JavaScript | 复制代码

```
export default class VNode {
  tag: string | void;
  data: VNodeData | void;
  children: ?Array<VNode>;
  text: string | void;
  elm: Node | void;
  ns: string | void;
  context: Component | void; // rendered in this component's scope
  functionalContext: Component | void; // only for functional component root nodes
  key: string | number | void;
  componentOptions: VNodeComponentOptions | void;
  componentInstance: Component | void; // component instance
  parent: VNode | void; // component placeholder node
  raw: boolean; // contains raw HTML? (server only)
  isStatic: boolean; // hoisted static node
  isRootInsert: boolean; // necessary for enter transition check
  isComment: boolean; // empty comment placeholder?
  isCloned: boolean; // is a cloned node?
  isOnce: boolean; // is a v-once node?

  constructor (
    tag?: string,
    data?: VNodeData,
    children?: ?Array<VNode>,
    text?: string,
    elm?: Node,
    context?: Component,
    componentOptions?: VNodeComponentOptions
  ) {
    /*当前节点的标签名*/
    this.tag = tag
    /*当前节点对应的对象，包含了具体的一些数据信息，是一个VNodeData类型，可以参考VNodeData类型中的数据信息*/
    this.data = data
    /*当前节点的子节点，是一个数组*/
    this.children = children
    /*当前节点的文本*/
    this.text = text
    /*当前虚拟节点对应的真实dom节点*/
    this.elm = elm
    /*当前节点的名字空间*/
    this.ns = undefined
    /*编译作用域*/
    this.context = context
```

```
    /*函数化组件作用域*/
    this.functionalContext = undefined
    /*节点的key属性，被当作节点的标志，用以优化*/
    this.key = data && data.key
    /*组件的option选项*/
    this.componentOptions = componentOptions
    /*当前节点对应的组件的实例*/
    this.componentInstance = undefined
    /*当前节点的父节点*/
    this.parent = undefined
    /*简而言之就是是否为原生HTML或只是普通文本，innerHTML的时候为true，textContent的时候为false*/
    this.raw = false
    /*静态节点标志*/
    this.isStatic = false
    /*是否作为跟节点插入*/
    this.isRootInsert = true
    /*是否为注释节点*/
    this.isComment = false
    /*是否为克隆节点*/
    this.isCloned = false
    /*是否有v-once指令*/
    this.isOnce = false
  }

  // DEPRECATED: alias for componentInstance for backwards compat.
  /* istanbul ignore next https://github.com/answershuto/learnVue*/
  get child (): Component | void {
    return this.componentInstance
  }
}
```

这里对 `VNode` 进行稍微的说明：

- 所有对象的 `context` 选项都指向了 `Vue` 实例
- `elm` 属性则指向了其相对应的真实 `DOM` 节点

`vue` 是通过 `createElement` 生成 `VNode`

源码位置：src/core/vdom/create-element.js

JavaScript | 复制代码

```javascript
export function createElement (
  context: Component,
  tag: any,
  data: any,
  children: any,
  normalizationType: any,
  alwaysNormalize: boolean
): VNode | Array<VNode> {
  if (Array.isArray(data) || isPrimitive(data)) {
    normalizationType = children
    children = data
    data = undefined
  }
  if (isTrue(alwaysNormalize)) {
    normalizationType = ALWAYS_NORMALIZE
  }
  return _createElement(context, tag, data, children, normalizationType)
}
```

上面可以看到 `createElement` 方法实际上是对 `_createElement` 方法的封装，对参数的传入进行了判断

JavaScript | 复制代码

```
export function _createElement(
    context: Component,
    tag?: string | Class<Component> | Function | Object,
    data?: VNodeData,
    children?: any,
    normalizationType?: number
): VNode | Array<VNode> {
    if (isDef(data) && isDef((data: any).__ob__)) {
        process.env.NODE_ENV !== 'production' && warn(
            `Avoid using observed data object as vnode data: ${JSON.stringify(data)}\n` +
            'Always create fresh vnode data objects in each render!',
            context`
        )
        return createEmptyVNode()
    }
    // object syntax in v-bind
    if (isDef(data) && isDef(data.is)) {
        tag = data.is
    }
    if (!tag) {
        // in case of component :is set to falsy value
        return createEmptyVNode()
    }
    ...
    // support single function children as default scoped slot
    if (Array.isArray(children) &&
        typeof children[0] === 'function'
    ) {
        data = data || {}
        data.scopedSlots = { default: children[0] }
        children.length = 0
    }
    if (normalizationType === ALWAYS_NORMALIZE) {
        children = normalizeChildren(children)
    } else if ( === SIMPLE_NORMALIZE) {
        children = simpleNormalizeChildren(children)
    }
  // 创建VNode
    ...
}
```

可以看到 `_createElement` 接收5个参数：

- `context` 表示 `VNode` 的上下文环境，是 `Component` 类型

- tag 表示标签，它可以是一个字符串，也可以是一个 `Component`
- `data` 表示 `VNode` 的数据，它是一个 `VNodeData` 类型
- `children` 表示当前 `VNode` 的子节点，它是任意类型的
- `normalizationType` 表示子节点规范的类型，类型不同规范的方法也就不一样，主要是参考 `render` 函数是编译生成的还是用户手写的

根据 `normalizationType` 的类型， `children` 会有不同的定义

JavaScript | 复制代码

```
if (normalizationType === ALWAYS_NORMALIZE) {
    children = normalizeChildren(children)
} else if ( === SIMPLE_NORMALIZE) {
    children = simpleNormalizeChildren(children)
}
```

`simpleNormalizeChildren` 方法调用场景是 `render` 函数是编译生成的

`normalizeChildren` 方法调用场景分为下面两种：

- `render` 函数是用户手写的
- 编译 `slot` 、 `v-for` 的时候会产生嵌套数组

无论是 `simpleNormalizeChildren` 还是 `normalizeChildren` 都是对 `children` 进行规范（使 `children` 变成了一个类型为 `VNode` 的 `Array` ），这里就不展开说了

规范化 `children` 的源码位置在：src/core/vdom/helpers/normalzie-children.js

在规范化 `children` 后，就去创建 `VNode`

```javascript
let vnode, ns
// 对tag进行判断
if (typeof tag === 'string') {
  let Ctor
  ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
  if (config.isReservedTag(tag)) {
    // 如果是内置的节点，则直接创建一个普通VNode
    vnode = new VNode(
      config.parsePlatformTagName(tag), data, children,
      undefined, undefined, context
    )
  } else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
    // component
    // 如果是component类型，则会通过createComponent创建VNode节点
    vnode = createComponent(Ctor, data, context, children, tag)
  } else {
    vnode = new VNode(
      tag, data, children,
      undefined, undefined, context
    )
  }
} else {
  // direct component options / constructor
  vnode = createComponent(tag, data, context, children)
}
```

`createComponent` 同样是创建 `VNode`

源码位置：src/core/vdom/create-component.js

JavaScript | 复制代码

```
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {
  if (isUndef(Ctor)) {
    return
  }
 // 构建子类构造函数
  const baseCtor = context.$options._base

  // plain options object: turn it into a constructor
  if (isObject(Ctor)) {
    Ctor = baseCtor.extend(Ctor)
  }

  // if at this stage it's not a constructor or an async component factory,
  // reject.
  if (typeof Ctor !== 'function') {
    if (process.env.NODE_ENV !== 'production') {
      warn(`Invalid Component definition: ${String(Ctor)}`, context)
    }
    return
  }

  // async component
  let asyncFactory
  if (isUndef(Ctor.cid)) {
    asyncFactory = Ctor
    Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)
    if (Ctor === undefined) {
      return createAsyncPlaceholder(
        asyncFactory,
        data,
        context,
        children,
        tag
      )
    }
  }

  data = data || {}
```

```
45
46      // resolve constructor options in case global mixins are applied after
47      // component constructor creation
48      resolveConstructorOptions(Ctor)
49
50      // transform component v-model data into props & events
51      if (isDef(data.model)) {
52        transformModel(Ctor.options, data)
53      }
54
55      // extract props
56      const propsData = extractPropsFromVNodeData(data, Ctor, tag)
57
58      // functional component
59      if (isTrue(Ctor.options.functional)) {
60        return createFunctionalComponent(Ctor, propsData, data, context, child
    ren)
61      }
62
63      // extract listeners, since these needs to be treated as
64      // child component listeners instead of DOM listeners
65      const listeners = data.on
66      // replace with listeners with .native modifier
67      // so it gets processed during parent component patch.
68      data.on = data.nativeOn
69
70      if (isTrue(Ctor.options.abstract)) {
71        const slot = data.slot
72        data = {}
73        if (slot) {
74          data.slot = slot
75        }
76      }
77
78      // 安装组件钩子函数，把钩子函数合并到data.hook中
79      installComponentHooks(data)
80
81      //实例化一个VNode返回。组件的VNode是没有children的
82      const name = Ctor.options.name || tag
83      const vnode = new VNode(
84        `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
85        data, undefined, undefined, undefined, context,
86        { Ctor, propsData, listeners, tag, children },
87        asyncFactory
88      )
89      if (__WEEX__ && isRecyclableComponent(vnode)) {
90        return renderRecyclableComponentTemplate(vnode)
91      }
```

```
92
93
94      return vnode
    }
```

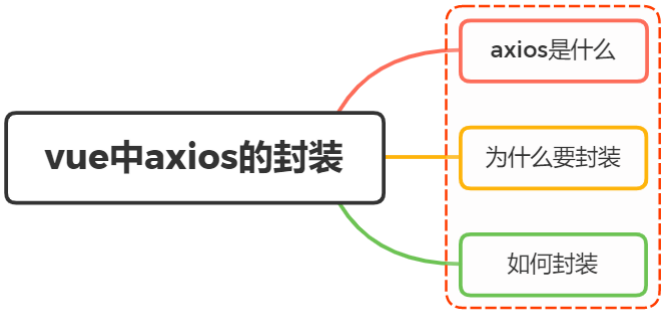
稍微提下 `createComponent` 生成 `VNode` 的三个关键流程：

- 构造子类构造函数 `Ctor`
- `installComponentHooks` 安装组件钩子函数
- 实例化 `vnode`

19.3.1. 小结

`createElement` 创建 `VNode` 的过程，每个 `VNode` 有 `children`， `children` 每个元素也是一个 `VNode`，这样就形成了一个虚拟树结构，用于描述真实的 `DOM` 树结构

20. Vue项目中有封装过axios吗？主要是封装哪方面的？



20.1. axios是什么

`axios` 是一个轻量的 `HTTP` 客户端

基于 `XMLHttpRequest` 服务来执行 `HTTP` 请求，支持丰富的配置，支持 `Promise`，支持浏览器端和 `Node.js` 端。自 `Vue` 2.0起，尤大宣布取消对 `vue-resource` 的官方推荐，转而推荐 `axios`。现在 `axios` 已经成为大部分 `Vue` 开发者的首选

20.2. 特性

- 从浏览器中创建 `XMLHttpRequests`
- 从 `node.js` 创建 `http` 请求
- 支持 `Promise` API
- 拦截请求和响应
- 转换请求数据和响应数据
- 取消请求
- 自动转换 `JSON` 数据
- 客户端支持防御 `XSRF`

20.2.1. 基本使用

安装

JavaScript | 复制代码

```
// 项目中安装
npm install axios --S
// cdn 引入
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

导入

JavaScript | 复制代码

```
import axios from 'axios'
```

发送请求

JavaScript | 复制代码

```
axios({
    url:'xxx',    // 设置请求的地址
    method:"GET", // 设置请求方法
    params:{      // get请求使用params进行参数凭借,如果是post请求用data
      type: '',
      page: 1
    }
}).then(res => {
    // res为后端返回的数据
    console.log(res);
})
```

并发请求 `axios.all([])`

JavaScript | 复制代码

```
function getUserAccount() {
    return axios.get('/user/12345');
}

function getUserPermissions() {
    return axios.get('/user/12345/permissions');
}

axios.all([getUserAccount(), getUserPermissions()])
    .then(axios.spread(function (res1, res2) {
    // res1第一个请求的返回的内容，res2第二个请求返回的内容
    // 两个请求都执行完成才会执行
}));
```

20.3. 为什么要封装

`axios` 的 API 很友好，你完全可以很轻松地在项目中直接使用。

不过随着项目规模增大，如果每发起一次 `HTTP` 请求，就要把这些比如设置超时时间、设置请求头、根据项目环境判断使用哪个请求地址、错误处理等等操作，都需要写一遍

这种重复劳动不仅浪费时间，而且让代码变得冗余不堪，难以维护。为了提高我们的代码质量，我们应该在项目中二次封装一下 `axios` 再使用

举个例子：

```javascript
axios('http://localhost:3000/data', {
  // 配置代码
  method: 'GET',
  timeout: 1000,
  withCredentials: true,
  headers: {
    'Content-Type': 'application/json',
    Authorization: 'xxx',
  },
  transformRequest: [function (data, headers) {
    return data;
  }],
  // 其他请求配置...
})
.then((data) => {
  // todo: 真正业务逻辑代码
  console.log(data);
}, (err) => {
  // 错误处理代码
  if (err.response.status === 401) {
  // handle authorization error
  }
  if (err.response.status === 403) {
  // handle server forbidden error
  }
  // 其他错误处理.....
  console.log(err);
});
```

如果每个页面都发送类似的请求，都要写一堆的配置与错误处理，就显得过于繁琐了

这时候我们就需要对 `axios` 进行二次封装，让使用更为便利

20.4. 如何封装

封装的同时，你需要和 后端协商好一些约定，请求头，状态码，请求超时时间.....

设置接口请求前缀：根据开发、测试、生产环境的不同，前缀需要加以区分

请求头： 来实现一些具体的业务，必须携带一些参数才可以请求(例如：会员业务)

状态码： 根据接口返回的不同 `status` ， 来执行不同的业务，这块需要和后端约定好

请求方法：根据 `get` 、 `post` 等方法进行一个再次封装，使用起来更为方便

请求拦截器：根据请求的请求头设定，来决定哪些请求可以访问

响应拦截器： 这块就是根据 后端`返回来的状态码判定执行不同业务

20.4.1. 设置接口请求前缀

利用 `node` 环境变量来作判断，用来区分开发、测试、生产环境

JavaScript | 复制代码

```
if (process.env.NODE_ENV === 'development') {
  axios.defaults.baseURL = 'http://dev.xxx.com'
} else if (process.env.NODE_ENV === 'production') {
  axios.defaults.baseURL = 'http://prod.xxx.com'
}
```

在本地调试的时候，还需要在 `vue.config.js` 文件中配置 `devServer` 实现代理转发，从而实现跨域

JavaScript | 复制代码

```
devServer: {
    proxy: {
      '/proxyApi': {
        target: 'http://dev.xxx.com',
        changeOrigin: true,
        pathRewrite: {
          '/proxyApi': ''
        }
      }
    }
  }
```

20.4.2. 设置请求头与超时时间

大部分情况下，请求头都是固定的，只有少部分情况下，会需要一些特殊的请求头，这里将普适性的请求头作为基础配置。当需要特殊请求头时，将特殊请求头作为参数传入，覆盖基础配置

```javascript
const service = axios.create({
    ...
    timeout: 30000,  // 请求 30s 超时
    headers: {
        get: {
          'Content-Type': 'application/x-www-form-urlencoded;charset=utf-8'
          // 在开发中，一般还需要单点登录或者其他功能的通用请求头，可以一并配置进来
        },
        post: {
          'Content-Type': 'application/json;charset=utf-8'
          // 在开发中，一般还需要单点登录或者其他功能的通用请求头，可以一并配置进来
        }
  },
})
```

20.4.3. 封装请求方法

先引入封装好的方法，在要调用的接口重新封装成一个方法暴露出去

JavaScript | 复制代码

```
// get 请求
export function httpGet({
  url,
  params = {}
}) {
  return new Promise((resolve, reject) => {
    axios.get(url, {
      params
    }).then((res) => {
      resolve(res.data)
    }).catch(err => {
      reject(err)
    })
  })
}

// post
// post请求
export function httpPost({
  url,
  data = {},
  params = {}
}) {
  return new Promise((resolve, reject) => {
    axios({
      url,
      method: 'post',
      transformRequest: [function (data) {
        let ret = ''
        for (let it in data) {
          ret += encodeURIComponent(it) + '=' + encodeURIComponent(data[it
]) + '&'
        }
        return ret
      }],
      // 发送的数据
      data,
      // url参数
      params

    }).then(res => {
      resolve(res.data)
    })
  })
}
```

把封装的方法放在一个 `api.js` 文件中

```JavaScript
import { httpGet, httpPost } from './http'
export const getorglist = (params = {}) => httpGet({ url: 'apps/api/org/list', params })
```

页面中就能直接调用

```JavaScript
// .vue
import { getorglist } from '@/assets/js/api'

getorglist({ id: 200 }).then(res => {
  console.log(res)
})
```

这样可以把 `api` 统一管理起来，以后维护修改只需要在 `api.js` 文件操作即可

20.4.4. 请求拦截器

请求拦截器可以在每个请求里加上token，做了统一处理后维护起来也方便

```JavaScript
// 请求拦截器
axios.interceptors.request.use(
  config => {
    // 每次发送请求之前判断是否存在token
    // 如果存在，则统一在http请求的header都加上token，这样后台根据token判断你的登录情况，此处token一般是用户完成登录后储存到localstorage里的
    token && (config.headers.Authorization = token)
    return config
  },
  error => {
    return Promise.error(error)
  })
```

20.4.5. 响应拦截器

响应拦截器可以在接收到响应后先做一层操作，如根据状态码判断登录状态、授权

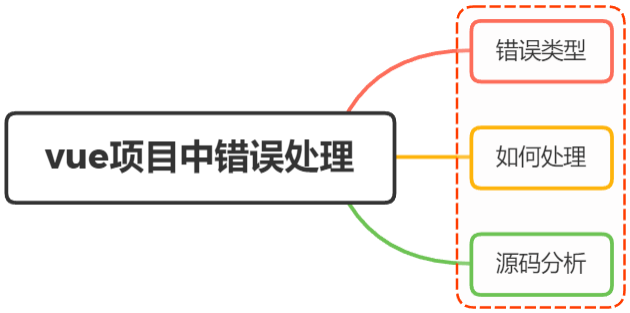
JavaScript 复制代码

```javascript
// 响应拦截器
axios.interceptors.response.use(response => {
  // 如果返回的状态码为200，说明接口请求成功，可以正常拿到数据
  // 否则的话抛出错误
  if (response.status === 200) {
    if (response.data.code === 511) {
      // 未授权调取授权接口
    } else if (response.data.code === 510) {
      // 未登录跳转登录页
    } else {
      return Promise.resolve(response)
    }
  } else {
    return Promise.reject(response)
  }
}, error => {
  // 我们可以在这里对异常状态作统一处理
  if (error.response.status) {
    // 处理请求失败的情况
    // 对不同返回码对相应处理
    return Promise.reject(error.response)
  }
})
```

20.5. 小结

- 封装是编程中很有意义的手段，简单的 `axios` 封装，就可以让我们可以领略到它的魅力
- 封装 `axios` 没有一个绝对的标准，只要你的封装可以满足你的项目需求，并且用起来方便，那就是一个好的封装方案

21. 是怎么处理vue项目中的错误的?



21.1. 错误类型

任何一个框架，对于错误的处理都是一种必备的能力

在 `Vue` 中，则是定义了一套对应的错误处理规则给到使用者，且在源代码级别，对部分必要的过程做了一定的错误处理。

主要的错误来源包括：

- 后端接口错误
- 代码中本身逻辑错误

21.2. 如何处理

21.2.1. 后端接口错误

通过 `axios` 的 `interceptor` 实现网络请求的 `response` 先进行一层拦截

JavaScript | 复制代码

```
apiClient.interceptors.response.use(
  response => {
    return response;
  },
  error => {
    if (error.response.status == 401) {
      router.push({ name: "Login" });
    } else {
      message.error("出错了");
      return Promise.reject(error);
    }
  }
);
```

21.2.2. 代码逻辑问题

21.2.2.1. 全局设置错误处理

设置全局错误处理函数

```JavaScript
Vue.config.errorHandler = function (err, vm, info) {
  // handle error
  // `info` 是 Vue 特定的错误信息，比如错误所在的生命周期钩子
  // 只在 2.2.0+ 可用
}
```

`errorHandler` 指定组件的渲染和观察期间未捕获错误的处理函数。这个处理函数被调用时，可获取错误信息和 `Vue` 实例

不过值得注意的是，在不同 `Vue` 版本中，该全局 `API` 作用的范围会有所不同：

> 从 2.2.0 起，这个钩子也会捕获组件生命周期钩子里的错误。同样的，当这个钩子是 `undefined` 时，被捕获的错误会通过 `console.error` 输出而避免应用崩

> 从 2.4.0 起，这个钩子也会捕获 Vue 自定义事件处理函数内部的错误了

> 从 2.6.0 起，这个钩子也会捕获 `v-on` DOM 监听器内部抛出的错误。另外，如果任何被覆盖的钩子或处理函数返回一个 Promise 链 (例如 async 函数)，则来自其 Promise 链的错误也会被处理

21.2.2.2. 生命周期钩子

`errorCaptured` 是 2.5.0 新增的一个生命钩子函数，当捕获到一个来自子孙组件的错误时被调用

基本类型

```JavaScript
(err: Error, vm: Component, info: string) => ?boolean
```

此钩子会收到三个参数：错误对象、发生错误的组件实例以及一个包含错误来源信息的字符串。此钩子可以返回 `false` 以阻止该错误继续向上传播

参考官网，错误传播规则如下：

- 默认情况下，如果全局的 `config.errorHandler` 被定义，所有的错误仍会发送它，因此这些错误仍然会向单一的分析服务的地方进行汇报
- 如果一个组件的继承或父级从属链路中存在多个 `errorCaptured` 钩子，则它们将会被相同的错

误逐个唤起。

- 如果此 `errorCaptured` 钩子自身抛出了一个错误，则这个新错误和原本被捕获的错误都会发送给全局的 `config.errorHandler`
- 一个 `errorCaptured` 钩子能够返回 `false` 以阻止错误继续向上传播。本质上是说“这个错误已经被搞定了且应该被忽略”。它会阻止其它任何会被这个错误唤起的 `errorCaptured` 钩子和全局的 `config.errorHandler`

下面来看个例子

定义一个父组件 `cat`

JavaScript | 复制代码

```
Vue.component('cat', {
    template:`
        <div>
      <h1>Cat: </h1>
          <slot></slot>
        </div>`,
    props:{
        name:{
            required:true,
            type:String
        }
    },
    errorCaptured(err,vm,info) {
        console.log(`cat EC: ${err.toString()}\ninfo: ${info}`);
        return false;
    }

 });
```

定义一个子组件 `kitten`，其中 `dontexist()` 并没有定义，存在错误

JavaScript | 复制代码

```
Vue.component('kitten', {
    template:'<div><h1>Kitten: {{ dontexist() }}</h1></div>',
    props:{
        name:{
            required:true,
            type:String
        }
    }
});
```