- commonjs: require / module.exports / exports
- amd: require / defined

require与import的区别

- require 支持 动态导入， import 不支持， 正在提案 (babel 下可支持)
- require 是 同步 导入， impor t属于 异步 导入
- require 是 值拷贝， 导出值变化不会影响导入值； import 指向 内存地址， 导入值会随导出值而变化

17. 防抖与节流

> 防抖与节流函数是一种最常用的 高频触发优化方式， 能对性能有较大的帮助。

- 防抖 (debounce):将多次高频操作优化为只在最后一次执行， 通常使用的场景是：用户输入， 只需再输入完成后做一次输入校验即可。

```js
function debounce(fn, wait, immediate) {
    let timer = null

    return function() {
        let args = arguments
        let context = this

        if (immediate && !timer) {
            fn.apply(context, args)
        }

        if (timer) clearTimeout(timer)
        timer = setTimeout(() => {
            fn.apply(context, args)
        }, wait)
    }
}
```

- 节流(throttle):每隔一段时间后执行一次，也就是降低频率，将高频操作优化成低频操作，通常使用场景: 滚动条事件 或者 resize 事件， 通常每隔 100~500 ms 执行一次即可。

```js
function throttle(fn, wait, immediate) {
    let timer = null
    let callNow = immediate
```

```
        return function() {
            let context = this,
                args = arguments

            if (callNow) {
                fn.apply(context, args)
                callNow = false
            }

            if ( !timer) {
                timer = setTimeout(() => {
                    fn.apply(context, args)
                    timer = null
                }, wait)
            }
        }
    }
```

18. 函数执行改变this

- 由于 JS 的设计原理: 在函数中， 可以引用运行环境中的变量 。因此就需要一个机制来让我们可以在函数体内部获取当前的运行环境， 这便是 `this` 。

> 因此要明白 `this` 指向， 其实就是要搞清楚 函数的运行环境，说人话就是，谁调用了函数 。例如

- `obj.fn()` ，便是 `obj` 调用了函数， 既函数中的 `this === obj`
- `fn()` ， 这里可以看成 `window.fn()` ， 因此 `this === window`

> 但这种机制并不完全能满足我们的业务需求， 因此提供了三种方式可以手动修改 `this` 的指向:

- `call: fn.call(target, 1, 2)`
- `apply: fn.apply(target, [1, 2])`
- `bind: fn.bind(target)(1,2)`

19. ES6/ES7

> 由于 Babel 的强大和普及，现在 ES6/ES7 基本上已经是现代化开发的必备了。通过新的语法糖， 能让代码整体更为简洁和易读。

声明

- let / const：块级作用域、不存在变量提升、暂时性死区、不允许重复声明
- const：声明常量， 无法修改

解构赋值

class / extend: 类声明与继承

Set / Map: 新的数据结构

异步解决方案:

- Promise 的使用与实现
- generator：
  - yield：暂停代码
  - next()：继续执行代码

```js
function* helloWorld() {
  yield 'hello';
  yield 'world';
  return 'ending';
}

const generator = helloWorld();

generator.next()  // { value: 'hello', done: false }

generator.next()  // { value: 'world', done: false }

generator.next()  // { value: 'ending', done: true }

generator.next()  // { value: undefined, done: true }
```

> await / async：是 generator 的语法糖， babel 中是基于 promise 实现。

```js
async function getUserByAsync() {
   let user = await fetchUser();
   return user;
}

const user = await getUserByAsync()
console.log(user)
```

20. AST

> 抽象语法树（Abstract Syntax Tree），是将代码逐字母解析成 树状对象 的形式。这是语言之间的转换、代码语法检查，代码风格检查，代码格式化，代码高亮，代码错误提示，代码自动补全等等的基础。例如:

```js
function square(n){
    return n * n
}
```

> 通过解析转化成的AST如下图:

21. babel编译原理

- babylon 将 ES6/ES7 代码解析成 AST
- babel-traverse 对 AST 进行遍历转译，得到新的 AST
- 新 AST 通过 babel-generator 转换成 ES5

22. 函数柯里化

> 在一个函数中，首先填充几个参数，然后再返回一个新的函数的技术，称为函数的柯里化。通常可用于在不侵入函数的前提下，为函数 预置通用参数，供多次重复调用。

```js
const add = function add(x) {
    return function (y) {
        return x + y
    }
}

const add1 = add(1)

add1(2) === 3
add1(20) === 21
```

23. 数组(array)

- map：遍历数组， 返回回调返回值组成的新数组
- forEach：无法 break ， 可以用 try/catch 中 throw new Error 来停止
- filter：过滤
- some：有一项返回 true ，则整体为 true
- every：有一项返回 false ，则整体为 false
- join：通过指定连接符生成字符串
- push / pop：末尾推入和弹出， 改变原数组， 返回推入/弹出项
- unshift / shift：头部推入和弹出， 改变原数组， 返回操作项
- sort(fn) / reverse：排序与反转， 改变原数组
- concat：连接数组，不影响原数组， 浅拷贝
- slice(start, end)：返回截断后的新数组，不改变原数组
- splice(start, number, value...)：返回删除元素组成的数组， value 为插入项， 改变原数组
- indexOf / lastIndexOf(value, fromIndex)：查找数组项， 返回对应的下标
- reduce / reduceRight(fn(prev, cur) ， defaultPrev)：两两执行， prev 为上次化简函数的 return 值， cur 为当前值(从第二项开始)

数组乱序：

```js
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
arr.sort(function () {
    return Math.random() - 0.5;
});
```

数组拆解: flat: [1,[2,3]] --> [1, 2, 3]

```js
Array.prototype.flat = function() {
    this.toString().split(',').map(item => +item )
}
```

三、浏览器

1. 跨标签页通讯

> 不同标签页间的通讯，本质原理就是去运用一些可以 共享的中间介质， 因此比较常用的有以下方法:

- 通过父页面 `window.open()` 和子页面 `postMessage`
  - 异步下， 通过 `window.open('about: blank')` 和 `tab.location.href = '*'`
- 设置同域下共享的 `localStorage` 与监听 `window.onstorage`
  - 重复写入相同的值无法触发
  - 会受到浏览器隐身模式等的限制
- 设置共享 `cookie` 与不断轮询脏检查(`setInterval`)
- 借助服务端或者中间层实现

2. 浏览器架构

- 用户界面
- 主进程
- 内核
  - 渲染引擎
  - `JS` 引擎
    - 执行栈
- 事件触发线程
  - 消息队列
    - 微任务
    - 宏任务
- 网络异步线程
- 定时器线程

3. 浏览器下事件循环(Event Loop)

> 事件循环是指:执行一个宏任务，然后执行清空微任务列表，循环再执行宏任务， 再清微任务列表

- 微任务 `microtask(jobs): promise / ajax / Object.observe` (该方法已废弃)
- 宏任务 `macrotask(task): setTimout / script / IO / UI Rendering`

4. 从输入 url 到展示的过程

- `DNS` 解析
- `TCP` 三次握手
- 发送请求，分析 `url` ，设置请求报文(头， 主体)
- 服务器返回请求的文件 (`html`)
- 浏览器渲染
  - `HTML parser --> DOM Tree`
    - 标记化算法， 进行元素状态的标记
    - `dom` 树构建
- `CSS parser --> Style Tree`
  - 解析 `css` 代码，生成样式树
- `attachment --> Render Tree`
  - 结合 dom树 与 style树，生成渲染树
- `layout` : 布局
- `GPU painting` : 像素绘制页面

5. 重绘与回流

> 当元素的样式发生变化时， 浏览器需要触发更新， 重新绘制元素 。这个过程中，有两种类型的操作， 即重绘与回流。

> - 重绘(repaint): 当元素样式的改变不影响布局时， 浏览器将使用重绘对元素进行更新，此时由于只需要UI层面的重新像素绘制， 因此 损耗较少
> - 回流(reflow): 当元素的尺寸 、结构或触发某些属性时， 浏览器会重新渲染页面，称为回流 。此时， 浏览器需要重新经过计算，计算后还需要重新页面布局， 因此是较重的操作 。会触发回流的操作:

- 页面初次渲染
- 浏览器窗口大小改变

- 元素尺寸、位置、内容发生改变
- 元素字体大小变化
- 添加或者删除可见的 dom 元素
- 激活 CSS 伪类（例如： :hover ）
- 查询某些属性或调用某些方法
  - clientWidth、clientHeight、clientTop、clientLeft
  - offsetWidth、offsetHeight、offsetTop、offsetLeft
  - scrollWidth、scrollHeight、scrollTop、scrollLeft
  - getComputedStyle()
  - getBoundingClientRect()
  - scrollTo()

> 回流必定触发重绘， 重绘不一定触发回流。重绘的开销较小， 回流的代价较高。

最佳实践:

css

- 避免使用 table 布局
- 将动画效果应用到 position 属性为 absolute 或 fixed 的元素上

javascript

- 避免频繁操作样式， 可汇总后统一 一次修改
- 尽量使用 class 进行样式修改
- 减少 dom 的增删次数， 可使用 字符串 或者 documentFragment 一次性插入
- 极限优化时，修改样式可将其 display: none 后修改
- 避免多次触发上面提到的那些会触发回流的方法， 可以的话尽量用 变量存住

6. 存储

> 我们经常需要对业务中的一些数据进行存储， 通常可以分为 短暂性存储 和 持久性储存。

- 短暂性的时候， 我们只需要将数据存在内存中， 只在运行时可用
- 持久性存储， 可以分为 浏览器端 与 服务器端
  - 浏览器:

- cookie：通常用于存储用户身份，登录状态等
  - http 中自动携带， 体积上限为 4K ， 可自行设置过期时间
- localStorage / sessionStorage：长久储存/窗口关闭删除， 体积限制为 4~5M
- indexDB

- 服务器:
  - 分布式缓存 redis
  - 数据库

7. Web Worker

> 现代浏览器为 JavaScript 创造的 多线程环境 。可以新建并将部分任务分配到 worker 线程并行运行，两个线程可 独立运行， 互不干扰， 可通过自带的消息机制 相互通信。

基本用法:

```js
// 创建 worker
const worker = new Worker( 'work.js');

// 向主进程推送消息
worker.postMessage( 'Hello World');

// 监听主进程来的消息
worker.onmessage = function (event) {
  console.log( 'Received message ' + event.data);
}
```

限制:

- 同源限制
- 无法使用 document / window / alert / confirm
- 无法加载本地资源

8. 内存泄露

- 意外的全局变量: 无法被回收
- 定时器: 未被正确关闭， 导致所引用的外部变量无法被释放
- 事件监听: 没有正确销毁 (低版本浏览器可能出现)
- 闭包: 会导致父级中的变量无法被释放

- `dom` 引用：`dom` 元素被删除时， 内存中的引用未被正确清空

> 可用 `chrome` 中的 `timeline` 进行内存标记， 可视化查看内存的变化情况，找出异常点。

四、服务端与网络

1. http/https 协议

1.0 协议缺陷:

- 无法复用链接， 完成即断开， 重新慢启动和 `TCP 3` 次握手
- `head of line blocking` : 线头阻塞， 导致请求之间互相影响

1.1 改进:

- 长连接(默认 `keep-alive`)， 复用
- `host` 字段指定对应的虚拟站点
- 新增功能:
  - 断点续传
  - 身份认证
  - 状态管理
  - `cache` 缓存
    - `Cache-Control`
    - `Expires`
    - `Last-Modified`
    - `Etag`

2.0:

- 多路复用
- 二进制分帧层: 应用层和传输层之间
- 首部压缩
- 服务端推送

https: 较为安全的网络传输协议

- 证书(公钥)
- `SSL` 加密
- 端口 `443`

TCP:

- 三次握手
- 四次挥手
- 滑动窗口: 流量控制
- 拥塞处理
  - 慢开始
  - 拥塞避免
  - 快速重传
  - 快速恢复

缓存策略: 可分为 强缓存 和 协商缓存

- `Cache-Control/Expires` : 浏览器判断缓存是否过期， 未过期时， 直接使用强缓存，`Cache-Control` 的 `max-age` 优先级高于 `Expires`
- 当缓存已经过期时，使用协商缓存
  - 唯一标识方案: `Etag` (`response` 携带) & `If-None-Match` (`request` 携带，上一次返回的 `Etag`): 服务器判断资源是否被修改
  - 最后一次修改时间: `Last-Modified(response) & If-Modified-Since` (`request` , 上一次返回的 `Last-Modified`)
    - 如果一致，则直接返回 304 通知浏览器使用缓存
    - 如不一致，则服务端返回新的资源
- `Last-Modified` 缺点:
  - 周期性修改，但内容未变时，会导致缓存失效
  - 最小粒度只到 `s` ， `s` 以内的改动无法检测到
- `Etag` 的优先级高于 `Last-Modified`

2. 常见状态码

- `1xx` : 接受， 继续处理
- `200` : 成功， 并返回数据
- `201` : 已创建
- `202` : 已接受
- `203` : 成为，但未授权
- `204` : 成功， 无内容
- `205` : 成功， 重置内容
- `206` : 成功， 部分内容
- `301` : 永久移动， 重定向
- `302` : 临时移动， 可使用原有URI
- `304` : 资源未修改， 可使用缓存

- 305 ：需代理访问
- 400 ：请求语法错误
- 401 ：要求身份认证
- 403 ：拒绝请求
- 404 ：资源不存在
- 500 ：服务器错误

3. get / post

- get ：缓存 、请求长度受限 、会被历史保存记录
  - 无副作用(不修改资源)，幂等(请求次数与资源无关)的场景
- post ：安全 、大数据 、更多编码类型

4. Websocket

> Websocket 是一个 持久化的协议， 基于 http ， 服务端可以 主动 push

兼容:

> - FLASH Socket
> - 长轮询： 定时发送 ajax
> - long poll ： 发送 --> 有消息时再 response

- new WebSocket(url)
- ws.onerror = fn
- ws.onclose = fn
- ws.onopen = fn
- ws.onmessage = fn
- ws.send()

5. TCP三次握手

> 建立连接前，客户端和服务端需要通过握手来确认对方:

- 客户端发送 `syn` (同步序列编号) 请求， 进入 `syn_send` 状态， 等待确认
- 服务端接收并确认 `syn` 包后发送 `syn+ack` 包， 进入 `syn_recv` 状态
- 客户端接收 `syn+ack` 包后，发送 `ack` 包，双方进入 `established` 状态

6. TCP四次挥手

- 客户端 -- FIN --> 服务端， FIN—WAIT
- 服务端 -- ACK --> 客户端， CLOSE-WAIT
- 服务端 -- ACK,FIN --> 客户端， LAST-ACK
- 客户端 -- ACK --> 服务端， CLOSED

7. Node 的 Event Loop: 6个阶段

- `timer` 阶段: 执行到期的 `setTimeout / setInterval` 队列回调
- `I/O` 阶段: 执行上轮循环残流的 `callback`
- `idle , prepare`
- `poll` : 等待回调
  - 1. 执行回调
  - 2. 执行定时器
    - 如有到期的 `setTimeout / setInterval` ， 则返回 `timer` 阶段
    - 如有 `setImmediate` ，则前往 `check` 阶段
- `check`
  - 执行 `setImmediate`
- `close callbacks`

8. 跨域

- `JSONP` : 利用 `<script>` 标签不受跨域限制的特点， 缺点是只能支持 `get` 请求

```js
function jsonp(url, jsonpCallback, success) {
  const script = document.createElement('script')
  script.src = url
  script.async = true
  script.type = 'text/javascript'
  window[jsonpCallback] = function(data) {
    success && success(data)
  }
  document.body.appendChild(script)
}
```

- 设置 CORS: Access-Control-Allow-Origin: *
- postMessage

9. 安全

- XSS 攻击: 注入恶意代码
  - cookie 设置 httpOnly
  - 转义页面上的输入内容和输出内容
- CSRF：跨站请求伪造， 防护:
  - get 不修改数据
  - 不被第三方网站访问到用户的 cookie
  - 设置白名单，不被第三方网站请求
  - 请求校验

五、框架：Vue

1. nextTick

> 在下次 dom 更新循环结束之后执行延迟回调， 可用于获取更新后的 dom 状态

- 新版本中默认是 mincrotasks ， v-on 中会使用 macrotasks
- macrotasks 任务的实现:
  - setImmediate / MessageChannel / setTimeout

2. 生命周期

init

- initLifecycle/Event ， 往vm上挂载各种属性
- callHook: beforeCreated：实例刚创建
- initInjection/initState：初始化注入和 data 响应性
- created: 创建完成，属性已经绑定， 但还未生成真实 dom、
- 进行元素的挂载： $el / vm.$mount()
- 是否有 template：解析成 render function
  - *.vue 文件: vue-loader 会将 <template> 编译成 render function
- beforeMount：模板编译/挂载之前
- 执行 render function ，生成真实的 dom ， 并替换到 dom tree 中

- mounted：组件已挂载

update

- 执行 diff 算法， 比对改变是否需要触发 UI 更新
- flushScheduleQueue
- watcher.before：触发 beforeUpdate 钩子 - watcher.run()：执行 watcher 中的 notify， 通知所有依赖项更新UI
- 触发 updated 钩子: 组件已更新
- actived / deactivated(keep-alive)：不销毁， 缓存， 组件激活与失活
- destroy
  - beforeDestroy：销毁开始
  - 销毁自身且递归销毁子组件以及事件监听
    - remove()：删除节点
    - watcher.teardown()：清空依赖
    - vm.$off()：解绑监听
  - destroyed：完成后触发钩子

> 上面是vue的声明周期的简单梳理，接下来我们直接以代码的形式来完成vue的初始化

```js
new Vue({})

// 初始化Vue实例
function _init() {
     // 挂载属性
    initLifeCycle(vm)
    // 初始化事件系统，钩子函数等
    initEvent(vm)
    // 编译slot、vnode
    initRender(vm)
    // 触发钩子
    callHook(vm, 'beforeCreate')
    // 添加inject功能
    initInjection(vm)
    // 完成数据响应性 props/data/watch/computed/methods
    initState(vm)
    // 添加 provide 功能
    initProvide(vm)
    // 触发钩子
    callHook(vm, 'created')
```

```
     // 挂载节点
    if (vm.$options.el) {
        vm.$mount(vm.$options.el)
    }
}

// 挂载节点实现
function mountComponent(vm) {
     // 获取 render function
    if ( !this.options.render) {
        // template to render
        // Vue.compile = compileToFunctions
        let { render } = compileToFunctions()
        this.options.render = render
    }
    // 触发钩子
    callHook( 'beforeMounte')
    // 初始化观察者
    // render 渲染 vdom,
    vdom = vm.render()
    // update: 根据 diff 出的 patchs 挂载成真实的 dom
    vm._update(vdom)
    // 触发钩子
    callHook(vm, 'mounted')
}

// 更新节点实现
funtion queueWatcher(watcher) {
    nextTick(flushScheduleQueue)
}

// 清空队列
function flushScheduleQueue() {
     // 遍历队列中所有修改
    for(){
        // beforeUpdate
        watcher.before()

        // 依赖局部更新节点
        watcher.update()
        callHook( 'updated')
    }
}

// 销毁实例实现
Vue.prototype.$destory = function() {
```

```
        // 触发钩子
        callHook(vm, 'beforeDestory')
        // 自身及子节点
        remove()
        // 删除依赖
        watcher.teardown()
        // 删除监听
        vm.$off()
        // 触发钩子
        callHook(vm, 'destoryed')
    }
```

3. Proxy 相比于 defineProperty 的优势

- 数组变化也能监听到
- 不需要深度遍历监听

```
let data = { a: 1 }
let reactiveData = new Proxy(data, {
    get: function(target, name){
        // ...
    },
    // ...
})
```

4. vue-router

mode

- `hash`
- `history`

跳转

- `this.$router.push()`
- `<router-link to=""></router-link>`

占位

```
<router-view></router-view>
```

5. vuex

- `state`：状态中心
- `mutations`：更改状态
- `actions`：异步更改状态
- `getters`：获取状态
- `modules`：将 `state` 分成多个 `modules`，便于管理

1. 合并两个有序链表

题目描述

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例：

```
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

前置知识

- 递归
- 链表

思路

本题可以使用递归来解，将两个链表头部较小的一个与剩下的元素合并，并返回排好序的链表头，当两条链表中的一条为空时终止递归。

关键点

- 掌握链表数据结构
- 考虑边界情况

代码

JS Code:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
```

```
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
const mergeTwoLists = function (l1, l2) {
  if (l1 === null) {
    return l2;
  }
  if (l2 === null) {
    return l1;
  }
  if (l1.val < l2.val) {
    l1.next = mergeTwoLists(l1.next, l2);
    return l1;
  } else {
    l2.next = mergeTwoLists(l1, l2.next);
    return l2;
  }
};
```

复杂度分析

M、N 是两条链表 l1、l2 的长度

- 时间复杂度：$O(M+N)$
- 空间复杂度：$O(M+N)$

扩展

- 你可以使用迭代的方式求解么?

迭代的 CPP 代码如下：

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* a, ListNode* b) {
        ListNode head, *tail = &head;
        while (a && b) {
            if (a->val <= b->val) {
                tail->next = a;
                a = a->next;
            } else {
                tail->next = b;
```

```
                b = b->next;
            }
            tail = tail->next;
        }
        tail->next = a ? a : b;
        return head.next;
    }
};
```

迭代的 JS 代码如下：

```
var mergeTwoLists = function (l1, l2) {
  const prehead = new ListNode(-1);

  let prev = prehead;
  while (l1 != null && l2 != null) {
    if (l1.val <= l2.val) {
      prev.next = l1;
      l1 = l1.next;
    } else {
      prev.next = l2;
      l2 = l2.next;
    }
    prev = prev.next;
  }
  prev.next = l1 === null ? l2 : l1;

  return prehead.next;
};
```

2. 括号生成

题目描述

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

示例：

```
输入：n = 3
输出：[
       "((()))",
       "(()())",
       "(())()",
```

```
      "()(())",
      "()()()"
    ]
```

前置知识

- DFS
- 回溯法

思路

本题是 20. 有效括号 的升级版。

由于我们需要求解所有的可能， 因此回溯就不难想到。回溯的思路和写法相对比较固定，并且回溯的优化手段大多是剪枝。

不难想到， 如果左括号的数目小于右括号，我们可以提前退出，这就是这道题的剪枝。 比如 `())....`，后面就不用看了，直接退出即可。回溯的退出条件也不难想到，那就是：

- 左括号数目等于右括号数目
- 左括号数目 + 右括号数目 = 2 * n

由于我们需要剪枝， 因此必须从左开始遍历。（WHY？）

因此这道题我们可以使用深度优先搜索(回溯思想)，从空字符串开始构造，做加法， 即 dfs(左括号数, 右括号数目, 路径)， 我们从 dfs(0, 0, '') 开始。

伪代码：

```
res = []
def dfs(l, r, s):
   if l > n or r > n: return
   if (l == r == n): res.append(s)
   # 剪枝，提高算法效率
   if l < r: return
   # 加一个左括号
   dfs(l + 1, r, s + '(')
   # 加一个右括号
   dfs(l, r + 1, s + ')')
dfs(0, 0, '')
return res
```

由于字符串的不可变性， 因此我们无需`撤销 s 的选择`。但是当你使用 C++ 等语言的时候， 就需要注意撤销 s 的选择了。类似：

```
s.push_back(')');
dfs(l, r + 1, s);
s.pop_back();
```

关键点

- 当 l < r 时记得剪枝

代码

JS Code:

```
/**
 * @param {number} n
 * @return {string[]}
 * @param l 左括号已经用了几个
 * @param r 右括号已经用了几个
 * @param str 当前递归得到的拼接字符串结果
 * @param res 结果集
 */
const generateParenthesis = function (n) {
  const res = [];

  function dfs(l, r, str) {
    if (l == n && r == n) {
      return res.push(str);
    }
    // l 小于 r 时不满足条件 剪枝
    if (l < r) {
      return;
    }
    // l 小于 n 时可以插入左括号，最多可以插入 n 个
    if (l < n) {
      dfs(l + 1, r, str + "(");
    }
    // r < l 时 可以插入右括号
    if (r < l) {
      dfs(l, r + 1, str + ")");
    }
  }
```

```
  dfs(0, 0, "");
  return res;
};
```

复杂度分析

- 时间复杂度：O(2^N)
- 空间复杂度：O(2^N)

3. 合并 K 个排序链表

题目描述

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

示例：

```
输入：
[
  1->4->5,
  1->3->4,
  2->6
]
输出：1->1->2->3->4->4->5->6
```

前置知识

- 链表
- 归并排序

思路

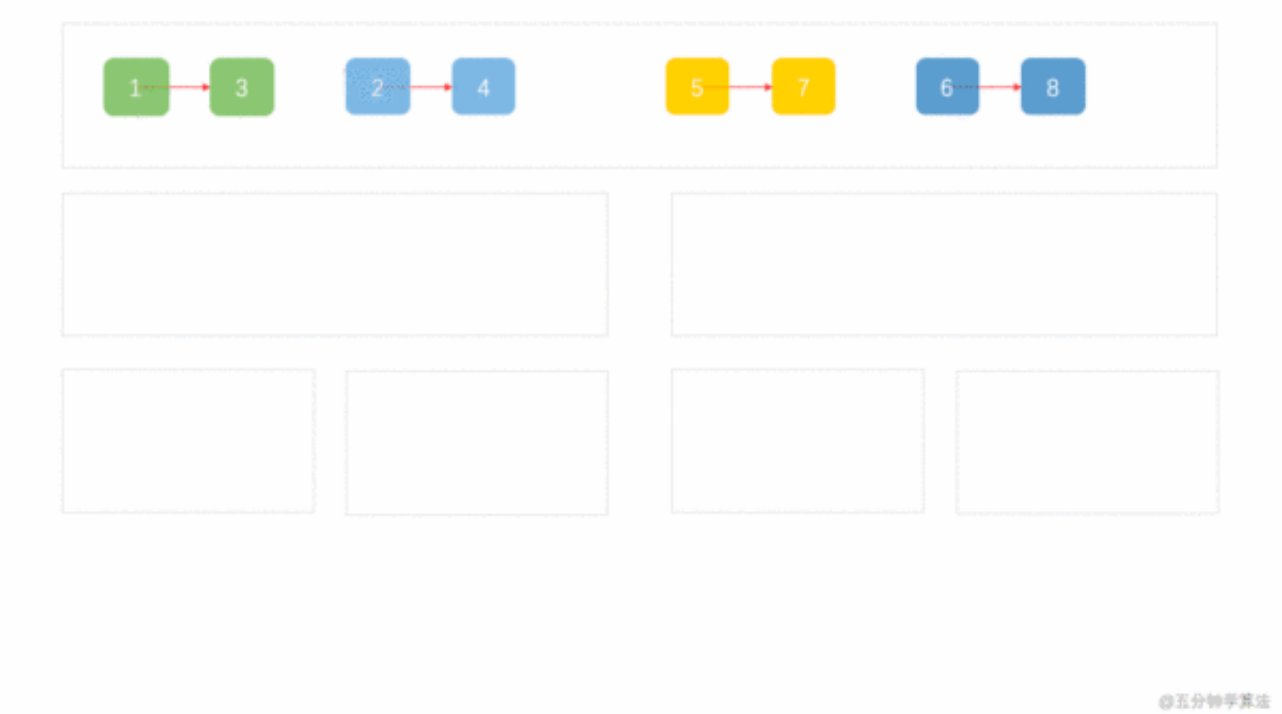
这道题目是合并 k 个已排序的链表，号称 leetcode 目前`最难`的链表题。 和之前我们解决的88.merge-sorted-array很像。
他们有两点区别：

1. 这道题的数据结构是链表，那道是数组。这个其实不复杂，毕竟都是线性的数据结构。

2. 这道题需要合并 k 个元素，那道则只需要合并两个。这个是两题的关键差别，也是这道题难度为 `hard` 的原因。

因此我们可以看出，这道题目是 `88.merge-sorted-array` 的进阶版本。其实思路也有点像，我们来具体分析下第二条。
如果你熟悉合并排序的话，你会发现它就是 `合并排序的一部分` 。

具体我们可以来看一个动画



（动画来自 https://zhuanlan.zhihu.com/p/61796021 ）

关键点解析

- 分治
- 归并排序(merge sort)

代码

JavaScript Code：

```
/*
 * @lc app=leetcode id=23 lang=javascript
 *
 * [23] Merge k Sorted Lists
 *
 * https://leetcode.com/problems/merge-k-sorted-lists/description/
```

```
 *
 */
function mergeTwoLists(l1, l2) {
  const dummyHead = {};
  let current = dummyHead;
  // l1: 1 -> 3 -> 5
  // l2: 2 -> 4 -> 6
  while (l1 !== null && l2 !== null) {
    if (l1.val < l2.val) {
      current.next = l1; // 把小的添加到结果链表
      current = current.next; // 移动结果链表的指针
      l1 = l1.next; // 移动小的那个链表的指针
    } else {
      current.next = l2;
      current = current.next;
      l2 = l2.next;
    }
  }

  if (l1 === null) {
    current.next = l2;
  } else {
    current.next = l1;
  }
  return dummyHead.next;
}
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode[]} lists
 * @return {ListNode}
 */
var mergeKLists = function (lists) {
  // 图参考:  https://zhuanlan.zhihu.com/p/61796021
  if (lists.length === 0) return null;
  if (lists.length === 1) return lists[0];
  if (lists.length === 2) {
    return mergeTwoLists(lists[0], lists[1]);
  }

  const mid = lists.length >> 1;
  const l1 = [];
  for (let i = 0; i < mid; i++) {
```

```
    l1[i] = lists[i];
  }

  const l2 = [];
  for (let i = mid, j = 0; i < lists.length; i++, j++) {
    l2[j] = lists[i];
  }

  return mergeTwoLists(mergeKLists(l1), mergeKLists(l2));
};
```

复杂度分析

- 时间复杂度：$O(kn * logk)$
- 空间复杂度：$O(logk)$

相关题目

- 88.merge-sorted-array

扩展

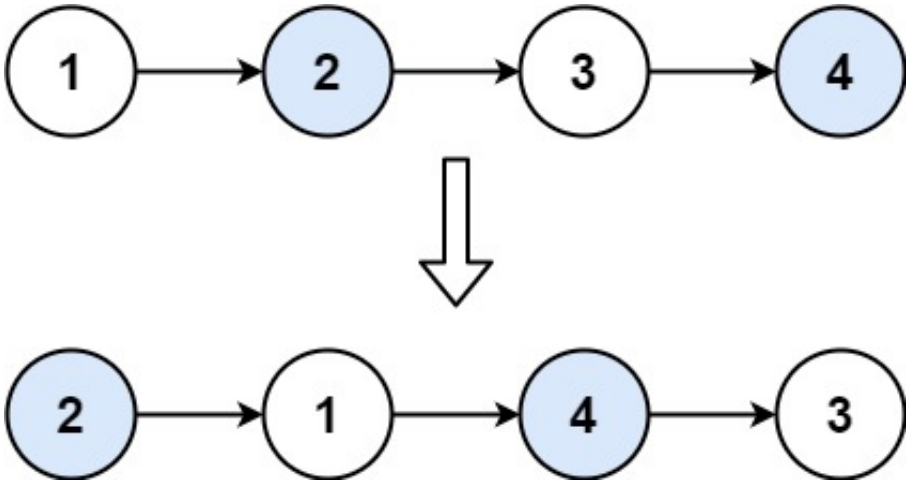
这道题其实可以用堆来做，感兴趣的同学尝试一下吧。

4. 两两交换链表中的节点

题目描述

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。




```
示例 1:
输入: head = [1,2,3,4]
输出: [2,1,4,3]

示例 2:
输入: head = []
输出: []

示例 3:
输入: head = [1]
输出: [1]

提示:
链表中节点的数目在范围 [0, 100] 内
0 <= Node.val <= 100
```

前置知识

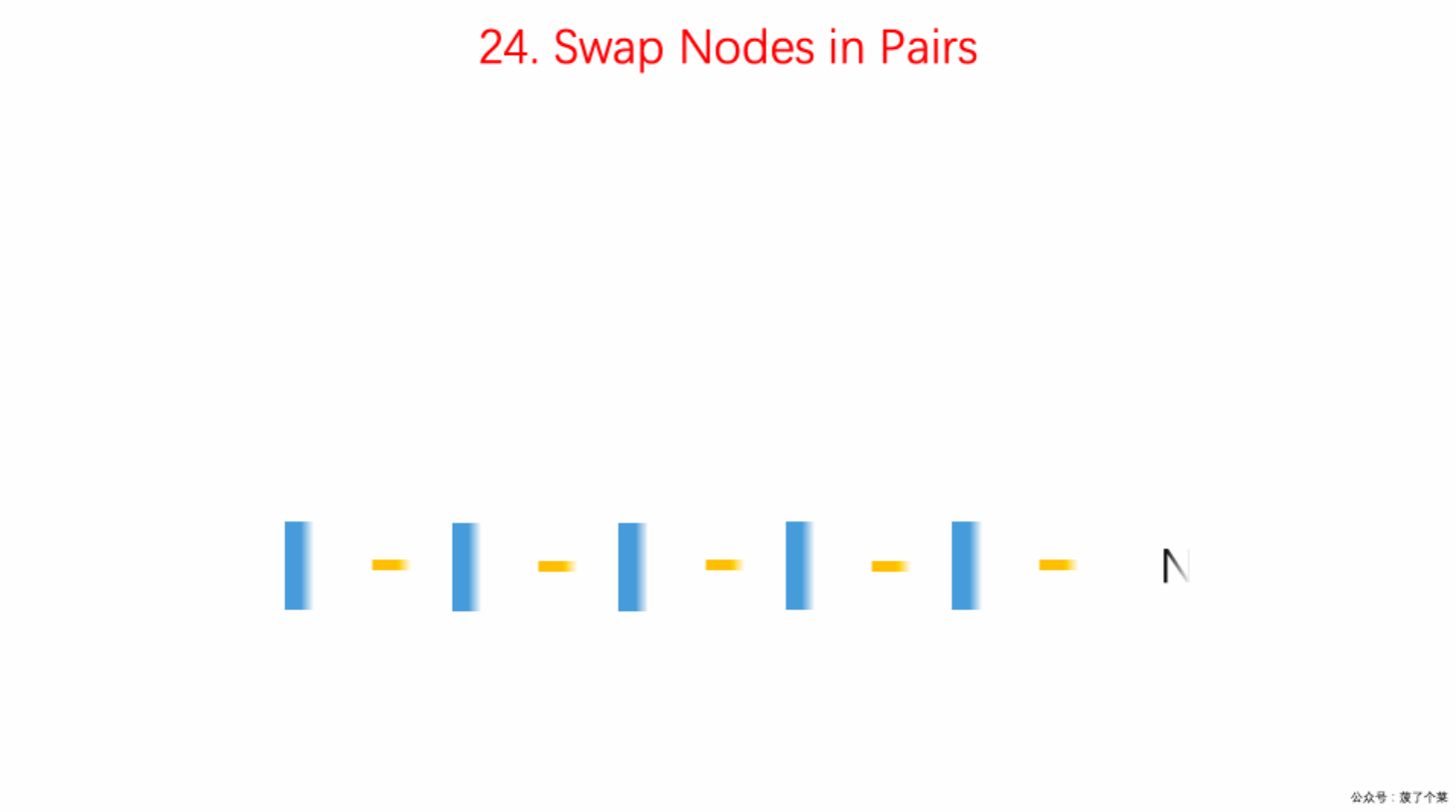
- 链表

思路

设置一个 dummy 节点简化操作，dummy next 指向 head。

1. 初始化 first 为第一个节点
2. 初始化 second 为第二个节点
3. 初始化 current 为 dummy
4. first.next = second.next
5. second.next = first
6. current.next = second

7. current 移动两格
8. 重复



（图片来自： https://github.com/MisterBooo/LeetCodeAnimation）

关键点解析

1. 链表这种数据结构的特点和使用
2. dummyHead 简化操作

代码

JS Code:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
```

```
 * @return {ListNode}
 */
var swapPairs = function (head) {
  const dummy = new ListNode(0);
  dummy.next = head;
  let current = dummy;
  while (current.next != null && current.next.next != null) {
    // 初始化双指针
    const first = current.next;
    const second = current.next.next;

    // 更新双指针和 current 指针
    first.next = second.next;
    second.next = first;
    current.next = second;

    // 更新指针
    current = current.next.next;
  }
  return dummy.next;
};
```

5. K 个一组翻转链表

题目描述

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

示例：

给你这个链表：1->2->3->4->5

当 k = 2 时，应当返回：2->1->4->3->5

当 k = 3 时，应当返回：3->2->1->4->5

说明：

你的算法只能使用常数的额外空间。
你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

前置知识

- 链表

思路

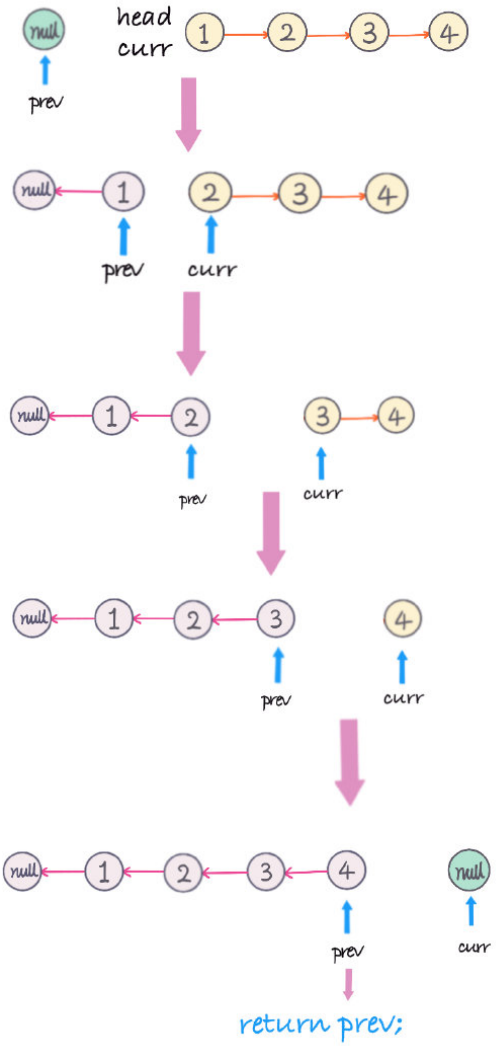
题意是以 `k` 个 nodes 为一组进行翻转，返回翻转后的`linked list`.

从左往右扫描一遍`linked list`，扫描过程中，以 k 为单位把数组分成若干段，对每一段进行翻转。给定首尾 nodes，如何对链表进行翻转。

链表的翻转过程，初始化一个为`null`的 `previous node (prev)` ，然后遍历链表的同时，当前`node (curr)`的下一个（next）指向前一个`node (prev)` ，
在改变当前 node 的指向之前，用一个临时变量记录当前 node 的下一个`node (curr.next)`. 即

```
ListNode temp = curr.next;
curr.next = prev;
prev = curr;
curr = temp;
```

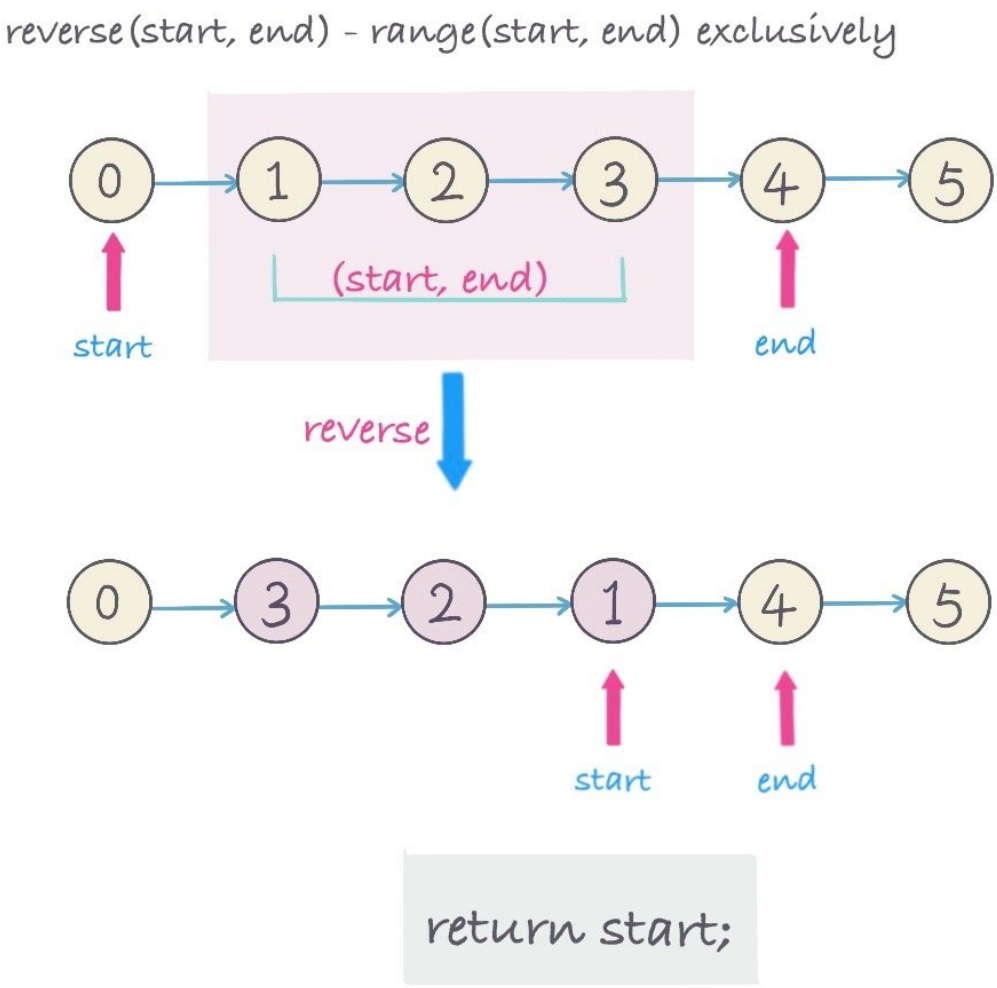
举例如图：翻转整个链表 `1->2->3->4->null` -> `4->3->2->1->null`



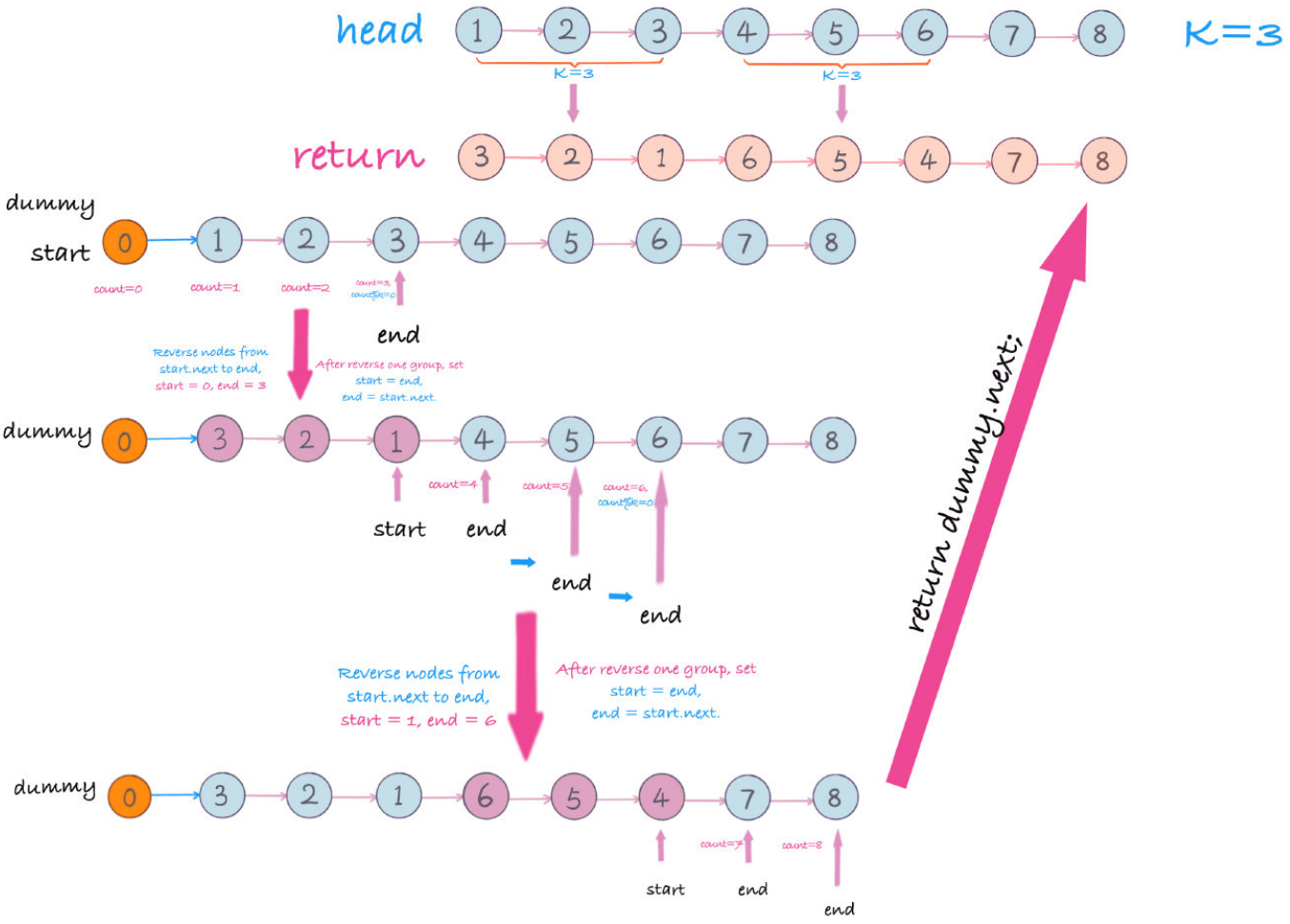
这里是对每一组（`k个nodes`）进行翻转，

1. 先分组，用一个`count`变量记录当前节点的个数
2. 用一个`start`变量记录当前分组的起始节点位置的前一个节点
3. 用一个`end`变量记录要翻转的最后一个节点位置
4. 翻转一组（`k个nodes`）即`(start, end) - start and end exclusively`。
5. 翻转后，`start`指向翻转后链表, 区间`(start, end)`中的最后一个节点, 返回`start`节点。
6. 如果不需要翻转，`end`就往后移动一个（`end=end.next`），每一次移动，都要`count+1`.

如图所示 步骤 4 和 5： 翻转区间链表区间`(start, end)`



举例如图，`head=[1,2,3,4,5,6,7,8], k = 3`



> **NOTE**: 一般情况下对链表的操作，都有可能会引入一个新的`dummy node`，因为`head`有可能会改变。这里`head 从1->3`，
> `dummy (List(0))`保持不变。

复杂度分析

- *时间复杂度:* `O(n) - n is number of Linked List`
- *空间复杂度:* `O(1)`

关键点分析

1. 创建一个 dummy node
2. 对链表以 k 为单位进行分组，记录每一组的起始和最后节点位置
3. 对每一组进行翻转，更换起始和最后的位置
4. 返回`dummy.next`.

代码

javascript code

```
/**
 * @param {ListNode} head
 * @param {number} k
 * @return {ListNode}
 */
var reverseKGroup = function (head, k) {
  // 标兵
  let dummy = new ListNode();
  dummy.next = head;
  let [start, end] = [dummy, dummy.next];
  let count = 0;
  while (end) {
    count++;
    if (count % k === 0) {
      start = reverseList(start, end.next);
      end = start.next;
    } else {
      end = end.next;
    }
  }
  return dummy.next;

  // 翻转stat -> end的链表
  function reverseList(start, end) {
    let [pre, cur] = [start, start.next];
    const first = cur;
    while (cur !== end) {
      let next = cur.next;
      cur.next = pre;
      pre = cur;
      cur = next;
    }
    start.next = pre;
    first.next = cur;
    return first;
  }
};
```

参考（References)

- Leetcode Discussion (yellowstone)

扩展 1

- 要求从后往前以 `k` 个为一组进行翻转。**(字节跳动（ByteDance）面试题)**

  例子， `1->2->3->4->5->6->7->8, k = 3`，

  从后往前以 `k=3` 为一组，

  - `6->7->8` 为一组翻转为 `8->7->6`，
  - `3->4->5` 为一组翻转为 `5->4->3`.
  - `1->2` 只有 2 个 nodes 少于 `k=3` 个，不翻转。

  最后返回： `1->2->5->4->3->8->7->6`

这里的思路跟从前往后以 `k` 个为一组进行翻转类似，可以进行预处理：

1. 翻转链表
2. 对翻转后的链表进行从前往后以 k 为一组翻转。
3. 翻转步骤 2 中得到的链表。

例子： `1->2->3->4->5->6->7->8, k = 3`

1. 翻转链表得到： `8->7->6->5->4->3->2->1`
2. 以 k 为一组翻转： `6->7->8->3->4->5->2->1`
3. 翻转步骤#2 链表： `1->2->5->4->3->8->7->6`

扩展 2

如果这道题你按照 92.reverse-linked-list-ii 提到的 `p1, p2, p3, p4` （四点法） 的思路来思考的话会很清晰。

代码如下（Python）：

```
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        if head is None or k < 2:
            return head
        dummy = ListNode(0)
        dummy.next = head
```

```
        pre = dummy
        cur = head
        count = 0
        while cur:
            count += 1
            if count % k == 0:
                pre = self.reverse(pre, cur.next)
                # end 调到下一个位置
                cur = pre.next
            else:
                cur = cur.next
        return dummy.next
    # (p1, p4)  左右都开放

    def reverse(self, p1, p4):
        prev, curr = p1, p1.next
        p2 = curr
        # 反转
        while curr != p4:
            next = curr.next
            curr.next = prev
            prev = curr
            curr = next
        # 将反转后的链表添加到原链表中
        # prev 相当于 p3
        p1.next = prev
        p2.next = p4
        # 返回反转前的头,  也就是反转后的尾部
        return p2

# @lc code=end
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

6. 删除排序数组中的重复项

题目描述

给定一个排序数组，你需要在 原地 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 原地 修改输入数组 并在使用 O(1) 额外空间的条件下完成。

示例 1:

给定数组 nums = [1,1,2],

函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 nums = [0,0,1,1,1,2,2,3,3,4],

函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

前置知识

- 数组
- 双指针

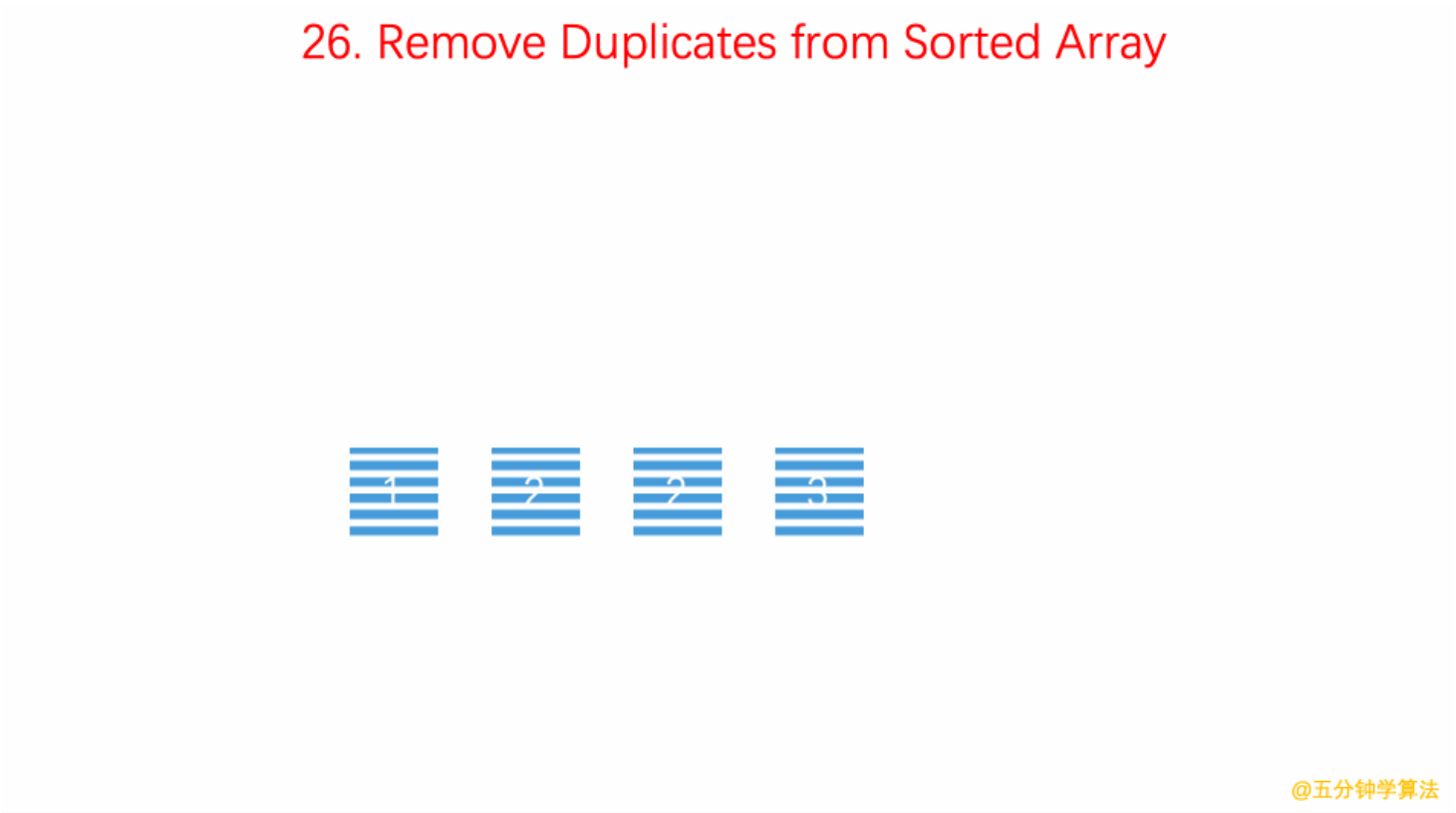
公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg
- facebook
- microsoft

思路

使用快慢指针来记录遍历的坐标。

- 开始时这两个指针都指向第一个数字
- 如果两个指针指的数字相同，则快指针向前走一步
- 如果不同，则两个指针都向前走一步
- 当快指针走完整个数组后，慢指针当前的坐标加 1 就是数组中不同数字的个数



（图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

实际上这就是双指针中的快慢指针。在这里快指针是读指针， 慢指针是写指针。**从读写指针考虑， 我觉得更符合本质**。