```
44        } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved
right
45
46          // patch oldStartVnode和newEndVnode
47          patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue)
            // 如果removeOnly是false，则将oldStartVnode.eml移动到oldEndVnode.elm
之后
48          canMove && nodeOps.insertBefore(parentElm, oldStartVnode.elm, nod
eOps.nextSibling(oldEndVnode.elm))
49          // oldStart索引右移，newEnd索引左移
50          oldStartVnode = oldCh[++oldStartIdx]
51          newEndVnode = newCh[--newEndIdx]
52
53        // 如果oldEndVnode和newStartVnode是同一个节点
54        } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved
left
55          // patch oldEndVnode和newStartVnode
56          patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue)
57          // 如果removeOnly是false，则将oldEndVnode.elm移动到oldStartVnode.elm
之前
58          canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldSt
artVnode.elm)
59          // oldEnd索引左移，newStart索引右移
60          oldEndVnode = oldCh[--oldEndIdx]
61          newStartVnode = newCh[++newStartIdx]
62
63        // 如果都不匹配
64        } else {
65          if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh,
oldStartIdx, oldEndIdx)
66
67          // 尝试在oldChildren中寻找和newStartVnode的具有相同的key的Vnode
68          idxInOld = isDef(newStartVnode.key)
69            ? oldKeyToIdx[newStartVnode.key]
70            : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)
71
72          // 如果未找到，说明newStartVnode是一个新的节点
73          if (isUndef(idxInOld)) { // New element
74            // 创建一个新Vnode
75            createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStar
tVnode.elm)
76
77          // 如果找到了和newStartVnodej具有相同的key的Vnode，叫vnodeToMove
78          } else {
79            vnodeToMove = oldCh[idxInOld]
80            /* istanbul ignore if */
81            if (process.env.NODE_ENV !== 'production' && !vnodeToMove) {
82              warn(
83
```

```
                'It seems there are duplicate keys that is causing an updat
e error. ' +
                'Make sure each v-for item has a unique key.'
              )
            }

            // 比较两个具有相同的key的新节点是否是同一个节点
            //不设key，newCh和oldCh只会进行头尾两端的相互比较，设key后，除了头尾两端
的比较外，还会从用key生成的对象oldKeyToIdx中查找匹配的节点，所以为节点设置key可以更
高效的利用dom。
            if (sameVnode(vnodeToMove, newStartVnode)) {
              // patch vnodeToMove和newStartVnode
              patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue)
              // 清除
              oldCh[idxInOld] = undefined
              // 如果removeOnly是false，则将找到的和newStartVnodej具有相同的key
的Vnode，叫vnodeToMove.elm
              // 移动到oldStartVnode.elm之前
              canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm, o
ldStartVnode.elm)

            // 如果key相同，但是节点不相同，则创建一个新的节点
            } else {
              // same key but different element. treat as new element
              createElm(newStartVnode, insertedVnodeQueue, parentElm, oldSt
artVnode.elm)
            }
          }

          // 右移
          newStartVnode = newCh[++newStartIdx]
        }
      }
```

`while` 循环主要处理了以下五种情景：

- 当新老 `VNode` 节点的 `start` 相同时，直接 `patchVnode` ，同时新老 `VNode` 节点的开始索引都加 1
- 当新老 `VNode` 节点的 `end` 相同时，同样直接 `patchVnode` ，同时新老 `VNode` 节点的结束索引都减 1
- 当老 `VNode` 节点的 `start` 和新 `VNode` 节点的 `end` 相同时，这时候在 `patchVnode` 后，还需要将当前真实 `dom` 节点移动到 `oldEndVnode` 的后面，同时老 `VNode` 节点开始索引加 1，新 `VNode` 节点的结束索引减 1
- 当老 `VNode` 节点的 `end` 和新 `VNode` 节点的 `start` 相同时，这时候在 `patchVnode` 后，还需要将当前真实 `dom` 节点移动到 `oldStartVnode` 的前面，同时老 `VNode` 节点结束

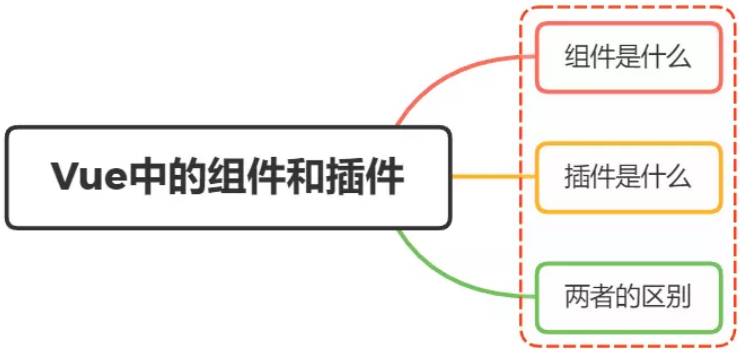
索引减 1，新 `VNode` 节点的开始索引加 1

- 如果都不满足以上四种情形，那说明没有相同的节点可以复用，则会分为以下两种情况：
  - 从旧的 `VNode` 为 `key` 值，对应 `index` 序列为 `value` 值的哈希表中找到与 `newStartVnode` 一致 `key` 的旧的 `VNode` 节点，再进行 `patchVnode`，同时将这个真实 `dom` 移动到 `oldStartVnode` 对应的真实 `dom` 的前面
  - 调用 `createElm` 创建一个新的 `dom` 节点放到当前 `newStartIdx` 的位置

13.4. 小结

- 当数据发生改变时，订阅者 `watcher` 就会调用 `patch` 给真实的 `DOM` 打补丁
- 通过 `isSameVnode` 进行判断，相同则调用 `patchVnode` 方法
- `patchVnode` 做了以下操作：
  - 找到对应的真实 `dom`，称为 `el`
  - 如果都有都有文本节点且不相等，将 `el` 文本节点设置为 `Vnode` 的文本节点
  - 如果 `oldVnode` 有子节点而 `VNode` 没有，则删除 `el` 子节点
  - 如果 `oldVnode` 没有子节点而 `VNode` 有，则将 `VNode` 的子节点真实化后添加到 `el`
  - 如果两者都有子节点，则执行 `updateChildren` 函数比较子节点
- `updateChildren` 主要做了以下操作：
  - 设置新旧 `VNode` 的头尾指针
  - 新旧头尾指针进行比较，循环向中间靠拢，根据情况调用 `patchVnode` 进行 `patch` 重复流程、调用 `createElem` 创建一个新节点，从哈希表寻找 `key` 一致的 `VNode` 节点再分情况操作

14. Vue中组件和插件有什么区别?



14.1. 组件是什么

回顾以前对组件的定义：

组件就是把图形、非图形的各种逻辑均抽象为一个统一的概念（组件）来实现开发的模式，在 `Vue` 中每一个 `.vue` 文件都可以视为一个组件

组件的优势

- 降低整个系统的耦合度，在保持接口不变的情况下，我们可以替换不同的组件快速完成需求，例如输入框，可以替换为日历、时间、范围等组件作具体的实现
- 调试方便，由于整个系统是通过组件组合起来的，在出现问题的时候，可以用排除法直接移除组件，或者根据报错的组件快速定位问题，之所以能够快速定位，是因为每个组件之间低耦合，职责单一，所以逻辑会比分析整个系统要简单
- 提高可维护性，由于每个组件的职责单一，并且组件在系统中是被复用的，所以对代码进行优化可获得系统的整体升级

14.2. 插件是什么

插件通常用来为 `Vue` 添加全局功能。插件的功能范围没有严格的限制——一般有下面几种：

- 添加全局方法或者属性。如: `vue-custom-element`
- 添加全局资源：指令/过滤器/过渡等。如 `vue-touch`
- 通过全局混入来添加一些组件选项。如 `vue-router`
- 添加 `Vue` 实例方法，通过把它们添加到 `Vue.prototype` 上实现。
- 一个库，提供自己的 `API` ，同时提供上面提到的一个或多个功能。如 `vue-router`

14.3. 两者的区别

两者的区别主要表现在以下几个方面：

- 编写形式
- 注册形式
- 使用场景

14.3.1. 编写形式

14.3.1.1. 编写组件

编写一个组件，可以有很多方式，我们最常见的就是 `vue` 单文件的这种格式，每一个 `.vue` 文件我们都可以看成是一个组件

`vue` 文件标准格式

Vue | 复制代码

```vue
<template>
</template>
<script>
export default{
    ...
}
</script>
<style>
</style>
```

我们还可以通过 `template` 属性来编写一个组件，如果组件内容多，我们可以在外部定义 `template` 组件内容，如果组件内容并不多，我们可直接写在 `template` 属性上

JavaScript | 复制代码

```javascript
<template id="testComponent">     // 组件显示的内容
    <div>component!</div>
</template>

Vue.component('componentA',{
    template: '#testComponent'
    template: `<div>component</div>`   // 组件内容少可以通过这种形式
})
```

14.3.1.2. 编写插件

`vue` 插件的实现应该暴露一个 `install` 方法。这个方法的第一个参数是 `Vue` 构造器，第二个参数是一个可选的选项对象

JavaScript | 复制代码

```
MyPlugin.install = function (Vue, options) {
  // 1. 添加全局方法或 property
  Vue.myGlobalMethod = function () {
    // 逻辑...
  }

  // 2. 添加全局资源
  Vue.directive('my-directive', {
    bind (el, binding, vnode, oldVnode) {
      // 逻辑...
    }
    ...
  })

  // 3. 注入组件选项
  Vue.mixin({
    created: function () {
      // 逻辑...
    }
    ...
  })

  // 4. 添加实例方法
  Vue.prototype.$myMethod = function (methodOptions) {
    // 逻辑...
  }
}
```

14.3.2. 注册形式

14.3.2.1. 组件注册

`vue` 组件注册主要分为全局注册与局部注册

全局注册通过 `Vue.component` 方法，第一个参数为组件的名称，第二个参数为传入的配置项

JavaScript | 复制代码

```
Vue.component('my-component-name', { /* ... */ })
```

局部注册只需在用到的地方通过 `components` 属性注册一个组件

```JavaScript
const component1 = {...} // 定义一个组件

export default {
  components:{
    component1   // 局部注册
  }
}
```

14.3.2.2. 插件注册

插件的注册通过 `Vue.use()` 的方式进行注册（安装），第一个参数为插件的名字，第二个参数是可选择的配置项

```JavaScript
Vue.use(插件名字,{ /* ... */} )
```

注意的是：

注册插件的时候，需要在调用 `new Vue()` 启动应用之前完成

`Vue.use` 会自动阻止多次注册相同插件，只会注册一次

14.4. 使用场景

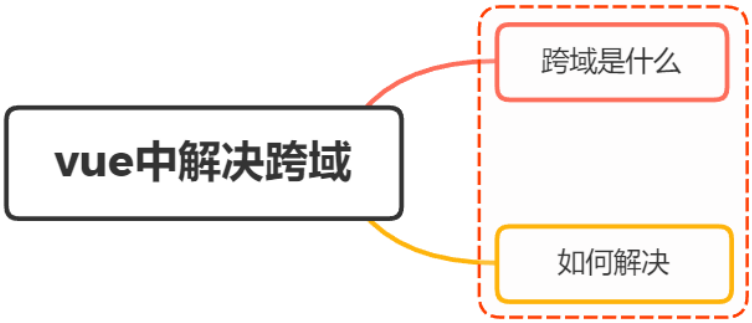
具体的其实在插件是什么章节已经表述了，这里在总结一下

组件 `(Component)` 是用来构成你的 `App` 的业务模块，它的目标是 `App.vue`

插件 `(Plugin)` 是用来增强你的技术栈的功能模块，它的目标是 `Vue` 本身

简单来说，插件就是指对 `Vue` 的功能的增强或补充

15. Vue项目中你是如何解决跨域的呢?



15.1. 跨域是什么

跨域本质是浏览器基于**同源策略**的一种安全手段

同源策略（Sameoriginpolicy），是一种约定，它是浏览器最核心也最基本的安全功能

所谓同源（即指在同一个域）具有以下三个相同点

- 协议相同（protocol）
- 主机相同（host）
- 端口相同（port）

反之非同源请求，也就是协议、端口、主机其中一项不相同的时候，这时候就会产生跨域

> 一定要注意跨域是浏览器的限制，你用抓包工具抓取接口数据，是可以看到接口已经把数据返回回来了，只是浏览器的限制，你获取不到数据。用postman请求接口能够请求到数据。这些再次印证了跨域是浏览器的限制。

15.2. 如何解决

解决跨域的方法有很多，下面列举了三种：

- JSONP
- CORS
- Proxy

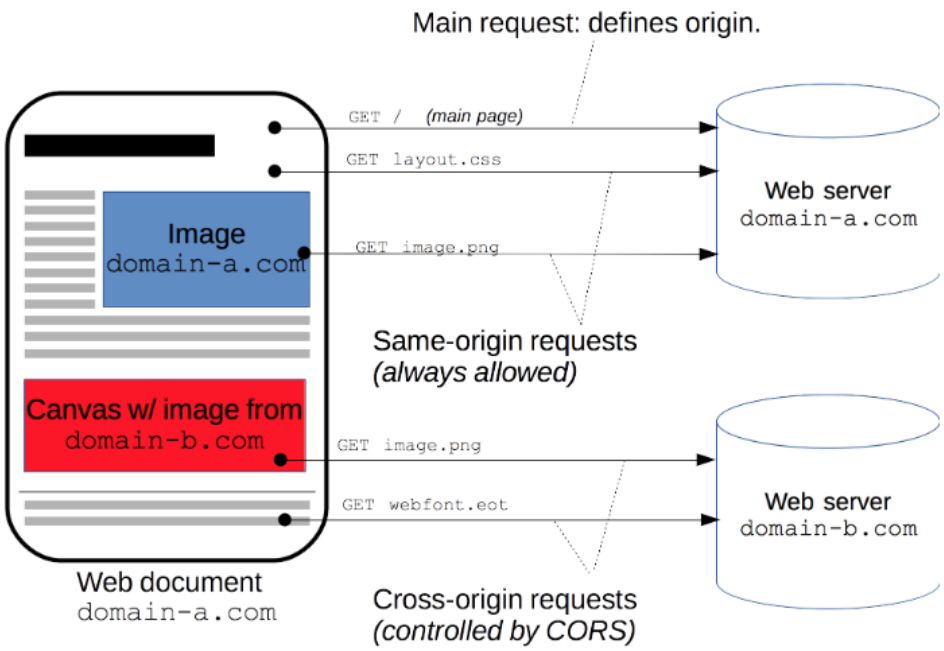
而在 `vue` 项目中，我们主要针对 `CORS` 或 `Proxy` 这两种方案进行展开

15.2.1. CORS

CORS （Cross-Origin Resource Sharing，跨域资源共享）是一个系统，它由一系列传输的HTTP头组成，这些HTTP头决定浏览器是否阻止前端 JavaScript 代码获取跨域请求的响应

`CORS` 实现起来非常方便，只需要增加一些 `HTTP` 头，让服务器能声明允许的访问来源

只要后端实现了 `CORS` ，就实现了跨域



以 `koa` 框架举例

添加中间件，直接设置 `Access-Control-Allow-Origin` 响应头

JavaScript | 复制代码

```
app.use(async (ctx, next)=> {
  ctx.set('Access-Control-Allow-Origin', '*');
  ctx.set('Access-Control-Allow-Headers', 'Content-Type, Content-Length, Authorization, Accept, X-Requested-With , yourHeaderFeild');
  ctx.set('Access-Control-Allow-Methods', 'PUT, POST, GET, DELETE, OPTIONS');
  if (ctx.method == 'OPTIONS') {
    ctx.body = 200;
  } else {
    await next();
  }
})
```

ps: `Access-Control-Allow-Origin` 设置为*其实意义不大，可以说是形同虚设，实际应用中，上线前我们会将 `Access-Control-Allow-Origin` 值设为我们目标 `host`

15.2.2. Proxy

代理（Proxy）也称网络代理，是一种特殊的网络服务，允许一个（一般为客户端）通过这个服务与另一个网络终端（一般为服务器）进行非直接的连接。一些网关、路由器等网络设备具备网络代理功能。一般认为代理服务有利于保障网络终端的隐私或安全，防止攻击

方案一

如果是通过 `vue-cli` 脚手架工具搭建项目，我们可以通过 `webpack` 为我们起一个本地服务器作为请求的代理对象

通过该服务器转发请求至目标服务器，得到结果再转发给前端，但是最终发布上线时如果web应用和接口服务器不在一起仍会跨域

在 `vue.config.js` 文件，新增以下代码

JavaScript 复制代码

```
amodule.exports = {
    devServer: {
        host: '127.0.0.1',
        port: 8084,
        open: true,// vue项目启动时自动打开浏览器
        proxy: {
            '/api': { // '/api'是代理标识，用于告诉node，url前面是/api的就是使用代理的
                target: "http://xxx.xxx.xx.xx:8080", //目标地址，一般是指后台服务器地址
                changeOrigin: true, //是否跨域
                pathRewrite: { // pathRewrite 的作用是把实际Request Url中的'/api'用""代替
                    '^/api': ""
                }
            }
        }
    }
}
```

通过 `axios` 发送请求中，配置请求的根路径

JavaScript 复制代码

```
axios.defaults.baseURL = '/api'
```

方案二

此外，还可通过服务端实现代理请求转发

以 `express` 框架为例

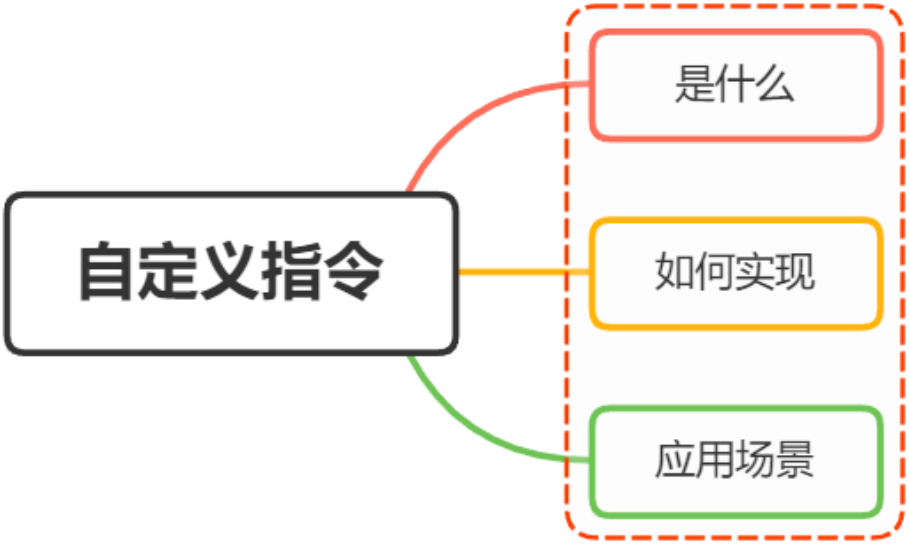
```javascript
var express = require('express');
const proxy = require('http-proxy-middleware')
const app = express()
app.use(express.static(__dirname + '/'))
app.use('/api', proxy({ target: 'http://localhost:4000', changeOrigin: false
                      }));
module.exports = app
```

方案三

通过配置 `nginx` 实现代理

```javascript
server {
    listen    80;
    # server_name www.josephxia.com;
    location / {
        root  /var/www/html;
        index  index.html index.htm;
        try_files $uri $uri/ /index.html;
    }
    location /api {
        proxy_pass  http://127.0.0.1:3000;
        proxy_redirect   off;
        proxy_set_header  Host       $host;
        proxy_set_header  X-Real-IP     $remote_addr;
        proxy_set_header  X-Forwarded-For  $proxy_add_x_forwarded_for;
    }
}
```

16. 有写过自定义指令吗？自定义指令的应用场景有哪些？



16.1. 什么是指令

开始之前我们先学习一下指令系统这个词

指令系统是计算机硬件的语言系统，也叫机器语言，它是系统程序员看到的计算机的主要属性。因此指令系统表征了计算机的基本功能决定了机器所要求的能力

在 `vue` 中提供了一套为数据驱动视图更为方便的操作，这些操作被称为指令系统

我们看到的 `v-` 开头的行内属性，都是指令，不同的指令可以完成或实现不同的功能

除了核心功能默认内置的指令（`v-model` 和 `v-show`），`Vue` 也允许注册自定义指令

指令使用的几种方式：

```
//会实例化一个指令，但这个指令没有参数
`v-xxx`

// -- 将值传到指令中
`v-xxx="value"`

// -- 将字符串传入到指令中，如`v-html="'<p>内容</p>'"`
`v-xxx="'string'"`

// -- 传参数（`arg`），如`v-bind:class="className"`
`v-xxx:arg="value"`

// -- 使用修饰符（`modifier`）
`v-xxx:arg.modifier="value"`
```

16.1.1. 如何实现

注册一个自定义指令有全局注册与局部注册

全局注册主要是通过 `Vue.directive` 方法进行注册

`Vue.directive` 第一个参数是指令的名字（不需要写上 `v-` 前缀），第二个参数可以是对象数据，也可以是一个指令函数

```
// 注册一个全局自定义指令 `v-focus`
Vue.directive('focus', {
  // 当被绑定的元素插入到 DOM 中时……
  inserted: function (el) {
    // 聚焦元素
    el.focus()  // 页面加载完成之后自动让输入框获取到焦点的小功能
  }
})
```

局部注册通过在组件 `options` 选项中设置 `directive` 属性

```javascript
directives: {
  focus: {
    // 指令的定义
    inserted: function (el) {
      el.focus() // 页面加载完成之后自动让输入框获取到焦点的小功能
    }
  }
}
```

然后你可以在模板中任何元素上使用新的 `v-focus` property，如下：

```javascript
<input v-focus />
```

自定义指令也像组件那样存在钩子函数：

- `bind`：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置
- `inserted`：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)
- `update`：所在组件的 `VNode` 更新时调用，但是可能发生在其子 `VNode` 更新之前。指令的值可能发生了改变，也可能没有。但是你可以通过比较更新前后的值来忽略不必要的模板更新
- `componentUpdated`：指令所在组件的 `VNode` 及其子 `VNode` 全部更新后调用
- `unbind`：只调用一次，指令与元素解绑时调用

所有的钩子函数的参数都有以下：

- `el`：指令所绑定的元素，可以用来直接操作 `DOM`
- `binding`：一个对象，包含以下 `property`：
  - `name`：指令名，不包括 `v-` 前缀。
  - `value`：指令的绑定值，例如：`v-my-directive="1 + 1"` 中，绑定值为 `2`。
  - `oldValue`：指令绑定的前一个值，仅在 `update` 和 `componentUpdated` 钩子中可用。无论值是否改变都可用。
  - `expression`：字符串形式的指令表达式。例如 `v-my-directive="1 + 1"` 中，表达式为 `"1 + 1"`。
  - `arg`：传给指令的参数，可选。例如 `v-my-directive:foo` 中，参数为 `"foo"`。
  - `modifiers`：一个包含修饰符的对象。例如：`v-my-directive.foo.bar` 中，修饰符对象为 `{ foo: true, bar: true }`
- `vnode`：`Vue` 编译生成的虚拟节点

- `oldVnode`：上一个虚拟节点，仅在 `update` 和 `componentUpdated` 钩子中可用

> 除了 `el` 之外，其它参数都应该是只读的，切勿进行修改。如果需要在钩子之间共享数据，建议通过元素的 `dataset` 来进行

举个例子：

HTML | 复制代码

```
<div v-demo="{ color: 'white', text: 'hello!' }"></div>
<script>
    Vue.directive('demo', function (el, binding) {
    console.log(binding.value.color) // "white"
    console.log(binding.value.text)  // "hello!"
    })
</script>
```

16.2. 应用场景

使用自定义指令可以满足我们日常一些场景，这里给出几个自定义指令的案例：

- 表单防止重复提交
- 图片懒加载
- 一键 Copy的功能

16.2.1. 表单防止重复提交

表单防止重复提交这种情况设置一个 `v-throttle` 自定义指令来实现

举个例子：

JavaScript | 复制代码

```
// 1.设置v-throttle自定义指令
Vue.directive('throttle', {
  bind: (el, binding) => {
    let throttleTime = binding.value; // 节流时间
    if (!throttleTime) { // 用户若不设置节流时间，则默认2s
      throttleTime = 2000;
    }
    let cbFun;
    el.addEventListener('click', event => {
      if (!cbFun) { // 第一次执行
        cbFun = setTimeout(() => {
          cbFun = null;
        }, throttleTime);
      } else {
        event && event.stopImmediatePropagation();
      }
    }, true);
  },
});
// 2.为button标签设置v-throttle自定义指令
<button @click="sayHello" v-throttle>提交</button>
```

16.2.2. 图片懒加载

设置一个 `v-lazy` 自定义指令完成图片懒加载

JavaScript | 复制代码

```javascript
const LazyLoad = {
    // install方法
    install(Vue,options){
        // 代替图片的loading图
        let defaultSrc = options.default;
        Vue.directive('lazy',{
            bind(el,binding){
                LazyLoad.init(el,binding.value,defaultSrc);
            },
            inserted(el){
                // 兼容处理
                if('IntersectionObserver' in window){
                    LazyLoad.observe(el);
                }else{
                    LazyLoad.listenerScroll(el);
                }

            },
        })
    },
    // 初始化
    init(el,val,def){
        // data-src 储存真实src
        el.setAttribute('data-src',val);
        // 设置src为loading图
        el.setAttribute('src',def);
    },
    // 利用IntersectionObserver监听el
    observe(el){
        let io = new IntersectionObserver(entries => {
            let realSrc = el.dataset.src;
            if(entries[0].isIntersecting){
                if(realSrc){
                    el.src = realSrc;
                    el.removeAttribute('data-src');
                }
            }
        });
        io.observe(el);
    },
    // 监听scroll事件
    listenerScroll(el){
        let handler = LazyLoad.throttle(LazyLoad.load,300);
        LazyLoad.load(el);
        window.addEventListener('scroll',() => {
```

```
                handler(el);
            });
        },
        // 加载真实图片
        load(el){
            let windowHeight = document.documentElement.clientHeight
            let elTop = el.getBoundingClientRect().top;
            let elBtm = el.getBoundingClientRect().bottom;
            let realSrc = el.dataset.src;
            if(elTop - windowHeight<0&&elBtm > 0){
                if(realSrc){
                    el.src = realSrc;
                    el.removeAttribute('data-src');
                }
            }
        },
        // 节流
        throttle(fn,delay){
            let timer;
            let prevTime;
            return function(...args){
                let currTime = Date.now();
                let context = this;
                if(!prevTime) prevTime = currTime;
                clearTimeout(timer);

                if(currTime - prevTime > delay){
                    prevTime = currTime;
                    fn.apply(context,args);
                    clearTimeout(timer);
                    return;
                }

                timer = setTimeout(function(){
                    prevTime = Date.now();
                    timer = null;
                    fn.apply(context,args);
                },delay);
            }
        }

    }
    export default LazyLoad;
```

16.2.3. 一键 Copy的功能

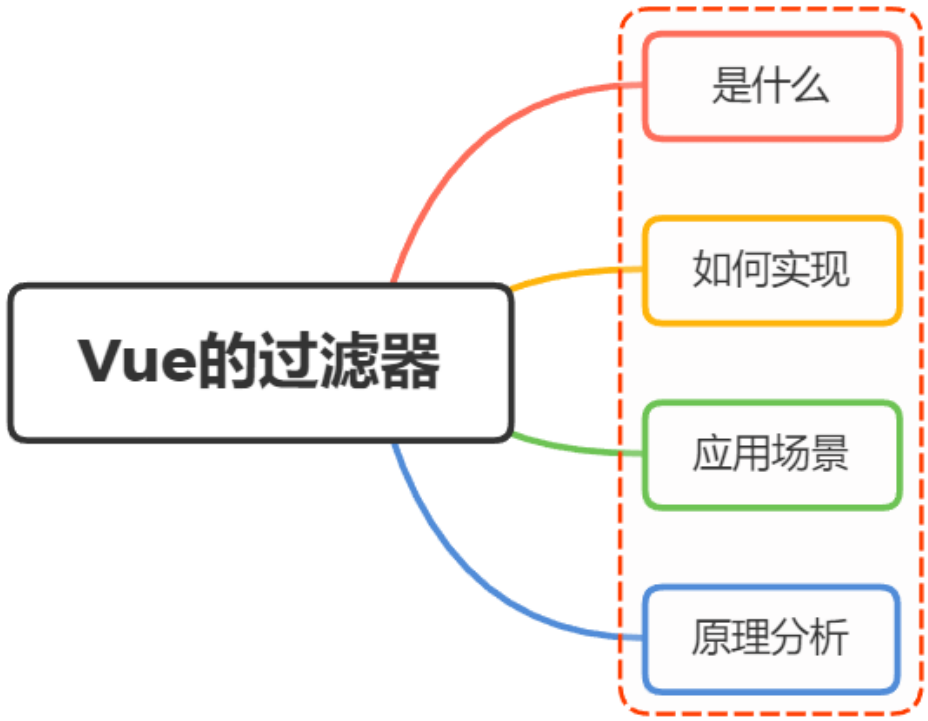
JavaScript | 复制代码

```javascript
import { Message } from 'ant-design-vue';

const vCopy = { //
  /*
    bind 钩子函数，第一次绑定时调用，可以在这里做初始化设置
    el: 作用的 dom 对象
    value: 传给指令的值，也就是我们要 copy 的值
  */
  bind(el, { value }) {
    el.$value = value; // 用一个全局属性来存传进来的值，因为这个值在别的钩子函数里还会用到
    el.handler = () => {
      if (!el.$value) {
      // 值为空的时候，给出提示，我这里的提示是用的 ant-design-vue 的提示，你们随意
        Message.warning('无复制内容');
        return;
      }
      // 动态创建 textarea 标签
      const textarea = document.createElement('textarea');
      // 将该 textarea 设为 readonly 防止 iOS 下自动唤起键盘，同时将 textarea 移出可视区域
      textarea.readOnly = 'readonly';
      textarea.style.position = 'absolute';
      textarea.style.left = '-9999px';
      // 将要 copy 的值赋给 textarea 标签的 value 属性
      textarea.value = el.$value;
      // 将 textarea 插入到 body 中
      document.body.appendChild(textarea);
      // 选中值并复制
      textarea.select();
      // textarea.setSelectionRange(0, textarea.value.length);
      const result = document.execCommand('Copy');
      if (result) {
        Message.success('复制成功');
      }
      document.body.removeChild(textarea);
    };
    // 绑定点击事件，就是所谓的一键 copy 啦
    el.addEventListener('click', el.handler);
  },
  // 当传进来的值更新的时候触发
  componentUpdated(el, { value }) {
    el.$value = value;
  },
  // 指令与元素解绑的时候，移除事件绑定
```

```
  unbind(el) {
    el.removeEventListener('click', el.handler);
  },
};

export default vCopy;
```

关于自定义指令还有很多应用场景，如：拖拽指令、页面水印、权限校验等等应用场景

17. Vue中的过滤器了解吗？过滤器的应用场景有哪些？



17.1. 是什么

过滤器（`filter`）是输送介质管道上不可缺少的一种装置

大白话，就是把一些不必要的东西过滤掉

过滤器实质不改变原始数据，只是对数据进行加工处理后返回过滤后的数据再进行调用处理，我们也可以理解其为一个纯函数

`Vue` 允许你自定义过滤器，可被用于一些常见的文本格式化

ps: `Vue3` 中已废弃 `filter`