prefix] = cur
    }
  }
  dfs(obj, '')
  // console.log(cost)
  return res
}

// console.time('flattenObject')
console.log(flattenObject(obj))
// console.timeEnd('flattenObject')
```

总结

剑指 Offer 简单中等

力扣 top100 简单中等

- DOM 转 JSON
- JSON 转 DOM
- 树转数组
- 数组转树
- 数组打平
- 对象打平