1. 自定义指令Vue.directions

Vue2:

bind : 指令绑定到元素后调用

inserted: 元素插入父DOM后调用  
 update: 元素更新，但子元素尚未更新，调用

componentUpdated: 一旦组件和子级被更新，就会调用

unbind: 一旦指令被移除，就会调用

Vue.directions(‘Name’, {

Inserted(el, binding, vNode, oldVNode){

const { value } = binding

const roles = store.getters && store.getters.roles

if( value && value instanceof Array && value.length > 0) {

const permissionRoles = value

const hasPermission = roles.some(role => {

return permissionRoles.includes(role)

})

If(!hasPermission) {

el.parentNode && el.parentNode.removeChild(el)

}

}

}

})

Vue3:

created:

beforeMount: 同bind, 在指令绑定到元素时，调用一次

mounted：元素挂载到父元素下调用

beforeUpdate：元素本身更新前调用

updated：组件和子级更新后调用

beforeUnmount：元素卸载前调用

unmounted：指令被移除时调用

Vue.directions(‘Name’, {

mounted(el, binding, vNode, oldVNode) {

}

})

1. Mixins / Vue.mixin全局混入

就是一个js文件, 我们在开发的时候经常会遇到一些相同的代码和逻辑，可以把这些东西给提取出来放在一个mixin.js文件中，然后通过mixins注入他们，注入的mixin的生命周期钩子函数会先于当前组件的钩子函数执行，当前组件中和mixin中相同的data，会对mixin覆盖

Vue2中，mixin的注入通过配置mixins传入一个mixin数组，这样的注入方式是隐示的注入，变量和方法的来源不明确很难知晓，而Vue3中是通过写hooks，useMixin的方式注入，变量和方法的来源都是明确的。

Vue2：(缺点：隐试引入，不知道变量和方法来源，不同mixin的同名方法，同名属性，可能会导致冲突。)

export default {

data() {

return {

aa: 0

}

}

}

Vue3:

export default function useMixin() {

const aa = ref(0)

return {

aa,

}  
}

1. Vue.Extend

Extend 传入的是一个对象，它会去继承Vue上的一些功能，然后返回一个构造器，由这个构造器构造出来的实例对象可以去调用Vue上的一些方法，比如$mount，然后将extend的内容挂载在某个dom节点下。

它的原理就是通过Object.create去继承Vue的prototype，然后将constructor指回子类，并且初始化了子类的props, computed等，返回子类构造器。

function Extend(extendOptions) {

const Super = this

const Sub = function VueComponent(options) {

this.\_init(options)

}

Sub.prototype = Object.create(Super.prototype)

Sub.prototype.constructor = Sub

Sub.options = mergeOptions(Super, extendOptions)

initProps(Sub)

initComputed(Sub)

// 继承静态方法

Sub.extend = Super.extend

Sub.mixin = Super.mixin

Sub.use = Super.use

return Sub

}

1. 防抖和节流 => 闭包 => 垃圾回收机制

防抖是一段时间内，频繁去触发某个事件，事件不会去响应，只有在触发了这个时间，等待n秒后，没有重复触发，才会去响应最后一次触发的事件。

节流是一段时间内，频繁触发某个事件，它不会一直响应，只会每n秒响应一次。

防抖应用场景 比如search搜索联想，window触发调整浏览器窗口大小。

function debounce(fn, wait) {

let timer = null

return function (…args) {

if(timer) {

clearTimeout(timer)

timer = null

}

timer = setTimeout(() => {

fn.apply(this, args)

}, wait)

}

}

节流应用场景 比如鼠标不断点击触发，监听滚动事件，比如滑动到底部自动加载更多。

function throtten(fn, wait){

let curTime = Date.now()

return function(…args) {

if(Date.now() – curTime >= wait) {

fn.apply(this, args)

curTime = Date.now()

}

}

}

function throtten(fn. wait) {

let timer = null

return function(…args) {

if(!timer) {

timer = setTimeout(() => {

fn.applt(this,args)

timer = null

}, wait)

}

}

}

闭包应用场景： 创建私有变量，对外不可见。 防抖，节流，柯里化，模块化，异步操作中的回调函数

异步操作中的回调函数：

function fetchData(url, cb) {

setTimeout(() => {

const data = fetch(url)

cb(data) // 闭包

})

}

纵然很多场景下，我们需要使用到闭包，但是闭包也有很多缺陷，比如可能会导致内存泄漏，可能引发一些性能问题。因为他们可以长时间保持对外部作用域的引用，从而阻止垃圾回收机制释放，这可能会占用更多的内存和处理时间。所以为了避免这种情况，要谨慎的使用闭包，我在编码的过程中一般会打开f12的performance 查看里面的js heap max，当它一直呈现一个上升不下降的情况，可能就导致了内存泄漏，从而去排查自己的代码，并且在开发的时候也会去注意哪些定时器的清除，事件的销毁，以及不需要使用闭包的时候手动断开等。

垃圾回收机制：

见brower篇章