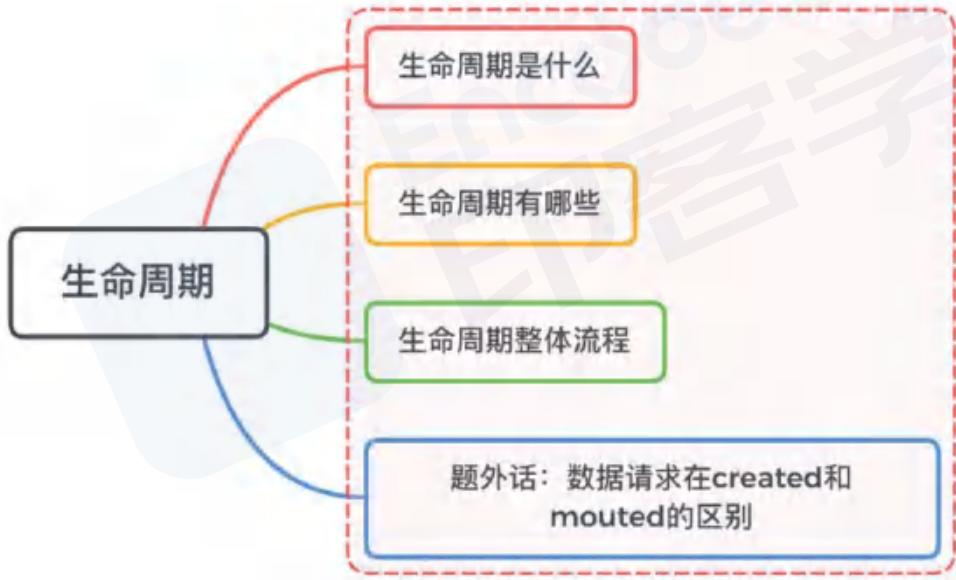


Vue2面试真题（29题）

1. 请描述下对vue生命周期的理解



1.1. 生命周期是什么

生命周期（Life Cycle）的概念应用很广泛，特别是在政治、经济、环境、技术、社会等诸多领域经常出现，其基本涵义可以通俗地理解为“从摇篮到坟墓”（Cradle-to-Grave）的整个过程。在 Vue 中实例从创建到销毁的过程就是生命周期，即指从创建、初始化数据、编译模板、挂载Dom→渲染、更新→渲染、卸载等一系列过程。我们可以把组件比喻成工厂里面的一条流水线，每个工人（生命周期）站在各自的岗位，当任务流转到工人身边的时候，工人就开始工作。PS：在 Vue 生命周期钩子会自动绑定 this 上下文到实例中，因此你可以访问数据，对 property 和方法进行运算。这意味着你不能使用箭头函数来定义一个生命周期方法（例如 created: () => this.fetchTodos()）。

1.2. 生命周期有哪些

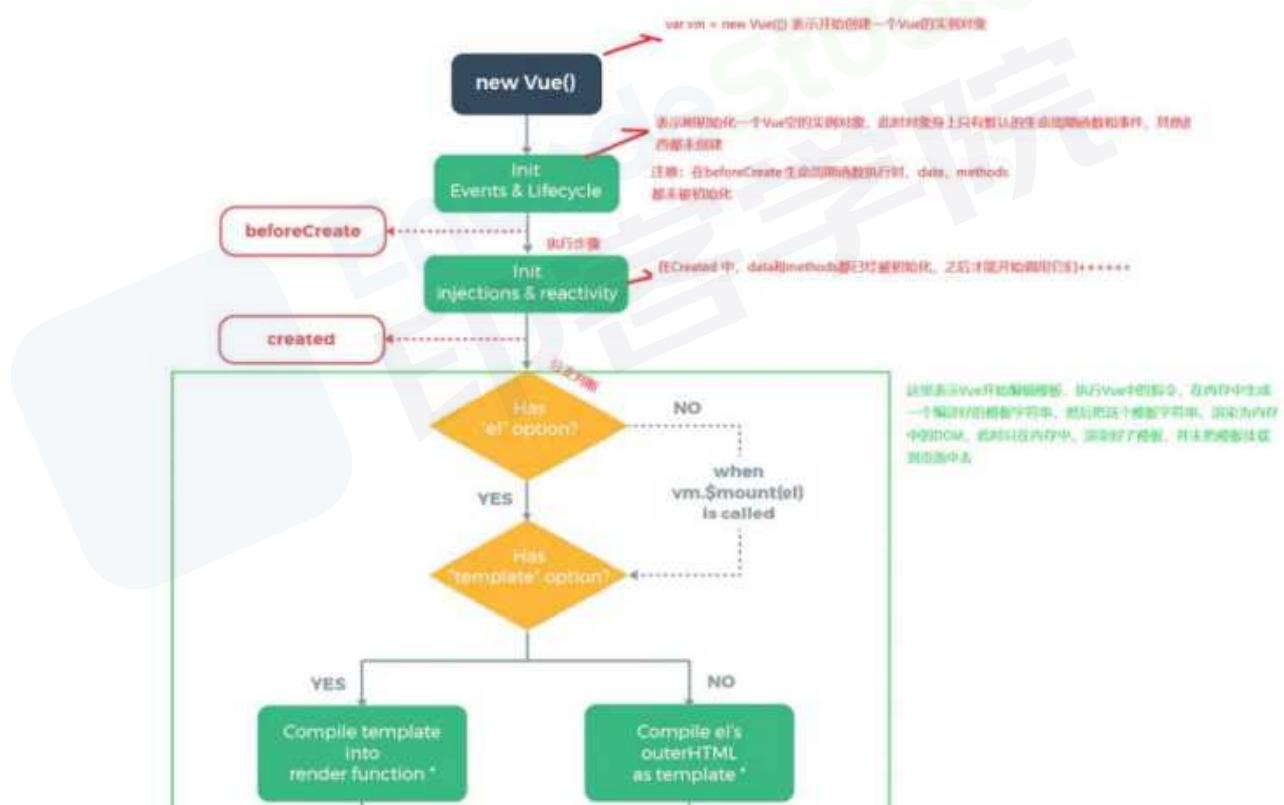
Vue生命周期总共可以分为8个阶段：创建前后，载入前后，更新前后，销毁前销毁后，以及一些特殊场景的生命周期。

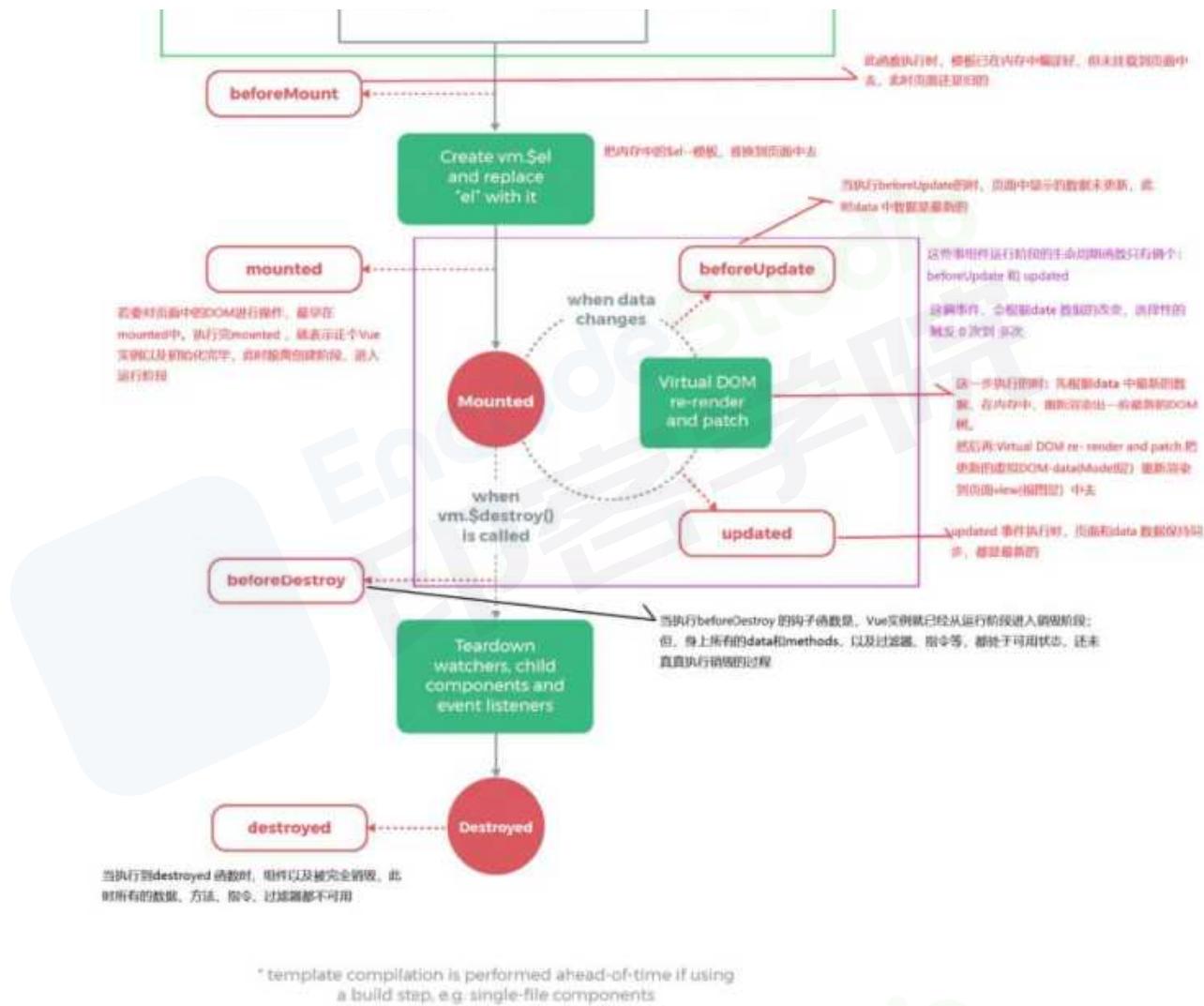
生命周期	描述
------	----

beforeCreate	组件实例被创建之初
created	组件实例已经完全创建
beforeMount	组件挂载之前
mounted	组件挂载到实例上去之后
beforeUpdate	组件数据发生变化，更新之前
updated	组件数据更新之后
beforeDestroy	组件实例销毁之前
destroyed	组件实例销毁之后
activated	keep-alive 缓存的组件激活时
deactivated	keep-alive 缓存的组件停用时调用
errorCaptured	捕获一个来自子孙组件的错误时被调用

1.3. 三、生命周期整体流程

Vue 生命周期流程图





1.3.1.1. 具体分析

`beforeCreate` → `created`

- 初始化 `vue` 实例，进行数据观测

`created`

- 完成数据观测，属性与方法的运算，`watch`、`event` 事件回调的配置
- 可调用 `methods` 中的方法，访问和修改 `data` 数据触发响应式渲染 `dom`，可通过 `computed` 和 `watch` 完成数据计算
- 此时 `vm.$el` 并没有被创建

`created` → `beforeMount`

- 判断是否存在 `el` 选项，若不存在则停止编译，直到调用 `vm.$mount(el)` 才会继续编译
- 优先级：`render` > `template` > `outerHTML`

- `vm.el` 获取到的是挂载 `DOM` 的

beforeMount

- 在此阶段可获取到 `vm.el`
- 此阶段 `vm.el` 虽已完成 `DOM` 初始化，但并未挂载在 `el` 选项上

beforeMount → mounted

- 此阶段 `vm.el` 完成挂载，`vm.$el` 生成的 `DOM` 替换了 `el` 选项所对应的 `DOM`

mounted

- `vm.el` 已完成 `DOM` 的挂载与渲染，此刻打印 `vm.$el`，发现之前的挂载点及内容已被替换成新的 `DOM`

beforeUpdate

- 更新的数据必须是被渲染在模板上的（`el`、`template`、`render` 之一）
- 此时 `view` 层还未更新
- 若在 `beforeUpdate` 中再次修改数据，不会再次触发更新方法

updated

- 完成 `view` 层的更新
- 若在 `updated` 中再次修改数据，会再次触发更新方法（`beforeUpdate`、`updated`）

beforeDestroy

- 实例被销毁前调用，此时实例属性与方法仍可访问

destroyed

- 完全销毁一个实例。可清理它与其它实例的连接，解绑它的全部指令及事件监听器
- 并不能清除 `DOM`，仅仅销毁实例

使用场景分析

生命周期	描述
beforeCreate	执行时组件实例还未创建，通常用于插件开发中执行一些初始化任务
created	组件初始化完毕，各种数据可以使用，常用于异步数据获取
beforeMount	未执行渲染、更新，dom未创建

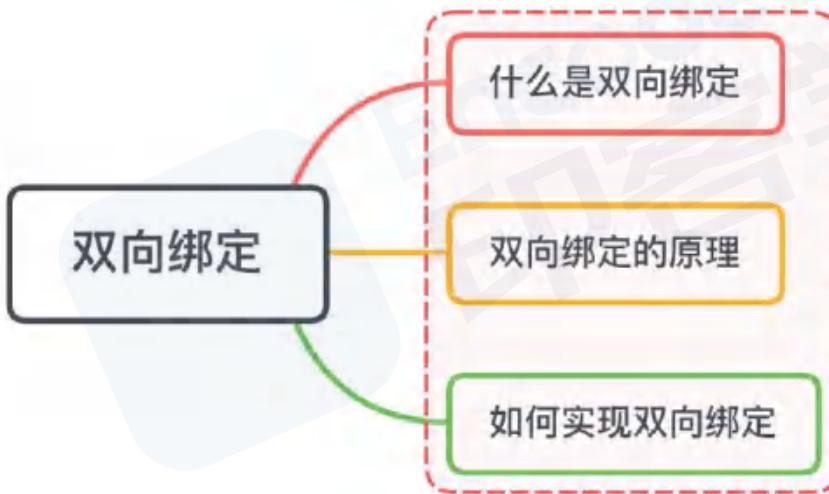
mounted	初始化结束，dom已创建，可用于获取访问数据和dom元素
beforeUpdate	更新前，可用于获取更新前各种状态
updated	更新后，所有状态已是最新
beforeDestroy	销毁前，可用于一些定时器或订阅的取消
destroyed	组件已销毁，作用同上

1.4. 数据请求在created和mouted的区别

`created` 是在组件实例一旦创建完成的时候立刻调用，这时候页面 `dom` 节点并未生成； `mounted` 是在页面 `dom` 节点渲染完毕之后就立刻执行的。触发时机上 `created` 是比 `mounted` 要更早的，两者的相同点：都能拿到实例对象的属性和方法。

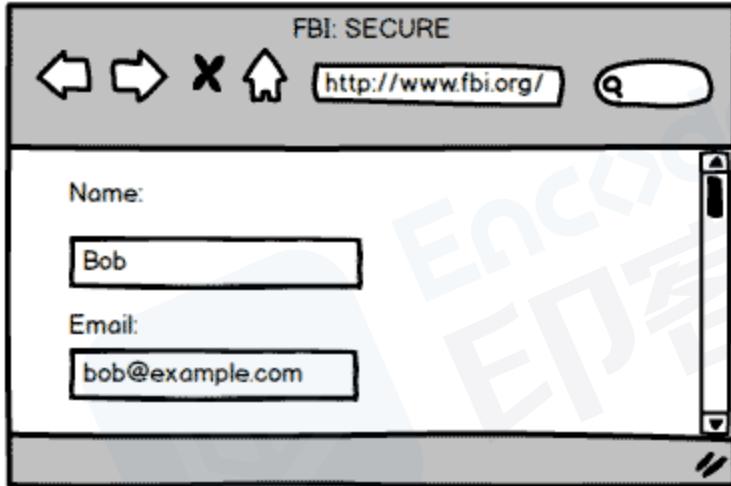
讨论这个问题本质就是触发的时机，放在 `mounted` 中的请求有可能导致页面闪动（因为此时页面 `dom` 结构已经生成），但如果在页面加载前完成请求，则不会出现此情况。建议对页面内容的改动放在 `created` 生命周期当中。

2. 双向数据绑定是什么

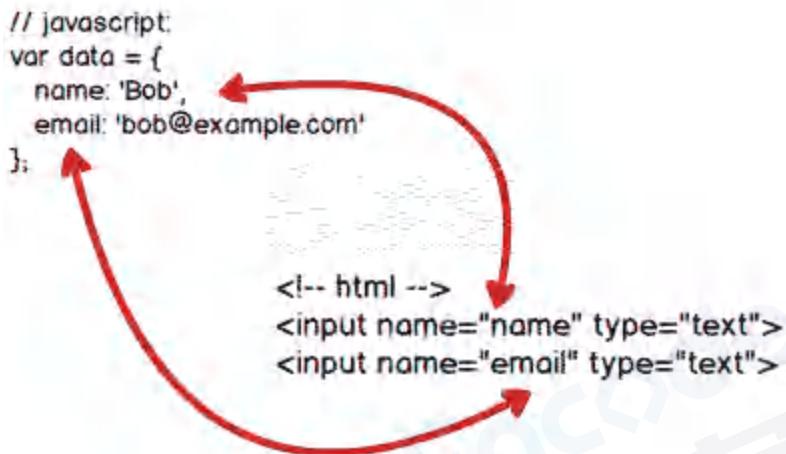


2.1. 什么是双向绑定

我们先从单向绑定切入双向绑定非常简单，就是把 **Model** 绑定到 **View**，当我们用 **JavaScript** 代码更新 **Model** 时，**View** 就会自动更新双向绑定就很容易联想到了，在单向绑定的基础上，用户更新了 **View**，**Model** 的数据也自动被更新了，这种情况就是双向绑定举个栗子



当用户填写表单时，**View** 的状态就被更新了，如果此时可以自动更新 **Model** 的状态，那就相当于我们把 **Model** 和 **View** 做了双向绑定关系图如下



2.2. 双向绑定的原理是什么

我们都知道 **Vue** 是数据双向绑定的框架，双向绑定由三个重要部分构成

- 数据层（Model）：应用的数据及业务逻辑
- 视图层（View）：应用的展示效果，各类UI组件
- 业务逻辑层（ViewModel）：框架封装的核心，它负责将数据与视图关联起来

而上面的这个分层的架构方案，可以用一个专业术语进行称呼：**MVVM** 这里的控制层的核心功能便是“数据双向绑定”。自然，我们只需弄懂它是什么，便可以进一步了解数据绑定的原理

2.2.1. 理解ViewModel

它的主要职责就是：

- 数据变化后更新视图
- 视图变化后更新数据

当然，它还有两个主要部分组成

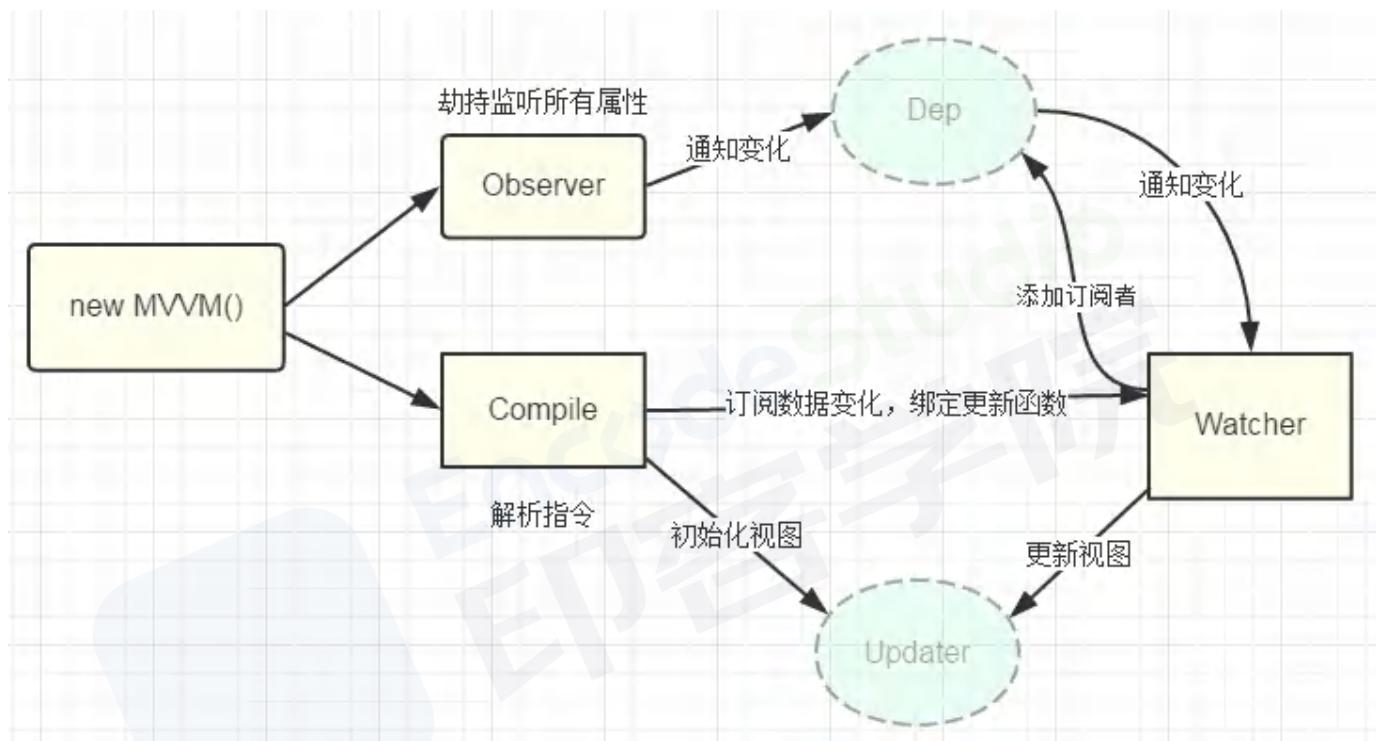
- 监听器（Observer）：对所有数据的属性进行监听
- 解析器（Compiler）：对每个元素节点的指令进行扫描跟解析，根据指令模板替换数据，以及绑定相应的更新函数

2.2.2. 实现双向绑定

我们还是以 `Vue` 为例，先来看看 `Vue` 中的双向绑定流程是什么的

1. `new Vue()` 首先执行初始化，对 `data` 执行响应化处理，这个过程发生在 `Observe` 中
2. 同时对模板执行编译，找到其中动态绑定的数据，从 `data` 中获取并初始化视图，这个过程发生在 `Compile` 中
3. 同时定义一个更新函数和 `Watcher`，将来对应数据变化时 `Watcher` 会调用更新函数
4. 由于 `data` 的某个 `key` 在一个视图中可能出现多次，所以每个 `key` 都需要一个管家 `Dep` 来管理多个 `Watcher`
5. 将来 `data` 中数据一旦发生变化，会首先找到对应的 `Dep`，通知所有 `Watcher` 执行更新函数

流程图如下：



2.2.3. 实现

先来一个构造函数：执行初始化，对 `data` 执行响应化处理

```

1  class Vue {
2    constructor(options) {
3      this.$options = options;
4      this.$data = options.data;
5
6      // 对data选项做响应式处理
7      observe(this.$data);
8
9      // 代理data到vm上
10     proxy(this);
11
12     // 执行编译
13     new Compile(options.el, this);
14   }
15 }
```

对 `data` 选项执行响应化具体操作

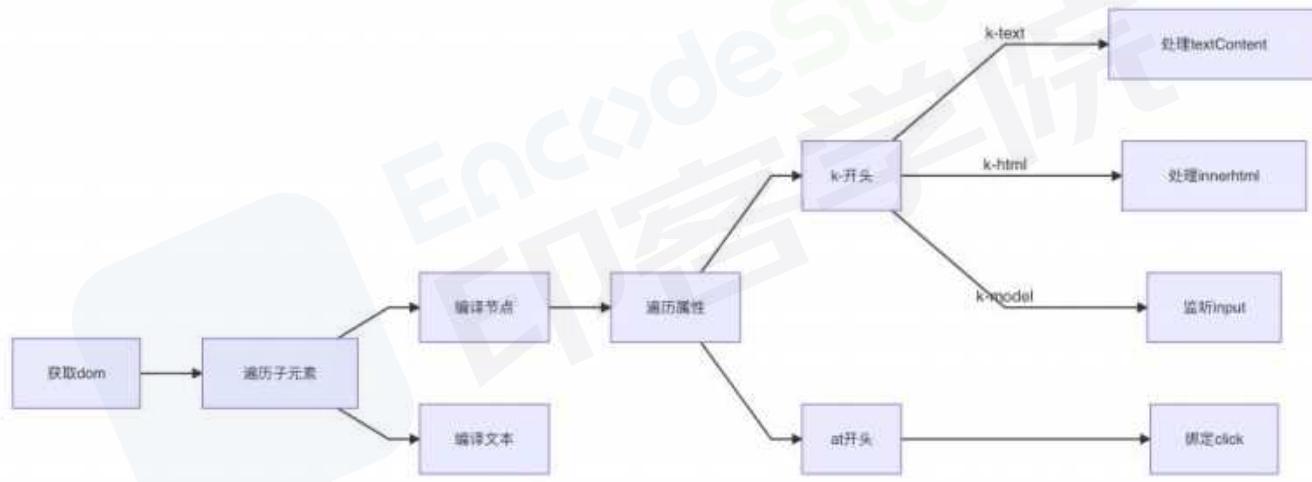
```

1  function observe(obj) {
2    if (typeof obj !== "object" || obj == null) {
3      return;
4    }
5    new Observer(obj);
6  }
7
8  class Observer {
9    constructor(value) {
10      this.value = value;
11      this.walk(value);
12    }
13    walk(obj) {
14      Object.keys(obj).forEach((key) => {
15        defineReactive(obj, key, obj[key]);
16      });
17    }
18  }

```

2.2.3.1. 编译 **Compile**

对每个元素节点的指令进行扫描跟解析,根据指令模板替换数据,以及绑定相应的更新函数



```

1  class Compile {
2    constructor(el, vm) {
3      this.$vm = vm;
4      this.$el = document.querySelector(el); // 获取dom
5      if (this.$el) {
6        this.compile(this.$el);
7      }
8    }
9    compile(el) {
10   const childNodes = el.childNodes;
11   Array.from(childNodes).forEach((node) => { // 遍历子元素
12     if (this.isElement(node)) { // 判断是否为节点
13       console.log("编译元素" + node.nodeName);
14     } else if (this.isInterpolation(node)) {
15       console.log("编译插值文本" + node.textContent); // 判断是否为插值文本
16     }
17     if (node.childNodes && node.childNodes.length > 0) { // 判断是否有子
元素
18       this.compile(node); // 对子元素进行递归遍历
19     }
20   });
21 }
22 isElement(node) {
23   return node.nodeType == 1;
24 }
25 isInterpolation(node) {
26   return node.nodeType == 3 && /\{\{(.*?)\}\}/.test(node.textContent);
27 }
28 }

```

2.2.3.2. 依赖收集

视图中会用到 `data` 中某 `key`，这称为依赖。同一个 `key` 可能出现多次，每次都需要收集出来用一个 `Watcher` 来维护它们，此过程称为依赖收集。多个 `Watcher` 需要一个 `Dep` 来管理，需要更新时由 `Dep` 统一通知。



实现思路

1. `defineReactive` 时为每一个 `key` 创建一个 `Dep` 实例
2. 初始化视图时读取某个 `key`，例如 `name1`，创建一个 `watcher1`
3. 由于触发 `name1` 的 `getter` 方法，便将 `watcher1` 添加到 `name1` 对应的 `Dep` 中
4. 当 `name1` 更新，`setter` 触发时，便可通知其管理所有 `Watcher` 更新

▼

JavaScript | 复制代码

```

1 // 负责更新视图
2 class Watcher {
3   constructor(vm, key, updater) {
4     this.vm = vm
5     this.key = key
6     this.updaterFn = updater
7
8     // 创建实例时，把当前实例指定到Dep.target静态属性上
9     Dep.target = this
10    // 读一下key，触发get
11    vm[key]
12    // 置空
13    Dep.target = null
14  }
15
16  // 未来执行dom更新函数，由dep调用的
17  update() {
18    this.updaterFn.call(this.vm, this.vm[this.key])
19  }
20}

```

声明 `Dep`

JavaScript | 复制代码

```

1  class Dep {
2    constructor() {
3      this.deps = []; // 依赖管理
4    }
5    addDep(dep) {
6      this.deps.push(dep);
7    }
8    notify() {
9      this.deps.forEach((dep) => dep.update());
10   }
11 }

```

创建 `watcher` 时触发 `getter`

JavaScript | 复制代码

```

1  class Watcher {
2    constructor(vm, key, updateFn) {
3      Dep.target = this;
4      this.vm[this.key];
5      Dep.target = null;
6    }
7  }

```

依赖收集，创建 `Dep` 实例

JavaScript | 复制代码

```

1  function defineReactive(obj, key, val) {
2    this.observe(val);
3    const dep = new Dep();
4    Object.defineProperty(obj, key, {
5      get() {
6        Dep.target && dep.addDep(Dep.target); // Dep.target也就是Watcher实例
7        return val;
8      },
9      set(newVal) {
10        if (newVal === val) return;
11        dep.notify(); // 通知dep执行更新方法
12      },
13    });
14  }

```

3. Vue组件之间的通信方式都有哪些？



3.1. 组件间通信的概念

开始之前，我们把组件间通信这个词进行拆分

- 组件
- 通信

都知道组件是 vue 最强大的功能之一， vue 中每一个 .vue 我们都可以视之为一个组件通信指的是发送者通过某种媒体以某种格式来传递信息到收信者以达到某个目的。广义上，任何信息的交通都是通信组件间通信即指组件(.vue)通过某种方式来传递信息以达到某个目的举个栗子我们在使用 UI 框架中的 table 组件，可能会往 table 组件中传入某些数据，这个本质就形成了组件之间的通信

3.2. 组件间通信解决了什么

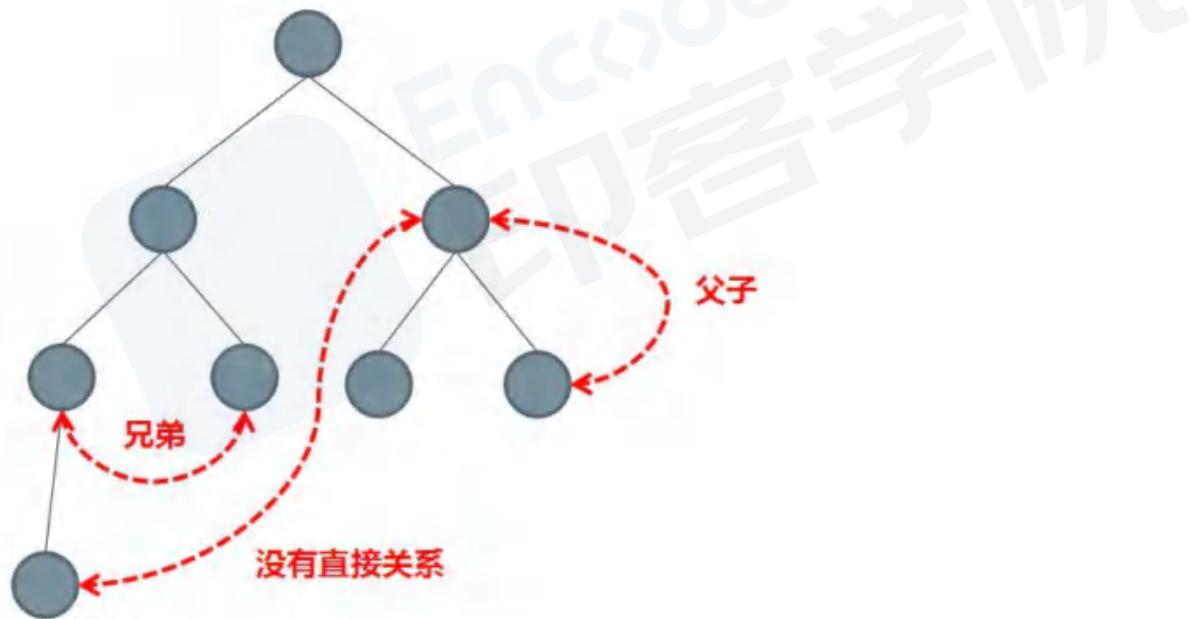
在古代，人们通过驿站、飞鸽传书、烽火报警、符号、语言、眼神、触碰等方式进行信息传递，到了今天，随着科技水平的飞速发展，通信基本完全利用有线或无线电完成，相继出现了有线电话、固定电话、无线电话、手机、互联网甚至视频电话等各种通信方式从上面这段话，我们可以看到通信的本质是信息同步，共享回到 vue 中，每个组件之间的都有独自的作用域，组件间的数据是无法共享的但实际开发工作中我们常常需要让组件之间共享数据，这也是组件通信的目的要让它们互相之间能进行通讯，这样才能构成一个有机的完整系统

3.3. 组件间通信的分类

组件间通信的分类可以分成以下

- 父子组件之间的通信
- 兄弟组件之间的通信
- 祖孙与后代组件之间的通信
- 非关系组件间之间的通信

关系图:



3.4. 组件间通信的方案

整理 vue 中8种常规的通信方案

1. 通过 props 传递
2. 通过 \$emit 触发自定义事件
3. 使用 ref
4. EventBus
5. parent 或 root
6. attrs 与 listeners
7. Provide 与 Inject
8. Vuex

3.4.1. props传递数据



- 适用场景：父组件传递数据给子组件
- 子组件设置 `props` 属性，定义接收父组件传递过来的参数
- 父组件在使用子组件标签中通过字面量来传递值

Children.vue

```
1  props:{  
2      // 字符串形式  
3      name:String // 接收的类型参数  
4      // 对象形式  
5      age:{  
6          type:Number, // 接收的类型为数值  
7          defaule:18, // 默认值为18  
8          require:true // age属性必须传递  
9      }  
10 }
```

JavaScript | 复制代码

Father.vue 组件

```
1  <Children name="jack" age=18 />
```

JavaScript | 复制代码

3.4.2. \$emit 触发自定义事件

- 适用场景：子组件传递数据给父组件
- 子组件通过 `$emit` 触发 自定义事件， `$emit` 第二个参数为传递的数值
- 父组件绑定监听器获取到子组件传递过来的参数

Chilfen.vue

▼

JavaScript

复制代码

```
1  this.$emit('add', good)
```

Father.vue

▼

JavaScript

复制代码

```
1  <Children @add="cartAdd($event)" />
```

3.4.3. ref

- 父组件在使用子组件的时候设置 `ref`
- 父组件通过设置子组件 `ref` 来获取数据

父组件

▼

JavaScript

复制代码

```
1  <Children ref="foo" />
2
3  this.$refs.foo // 获取子组件实例，通过子组件实例我们就能拿到对应的数据
```

3.4.4. EventBus

- 使用场景：兄弟组件传值
- 创建一个中央事件总线 `EventBus`
- 兄弟组件通过 `$emit` 触发自定义事件，`$emit` 第二个参数为传递的数值
- 另一个兄弟组件通过 `$on` 监听自定义事件

Bus.js

JavaScript | 复制代码

```

1 // 创建一个中央时间总线类
2 class Bus {
3   constructor() {
4     this.callbacks = {};  
      // 存放事件的名字
5   }
6   $on(name, fn) {
7     this.callbacks[name] = this.callbacks[name] || [];
8     this.callbacks[name].push(fn);
9   }
10  $emit(name, args) {
11    if (this.callbacks[name]) {
12      this.callbacks[name].forEach((cb) => cb(args));
13    }
14  }
15 }
16
17 // main.js
18 Vue.prototype.$bus = new Bus() // 将$bus挂载到vue实例的原型上
19 // 另一种方式
20 Vue.prototype.$bus = new Vue() // Vue已经实现了Bus的功能

```

Children1.vue

JavaScript | 复制代码

```
1 this.$bus.$emit('foo')
```

Children2.vue

JavaScript | 复制代码

```
1 this.$bus.$on('foo', this.handle)
```

3.4.5. parent、root

- 通过共同祖辈 `$parent` 或者 `$root` 搭建通信桥连

兄弟组件

```
this.$parent.on('add', this.add)
```

另一个兄弟组件

```
this.$parent.emit('add')
```

3.4.6. attrs与listeners

- 适用场景：祖先传递数据给子孙
- 设置批量向下传属性 `$attrs` 和 `$listeners`
- 包含了父级作用域中不作为 `prop` 被识别（且获取）的特性绑定（`class` 和 `style` 除外）。
- 可以通过 `v-bind="$attrs"` 传入内部组件

```

1 // child: 并未在props中声明foo
2 <p>{{$attrs.foo}}</p>
3
4 // parent
5 <HelloWorld foo="foo"/>

```

```

1 // 给Grandson隔代传值, communication/index.vue
2 <Child2 msg="lalala" @some-event="onSomeEvent"></Child2>
3
4 // Child2做展开
5 <Grandson v-bind="$attrs" v-on="$listeners"></Grandson>
6
7 // Grandson使用
8 <div @click="$emit('some-event', 'msg from grandson')">
9 {{msg}}
10 </div>

```

3.4.7. provide 与 inject

- 在祖先组件定义 `provide` 属性，返回传递的值
- 在后代组件通过 `inject` 接收组件传递过来的值

祖先组件

```

1 ▾ provide(){
2   return {
3     foo:'foo'
4   }
5 }

```

后代组件

▼

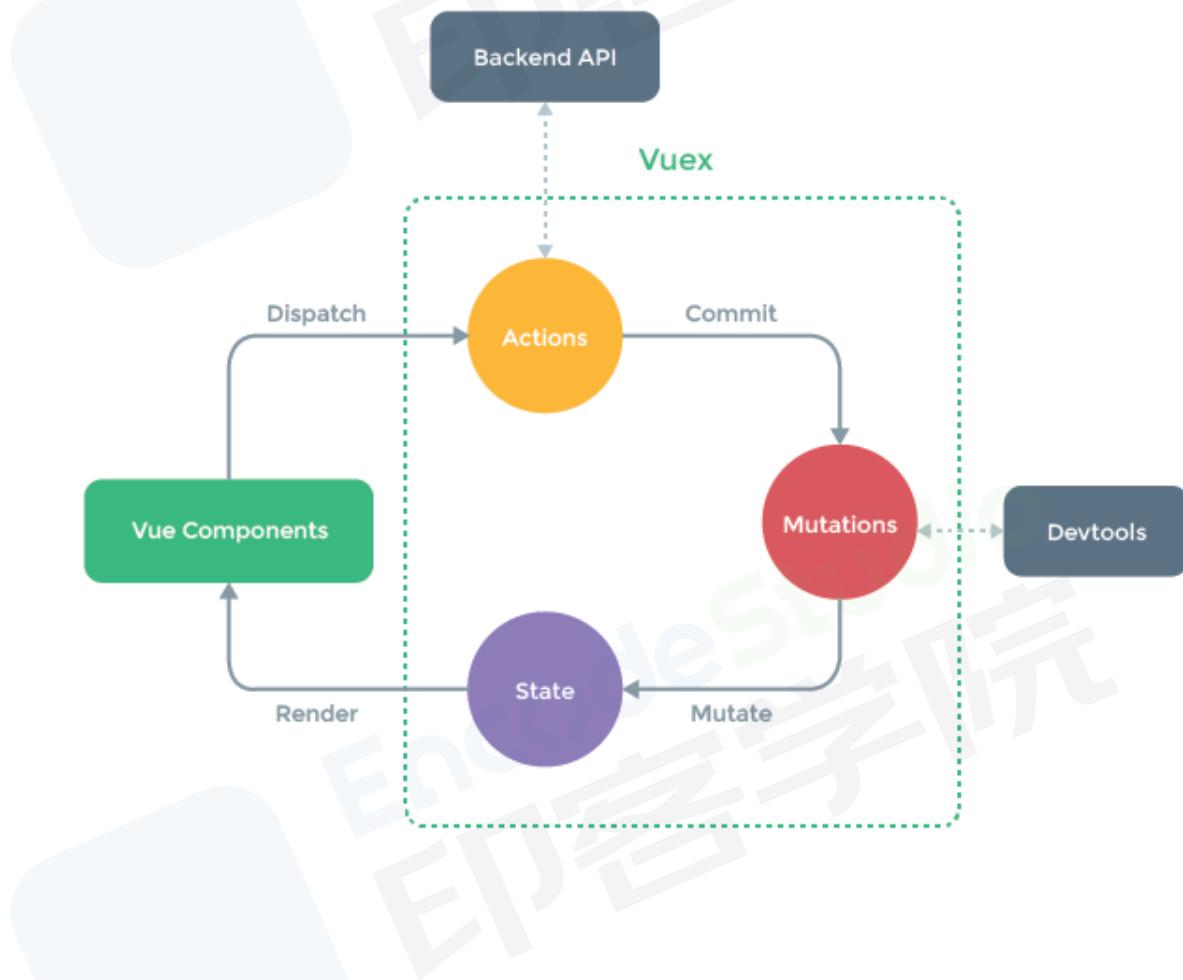
JavaScript

复制代码

```
1 inject: ['foo'] // 获取到祖先组件传递过来的值
```

3.4.8. vuex

- 适用场景: 复杂关系的组件数据传递
- Vuex 作用相当于一个用来存储共享变量的容器



- `state` 用来存放共享变量的地方
- `getter`，可以增加一个 `getter` 派生状态，(相当于 `store` 中的计算属性)，用来获得共享变量的值
- `mutations` 用来存放修改 `state` 的方法。
- `actions` 也是用来存放修改 `state` 的方法，不过 `action` 是在 `mutations` 的基础上进行。常

用来做一些异步操作

3.5. 小结

- 父子关系的组件数据传递选择 `props` 与 `$emit` 进行传递，也可选择 `ref`
- 兄弟关系的组件数据传递可选择 `$bus`，其次可以选择 `$parent` 进行传递
- 祖先与后代组件数据传递可选择 `attrs` 与 `listeners` 或者 `Provide` 与 `Inject`
- 复杂关系的组件数据传递可以通过 `vuex` 存放共享的变量

4. 为什么data属性是一个函数而不是一个对象？



4.1. 实例和组件定义data的区别

`vue` 实例的时候定义 `data` 属性既可以是一个对象，也可以是一个函数

JavaScript | 复制代码

```

1 const app = new Vue({
2   el:"#app",
3   // 对象格式
4   data:{
5     foo:"foo"
6   },
7   // 函数格式
8   data(){
9     return {
10       foo:"foo"
11     }
12   }
13 })

```

组件中定义 `data` 属性，只能是一个函数

如果为组件 `data` 直接定义为一个对象

JavaScript | 复制代码

```

1 Vue.component('component1',{
2   template:`<div>组件</div>`,
3   data:{
4     foo:"foo"
5   }
6 })

```

则会得到警告信息

⚠ [Vue warn]: The "data" option should be a function that returns a per-instance value in component definitions. vue.js:634

警告说明：返回的 `data` 应该是一个函数在每一个组件实例中

4.2. 组件data定义函数与对象的区别

上面讲到组件 `data` 必须是一个函数，不知道大家有没有思考过这是为什么呢？

在我们定义好一个组件的时候，`vue` 最终都会通过 `Vue.extend()` 构成组件实例

这里我们模仿组件构造函数，定义 `data` 属性，采用对象的形式

JavaScript | 复制代码

```
1 function Component(){
2
3 }
4 Component.prototype.data = {
5   count : 0
6 }
```

创建两个组件实例

Plain Text | 复制代码

```
1 const componentA = new Component()
2 const componentB = new Component()
```

修改 `componentA` 组件 `data` 属性的值，`componentB` 中的值也发生了改变

JavaScript | 复制代码

```
1 console.log(componentB.data.count) // 0
2 componentA.data.count = 1
3 console.log(componentB.data.count) // 1
```

产生这样的原因这是两者共用了同一个内存地址，`componentA` 修改的内容，同样对 `componentB` 产生了影响

如果我们采用函数的形式，则不会出现这种情况（函数返回的对象内存地址并不相同）

JavaScript | 复制代码

```
1 function Component(){
2   this.data = this.data()
3 }
4 Component.prototype.data = function (){
5   return {
6     count : 0
7   }
8 }
```

修改 `componentA` 组件 `data` 属性的值，`componentB` 中的值不受影响

JavaScript | 复制代码

```

1 console.log(componentB.data.count) // 0
2 componentA.data.count = 1
3 console.log(componentB.data.count) // 0

```

`vue` 组件可能会有很多个实例，采用函数返回一个全新 `data` 形式，使每个实例对象的数据不会受到其他实例对象数据的污染

4.3. 原理分析

首先可以看看 `vue` 初始化 `data` 的代码，`data` 的定义可以是函数也可以是对象

源码位置： [/vue-dev/src/core-instance/state.js](#)

```

1 function initData (vm: Component) {
2   let data = vm.$options.data
3   data = vm._data = typeof data === 'function'
4     ? getData(data, vm)
5     : data || {}
6   ...
7 }

```

`data` 既能是 `object` 也能是 `function`，那为什么还会出现上文警告呢？

别急，继续看下文

组件在创建的时候，会进行选项的合并

源码位置： [/vue-dev/src/core/util/options.js](#)

自定义组件会进入 `mergeOptions` 进行选项合并

JavaScript | 复制代码

```
1  Vue.prototype._init = function (options?: Object) {
2    ...
3    // merge options
4    if (options && options._isComponent) {
5      // optimize internal component instantiation
6      // since dynamic options merging is pretty slow, and none of the
7      // internal component options needs special treatment.
8      initInternalComponent(vm, options)
9    } else {
10      vm.$options = mergeOptions(
11        resolveConstructorOptions(vm.constructor),
12        options || {},
13        vm
14      )
15    }
16    ...
17  }
```

定义 `data` 会进行数据校验

源码位置: `/vue-dev/src/core/instance/init.js`

这时候 `vm` 实例为 `undefined`，进入 `if` 判断，若 `data` 类型不是 `function`，则出现警告提示

```

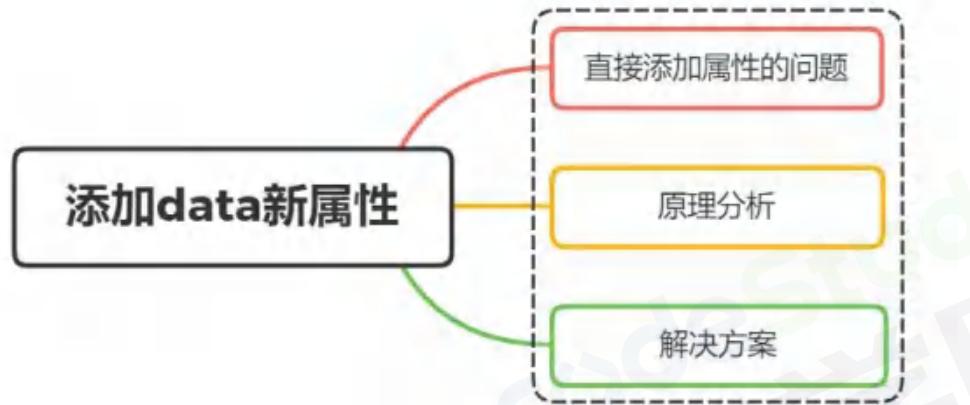
1 strats.data = function (
2   parentVal: any,
3   childVal: any,
4   vm?: Component
5 ): ?Function {
6   if (!vm) {
7     if (childVal && typeof childVal !== "function") {
8       process.env.NODE_ENV !== "production" &&
9       warn(
10         'The "data" option should be a function ' +
11         "that returns a per-instance value in component " +
12         "definitions.",
13         vm
14       );
15     }
16     return parentVal;
17   }
18   return mergeDataOrFn(parentVal, childVal);
19 }
20 return mergeDataOrFn(parentVal, childVal, vm);
21 };

```

4.4. 结论

- 根实例对象 `data` 可以是对象也可以是函数（根实例是单例），不会产生数据污染情况
- 组件实例对象 `data` 必须为函数，目的是为了防止多个组件实例对象之间共用一个 `data`，产生数据污染。采用函数的形式，`initData` 时会将其作为工厂函数都会返回全新 `data` 对象

5. 动态给vue的data添加一个新的属性时会发生什么？怎样解决？



5.1. 直接添加属性的问题

我们从一个例子开始

定义一个 `p` 标签，通过 `v-for` 指令进行遍历

然后给 `button` 标签绑定点击事件，我们预期点击按钮时，数据新增一个属性，界面也 新增一行

```
HTML | 复制代码
```

```
1 <p v-for="(value,key) in item" :key="key">
2   {{ value }}
3 </p>
4 <button @click="addProperty">动态添加新属性</button>
```

实例化一个 `Vue` 实例，定义 `data` 属性和 `methods` 方法

```
JavaScript | 复制代码
```

```
1 const app = new Vue({
2   el:"#app",
3   data:()=>{
4     item:{
5       oldProperty:"旧属性"
6     }
7   },
8   methods:{
9     addProperty(){
10       this.items.newProperty = "新属性" // 为items添加新属性
11       console.log(this.items) // 输出带有newProperty的items
12     }
13   }
14 })
```

点击按钮，发现结果不及预期，数据虽然更新了（`console` 打印出了新属性），但页面并没有更新

5.2. 原理分析

为什么产生上面的情况呢？

下面来分析一下

`vue2` 是用过 `Object.defineProperty` 实现数据响应式

```
1 const obj = {}
2 Object.defineProperty(obj, 'foo', {
3   get() {
4     console.log(`get foo:${val}`);
5     return val
6   },
7   set(newVal) {
8     if (newVal !== val) {
9       console.log(`set foo:${newVal}`);
10      val = newVal
11    }
12  }
13})
14 }
```

当我们访问 `foo` 属性或者设置 `foo` 值的时候都能够触发 `setter` 与 `getter`

```
1 obj.foo
2 obj.foo = 'new'
```

但是我们为 `obj` 添加新属性的时候，却无法触发事件属性的拦截

```
1 obj.bar = '新属性'
```

原因是一开始 `obj` 的 `foo` 属性被设成了响应式数据，而 `bar` 是后面新增的属性，并没有通过 `Object.defineProperty` 设置成响应式数据

5.3. 解决方案

Vue 不允许在已经创建的实例上动态添加新的响应式属性

若想实现数据与视图同步更新，可采取下面三种解决方案：

- Vue.set()
- Object.assign()
- \$forceUpdate()

5.3.1. Vue.set()

Vue.set(target, propertyName/index, value)

参数

- {Object | Array} target
- {string | number} propertyName/index
- {any} value

返回值：设置的值

通过 `Vue.set` 向响应式对象中添加一个 `property`，并确保这个新 `property` 同样是响应式的，且触发视图更新

关于 `Vue.set` 源码（省略了很多与本节不相关的代码）

源码位置：`src\core\observer\index.js`

```
1 function set (target: Array<any> | Object, key: any, val: any): any {
2   ...
3   defineReactive(ob.value, key, val)
4   ob.dep.notify()
5   return val
6 }
```

这里无非再次调用 `defineReactive` 方法，实现新增属性的响应式

关于 `defineReactive` 方法，内部还是通过 `Object.defineProperty` 实现属性拦截

大致代码如下：

JavaScript | 复制代码

```

1  function defineReactive(obj, key, val) {
2    Object.defineProperty(obj, key, {
3      get() {
4        console.log(`get ${key}:${val}`);
5        return val
6      },
7      set(newVal) {
8        if (newVal !== val) {
9          console.log(`set ${key}:${newVal}`);
10         val = newVal
11       }
12     }
13   })
14 }
```

5.3.2. Object.assign()

直接使用 `Object.assign()` 添加到对象的新属性不会触发更新

应创建一个新的对象，合并原对象和混入对象的属性

JavaScript | 复制代码

```

1  this.someObject = Object.assign({}, this.someObject, {newProperty1:1,newPrope
rty2:2 ...})
```

5.3.3. \$forceUpdate

如果你发现自己需要在 `Vue` 中做一次强制更新，99.9% 的情况，是你在某个地方做错了事

`$forceUpdate` 迫使 `Vue` 实例重新渲染

PS：仅仅影响实例本身和插入插槽内容的子组件，而不是所有子组件。

5.4. 小结

- 如果为对象添加少量的新属性，可以直接采用 `Vue.set()`
- 如果需要为新对象添加大量的新属性，则通过 `Object.assign()` 创建新对象
- 如果你实在不知道怎么操作时，可采取 `$forceUpdate()` 进行强制刷新（不建议）

PS: vue3 是用过 proxy 实现数据响应式的，直接动态添加新属性仍可以实现数据响应式

6. v-if和v-for的优先级是什么？



6.1. 作用

v-if 指令用于条件性地渲染一块内容。这块内容只会在指令的表达式返回 true 值的时候被渲染

v-for 指令基于一个数组来渲染一个列表。v-for 指令需要使用 item in items 形式的特殊语法，其中 items 是源数据数组或者对象，而 item 则是被迭代的数组元素的别名

在 v-for 的时候，建议设置 key 值，并且保证每个 key 值是独一无二的，这便于 diff 算法进行优化

两者在用法上

```
1 <Modal v-if="isShow" />
2
3 <li v-for="item in items" :key="item.id">
4   {{ item.label }}
5 </li>
```

JavaScript | 复制代码

6.2. 优先级

v-if 与 v-for 都是 vue 模板系统中的指令

在 vue 模板编译的时候，会将指令系统转化成可执行的 render 函数

6.2.1. 示例

编写一个 `p` 标签，同时使用 `v-if` 与 `v-for`

```
HTML | 复制代码

1 <div id="app">
2   <p v-if="isShow" v-for="item in items">
3     {{ item.title }}
4   </p>
5 </div>
```

创建 `Vue` 实例，存放 `isShow` 与 `items` 数据

```
JavaScript | 复制代码

1 const app = new Vue({
2   el: "#app",
3   data() {
4     return {
5       items: [
6         { title: "foo" },
7         { title: "baz" }
8       ]
9     },
10  computed: {
11    isShow() {
12      return this.items && this.items.length > 0
13    }
14  }
15})
```

模板指令的代码都会生成在 `render` 函数中，通过 `app.$options.render` 就能得到渲染函数

```
JavaScript | 复制代码

1 f anonymous() {
2   with (this) { return
3     _c('div', { attrs: { id: "app" } },
4     _l(items, function (item)
5       { return (isShow) ? _c('p', [_v("\n" + _s(item.title) + "\n")]) : _e()
6     }), 0)
7 }
```

`_l` 是 `Vue` 的列表渲染函数，函数内部都会进行一次 `if` 判断

初步得到结论：`v-for` 优先级是比 `v-if` 高

再将 `v-for` 与 `v-if` 置于不同标签

```
1 <div id="app">
2   <template v-if="isShow">
3     <p v-for="item in items">{{item.title}}</p>
4   </template>
5 </div>
```

再输出下 `render` 函数

```
1 f anonymous() {
2   with(this){return
3     _c('div',{attrs:{'id':"app"}},
4     [(isShow)?[_v("\n"),
5       _l((items),function(item){return _c('p',[_v(_s(item.title))])})]:_e()],
6   2)}
7 }
```

这时候我们可以看到，`v-for` 与 `v-if` 作用在不同标签时候，是先进行判断，再进行列表的渲染

我们再在查看下 `vue` 源码

源码位置： `\vue-dev\src\compiler\codegen\index.js`

JavaScript | 复制代码

```

1 export function genElement (el: ASTElement, state: CodegenState): string {
2   if (el.parent) {
3     el.pre = el.pre || el.parent.pre
4   }
5   if (el.staticRoot && !el.staticProcessed) {
6     return genStatic(el, state)
7   } else if (el.once && !el.onceProcessed) {
8     return genOnce(el, state)
9   } else if (el.for && !el.forProcessed) {
10    return genFor(el, state)
11  } else if (el.if && !el.ifProcessed) {
12    return genIf(el, state)
13  } else if (el.tag === 'template' && !el.slotTarget && !state.pre) {
14    return genChildren(el, state) || 'void 0'
15  } else if (el.tag === 'slot') {
16    return genSlot(el, state)
17  } else {
18    // component or element
19    ...
20  }

```

在进行 `if` 判断的时候，`v-for` 是比 `v-if` 先进行判断

最终结论：`v-for` 优先级比 `v-if` 高

6.3. 注意事项

- 永远不要把 `v-if` 和 `v-for` 同时用在同一个元素上，带来性能方面的浪费（每次渲染都会先循环再进行条件判断）
- 如果避免出现这种情况，则在外层嵌套 `template`（页面渲染不生成 `dom` 节点），在这一层进行 `v-if` 判断，然后在内部进行 `v-for` 循环

JavaScript | 复制代码

```

1 <template v-if="isShow">
2   <p v-for="item in items">
3     </template>

```

- 如果条件出现在循环内部，可通过计算属性 `computed` 提前过滤掉那些不需要显示的项

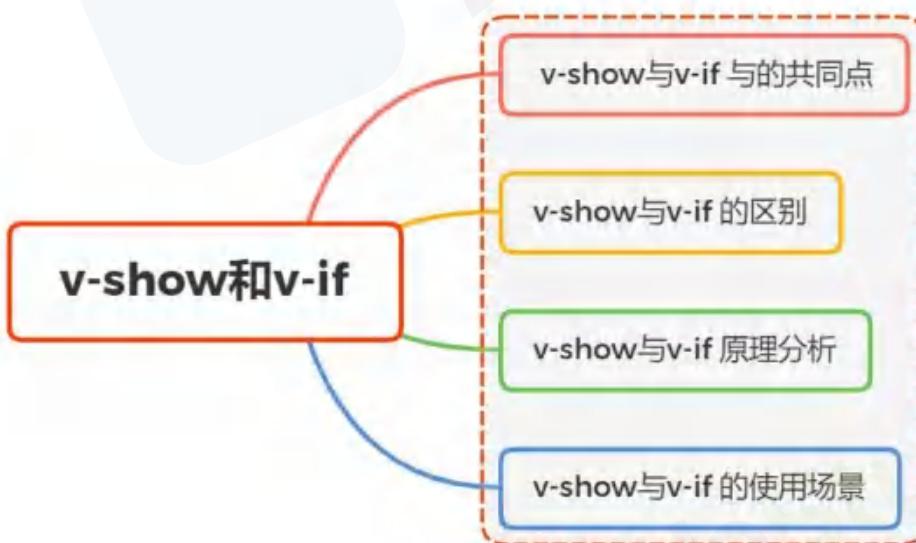
JavaScript | 复制代码

```

1  computed: {
2    items: function() {
3      return this.list.filter(function (item) {
4        return item.isShow
5      })
6    }
7  }

```

7. v-show和v-if有什么区别？使用场景分别是什么？



7.1. v-show与v-if的共同点

我们都知道在 vue 中 v-show 与 v-if 的作用效果是相同的(不含v-else)，都能控制元素在页面是否显示

在用法上也是相同的

JavaScript | 复制代码

```

1 <Model v-show="isShow" />
2 <Model v-if="isShow" />

```

- 当表达式为 true 的时候，都会占据页面的位置

- 当表达式都为 `false` 时，都不会占据页面位置

7.2. v-show与v-if的区别

- 控制手段不同
- 编译过程不同
- 编译条件不同

控制手段：`v-show` 隐藏则是为该元素添加 `css--display:none`，`dom` 元素依旧还在。`v-if` 显示隐藏是将 `dom` 元素整个添加或删除

编译过程：`v-if` 切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件；`v-show` 只是简单的基于css切换

编译条件：`v-if` 是真正的条件渲染，它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。只有渲染条件为假时，并不做操作，直到为真才渲染

- `v-show` 由 `false` 变为 `true` 的时候不会触发组件的生命周期
- `v-if` 由 `false` 变为 `true` 的时候，触发组件的 `beforeCreate`、`create`、`beforeMount`、`mounted` 钩子，由 `true` 变为 `false` 的时候触发组件的 `beforeDestory`、`destoryed` 方法

性能消耗：`v-if` 有更高的切换消耗；`v-show` 有更高的初始渲染消耗；

7.3. v-show与v-if原理分析

具体解析流程这里不展开讲，大致流程如下

- 将模板 `template` 转为 `ast` 结构的 `JS` 对象
- 用 `ast` 得到的 `JS` 对象拼装 `render` 和 `staticRenderFns` 函数
- `render` 和 `staticRenderFns` 函数被调用后生成虚拟 `VNODE` 节点，该节点包含创建 `DOM` 节点所需信息
- `vm.patch` 函数通过虚拟 `DOM` 算法利用 `VNODE` 节点创建真实 `DOM` 节点

7.3.1. v-show原理

不管初始条件是什么，元素总是会被渲染

我们看一下在 `vue` 中是如何实现的

代码很好理解，有 `transition` 就执行 `transition`，没有就直接设置 `display` 属性

JavaScript | 复制代码

```

1 // https://github.com/vuejs/vue-next/blob/3cd30c5245da0733f9eb6f29d220f39c
2 // 46518162/packages/runtime-dom/src/directives/vShow.ts
3 export const vShow: ObjectDirective<VShowElement> = {
4   beforeMount(el, { value }, { transition }) {
5     el._vod = el.style.display === 'none' ? '' : el.style.display
6     if (transition && value) {
7       transition.beforeEnter(el)
8     } else {
9       setDisplay(el, value)
10    }
11  },
12  mounted(el, { value }, { transition }) {
13    if (transition && value) {
14      transition.enter(el)
15    }
16  },
17  updated(el, { value, oldValue }, { transition }) {
18    // ...
19  },
20  beforeUnmount(el, { value }) {
21    setDisplay(el, value)
22  }

```

7.3.2. v-if原理

`v-if` 在实现上比 `v-show` 要复杂的多，因为还有 `else` `else-if` 等条件需要处理，这里我们也只摘抄源码中处理 `v-if` 的一小部分

返回一个 `node` 节点，`render` 函数通过表达式的值来决定是否生成 `DOM`

```

1 // https://github.com/vuejs/vue-next/blob/cdc9f336fd/packages/compiler-core/src/transforms/vIf.ts
2 export const transformIf = createStructuralDirectiveTransform(
3   /^(if|else|else-if)$/,
4   (node, dir, context) => {
5     return processIf(node, dir, context, (ifNode, branch, isRoot) => {
6       // ...
7       return () => {
8         if (isRoot) {
9           ifNode.codegenNode = createCodegenNodeForBranch(
10             branch,
11             key,
12             context
13             ) as IfConditionalExpression
14         } else {
15           // attach this branch's codegen node to the v-if root.
16           const parentCondition = getParentCondition(ifNode.codegenNode!)
17           parentCondition.alternate = createCodegenNodeForBranch(
18             branch,
19             key + ifNode.branches.length - 1,
20             context
21             )
22         }
23       }
24     })
25   }
26 )

```

7.4. v-show与v-if的使用场景

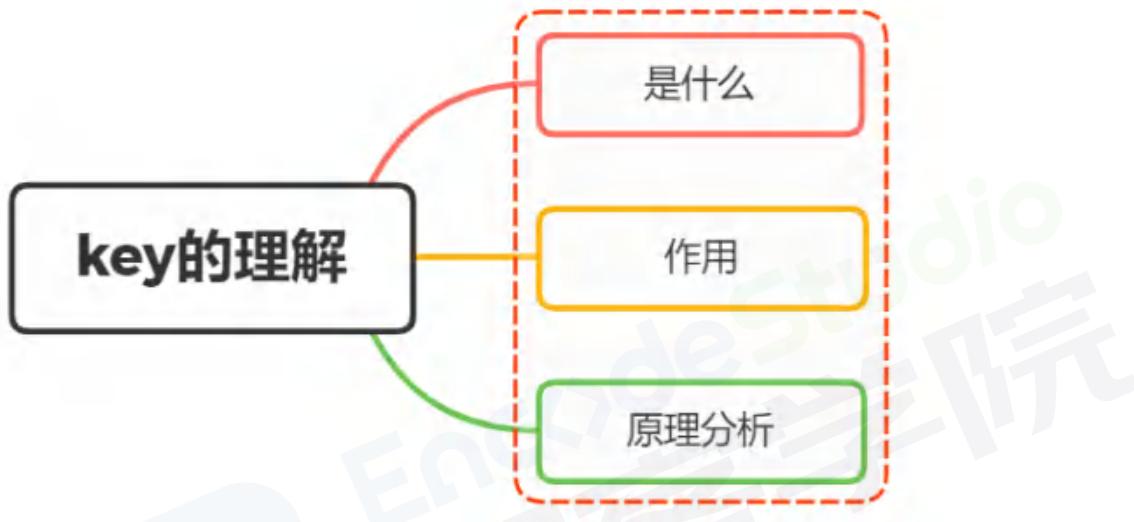
v-if 与 v-show 都能控制 dom 元素在页面的显示

v-if 相比 v-show 开销更大的（直接操作 dom 节点增加与删除）

如果需要非常频繁地切换，则使用 v-show 较好

如果在运行时条件很少改变，则使用 v-if 较好

8. 你知道vue中key的原理吗？说说你对它的理解



8.1. Key是什么

开始之前，我们先还原两个实际工作场景

1. 当我们在使用 `v-for` 时，需要给单元加上 `key`

```
1 <ul>
2   <li v-for="item in items" :key="item.id">...</li>
3 </ul>
```

2. 用 `+new Date()` 生成的时间戳作为 `key`，手动强制触发重新渲染

```
1 <Comp :key="'+new Date()'" />
```

那么这背后的逻辑是什么，`key` 的作用又是什么？

一句话来讲

key是给每一个vnode的唯一id，也是diff的一种优化策略，可以根据key，更准确，更快的找到对应的vnode节点

8.1.1. 场景背后的逻辑

当我们在使用 `v-for` 时，需要给单元加上 `key`

- 如果不用key，Vue会采用就地复用原则：最小化element的移动，并且会尝试尽最大程度在同适当

的地方对相同类型的element，做patch或者reuse。

- 如果使用了key，Vue会根据keys的顺序记录element，曾经拥有了key的element如果不再出现的话，会被直接remove或者destroyed

用`+new Date()`生成的时间戳作为`key`，手动强制触发重新渲染

- 当拥有新值的rerender作为key时，拥有了新key的Comp出现了，那么旧key Comp会被移除，新key Comp触发渲染

8.2. 设置key与不设置key区别

举个例子：

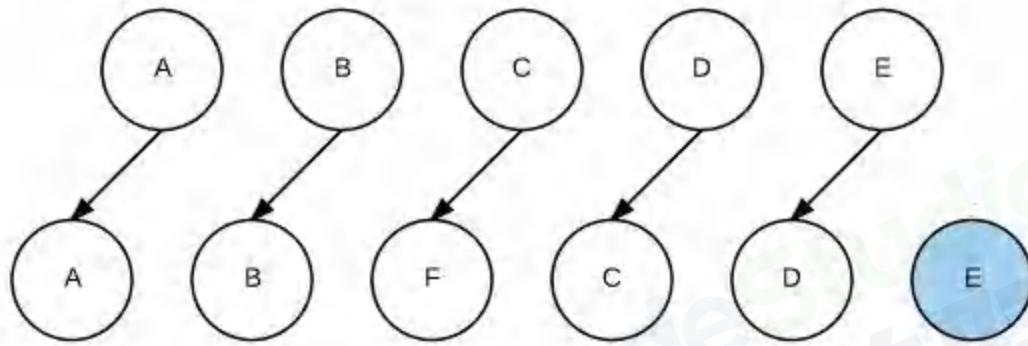
创建一个实例，2秒后往`items`数组插入数据

```

1  <body>
2    <div id="demo">
3      <p v-for="item in items" :key="item">{{item}}</p>
4    </div>
5    <script src="../../dist/vue.js"></script>
6    <script>
7      // 创建实例
8      const app = new Vue({
9        el: '#demo',
10       data: { items: ['a', 'b', 'c', 'd', 'e'] },
11       mounted () {
12         setTimeout(() => {
13           this.items.splice(2, 0, 'f') // 
14         }, 2000);
15       },
16     });
17   </script>
18 </body>

```

在不使用`key`的情况下，`vue`会进行这样的操作：



分析下整体流程：

- 比较A, A, 相同类型的节点, 进行 `patch`, 但数据相同, 不发生 `dom` 操作
- 比较B, B, 相同类型的节点, 进行 `patch`, 但数据相同, 不发生 `dom` 操作
- 比较C, F, 相同类型的节点, 进行 `patch`, 数据不同, 发生 `dom` 操作
- 比较D, C, 相同类型的节点, 进行 `patch`, 数据不同, 发生 `dom` 操作
- 比较E, D, 相同类型的节点, 进行 `patch`, 数据不同, 发生 `dom` 操作
- 循环结束, 将E插入到 `DOM` 中

一共发生了3次更新, 1次插入操作

在使用 `key` 的情况: `vue` 会进行这样的操作:

- 比较A, A, 相同类型的节点, 进行 `patch`, 但数据相同, 不发生 `dom` 操作
- 比较B, B, 相同类型的节点, 进行 `patch`, 但数据相同, 不发生 `dom` 操作
- 比较C, F, 不相同类型的节点
 - 比较E、E, 相同类型的节点, 进行 `patch`, 但数据相同, 不发生 `dom` 操作
- 比较D、D, 相同类型的节点, 进行 `patch`, 但数据相同, 不发生 `dom` 操作
- 比较C、C, 相同类型的节点, 进行 `patch`, 但数据相同, 不发生 `dom` 操作
- 循环结束, 将F插入到C之前

一共发生了0次更新, 1次插入操作

通过上面两个小例子, 可见设置 `key` 能够大大减少对页面的 `DOM` 操作, 提高了 `diff` 效率

8.2.1. 设置key值一定能提高diff效率吗?

其实不然, 文档中也明确表示

当 `Vue.js` 用 `v-for` 正在更新已渲染过的元素列表时, 它默认用“就地复用”策略。如果数据项的顺序被改变, `Vue` 将不会移动 `DOM` 元素来匹配数据项的顺序, 而是简单复用此处每个元素, 并且确保它在

特定索引下显示已被渲染过的每个元素

这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时 DOM 状态（例如：表单输入值）的列表渲染输出

建议尽可能在使用 `v-for` 时提供 `key`，除非遍历输出的 DOM 内容非常简单，或者是刻意依赖默认行为以获取性能上的提升

8.3. 原理分析

源码位置：core/vdom/patch.js

这里判断是否为同一个 `key`，首先判断的是 `key` 值是否相等如果没有设置 `key`，那么 `key` 为 `undefined`，这时候 `undefined` 是恒等于 `undefined`

```

1  function sameVnode (a, b) {
2      return (
3          a.key === b.key && (
4              (
5                  a.tag === b.tag &&
6                  a.isComment === b.isComment &&
7                  isDef(a.data) === isDef(b.data) &&
8                  sameInputType(a, b)
9              ) || (
10                  isTrue(a.isAsyncPlaceholder) &&
11                  a.asyncFactory === b.asyncFactory &&
12                  isUndef(b.asyncFactory.error)
13              )
14          )
15      )
16  }

```

`updateChildren` 方法中会对新旧 `vnode` 进行 `diff`，然后将比对出的结果用来更新真实的 `DOM`

JavaScript | 复制代码

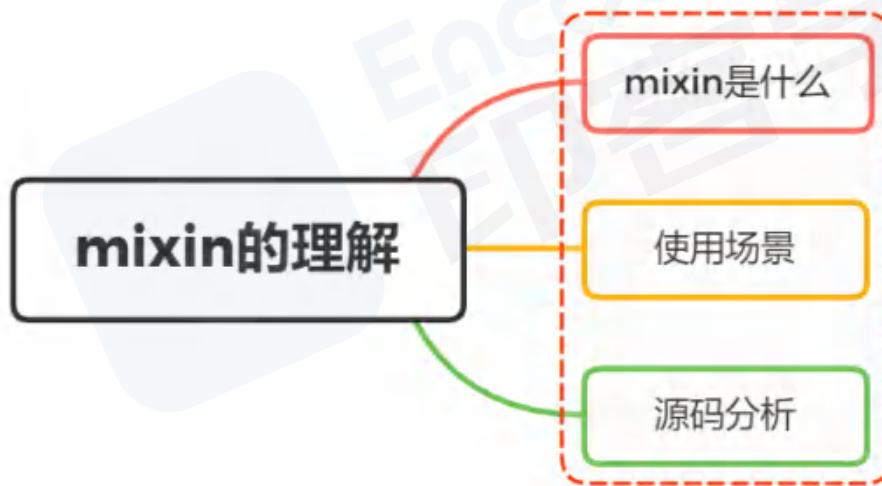
```

1  function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue, removeOnly) {
2      ...
3      while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
4          if (isUndef(oldStartVnode)) {
5              ...
6          } else if (isUndef(oldEndVnode)) {
7              ...
8          } else if (sameVnode(oldStartVnode, newStartVnode)) {
9              ...
10         } else if (sameVnode(oldEndVnode, newEndVnode)) {
11             ...
12         } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode move
13             d right
14             ...
15         } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode move
16             d left
17             ...
18         } else {
19             if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldC
20             h, oldStartIdx, oldEndIdx)
21             idxInOld = isDef(newStartVnode.key)
22             ? oldKeyToIdx[newStartVnode.key]
23             : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndId
24             x)
25             if (isUndef(idxInOld)) { // New element
26                 createElm(newStartVnode, insertedVnodeQueue, parentElm, ol
27                 dStartVnode.elm, false, newCh, newStartIdx)
28             } else {
29                 vnodeToMove = oldCh[idxInOld]
30                 if (sameVnode(vnodeToMove, newStartVnode)) {
31                     patchVnode(vnodeToMove, newStartVnode, insertedVnodeQu
32                     eue, newCh, newStartIdx)
33                     oldCh[idxInOld] = undefined
34                     canMove && nodeOps.insertBefore(parentElm, vnodeToMove
35                     .elm, oldStartVnode.elm)
36                 } else {
37                     // same key but different element. treat as new elemen
38                     t
39                     createElm(newStartVnode, insertedVnodeQueue, parentElm
40                     , oldStartVnode.elm, false, newCh, newStartIdx)
41                 }
42             }
43             newStartVnode = newCh[++newStartIdx]
44         }
45     }
46 }

```

```
36      }  
37      ...  
38  }
```

9. 说说你对vue的Mixin的理解，有什么应用场景？



9.1. mixin是什么

Mixin 是面向对象程序设计语言中的类，提供了方法的实现。其他类可以访问 mixin 类的方法而不必成为其子类

Mixin 类通常作为功能模块使用，在需要该功能时“混入”，有利于代码复用又避免了多继承的复杂

9.1.1. Vue中的mixin

先来看一下官方定义

mixin (混入)，提供了一种非常灵活的方式，来分发 Vue 组件中的可复用功能。

本质其实就是一个 js 对象，它可以包含我们组件中任意功能选项，如 data 、 components 、 methods 、 created 、 computed 等等

我们只要将共用的功能以对象的方式传入 mixins 选项中，当组件使用 mixins 对象时所有 mixins 对象的选项都将被混入该组件本身的选项中来

在 Vue 中我们可以局部混入跟全局混入

9.1.2. 局部混入

定义一个 `mixin` 对象，有组件 `options` 的 `data`、`methods` 属性

```
▼ JavaScript | 复制代码  
1 var myMixin = {  
2   created: function () {  
3     this.hello()  
4   },  
5   methods: {  
6     hello: function () {  
7       console.log('hello from mixin!')  
8     }  
9   }  
10 }  
  
```

组件通过 `mixins` 属性调用 `mixin` 对象

```
▼ JavaScript | 复制代码  
1 Vue.component('componentA', {  
2   mixins: [myMixin]  
3 })  
  
```

该组件在使用的时候，混合了 `mixin` 里面的方法，在自动执行 `created` 生命钩子，执行 `hello` 方法

9.1.3. 全局混入

通过 `Vue.mixin()` 进行全局的混入

```
▼ JavaScript | 复制代码  
1 Vue.mixin({  
2   created: function () {  
3     console.log("全局混入")  
4   }  
5 })  
  
```

使用全局混入需要特别注意，因为它会影响到每一个组件实例（包括第三方组件）

PS：全局混入常用于插件的编写

9.1.4. 注意事项：

当组件存在与 `.mixin` 对象相同的选项的时候，进行递归合并的时候组件的选项会覆盖 `Mixin` 的选项
但是如果相同选项为生命周期钩子的时候，会合并成一个数组，先执行 `Mixin` 的钩子，再执行组件的钩子

9.2. 使用场景

在日常的开发中，我们经常会遇到在不同的组件中经常会需要用到一些相同或者相似的代码，这些代码的功能相对独立

这时，可以通过 `Vue` 的 `Mixin` 功能将相同或者相似的代码提出来

举个例子

定义一个 `modal` 弹窗组件，内部通过 `isShowing` 来控制显示

```
▼ JavaScript | ⌂ 复制代码
1  const Modal = {
2    template: '#modal',
3    data() {
4      return {
5        isShowing: false
6      }
7    },
8    methods: {
9      toggleShow() {
10        this.isShowing = !this.isShowing;
11      }
12    }
13  }
```

定义一个 `tooltip` 提示框，内部通过 `isShowing` 来控制显示

JavaScript | 复制代码

```
1 const Tooltip = {
2   template: '#tooltip',
3   data() {
4     return {
5       isShowing: false
6     }
7   },
8   methods: {
9     toggleShow() {
10       this.isShowing = !this.isShowing;
11     }
12   }
13 }
```

通过观察上面两个组件，发现两者的逻辑是相同的，代码控制显示也是相同的，这时候 `mixin` 就派上用场了

首先抽出共同代码，编写一个 `mixin`

JavaScript | 复制代码

```
1 const toggle = {
2   data() {
3     return {
4       isShowing: false
5     }
6   },
7   methods: {
8     toggleShow() {
9       this.isShowing = !this.isShowing;
10    }
11  }
12 }
```

两个组件在用上，只需要引入 `mixin`

JavaScript | 复制代码

```

1 const Modal = {
2   template: '#modal',
3   mixins: [toggle]
4 };
5
6 const Tooltip = {
7   template: '#tooltip',
8   mixins: [toggle]
9 }

```

通过上面小小例子，让我们知道了 `Mixin` 对于封装一些可复用的功能如此有趣、方便、实用

9.3. 源码分析

首先从 `Vue.mixin` 入手

源码位置：/src/core/global-api/mixin.js

JavaScript | 复制代码

```

1 export function initMixin (Vue: GlobalAPI) {
2   Vue.mixin = function (mixin: Object) {
3     this.options = mergeOptions(this.options, mixin)
4     return this
5   }
6 }

```

主要是调用 `merOptions` 方法

源码位置：/src/core/util/options.js

```

1  export function mergeOptions (
2    parent: Object,
3    child: Object,
4    vm?: Component
5  ): Object {
6
7    if (child.mixins) { // 判断有没有mixin 也就是 mixin里面挂 mixin的情况 有的话递归进行合并
8      for (let i = 0, l = child.mixins.length; i < l; i++) {
9        parent = mergeOptions(parent, child.mixins[i], vm)
10      }
11    }
12
13    const options = {}
14    let key
15    for (key in parent) {
16      mergeField(key) // 先遍历parent的key 调对应的strats[XXX]方法进行合并
17    }
18    for (key in child) {
19      if (!hasOwn(parent, key)) { // 如果parent已经处理过某个key 就不处理了
20        mergeField(key) // 处理child中的key 也就parent中没有处理过的key
21      }
22    }
23    function mergeField (key) {
24      const strat = strats[key] || defaultStrat
25      options[key] = strat(parent[key], child[key], vm, key) // 根据不同类型的options调用strats中不同的方法进行合并
26    }
27    return options
28  }

```

从上面的源码，我们得到以下几点：

- 优先递归处理 `mixins`
- 先遍历合并 `parent` 中的 `key`，调用 `mergeField` 方法进行合并，然后保存在变量 `options`
- 再遍历 `child`，合并补上 `parent` 中没有的 `key`，调用 `mergeField` 方法进行合并，保存在变量 `options`
- 通过 `mergeField` 函数进行了合并

下面是关于 `Vue` 的几种类型的合并策略

- 替换型

- 合并型
- 队列型
- 叠加型

9.3.1. 替换型

替换型合并有 `props`、`methods`、`inject`、`computed`

```
▼ JavaScript | 复制代码

1 strats.props =
2 strats.methods =
3 strats.inject =
4 strats.computed = function (
5   parentVal: ?Object,
6   childVal: ?Object,
7   vm?: Component,
8   key: string
9 ): ?Object {
10   if (!parentVal) return childVal // 如果parentVal没有值, 直接返回childVal
11   const ret = Object.create(null) // 创建一个第三方对象 ret
12   extend(ret, parentVal) // extend方法实际是把parentVal的属性复制到ret中
13   if (childVal) extend(ret, childVal) // 把childVal的属性复制到ret中
14   return ret
15 }
16 strats.provide = mergeDataOrFn
```

同名的 `props`、`methods`、`inject`、`computed` 会被后来者代替

9.3.2. 合并型

合并型合并有: `data`

```

1 strats.data = function(parentVal, childVal, vm) {
2     return mergeDataOrFn(
3         parentVal, childVal, vm
4     )
5 };
6
7 function mergeDataOrFn(parentVal, childVal, vm) {
8     return function mergedInstanceDataFn() {
9         var childData = childVal.call(vm, vm) // 执行data挂的函数得到对象
10        var parentData = parentVal.call(vm, vm)
11        if (childData) {
12            return mergeData(childData, parentData) // 将2个对象进行合并
13        } else {
14            return parentData // 如果没有childData 直接返回parentData
15        }
16    }
17 }
18
19 function mergeData(to, from) {
20     if (!from) return to
21     var key, toVal, fromVal;
22     var keys = Object.keys(from);
23     for (var i = 0; i < keys.length; i++) {
24         key = keys[i];
25         toVal = to[key];
26         fromVal = from[key];
27         // 如果不存在这个属性, 就重新设置
28         if (!to.hasOwnProperty(key)) {
29             set(to, key, fromVal);
30         }
31         // 存在相同属性, 合并对象
32         else if (typeof toVal == "object" && typeof fromVal == "object") {
33             mergeData(toVal, fromVal);
34         }
35     }
36     return to
37 }

```

`mergeData` 函数遍历了要合并的 data 的所有属性, 然后根据不同情况进行合并:

- 当目标 data 对象不包含当前属性时, 调用 `set` 方法进行合并 (`set`方法其实就是一些合并重新赋值的方法)
- 当目标 data 对象包含当前属性并且当前值为纯对象时, 递归合并当前对象值, 这样做是为了防止对

象存在新增属性

9.3.3. 队列性

队列性合并有：全部生命周期和 `watch`

```
1  function mergeHook (
2    parentVal: ?Array<Function>,
3    childVal: ?Function | ?Array<Function>
4  ): ?Array<Function> {
5    return childVal
6      ? parentVal
7        ? parentVal.concat(childVal)
8        : Array.isArray(childVal)
9        ? childVal
10       : [childVal]
11     : parentVal
12   }
13
14  LIFECYCLE_HOOKS.forEach(hook => {
15    strats[hook] = mergeHook
16  })
17
18 // watch
19 strats.watch = function (
20   parentVal,
21   childVal,
22   vm,
23   key
24 ) {
25   // work around Firefox's Object.prototype.watch...
26   if (parentVal === nativeWatch) { parentVal = undefined; }
27   if (childVal === nativeWatch) { childVal = undefined; }
28   /* istanbul ignore if */
29   if (!childVal) { return Object.create(parentVal || null); }
30   {
31     assertObjectType(key, childVal, vm);
32   }
33   if (!parentVal) { return childVal; }
34   var ret = {};
35   extend(ret, parentVal);
36   for (var key$1 in childVal) {
37     var parent = ret[key$1];
38     var child = childVal[key$1];
39     if (parent && !Array.isArray(parent)) {
40       parent = [parent];
41     }
42     ret[key$1] = parent
43       ? parent.concat(child)
44       : Array.isArray(child) ? child : [child];
45   }
}
```

```
46     return ret  
47 };
```

生命周期钩子和 `watch` 被合并为一个数组，然后正序遍历一次执行

9.3.4. 叠加型

叠加型有：`component`、`directives`、`filters`

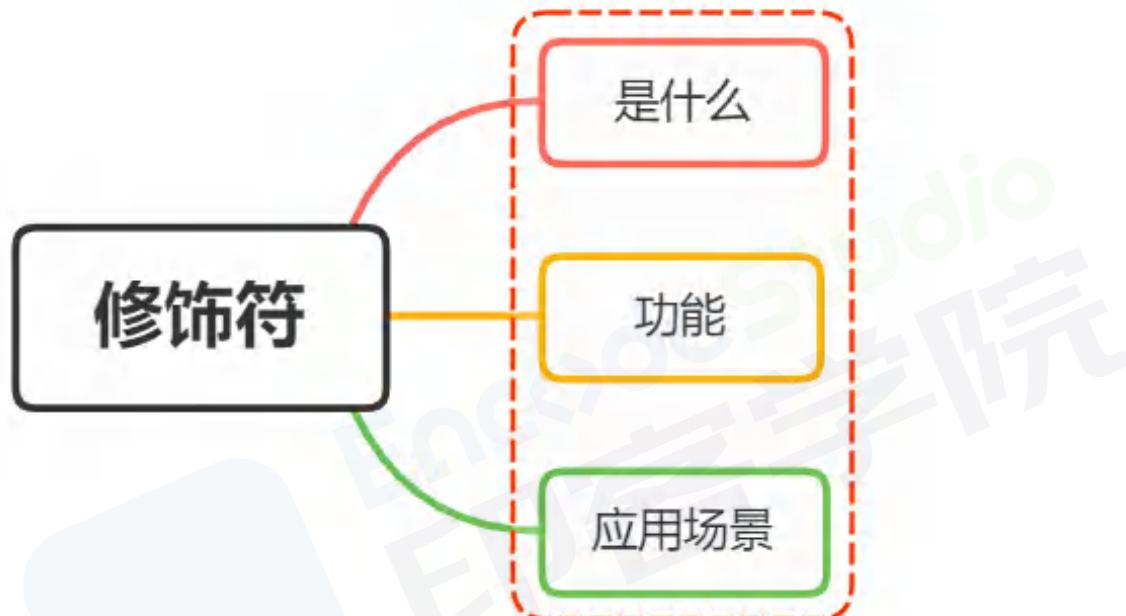
```
JavaScript | 复制代码  
  
1 strats.components =  
2 strats.directives =  
3  
4 strats.filters = function mergeAssets(  
5     parentVal, childVal, vm, key  
6   ) {  
7     var res = Object.create(parentVal || null);  
8     if (childVal) {  
9       for (var key in childVal) {  
10         res[key] = childVal[key];  
11     }  
12   }  
13   return res  
14 }
```

叠加型主要是通过原型链进行层层的叠加

9.4. 小结：

- 替换型策略有 `props`、`methods`、`inject`、`computed`，就是将新的同名参数替代旧的参数
- 合并型策略是 `data`，通过 `set` 方法进行合并和重新赋值
- 队列型策略有生命周期函数和 `watch`，原理是将函数存入一个数组，然后正序遍历依次执行
- 叠加型有 `component`、`directives`、`filters`，通过原型链进行层层的叠加

10. Vue常用的修饰符有哪些有什么应用场景



10.1. 修饰符是什么

在程序世界里，修饰符是用于限定类型以及类型成员的声明的一种符号

在 `Vue` 中，修饰符处理了许多 `DOM` 事件的细节，让我们不再需要花大量的时间去处理这些烦恼的事情，而能有更多的精力专注于程序的逻辑处理

`vue` 中修饰符分为以下五种：

- 表单修饰符
- 事件修饰符
- 鼠标按键修饰符
- 键值修饰符
- `v-bind`修饰符

10.2. 修饰符的作用

10.2.1. 表单修饰符

在我们填写表单的时候用得最多的是 `input` 标签，指令用得最多的是 `v-model`

关于表单的修饰符有如下：

- `lazy`

- trim
- number

10.2.1.1. lazy

在我们填完信息，光标离开标签的时候，才会将值赋予给 `value`，也就是在 `change` 事件之后再进行信息同步

```
1 <input type="text" v-model.lazy="value">
2 <p>{{value}}</p>
```

JavaScript | 复制代码

10.2.1.2. trim

自动过滤用户输入的首空格字符，而中间的空格不会过滤

```
1 <input type="text" v-model.trim="value">
```

JavaScript | 复制代码

10.2.1.3. number

自动将用户的输入值转为数值类型，但如果这个值无法被 `parseFloat` 解析，则会返回原来的值

```
1 <input v-model.number="age" type="number">
```

JavaScript | 复制代码

10.2.2. 事件修饰符

事件修饰符是对事件捕获以及目标进行了处理，有如下修饰符：

- stop
- prevent
- self
- once
- capture
- passive
- native

10.2.2.1. stop

阻止了事件冒泡，相当于调用了 `event.stopPropagation` 方法

```
1 <div @click="shout(2)">
2   <button @click.stop="shout(1)">ok</button>
3 </div>
4 //只输出1
```

JavaScript | 复制代码

10.2.2.2. prevent

阻止了事件的默认行为，相当于调用了 `event.preventDefault` 方法

```
1 <form v-on:submit.prevent="onSubmit"></form>
```

JavaScript | 复制代码

10.2.2.3. self

只当在 `event.target` 是当前元素自身时触发处理函数

```
1 <div v-on:click.self="doThat">...</div>
```

JavaScript | 复制代码

使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 `v-on:click.prevent.self` 会阻止所有的点击，而 `v-on:click.self.prevent` 只会阻止对元素自身的点击

10.2.2.4. once

绑定了事件以后只能触发一次，第二次就不会触发

```
1 <button @click.once="shout(1)">ok</button>
```

JavaScript | 复制代码

10.2.2.5. capture

使事件触发从包含这个元素的顶层开始往下触发

JavaScript | 复制代码

```

1 <div @click.capture="shout(1)">
2   obj1
3   <div @click.capture="shout(2)">
4     obj2
5   <div @click="shout(3)">
6     obj3
7   <div @click="shout(4)">
8     obj4
9 </div>
10 </div>
11 </div>
12 </div>
13 // 输出结构: 1 2 4 3

```

10.2.2.6. passive

在移动端，当我们在监听元素滚动事件的时候，会一直触发 `onscroll` 事件会让我们的网页变卡，因此我们使用这个修饰符的时候，相当于给 `onscroll` 事件整了一个 `.lazy` 修饰符

JavaScript | 复制代码

```

1 <!-- 滚动事件的默认行为（即滚动行为）将会立即触发 -->
2 <!-- 而不会等待 `onScroll` 完成 -->
3 <!-- 这其中包括 `event.preventDefault()` 的情况 -->
4 <div v-on:scroll.passive="onScroll">...</div>

```

不要把 `.passive` 和 `.prevent` 一起使用，因为 `.prevent` 将会被忽略，同时浏览器可能会向你展示一个警告。

`passive` 会告诉浏览器你不想阻止事件的默认行为

10.2.2.7. native

让组件变成像 `html` 内置标签那样监听根元素的原生事件，否则组件上使用 `v-on` 只会监听自定义事件

JavaScript | 复制代码

```
1 <my-component v-on:click.native="doSomething"></my-component>
```

| 使用.native修饰符来操作普通HTML标签是会令事件失效的

10.2.3. 鼠标按钮修饰符

鼠标按钮修饰符针对的就是左键、右键、中键点击，有如下：

- left 左键点击
- right 右键点击
- middle 中键点击

▼
JavaScript
 复制代码

```

1 <button @click.left="shout(1)">ok</button>
2 <button @click.right="shout(1)">ok</button>
3 <button @click.middle="shout(1)">ok</button>

```

10.2.4. 键盘修饰符

键盘修饰符是用来修饰键盘事件（`onkeyup`，`onkeydown`）的，有如下：

`keyCode` 存在很多，但 `vue` 为我们提供了别名，分为以下两种：

- 普通键（enter、tab、delete、space、esc、up...）
- 系统修饰键（ctrl、alt、meta、shift...）

▼
JavaScript
 复制代码

```

1 // 只有按键为keyCode的时候才触发
2 <input type="text" @keyup.keyCode="shout()">

```

还可以通过以下方式自定义一些全局的键盘码别名

▼
JavaScript
 复制代码

```

1 Vue.config.keyCodes.f2 = 113

```

10.2.5. v-bind修饰符

v-bind修饰符主要是为属性进行操作，用来分别有如下：

- `async`
- `prop`

- camel

10.2.5.1. async

能对 `props` 进行一个双向绑定

```
▼ JavaScript | 复制代码
1 //父组件
2 <comp :myMessage.sync="bar"></comp>
3 //子组件
4 this.$emit('update:myMessage',params);
```

以上这种方法相当于以下的简写

```
▼ JavaScript | 复制代码
1 //父亲组件
2 <comp :myMessage="bar" @update:myMessage="func"></comp>
3 func(e){
4   this.bar = e;
5 }
6 //子组件js
7 func2(){
8   this.$emit('update:myMessage',params);
9 }
```

使用 `async` 需要注意以下两点：

- 使用 `sync` 的时候，子组件传递的事件名格式必须为 `update:value`，其中 `value` 必须与子组件中 `props` 中声明的名称完全一致
- 注意带有 `.sync` 修饰符的 `v-bind` 不能和表达式一起使用
- 将 `v-bind.sync` 用在一个字面量的对象上，例如 `v-bind.sync="{ title: doc.title }"`，是无法正常工作的

10.2.5.2. props

设置自定义标签属性，避免暴露数据，防止污染HTML结构

```
▼ JavaScript | 复制代码
1 <input id="uid" title="title1" value="1" :index.prop="index">
```

10.2.5.3. camel

将命名变为驼峰命名法，如将 `view-Box` 属性名转换为 `viewBox`

```
1 <svg :viewBox="viewBox"></svg>
```

10.3. 应用场景

根据每一个修饰符的功能，我们可以得到以下修饰符的应用场景：

- `.stop`: 阻止事件冒泡
- `.native`: 绑定原生事件
- `.once`: 事件只执行一次
- `.self` : 将事件绑定在自身身上，相当于阻止事件冒泡
- `.prevent`: 阻止默认事件
- `.caption`: 用于事件捕获
- `.once`: 只触发一次
- `.keyCode`: 监听特定键盘按下
- `.right`: 右键

11. Vue中的`$nextTick`有什么作用？



11.1. NextTick是什么

官方对其的定义

在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM

什么意思呢？

我们可以理解成，Vue 在更新 DOM 时是异步执行的。当数据发生变化，Vue 将开启一个异步更新队列，视图需要等队列中所有数据变化完成之后，再统一进行更新

举例一下

HTML 结构

```
▼ HTML | ⌂ 复制代码
1 <div id="app"> {{ message }} </div>
```

构建一个 vue 实例

```
▼ JavaScript | ⌂ 复制代码
1 const vm = new Vue({
2   el: '#app',
3   data: {
4     message: '原始值'
5   }
6 })
```

修改 message

```
▼ JavaScript | ⌂ 复制代码
1 this.message = '修改后的值1'
2 this.message = '修改后的值2'
3 this.message = '修改后的值3'
```

这时候想获取页面最新的 DOM 节点，却发现获取到的是旧值

```
▼ JavaScript | ⌂ 复制代码
1 console.log(vm.$el.textContent) // 原始值
```

这是因为 message 数据在发现变化的时候，vue 并不会立刻去更新 Dom，而是将修改数据的操作放在了一个异步操作队列中

如果我们一直修改相同数据，异步操作队列还会进行去重

等待同一事件循环中的所有数据变化完成之后，会将队列中的事件拿来处理，进行 DOM 的更新

11.1.1.1. 为什么要有nexttick

举个例子

```
1  {{num}}
2  for(let i=0; i<100000; i++){
3      num = i
4 }
```

如果没有 nextTick 更新机制，那么 num 每次更新值都会触发视图更新(上面这段代码也就是会更新10万次视图)，有了 nextTick 机制，只需要更新一次，所以 nextTick 本质是一种优化策略

11.2. 使用场景

如果想要在修改数据后立刻得到更新后的 DOM 结构，可以使用 Vue.nextTick()

第一个参数为：回调函数（可以获取最近的 DOM 结构）

第二个参数为：执行函数上下文

```
1 // 修改数据
2 vm.message = '修改后的值'
3 // DOM 还没有更新
4 console.log(vm.$el.textContent) // 原始的值
5 Vue.nextTick(function () {
6     // DOM 更新了
7     console.log(vm.$el.textContent) // 修改后的值
8 })
```

组件内使用 vm.\$nextTick() 实例方法只需要通过 this.\$nextTick()，并且回调函数中的 this 将自动绑定到当前的 Vue 实例上

JavaScript | 复制代码

```
1 this.message = '修改后的值'
2 console.log(this.$el.textContent) // => '原始的值'
3 this.$nextTick(function () {
4     console.log(this.$el.textContent) // => '修改后的值'
5 })
```

`$nextTick()` 会返回一个 `Promise` 对象，可以是用 `async/await` 完成相同作用的事情

JavaScript | 复制代码

```
1 this.message = '修改后的值'
2 console.log(this.$el.textContent) // => '原始的值'
3 await this.$nextTick()
4 console.log(this.$el.textContent) // => '修改后的值'
```

11.3. 实现原理

源码位置: `/src/core/util/next-tick.js`

`callbacks` 也就是异步操作队列

`callbacks` 新增回调函数后又执行了 `timerFunc` 函数，`pending` 是用来标识同一个时间只能执行一次

```

1  export function nextTick(cb?: Function, ctx?: Object) {
2      let _resolve;
3
4      // cb 回调函数会经统一处理压入 callbacks 数组
5      callbacks.push(() => {
6          if (cb) {
7              // 给 cb 回调函数执行加上了 try-catch 错误处理
8              try {
9                  cb.call(ctx);
10             } catch (e) {
11                 handleError(e, ctx, 'nextTick');
12             }
13         } else if (_resolve) {
14             _resolve(ctx);
15         }
16     });
17
18     // 执行异步延迟函数 timerFunc
19     if (!pending) {
20         pending = true;
21         timerFunc();
22     }
23
24     // 当 nextTick 没有传入函数参数的时候，返回一个 Promise 化的调用
25     if (!cb && typeof Promise !== 'undefined') {
26         return new Promise(resolve => {
27             _resolve = resolve;
28         });
29     }
30 }

```

`timerFunc` 函数定义，这里是根据当前环境支持什么方法则确定调用哪个，分别有：

`Promise.then`、`MutationObserver`、`setImmediate`、`setTimeout`

通过上面任意一种方法，进行降级操作

```

1  export let isUsingMicroTask = false
2  if (typeof Promise !== 'undefined' && isNative(Promise)) {
3      //判断1: 是否原生支持Promise
4      const p = Promise.resolve()
5      timerFunc = () => {
6          p.then(flushCallbacks)
7          if (isIOS) setTimeout(noop)
8      }
9      isUsingMicroTask = true
10 } else if (!isIE && typeof MutationObserver !== 'undefined' && (
11     isNative(MutationObserver) ||
12     MutationObserver.toString() === '[object MutationObserverConstructor]')
13 )) {
14     //判断2: 是否原生支持MutationObserver
15     let counter = 1
16     const observer = new MutationObserver(flushCallbacks)
17     const textNode = document.createTextNode(String(counter))
18     observer.observe(textNode, {
19         characterData: true
20     })
21     timerFunc = () => {
22         counter = (counter + 1) % 2
23         textNode.data = String(counter)
24     }
25     isUsingMicroTask = true
26 } else if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
27     //判断3: 是否原生支持setImmediate
28     timerFunc = () => {
29         setImmediate(flushCallbacks)
30     }
31 } else {
32     //判断4: 上面都不行, 直接用setTimeout
33     timerFunc = () => {
34         setTimeout(flushCallbacks, 0)
35     }
36 }

```

无论是微任务还是宏任务，都会放到 `flushCallbacks` 使用

这里将 `callbacks` 里面的函数复制一份，同时 `callbacks` 置空

依次执行 `callbacks` 里面的函数

```

1 function flushCallbacks () {
2     pending = false
3     const copies = callbacks.slice(0)
4     callbacks.length = 0
5     for (let i = 0; i < copies.length; i++) {
6         copies[i]()
7     }
8 }

```

小结：

1. 把回调函数放入callbacks等待执行
2. 将执行函数放到微任务或者宏任务中
3. 事件循环到了微任务或者宏任务，执行函数依次执行callbacks中的回调

12. Vue实例挂载的过程



12.1. 思考

我们都听过知其然知其所以然这句话

那么不知道大家是否思考过 `new Vue()` 这个过程中究竟做了些什么？

过程中是如何完成数据的绑定，又是如何将数据渲染到视图的等等

12.2. 分析

首先找到 `vue` 的构造函数

源码位置: `src\core\instance\index.js`

```
1 function Vue (options) {
2   if (process.env.NODE_ENV !== 'production' &&
3     !(this instanceof Vue)
4   ) {
5     warn('Vue is a constructor and should be called with the `new` keyword'
6   )
7   }
8   this._init(options)
}
```

`options` 是用户传递过来的配置项，如 `data、methods` 等常用的方法

`vue` 构建函数调用 `_init` 方法，但我们发现本文件中并没有此方法，但仔细可以看到文件下方定义了很多初始化方法

```
1 initMixin(Vue);      // 定义 _init
2 stateMixin(Vue);     // 定义 $set $get $delete $watch 等
3 eventsMixin(Vue);   // 定义事件 $on $once $off $emit
4 lifecycleMixin(Vue); // 定义 _update $forceUpdate $destroy
5 renderMixin(Vue);   // 定义 _render 返回虚拟dom
```

首先可以看 `initMixin` 方法，发现该方法在 `Vue` 原型上定义了 `_init` 方法

源码位置: `src\core\instance\init.js`

JavaScript | 复制代码

```
1 - Vue.prototype._init = function (options?: Object) {
2     const vm: Component = this
3     // a uid
4     vm._uid = uid++
5     let startTag, endTag
6     /* istanbul ignore if */
7     if (process.env.NODE_ENV !== 'production' && config.performance && mar
k) {
8         startTag = `vue-perf-start:${vm._uid}`
9         endTag = `vue-perf-end:${vm._uid}`
10        mark(startTag)
11    }
12
13    // a flag to avoid this being observed
14    vm._isVue = true
15    // merge options
16    // 合并属性，判断初始化的是否是组件，这里合并主要是 mixins 或 extends 的方法
17    if (options && options._isComponent) {
18        // optimize internal component instantiation
19        // since dynamic options merging is pretty slow, and none of the
20        // internal component options needs special treatment.
21        initInternalComponent(vm, options)
22    } else { // 合并vue属性
23        vm.$options = mergeOptions(
24            resolveConstructorOptions(vm.constructor),
25            options || {},
26            vm
27        )
28    }
29    /* istanbul ignore else */
30    if (process.env.NODE_ENV !== 'production') {
31        // 初始化proxy拦截器
32        initProxy(vm)
33    } else {
34        vm._renderProxy = vm
35    }
36    // expose real self
37    vm._self = vm
38    // 初始化组件生命周期标志位
39    initLifecycle(vm)
40    // 初始化组件事件侦听
41    initEvents(vm)
42    // 初始化渲染方法
43    initRender(vm)
44    callHook(vm, 'beforeCreate')
```

```

45      // 初始化依赖注入内容，在初始化data、props之前
46      initInjections(vm) // resolve injections before data/props
47      // 初始化props/data/method/watch/methods
48      initState(vm)
49      initProvide(vm) // resolve provide after data/props
50      callHook(vm, 'created')
51
52      /* istanbul ignore if */
53      if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
54          vm._name = formatComponentName(vm, false)
55          mark(endTag)
56          measure(`vue ${vm._name} init`, startTag, endTag)
57      }
58      // 挂载元素
59      if (vm.$options.el) {
60          vm.$mount(vm.$options.el)
61      }
62  }

```

仔细阅读上面的代码，我们得到以下结论：

- 在调用 `beforeCreate` 之前，数据初始化并未完成，像 `data`、`props` 这些属性无法访问到
- 到了 `created` 的时候，数据已经初始化完成，能够访问 `data`、`props` 这些属性，但这时候并未完成 `dom` 的挂载，因此无法访问到 `dom` 元素
- 挂载方法是调用 `vm.$mount` 方法

`initState` 方法是完成 `props/data/method/watch/methods` 的初始化

源码位置：src\core\instance\state.js

```
1 export function initState (vm: Component) {
2     // 初始化组件的watcher列表
3     vm._watchers = []
4     const opts = vm.$options
5     // 初始化props
6     if (opts.props) initProps(vm, opts.props)
7     // 初始化methods方法
8     if (opts.methods) initMethods(vm, opts.methods)
9     if (opts.data) {
10         // 初始化data
11         initData(vm)
12     } else {
13         observe(vm._data = {}, true /* asRootData */)
14     }
15     if (opts.computed) initComputed(vm, opts.computed)
16     if (opts.watch && opts.watch !== nativeWatch) {
17         initWatch(vm, opts.watch)
18     }
19 }
```

我们主要看初始化 `data` 的方法为 `initData`，它与 `initState` 在同一文件上

```
1  function initData (vm: Component) {
2      let data = vm.$options.data
3      // 获取到组件上的data
4      data = vm._data = typeof data === 'function'
5          ? getData(data, vm)
6          : data || {}
7      if (!isPlainObject(data)) {
8          data = {}
9          process.env.NODE_ENV !== 'production' && warn(
10              'data functions should return an object:\n' +
11              'https://vuejs.org/v2/guide/components.html#data-Must-Be-a-Function'
12          ,
13          vm
14      )
15      // proxy data on instance
16      const keys = Object.keys(data)
17      const props = vm.$options.props
18      const methods = vm.$options.methods
19      let i = keys.length
20      while (i--) {
21          const key = keys[i]
22          if (process.env.NODE_ENV !== 'production') {
23              // 属性名不能与方法名重复
24              if (methods && hasOwn(methods, key)) {
25                  warn(
26                      `Method "${key}" has already been defined as a data property.`,
27                      vm
28                  )
29              }
30          }
31          // 属性名不能与state名称重复
32          if (props && hasOwn(props, key)) {
33              process.env.NODE_ENV !== 'production' && warn(
34                  `The data property "${key}" is already declared as a prop. ` +
35                  `Use prop default value instead.`,
36                  vm
37              )
38          } else if (!isReserved(key)) { // 验证key值的合法性
39              // 将_data中的数据挂载到组件vm上,这样就可以通过this.xxx访问到组件上的数据
40              proxy(vm, `_data`, key)
41          }
42      }
43      // observe data
44      // 响应式监听data是数据的变化
```

```
45     observe(data, true /* asRootData */)
46 }
```

仔细阅读上面的代码，我们可以得到以下结论

- 初始化顺序：`props`、`methods`、`data`
- `data` 定义的时候可选择函数形式或者对象形式（组件只能为函数形式）

关于数据响应式在这就不展开详细说明

上文提到挂载方法是调用 `vm.$mount` 方法

源码位置：

```
1  Vue.prototype.$mount = function (
2    el?: string | Element,
3    hydrating?: boolean
4  ): Component {
5    // 获取或查询元素
6    el = el && query(el)
7
8    /* istanbul ignore if */
9    // vue 不允许直接挂载到body或页面文档上
10   if (el === document.body || el === document.documentElement) {
11     process.env.NODE_ENV !== 'production' && warn(
12       `Do not mount Vue to <html> or <body> - mount to normal elements instead.`
13     )
14     return this
15   }
16
17   const options = this.$options
18   // resolve template/el and convert to render function
19   if (!options.render) {
20     let template = options.template
21     // 存在template模板, 解析vue模板文件
22     if (template) {
23       if (typeof template === 'string') {
24         if (template.charAt(0) === '#') {
25           template = idToTemplate(template)
26           /* istanbul ignore if */
27           if (process.env.NODE_ENV !== 'production' && !template) {
28             warn(
29               `Template element not found or is empty: ${options.template}`
30             ,
31             this
32           )
33         }
34       } else if (template.nodeType) {
35         template = template.innerHTML
36       } else {
37         if (process.env.NODE_ENV !== 'production') {
38           warn('invalid template option:' + template, this)
39         }
40         return this
41       }
42     } else if (el) {
43       // 通过选择器获取元素内容
```

```

44         template = getOuterHTML(el)
45     }
46     if (template) {
47       /* istanbul ignore if */
48       if (process.env.NODE_ENV !== 'production' && config.performance && m
ark) {
49         mark('compile')
50     }
51   /**
52    * 1.将temmplate解析ast tree
53    * 2.将ast tree转换成render语法字符串
54    * 3.生成render方法
55    */
56   const { render, staticRenderFns } = compileToFunctions(template, {
57     outputSourceRange: process.env.NODE_ENV !== 'production',
58     shouldDecodeNewlines,
59     shouldDecodeNewlinesForHref,
60     delimiters: options.delimiters,
61     comments: options.comments
62   }, this)
63   options.render = render
64   options.staticRenderFns = staticRenderFns
65
66   /* istanbul ignore if */
67   if (process.env.NODE_ENV !== 'production' && config.performance && m
ark) {
68     mark('compile end')
69     measure(`vue ${this._name} compile`, 'compile', 'compile end')
70   }
71 }
72 }
73 return mount.call(this, el, hydrating)
74 }

```

阅读上面代码，我们能得到以下结论：

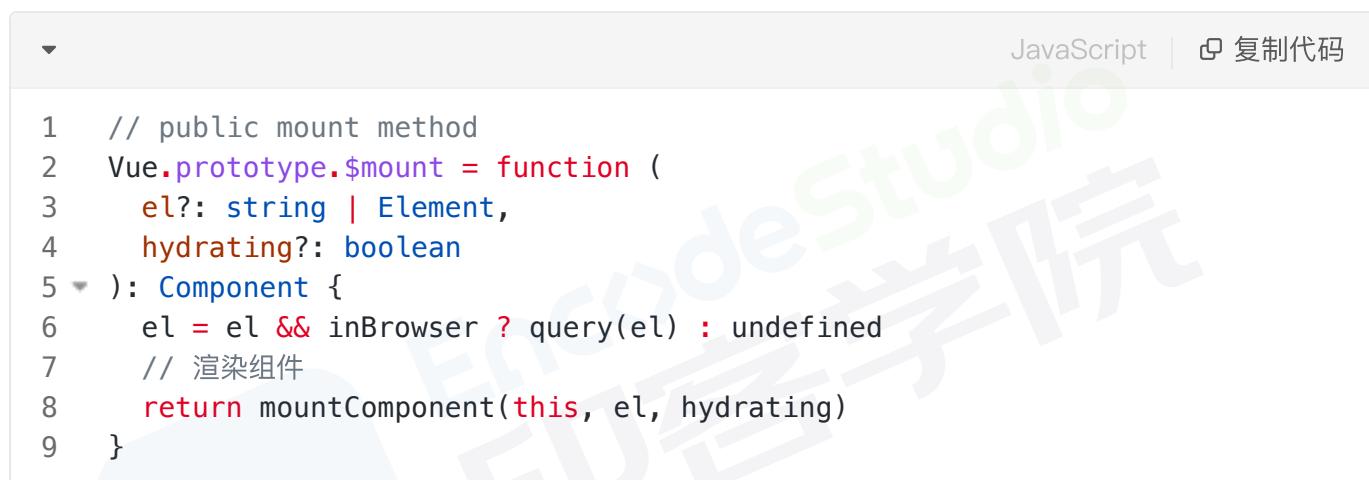
- 不要将根元素放到 `body` 或者 `html` 上
- 可以在对象中定义 `template/render` 或者直接使用 `template`、`el` 表示元素选择器
- 最终都会解析成 `render` 函数，调用 `compileToFunctions`，会将 `template` 解析成 `rend
er` 函数

对 `template` 的解析步骤大致分为以下几步：

- 将 `html` 文档片段解析成 `ast` 描述符
- 将 `ast` 描述符解析成字符串
- 生成 `render` 函数

生成 `render` 函数，挂载到 `vm` 上后，会再次调用 `mount` 方法

源码位置：src\platforms\web\runtime\index.js



The screenshot shows a code editor window with the following code:

```
1 // public mount method
2 Vue.prototype.$mount = function (
3   el?: string | Element,
4   hydrating?: boolean
5 ): Component {
6   el = el && inBrowser ? query(el) : undefined
7   // 渲染组件
8   return mountComponent(this, el, hydrating)
9 }
```

The code is written in JavaScript, showing the implementation of the `$mount` method on the `Vue` prototype. It takes an optional `el` parameter (either a string or an `Element`) and a `hydrating` boolean. If `el` is provided and the environment is a browser, it uses the `query` function to get the actual element. Then, it calls the `mountComponent` method with `this`, `el`, and `hydrating` as arguments.

调用 `mountComponent` 渲染组件

```
1  export function mountComponent (
2    vm: Component,
3    el: ?Element,
4    hydrating?: boolean
5  ): Component {
6    vm.$el = el
7    // 如果没有获取解析的render函数，则会抛出警告
8    // render是解析模板文件生成的
9    if (!vm.$options.render) {
10      vm.$options.render = createEmptyVNode
11      if (process.env.NODE_ENV !== 'production') {
12        /* istanbul ignore if */
13        if ((vm.$options.template && vm.$options.template.charAt(0) !== '#')
14          ||
15          vm.$options.el || el) {
16          warn(
17            'You are using the runtime-only build of Vue where the template
18            '
19            + 'compiler is not available. Either pre-compile the templates int
20            o '
21            + 'render functions, or use the compiler-included build.',
22            vm
23          )
24        } else {
25          // 没有获取到vue的模板文件
26          warn(
27            'Failed to mount component: template or render function not defi
28            ned.',
29            vm
30          )
31        }
32      }
33      // 执行beforeMount钩子
34      callHook(vm, 'beforeMount')
35      if (process.env.NODE_ENV !== 'production' && config.performance && mark)
36      {
37        updateComponent = () => {
38          const name = vm._name
39          const id = vm._uid
40          const startTag = `vue-perf-start:${id}`
41          const endTag = `vue-perf-end:${id}`
```

```

41      mark(startTag)
42      const vnode = vm._render()
43      mark(endTag)
44      measure(`vue ${name} render`, startTag, endTag)
45
46      mark(startTag)
47      vm._update(vnode, hydrating)
48      mark(endTag)
49      measure(`vue ${name} patch`, startTag, endTag)
50
51    }
52  } else {
53   // 定义更新函数
54   updateComponent = () => {
55     // 实际调用是在lifeCycleMixin中定义的_update和renderMixin中定义的_render
56     vm._update(vm._render(), hydrating)
57   }
58 }
59 // we set this to vm._watcher inside the watcher's constructor
60 // since the watcher's initial patch may call $forceUpdate (e.g. inside
61 child
62   // component's mounted hook), which relies on vm._watcher being already
63   // defined
64   // 监听当前组件状态, 当有数据变化时, 更新组件
65   new Watcher(vm, updateComponent, noop, {
66     before () {
67       if (vm._isMounted && !vm._isDestroyed) {
68         // 数据更新引发的组件更新
69         callHook(vm, 'beforeUpdate')
70       }
71     },
72   }, true /* isRenderWatcher */)
73   hydrating = false
74
75   // manually mounted instance, call mounted on self
76   // mounted is called for render-created child components in its inserted
77   // hook
78   if (vm.$vnode == null) {
79     vm._isMounted = true
80     callHook(vm, 'mounted')
81   }
82   return vm
83 }

```

阅读上面代码，我们得到以下结论：

- 会触发 `beforeCreate` 钩子

- 定义 `updateComponent` 渲染页面视图的方法
- 监听组件数据，一旦发生变化，触发 `beforeUpdate` 生命钩子

`updateComponent` 方法主要执行在 `vue` 初始化时声明的 `render`，`update` 方法

`render` 的作用主要是生成 `vnode`

源码位置：src\core\instance\render.js

```
1 // 定义vue 原型上的render方法
2 - Vue.prototype._render = function (): VNode {
3     const vm: Component = this
4     // render函数来自于组件的option
5     const { render, _parentVnode } = vm.$options
6
7     if (_parentVnode) {
8         vm.$scopedSlots = normalizeScopedSlots(
9             _parentVnode.data.scopedSlots,
10            vm.$slots,
11            vm.$scopedSlots
12        )
13    }
14
15    // set parent vnode. this allows render functions to have access
16    // to the data on the placeholder node.
17    vm.$vnode = _parentVnode
18    // render self
19    let vnode
20    try {
21        // There's no need to maintain a stack because all render fns are
22        // called
23        // separately from one another. Nested component's render fns are
24        // called
25        // when parent component is patched.
26        currentRenderingInstance = vm
27        // 调用render方法, 自己的独特的render方法, 传入createElement参数, 生成vN
28        // ode
29        vnode = render.call(vm._renderProxy, vm.$createElement)
30    } catch (e) {
31        handleError(e, vm, `render`)
32        // return error render result,
33        // or previous vnode to prevent render error causing blank compone
34        nt
35        /* istanbul ignore else */
36        if (process.env.NODE_ENV !== 'production' && vm.$options.renderErr
37        or) {
38            try {
39                vnode = vm.$options.renderError.call(vm._renderProxy, vm.
$createElement, e)
40            } catch (e) {
41                handleError(e, vm, `renderError`)
42                vnode = vm._vnode
43            }
44        } else {
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
888
889
889
890
891
892
893
894
895
896
897
898
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
948
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1089
1090
1091
1092
1093
1094
1095
1096
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1188
1189
1189
1190
1191
1192
1193
1194
1195
1196
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1288
1289
1289
1290
1291
1292
1293
1294
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1388
1389
1389
1390
1391
1392
1393
1394
1395
1396
1397
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1478
1479
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1488
1489
1489
1490
1491
1492
1493
1494
1495
1496
1497
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1577
1578
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1587
1587
1588
1588
1589
1589
1590
1591
1592
1593
1594
1595
1596
1597
1597
1598
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1677
1678
1678
1679
1679
1680
1681
1682
1683
1684
1685
1686
1687
1687
1688
1688
1689
1689
1690
1691
1692
1693
1694
1695
1696
1697
1697
1698
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1777
1778
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1787
1788
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1797
1798
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1877
1878
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1897
1898
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1977
1978
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2077
2078
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2097
2098
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2149
2150
2151
2152
2153
21
```

```

40             vnode = vm._vnode
41         }
42     } finally {
43         currentRenderingInstance = null
44     }
45     // if the returned array contains only a single node, allow it
46     if (Array.isArray(vnode) && vnode.length === 1) {
47         vnode = vnode[0]
48     }
49     // return empty vnode in case the render function errored out
50     if (!(vnode instanceof VNode)) {
51         if (process.env.NODE_ENV !== 'production' && Array.isArray(vnode))
52         {
53             warn(
54                 'Multiple root nodes returned from render function. Rende
r function ' +
55                 'should return a single root node.',
56                 vm
57             )
58             vnode = createEmptyVNode()
59         }
60         // set parent
61         vnode.parent = _parentVnode
62     }
63 }

```

`_update` 主要功能是调用 `patch`，将 `vnode` 转换为真实 `DOM`，并且更新到页面中

源码位置：src\core\instance\lifecycle.js

```

1  Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
2      const vm: Component = this
3      const prevEl = vm.$el
4      const prevVnode = vm._vnode
5      // 设置当前激活的作用域
6      const restoreActiveInstance = setActiveInstance(vm)
7      vm._vnode = vnode
8      // Vue.prototype._patch_ is injected in entry points
9      // based on the rendering backend used.
10     if (!prevVnode) {
11         // initial render
12         // 执行具体的挂载逻辑
13         vm.$el = vm._patch_(vm.$el, vnode, hydrating, false /* removeOnly
   */)
14     } else {
15         // updates
16         vm.$el = vm._patch_(prevVnode, vnode)
17     }
18     restoreActiveInstance()
19     // update __vue__ reference
20     if (prevEl) {
21         prevEl.__vue__ = null
22     }
23     if (vm.$el) {
24         vm.$el.__vue__ = vm
25     }
26     // if parent is an HOC, update its $el as well
27     if (vm.$vnode && vm.$parent && vm.$vnode === vm.$parent._vnode) {
28         vm.$parent.$el = vm.$el
29     }
30     // updated hook is called by the scheduler to ensure that children are
31     // updated in a parent's updated hook.
32 }

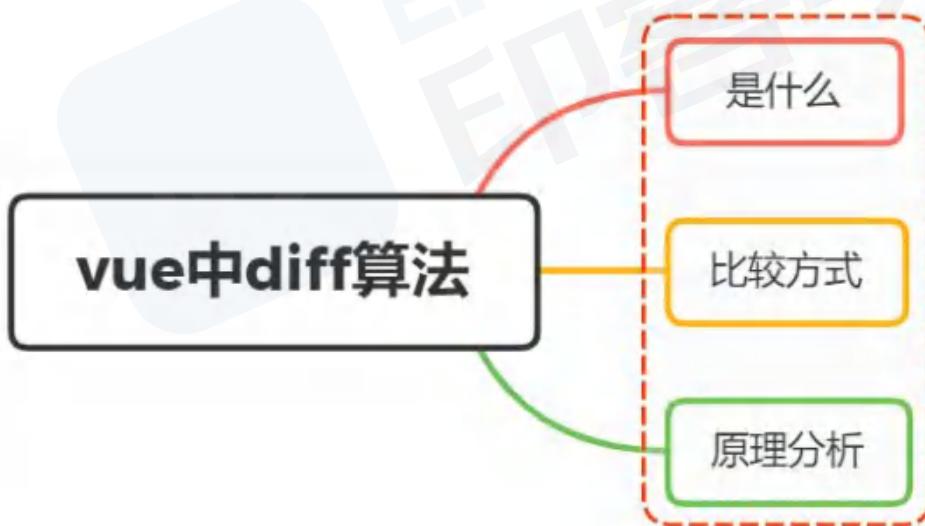
```

12.3. 结论

- `new Vue` 的时候调用会调用 `_init` 方法
 - 定义 `$set`、`$get`、`$delete`、`$watch` 等方法
 - 定义 `$on`、`$off`、`$emit`、`$off` 等事件
 - 定义 `_update`、`$forceUpdate`、`$destroy` 生命周期
- 调用 `$mount` 进行页面的挂载

- 挂载的时候主要是通过 `mountComponent` 方法
- 定义 `updateComponent` 更新函数
- 执行 `render` 生成虚拟 DOM
- `_update` 将虚拟 DOM 生成真实 DOM 结构，并且渲染到页面中

13. 你了解vue的diff算法吗？



13.1. 是什么

`diff` 算法是一种通过同层的树节点进行比较的高效算法

其有两个特点：

- 比较只会在同层级进行，不会跨层级比较
- 在diff比较的过程中，循环从两边向中间比较

`diff` 算法在很多场景下都有应用，在 `vue` 中，作用于虚拟 `dom` 渲染成真实 `dom` 的新旧 `vnode` 节点比较

13.2. 比较方式

`diff` 整体策略为：深度优先，同层比较

1. 比较只会在同层级进行，不会跨层级比较

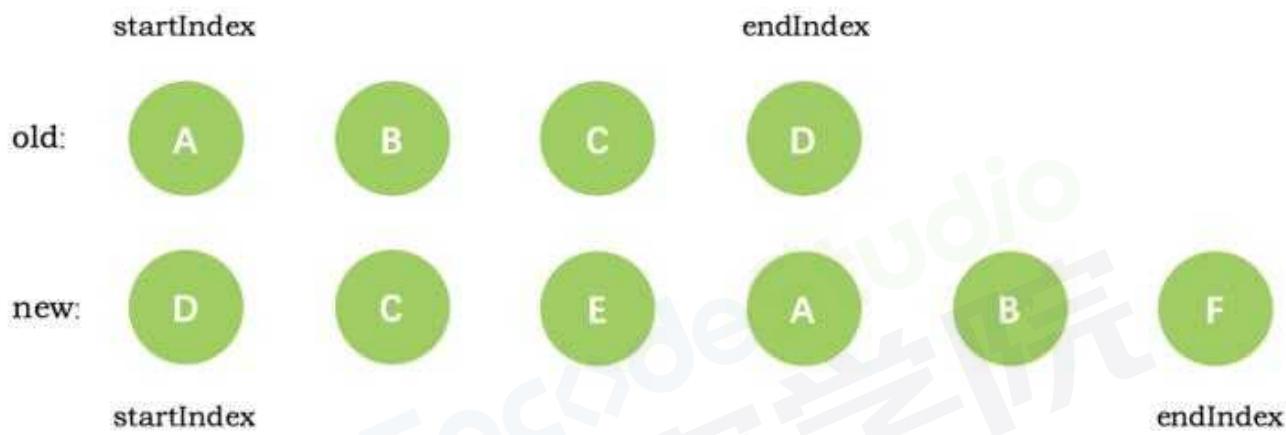


2. 比较的过程中，循环从两边向中间收拢



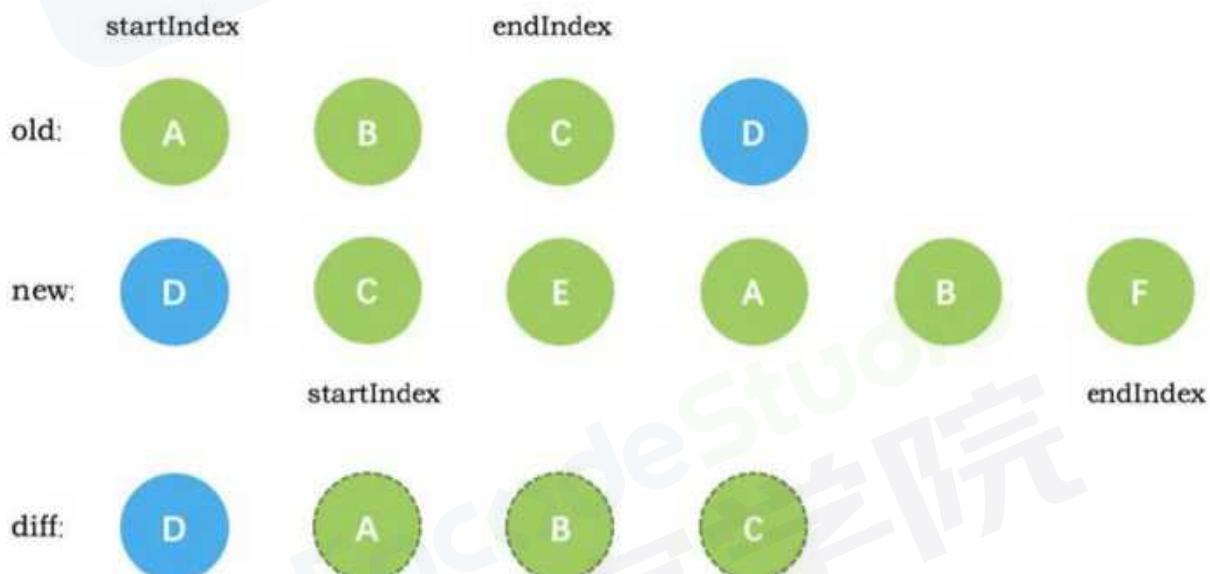
下面举个 `vue` 通过 `diff` 算法更新的例子：

新旧 `VNode` 节点如下图所示：



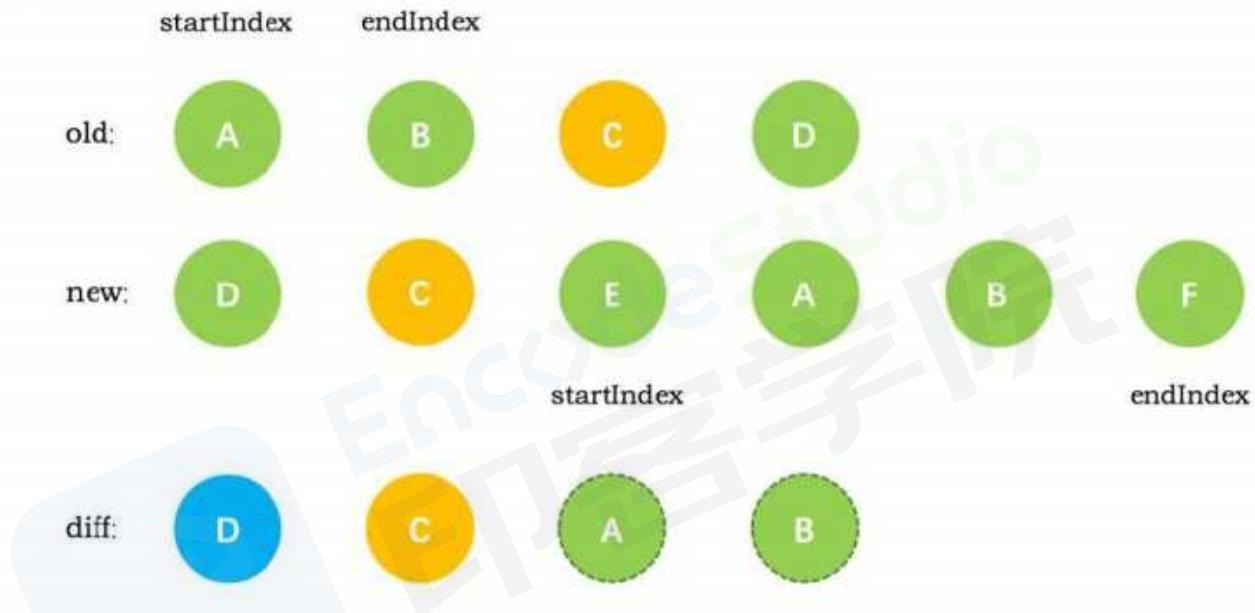
第一次循环后，发现旧节点D与新节点D相同，直接复用旧节点D作为 `diff` 后的第一个真实节点，同时旧节点 `endIndex` 移动到C，新节点的 `startIndex` 移动到了 C

第一次：



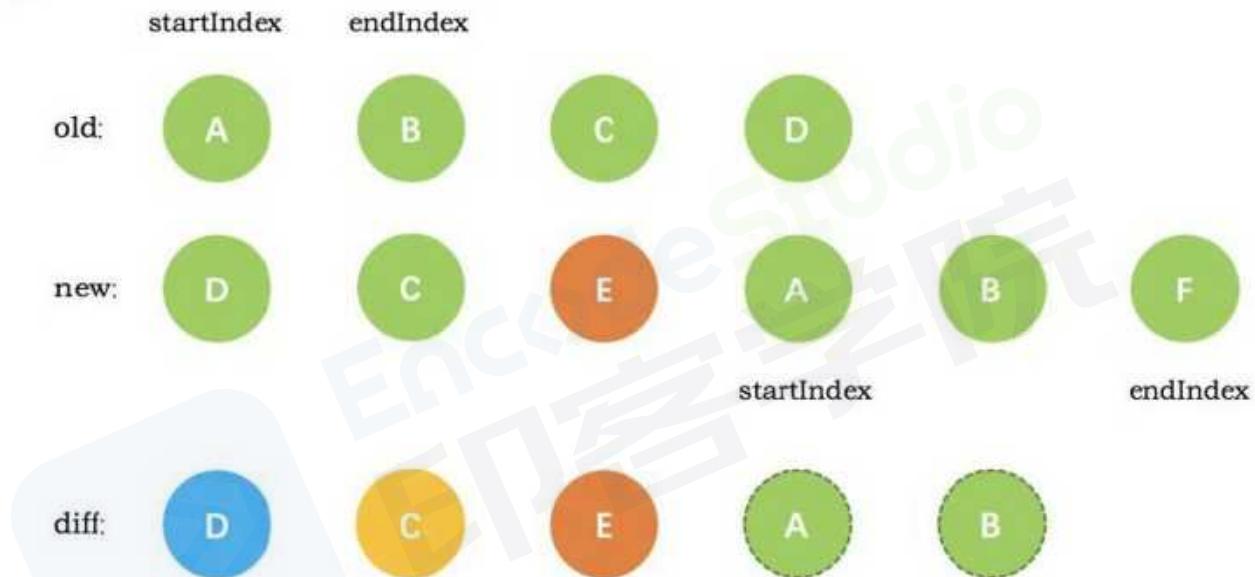
第二次循环后，同样是旧节点的末尾和新节点的开头(都是 C)相同，同理，`diff` 后创建了 C 的真实节点插入到第一次创建的 D 节点后面。同时旧节点的 `endIndex` 移动到了 B，新节点的 `startIndex` 移动到了 E

第二次：



第三次循环中，发现E没有找到，这时候只能直接创建新的真实节点 E，插入到第二次创建的 C 节点之后。同时新节点的 `startIndex` 移动到了 A。旧节点的 `startIndex` 和 `endIndex` 都保持不动

第三次：



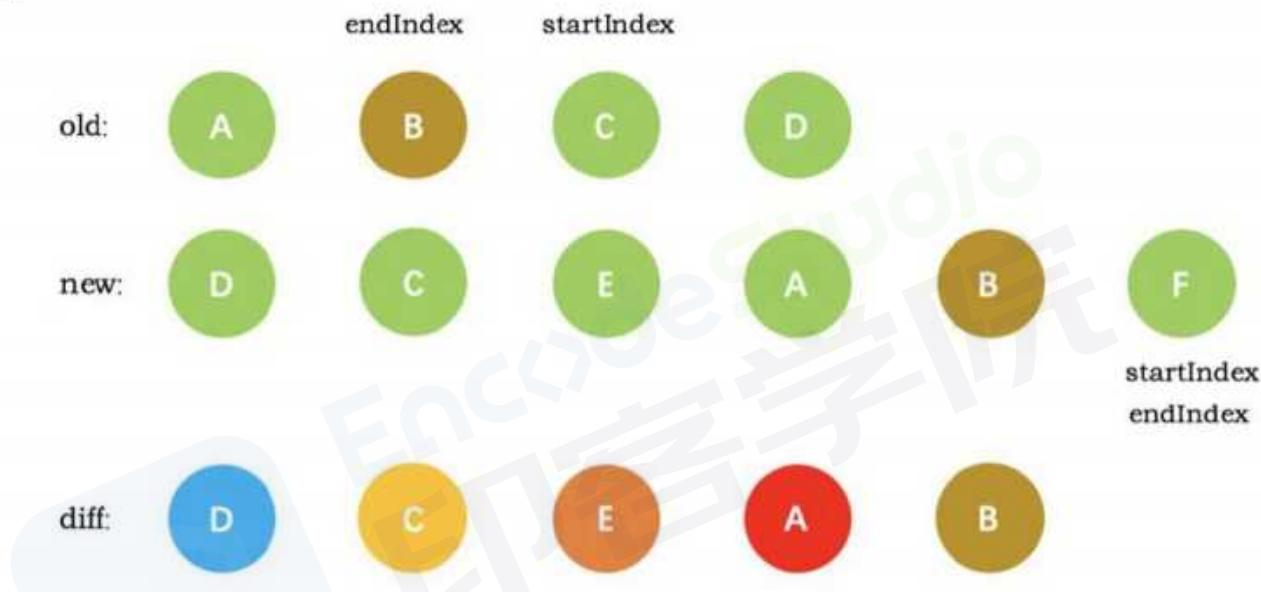
第四次循环中，发现了新旧节点的开头(都是 A)相同，于是 `diff` 后创建了 A 的真实节点，插入到前一次创建的 E 节点后面。同时旧节点的 `startIndex` 移动到了 B，新节点的 `startIndex` 移动到了 B

第四次：



第五次循环中，情形同第四次循环一样，因此 `diff` 后创建了 B 真实节点 插入到前一次创建的 A 节点后面。同时旧节点的 `startIndex` 移动到了 C，新节点的 `startIndex` 移动到了 F

第五次：



新节点的 `startIndex` 已经大于 `endIndex` 了，需要创建 `newStartIdx` 和 `newEndIdx` 之间的所有节点，也就是节点F，直接创建 F 节点对应的真实节点放到 B 节点后面

第六次：

因 `oldStartIndex > oldEndIndex`, 退出循环, 补充new增加的节点F



13.3. 原理分析

当数据发生改变时, `set` 方法会调用 `Dep.notify` 通知所有订阅者 `Watcher`, 订阅者就会调用 `patch` 给真实的 `DOM` 打补丁, 更新相应的视图

源码位置: `src/core/vdom/patch.js`

JavaScript | 复制代码

```

1  function patch(oldVnode, vnode, hydrating, removeOnly) {
2    if (isUndef(vnode)) { // 没有新节点, 直接执行destory钩子函数
3      if (isDef(oldVnode)) invokeDestroyHook(oldVnode)
4      return
5    }
6
7    let isInitialPatch = false
8    const insertedVnodeQueue = []
9
10   if (isUndef(oldVnode)) {
11     isInitialPatch = true
12     createElm(vnode, insertedVnodeQueue) // 没有旧节点, 直接用新节点生成dom元素
13   } else {
14     const isRealElement = isDef(oldVnode.nodeType)
15     if (!isRealElement && sameVnode(oldVnode, vnode)) {
16       // 判断旧节点和新节点自身一样, 一致执行patchVnode
17       patchVnode(oldVnode, vnode, insertedVnodeQueue, null, null, removeOnly)
18     } else {
19       // 否则直接销毁及旧节点, 根据新节点生成dom元素
20       if (isRealElement) {
21
22         if (oldVnode.nodeType === 1 && oldVnode.hasAttribute(SSR_ATTR_TTR)) {
23           oldVnode.removeAttribute(SSR_ATTR)
24           hydrating = true
25         }
26         if (isTrue(hydrating)) {
27           if (hydrate(oldVnode, vnode, insertedVnodeQueue)) {
28             invokeInsertHook(vnode, insertedVnodeQueue, true)
29             return oldVnode
30           }
31         }
32         oldVnode = emptyNodeAt(oldVnode)
33       }
34     }
35   }
36 }
37 }

```

`patch` 函数前两个参数位为 `oldVnode` 和 `Vnode`，分别代表新的节点和之前的旧节点，主要做了四个判断：

- 没有新节点，直接触发旧节点的 `destory` 钩子

- 没有旧节点，说明是页面刚开始初始化的时候，此时，根本不需要比较了，直接全是新建，所以只调用 `createElm`
- 旧节点和新节点自身一样，通过 `sameVnode` 判断节点是否一样，一样时，直接调用 `patchVnode` 去处理这两个节点
- 旧节点和新节点自身不一样，当两个节点不一样的时候，直接创建新节点，删除旧节点

下面主要讲的是 `patchVnode` 部分

JavaScript

复制代码

```
1  function patchVnode (oldVnode, vnode, insertedVnodeQueue, removeOnly) {
2      // 如果新旧节点一致，什么都不做
3      if (oldVnode === vnode) {
4          return
5      }
6
7      // 让vnode.el引用到现在的真实dom，当el修改时，vnode.el会同步变化
8      const elm = vnode.elm = oldVnode.elm
9
10     // 异步占位符
11     if (isTrue(oldVnode.isAsyncPlaceholder)) {
12         if (isDef(vnode.asyncFactory.resolved)) {
13             hydrate(oldVnode.elm, vnode, insertedVnodeQueue)
14         } else {
15             vnode.isAsyncPlaceholder = true
16         }
17         return
18     }
19     // 如果新旧都是静态节点，并且具有相同的key
20     // 当vnode是克隆节点或是v-once指令控制的节点时，只需要把oldVnode.elm和oldVnode.child都复制到vnode上
21     // 也不用再有其他操作
22     if (isTrue(vnode.isStatic) &&
23         isTrue(oldVnode.isStatic) &&
24         vnode.key === oldVnode.key &&
25         (isTrue(vnode.isCloned) || isTrue(vnode.isOnce)))
26     ) {
27         vnode.componentInstance = oldVnode.componentInstance
28         return
29     }
30
31     let i
32     const data = vnode.data
33     if (isDef(data) && isDef(i = data.hook) && isDef(i = i.prepatch)) {
34         i(oldVnode, vnode)
35     }
36
37     const oldCh = oldVnode.children
38     const ch = vnode.children
39     if (isDef(data) && isPatchable(vnode)) {
40         for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldVnode, vnode)
41         if (isDef(i = data.hook) && isDef(i = i.update)) i(oldVnode, vnode)
42     }
43     // 如果vnode不是文本节点或者注释节点
```

```

44      if (isUndef(vnode.text)) {
45          // 并且都有子节点
46          if (isDef(oldCh) && isDef(ch)) {
47              // 并且子节点不完全一致，则调用updateChildren
48              if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue,
49                  removeOnly)
50
51          // 如果只有新的vnode有子节点
52      } else if (isDef(ch)) {
53          if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
54          // elm已经引用了老的dom节点，在老的dom节点上添加子节点
55          addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
56
57      // 如果新vnode没有子节点，而vnode有子节点，直接删除老的oldCh
58  } else if (isDef(oldCh)) {
59      removeVnodes(elm, oldCh, 0, oldCh.length - 1)
60
61      // 如果老节点是文本节点
62  } else if (isDef(oldVnode.text)) {
63      nodeOps.setTextContent(elm, '')
64  }
65
66      // 如果新vnode和老vnode是文本节点或注释节点
67      // 但是vnode.text != oldVnode.text时，只需要更新vnode.elm的文本内容就可以
68  } else if (oldVnode.text !== vnode.text) {
69      nodeOps.setTextContent(elm, vnode.text)
70  }
71  if (isDef(data)) {
72      if (isDef(i = data.hook) && isDef(i = i.postpatch)) i(oldVnode, vnode)
73  }
74 }
```

`patchVnode` 主要做了几个判断：

- 新节点是否是文本节点，如果是，则直接更新 `dom` 的文本内容为新节点的文本内容
- 新节点和旧节点如果都有子节点，则处理比较更新子节点
- 只有新节点有子节点，旧节点没有，那么不用比较了，所有节点都是全新的，所以直接全部新建就好了，新建是指创建出所有新 `DOM`，并且添加进父节点
- 只有旧节点有子节点而新节点没有，说明更新后的页面，旧节点全部都不见了，那么要做的，就是把所有的旧节点删除，也就是直接把 `DOM` 删除

子节点不完全一致，则调用 `updateChildren`

```

1  function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue, removeOnly) {
2      let oldStartIdx = 0 // 旧头索引
3      let newStartIdx = 0 // 新头索引
4      let oldEndIdx = oldCh.length - 1 // 旧尾索引
5      let newEndIdx = newCh.length - 1 // 新尾索引
6      let oldStartVnode = oldCh[0] // oldVnode的第一个child
7      let oldEndVnode = oldCh[oldEndIdx] // oldVnode的最后一个child
8      let newStartVnode = newCh[0] // newVnode的第一个child
9      let newEndVnode = newCh[newEndIdx] // newVnode的最后一个child
10     let oldKeyToIdx, idxInOld, vnodeToMove, refElm
11
12     // removeOnly is a special flag used only by <transition-group>
13     // to ensure removed elements stay in correct relative positions
14     // during leaving transitions
15     const canMove = !removeOnly
16
17     // 如果oldStartVnode和oldEndVnode重合，并且新的也都重合了，证明diff完了，循环结束
18     while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
19         // 如果oldVnode的第一个child不存在
20         if (isUndef(oldStartVnode)) {
21             // oldStart索引右移
22             oldStartVnode = oldCh[++oldStartIdx] // Vnode has been moved left
23
24             // 如果oldVnode的最后一个child不存在
25         } else if (isUndef(oldEndVnode)) {
26             // oldEnd索引左移
27             oldEndVnode = oldCh[--oldEndIdx]
28
29             // oldStartVnode和newStartVnode是同一个节点
30         } else if (sameVnode(oldStartVnode, newStartVnode)) {
31             // patch oldStartVnode和newStartVnode, 索引左移, 继续循环
32             patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue)
33             oldStartVnode = oldCh[++oldStartIdx]
34             newStartVnode = newCh[++newStartIdx]
35
36             // oldEndVnode和newEndVnode是同一个节点
37         } else if (sameVnode(oldEndVnode, newEndVnode)) {
38             // patch oldEndVnode和newEndVnode, 索引右移, 继续循环
39             patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue)
40             oldEndVnode = oldCh[--oldEndIdx]
41             newEndVnode = newCh[--newEndIdx]
42
43             // oldStartVnode和newEndVnode是同一个节点

```

```

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

```

```

84           'It seems there are duplicate keys that is causing an updat
85   e error. ' +
86           'Make sure each v-for item has a unique key.'
87       )
88   }
89
// 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
90 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
91 高效的利用dom。
92     if (sameVnode(vnodeToMove, newStartVnode)) {
93         // patch vnodeToMove和newStartVnode
94         patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue)
95         // 清除
96         oldCh[idxInOld] = undefined
97         // 如果removeOnly是false, 则将找到的和newStartVnodej具有相同的key
98         的Vnode, 叫vnodeToMove.elm
99         // 移动到oldStartVnode.elm之前
100        canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm, o
ldStartVnode.elm)
101
102        // 如果key相同, 但是节点不相同, 则创建一个新的节点
103    } else {
104        // same key but different element. treat as new element
105        createElm(newStartVnode, insertedVnodeQueue, parentElm, oldSt
artVnode.elm)
106    }
107
108    // 右移
109    newStartVnode = newCh[++newStartIdx]
110
}

```

`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

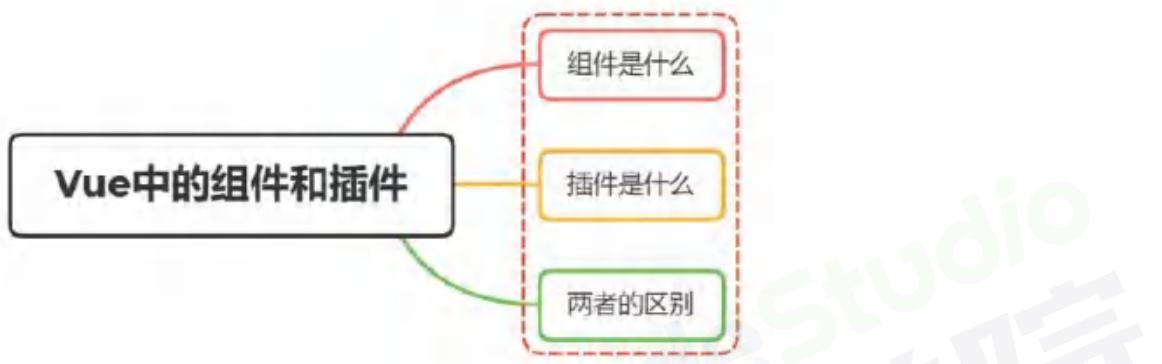
索引减 1，新 `VNode` 节点的开始索引加 1

- 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 `createElement` 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 `createElement` 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` 文件我们都可以说成是一个组件

`vue` 文件标准格式

```

 1 <template>
 2   ...
 3 <script>
 4 export default{
 5   ...
 6 }
 7 </script>
 8 <style>
 9 </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

```

 1 <template id="testComponent">      // 组件显示的内容
 2   <div>component!</div>
 3 </template>
 4
 5 Vue.component('componentA', {
 6   template: '#testComponent'
 7   template: `<div>component</div>` // 组件内容少可以通过这种形式
 8 })

```

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

```

1 - MyPlugin.install = function (Vue, options) {
2     // 1. 添加全局方法或 property
3 -     Vue.myGlobalMethod = function () {
4         // 逻辑...
5     }
6
7     // 2. 添加全局资源
8 -     Vue.directive('my-directive', {
9 -         bind (el, binding, vnode, oldVnode) {
10            // 逻辑...
11        }
12        ...
13    })
14
15     // 3. 注入组件选项
16 -     Vue.mixin({
17 -         created: function () {
18             // 逻辑...
19         }
20         ...
21     })
22
23     // 4. 添加实例方法
24 -     Vue.prototype.$myMethod = function (methodOptions) {
25         // 逻辑...
26     }
27 }
```

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

```

1  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

```
1 const component1 = {...} // 定义一个组件
2
3 export default {
4   components: {
5     component1 // 局部注册
6   }
7 }
```

JavaScript | 复制代码

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

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

JavaScript | 复制代码

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

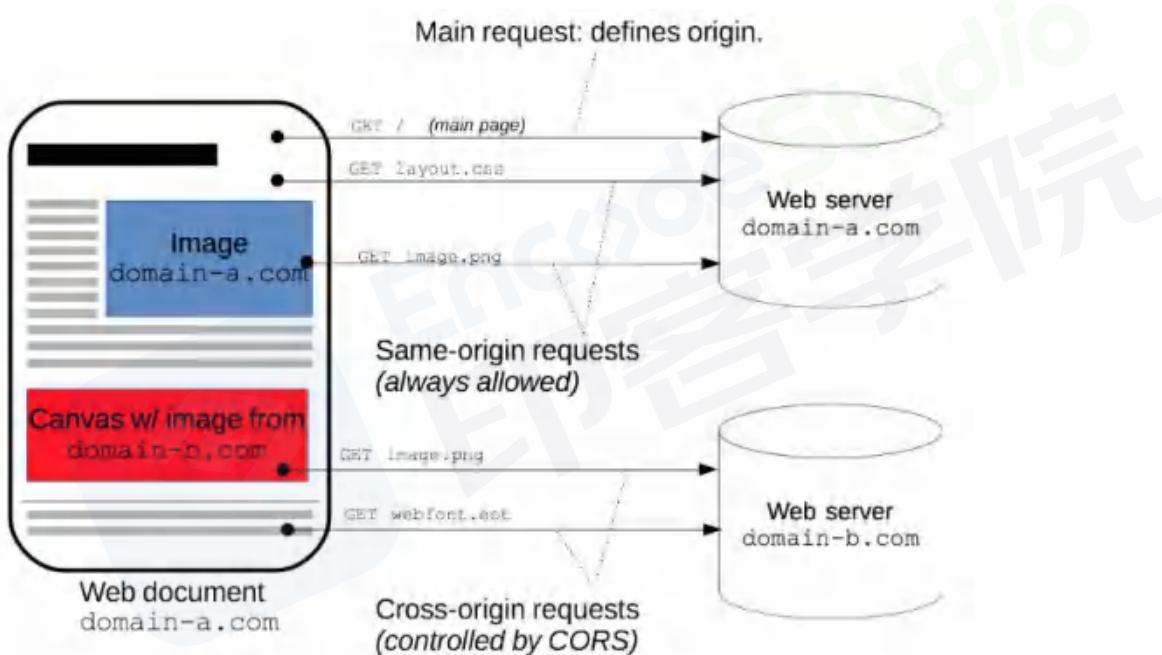
而在 vue 项目中，我们主要针对 CORS 或 Proxy 这两种方案进行展开

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

```
1 app.use(async (ctx, next) => {
2     ctx.set('Access-Control-Allow-Origin', '*');
3     ctx.set('Access-Control-Allow-Headers', 'Content-Type, Content-Length, Authorization, Accept, X-Requested-With , yourHeaderFeild');
4     ctx.set('Access-Control-Allow-Methods', 'PUT, POST, GET, DELETE, OPTIONS');
5     if (ctx.method == 'OPTIONS') {
6         ctx.body = 200;
7     } else {
8         await next();
9     }
10 })
```

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

1 amodule.exports = {
2   devServer: {
3     host: '127.0.0.1',
4     port: 8084,
5     open: true, // vue项目启动时自动打开浏览器
6     proxy: {
7       '/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
8         target: "http://xxx.xxx.xx.xx:8080", //目标地址，一般是指后台
      服务器地址
9         changeOrigin: true, //是否跨域
10        pathRewrite: { // pathRewrite 的作用是把实际Request Url中的'/
        api'用""代替
11          '^/api': ""
12        }
13      }
14    }
15  }
16}
```

通过 `axios` 发送请求中，配置请求的根路径

```
▼ JavaScript | 复制代码

1 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 | 复制代码

```

1 var express = require('express');
2 const proxy = require('http-proxy-middleware')
3 const app = express()
4 app.use(express.static(__dirname + '/'))
5 app.use('/api', proxy({ target: 'http://localhost:4000', changeOrigin: false
6 }));
7 module.exports = app

```

方案三

通过配置 `nginx` 实现代理

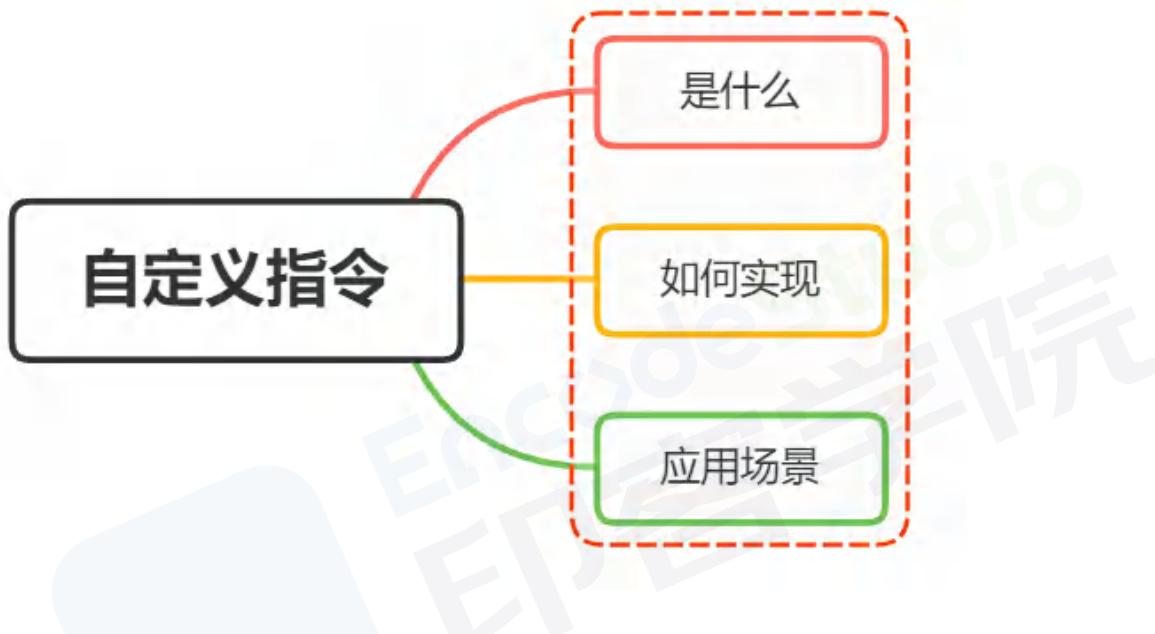
JavaScript | 复制代码

```

1 server {
2     listen 80;
3     # server_name www.josephxia.com;
4     location / {
5         root /var/www/html;
6         index index.html index.htm;
7         try_files $uri $uri/ /index.html;
8     }
9     location /api {
10         proxy_pass http://127.0.0.1:3000;
11         proxy_redirect off;
12         proxy_set_header Host $host;
13         proxy_set_header X-Real-IP $remote_addr;
14         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
15     }
16 }

```

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：

JavaScript | 复制代码

```

1 //会实例化一个指令，但这个指令没有参数
2 `v-xxx`
3
4 // -- 将值传到指令中
5 `v-xxx="value"`
6
7 // -- 将字符串传入到指令中，如`v-html=""<p>内容</p>``
8 `v-xxx="'string'"`
9
10 // -- 传参数 (`arg`)，如`v-bind:class="className"`
11 `v-xxx:arg="value"`
12
13 // -- 使用修饰符 (`modifier`)
14 `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

JavaScript | 复制代码

```

1 // 注册一个全局自定义指令 `v-focus`
2 Vue.directive('focus', {
3     // 当被绑定的元素插入到 DOM 中时....
4     inserted: function (el) {
5         // 聚焦元素
6         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
7     }
8 })

```

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

JavaScript | 复制代码

```

1  directives: {
2    focus: {
3      // 指令的定义
4      inserted: function (el) {
5        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
6      }
7    }
8  }

```

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

JavaScript | 复制代码

```

1  <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：

The screenshot shows a code editor interface with a tab bar at the top labeled "HTML" and "复制代码". Below the tabs is a code editor area containing the following code:

```
1 <div v-demo="{ color: 'white', text: 'hello!' }"></div>
2 <script>
3   Vue.directive('demo', function (el, binding) {
4     console.log(binding.value.color) // "white"
5     console.log(binding.value.text) // "hello!"
6   })
7 </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：

```
1 // 1.设置v-throttle自定义指令
2 Vue.directive('throttle', {
3   bind: (el, binding) => {
4     let throttleTime = binding.value; // 节流时间
5     if (!throttleTime) { // 用户若不设置节流时间，则默认2s
6       throttleTime = 2000;
7     }
8     let cbFun;
9     el.addEventListener('click', event => {
10    if (!cbFun) { // 第一次执行
11      cbFun = setTimeout(() => {
12        cbFun = null;
13      }, throttleTime);
14    } else {
15      event && event.stopImmediatePropagation();
16    }
17  }, true);
18 },
19 });
20 // 2.为button标签设置v-throttle自定义指令
21 <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

```
1  const LazyLoad = {
2      // install方法
3      install(Vue,options){
4          // 代替图片的loading图
5          let defaultSrc = options.default;
6          Vue.directive('lazy',{
7              bind(el,binding){
8                  LazyLoad.init(el,binding.value,defaultSrc);
9              },
10             inserted(el){
11                 // 兼容处理
12                 if('IntersectionObserver' in window){
13                     LazyLoad.observe(el);
14                 }else{
15                     LazyLoad.listenerScroll(el);
16                 }
17             },
18         },
19     })
20 },
21 // 初始化
22 init(el,val,def){
23     // data-src 储存真实src
24     el.setAttribute('data-src',val);
25     // 设置src为loading图
26     el.setAttribute('src',def);
27 },
28 // 利用IntersectionObserver监听el
29 observe(el){
30     let io = new IntersectionObserver(entries => {
31         let realSrc = el.dataset.src;
32         if(entries[0].isIntersecting){
33             if(realSrc){
34                 el.src = realSrc;
35                 el.removeAttribute('data-src');
36             }
37         }
38     });
39     io.observe(el);
40 },
41 // 监听scroll事件
42 listenerScroll(el){
43     let handler = LazyLoad.throttle(LazyLoad.load,300);
44     LazyLoad.load(el);
45     window.addEventListener('scroll',() => {
```

```

46         handler(el);
47     });
48   },
49   // 加载真实图片
50   load(el){
51     let windowHeight = document.documentElement.clientHeight;
52     let elTop = el.getBoundingClientRect().top;
53     let elBtm = el.getBoundingClientRect().bottom;
54     let realSrc = el.dataset.src;
55     if(elTop - windowHeight<0&&elBtm > 0){
56       if(realSrc){
57         el.src = realSrc;
58         el.removeAttribute('data-src');
59       }
60     }
61   },
62   // 节流
63   throttle(fn,delay){
64     let timer;
65     let prevTime;
66     return function(...args){
67       let currTime = Date.now();
68       let context = this;
69       if(!prevTime) prevTime = currTime;
70       clearTimeout(timer);
71
72       if(currTime - prevTime > delay){
73         prevTime = currTime;
74         fn.apply(context,args);
75         clearTimeout(timer);
76         return;
77       }
78
79       timer = setTimeout(function(){
80         prevTime = Date.now();
81         timer = null;
82         fn.apply(context,args);
83       },delay);
84     }
85   }
86 }
87
88 export default LazyLoad;

```

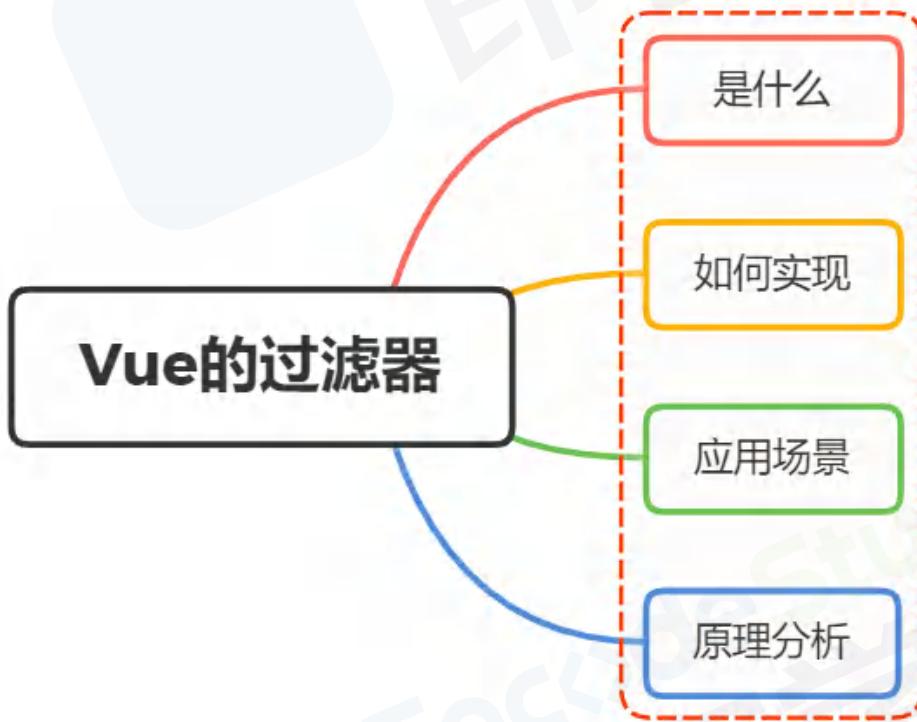
16.2.3. 一键 Copy的功能

```
1 import { Message } from 'ant-design-vue';
2
3 - const vCopy = { // 
4 -   /*
5     bind 钩子函数, 第一次绑定时调用, 可以在这里做初始化设置
6     el: 作用的 dom 对象
7     value: 传给指令的值, 也就是我们要 copy 的值
8   */
9 -   bind(el, { value }) {
10     el.$value = value; // 用一个全局属性来存传进来的值, 因为这个值在别的钩子函数里
11     // 还会用到
12     el.handler = () => {
13       if (!el.$value) {
14         // 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
15         Message.warning('无复制内容');
16         return;
17       }
18       // 动态创建 textarea 标签
19       const textarea = document.createElement('textarea');
20       // 将该 textarea 设为 readonly 防止 iOS 下自动唤起键盘, 同时将 textarea 移
21       // 出可视区域
22       textarea.readOnly = 'readonly';
23       textarea.style.position = 'absolute';
24       textarea.style.left = '-9999px';
25       // 将要 copy 的值赋给 textarea 标签的 value 属性
26       textarea.value = el.$value;
27       // 将 textarea 插入到 body 中
28       document.body.appendChild(textarea);
29       // 选中值并复制
30       textarea.select();
31       // textarea.setSelectionRange(0, textarea.value.length);
32       const result = document.execCommand('Copy');
33       if (result) {
34         Message.success('复制成功');
35       }
36       document.body.removeChild(textarea);
37     };
38   },
39   // 当传进来的值更新的时候触发
40   componentUpdated(el, { value }) {
41     el.$value = value;
42   },
43   // 指令与元素解绑的时候, 移除事件绑定
```

```
44     unbind(el) {
45       el.removeEventListener('click', el.handler);
46     },
47   },
48
49 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`

17.2. 如何用

`vue` 中的过滤器可以用在两个地方：双花括号插值和 `v-bind` 表达式，过滤器应该被添加在 `JavaScript` 表达式的尾部，由“管道”符号指示：

```
1 <!-- 在双花括号中 -->
2 {{ message | capitalize }}
3
4 <!-- 在 `v-bind` 中 -->
5 <div v-bind:id="rawId | formatId"></div>
```

17.2.1. 定义filter

在组件的选项中定义本地的过滤器

```
1 filters: {
2   capitalize: function (value) {
3     if (!value) return ''
4     value = value.toString()
5     return value.charAt(0).toUpperCase() + value.slice(1)
6   }
7 }
```

定义全局过滤器：

```
1 Vue.filter('capitalize', function (value) {
2   if (!value) return ''
3   value = value.toString()
4   return value.charAt(0).toUpperCase() + value.slice(1)
5 })
6
7 new Vue({
8   // ...
9 })
```

注意：当全局过滤器和局部过滤器重名时，会采用局部过滤器

过滤器函数总接收表达式的值（之前的操作链的结果）作为第一个参数。在上述例子中，`capitalize`

过滤器函数将会收到 `message` 的值作为第一个参数

过滤器可以串联：

```
1 {{ message | filterA | filterB }}
```

Plain Text | 复制代码

在这个例子中，`filterA` 被定义为接收单个参数的过滤器函数，表达式 `message` 的值将作为参数传入到函数中。然后继续调用同样被定义为接收单个参数的过滤器函数 `filterB`，将 `filterA` 的结果传递到 `filterB` 中。

过滤器是 `JavaScript` 函数，因此可以接收参数：

```
1 {{ message | filterA('arg1', arg2) }}
```

Plain Text | 复制代码

这里，`filterA` 被定义为接收三个参数的过滤器函数。

其中 `message` 的值作为第一个参数，普通字符串 `'arg1'` 作为第二个参数，表达式 `arg2` 的值作为第三个参数

举个例子：

```
1 <div id="app">
2   <p>{{ msg | msgFormat('疯狂', '--') }}</p>
3 </div>
4
5 <script>
6   // 定义一个 Vue 全局的过滤器，名字叫做 msgFormat
7   Vue.filter('msgFormat', function(msg, arg, arg2) {
8     // 字符串的 replace 方法，第一个参数，除了可写一个字符串之外，还可以定义一个正则
9     return msg.replace(/单纯/g, arg+arg2)
10    })
11 </script>
```

17.2.2. 小结：

- 部过滤器优先于全局过滤器被调用
- 一个表达式可以使用多个过滤器。过滤器之间需要用管道符“|”隔开。其执行顺序从左往右

17.3. 应用场景

平时开发中，需要用到过滤器的地方有很多，比如单位转换、数字打点、文本格式化、时间格式化之类的等

比如我们要实现将30000 => 30,000，这时候我们就需要使用过滤器

```
▼
1 Vue.filter('toThousandFilter', function (value) {
2     if (!value) return ''
3     value = value.toString()
4     return .replace(str.indexOf(',') > -1 ? /(\d)(?=(\d{3})+\.)/g : /(\d)
5         (?=(?:\d{3})+$)/g, '$1,')
```

17.4. 原理分析

使用过滤器

```
▼
1 {{ message | capitalize }}
```

在模板编译阶段过滤器表达式将会被编译为过滤器函数，主要是用过 `parseFilters`，我们放到最后讲

```
▼
1 _s(_f('filterFormat')(message))
```

首先分析一下 `_f`：

`_f` 函数全名是： `resolveFilter`，这个函数的作用是从 `this.$options.filters` 中找出注册的过滤器并返回

```
▼
1 // 变为
2 this.$options.filters['filterFormat'](message) // message为参数
```

关于 `resolveFilter`

JavaScript | 复制代码

```

1 import { identity, resolveAsset } from 'core/util/index'
2
3 export function resolveFilter(id){
4     return resolveAsset(this.$options,'filters',id,true) || identity
5 }

```

内部直接调用 `resolveAsset`，将 `option` 对象，类型，过滤器 `id`，以及一个触发警告的标志作为参数传递，如果找到，则返回过滤器；

`resolveAsset` 的代码如下：

JavaScript | 复制代码

```

1 export function resolveAsset(options,type,id,warnMissing){ // 因为我们找的是
  过滤器，所以在 resolveFilter 函数中调用时 type 的值直接给的 'filters'，实际这个函数
  还可以拿到其他很多东西
2 if(typeof id !== 'string'){ // 判断传递的过滤器 id 是不是字符串，不是则直接返
  回
3     return
4 }
5 const assets = options[type] // 将我们注册的所有过滤器保存在变量中
6 // 接下来的逻辑便是判断 id 是否在 assets 中存在，即进行匹配
7 if(hasOwn(assets,id)) return assets[id] // 如找到，直接返回过滤器
8 // 没有找到，代码继续执行
9 const camelizedId = camelize(id) // 万一你是驼峰的呢
10 if(hasOwn(assets,camelizedId)) return assets[camelizedId]
11 // 没找到，继续执行
12 const PascalCaseId = capitalize(camelizedId) // 万一你是首字母大写的驼峰呢
13 if(hasOwn(assets,PascalCaseId)) return assets[PascalCaseId]
14 // 如果还是没找到，则检查原型链（即访问属性）
15 const result = assets[id] || assets[camelizedId] || assets[PascalCaseI
d]
16 // 如果依然没找到，则在非生产环境的控制台打印警告
17 if(process.env.NODE_ENV !== 'production' && warnMissing && !result){
18     warn('Failed to resolve ' + type.slice(0,-1) + ':' + id, options)
19 }
20 // 无论是否找到，都返回查找结果
21 return result
22 }

```

下面再来分析一下 `_s`：

`_s` 函数的全称是 `toString`，过滤器处理后的结果会当作参数传递给 `toString` 函数，最终 `toString` 函数执行后的结果会保存到 `Vnode` 中的 `text` 属性中，渲染到视图中

JavaScript | 复制代码

```

1  function toString(value){
2      return value == null
3      ?
4      : typeof value === 'object'
5      ? JSON.stringify(value,null,2)// JSON.stringify()第三个参数可用来控制字符串里面的间距
6      : String(value)
7 }

```

最后，在分析下 `parseFilters`，在模板编译阶段使用该函数阶段将模板过滤器解析为过滤器函数调用表达式

JavaScript | 复制代码

```

1  function parseFilters (filter) {
2      let filters = filter.split('|')
3      let expression = filters.shift().trim() // shift()删除数组第一个元素并将其返回，该方法会更改原数组
4      let i
5      if (filters) {
6          for(i = 0;i < filters.length;i++){
7              expression = warpFilter(expression,filters[i].trim()) // 这里传进去的expression实际上是管道符号前面的字符串，即过滤器的第一个参数
8          }
9      }
10     return expression
11 }
12 // warpFilter函数实现
13 function warpFilter(exp,filter){
14     // 首先判断过滤器是否有其他参数
15     const i = filter.indexOf('(')
16     if(i<0){ // 不含其他参数，直接进行过滤器表达式字符串的拼接
17         return `_f("${filter}")(${exp})`
18     }else{
19         const name = filter.slice(0,i) // 过滤器名称
20         const args = filter.slice(i+1) // 参数，但还多了 ')'
21         return `_f('${name}')(${exp},${args})` // 注意这一步少给了一个 ')'
22     }
23 }

```

17.5. 小结：

- 在编译阶段通过 `parseFilters` 将过滤器编译成函数调用（串联过滤器则是一个嵌套的函数调用，前一个过滤器执行的结果是后一个过滤器函数的参数）
- 编译后通过调用 `resolveFilter` 函数找到对应过滤器并返回结果
- 执行结果作为参数传递给 `toString` 函数，而 `toString` 执行后，其结果会保存在 `Vnode` 的 `text` 属性中，渲染到视图

18. 说说你对slot的理解？slot使用场景有哪些？



18.1. slot是什么

在HTML中 `slot` 元素，作为 `Web Components` 技术套件的一部分，是Web组件内的一个占位符
该占位符可以在后期使用自己的标记语言填充

举个栗子

HTML | 复制代码

```

1 <template id="element-details-template">
2   <slot name="element-name">Slot template</slot>
3 </template>
4 <element-details>
5   <span slot="element-name">1</span>
6 </element-details>
7 <element-details>
8   <span slot="element-name">2</span>
9 </element-details>

```

`template` 不会展示到页面中，需要用先获取它的引用，然后添加到 `DOM` 中，

JavaScript | 复制代码

```

1 customElements.define('element-details',
2   class extends HTMLElement {
3     constructor() {
4       super();
5       const template = document
6         .getElementById('element-details-template')
7         .content;
8       const shadowRoot = this.attachShadow({mode: 'open'})
9         .appendChild(template.cloneNode(true));
10    }
11  })

```

在 `Vue` 中的概念也是如此

`Slot` 艺名插槽，花名“占坑”，我们可以理解为 `slot` 在组件模板中占好了位置，当使用该组件标签时候，组件标签里面的内容就会自动填坑（替换组件模板中 `slot` 位置），作为承载分发内容的出口
可以将其类比为插卡式的FC游戏机，游戏机暴露卡槽（插槽）让用户插入不同的游戏磁条（自定义内容）

放张图感受一下



18.2. 使用场景

通过插槽可以让用户可以拓展组件，去更好地复用组件和对其做定制化处理

如果父组件在使用到一个复用组件的时候，获取这个组件在不同的地方有少量的更改，如果去重写组件是一件不明智的事情

通过 `slot` 插槽向组件内部指定位置传递内容，完成这个复用组件在不同场景的应用

比如布局组件、表格列、下拉选、弹框显示内容等

18.3. 分类

`slot` 可以分来以下三种：

- 默认插槽
- 具名插槽
- 作用域插槽

18.3.1. 默认插槽

子组件用 `<slot>` 标签来确定渲染的位置，标签里面可以放 `DOM` 结构，当父组件使用的时候没有往插槽传入内容，标签内 `DOM` 结构就会显示在页面

父组件在使用的时候，直接在子组件的标签内写入内容即可

子组件 `Child.vue`

```
1 <template>
2   <slot>
3     <p>插槽后备的内容</p>
4   </slot>
5 </template>
```

父组件

```
1 <Child>
2   <div>默认插槽</div>
3 </Child>
```

18.3.2. 具名插槽

子组件用 `name` 属性来表示插槽的名字，不传为默认插槽

父组件中在使用时在默认插槽的基础上加上 `slot` 属性，值为子组件插槽 `name` 属性值

子组件 `Child.vue`

```
1 <template>
2   <slot>插槽后备的内容</slot>
3   <slot name="content">插槽后备的内容</slot>
4 </template>
```

父组件

HTML | 复制代码

```

1 <child>
2   <template v-slot:default>具名插槽</template>
3   <!-- 具名插槽用插槽名做参数 -->
4   <template v-slot:content>内容...</template>
5 </child>

```

18.3.3. 作用域插槽

子组件在作用域上绑定属性来将子组件的信息传给父组件使用，这些属性会被挂在父组件 `v-slot` 接受的对象上

父组件中在使用时通过 `v-slot:` (简写: `#`) 获取子组件的信息，在内容中使用

子组件 `Child.vue`

```

1 <template>
2   <slot name="footer" testProps="子组件的值">
3     <h3>没传footer插槽</h3>
4   </slot>
5 </template>

```

父组件

```

1 <child>
2   <!-- 把v-slot的值指定为作用域上下文对象 -->
3   <template v-slot:default="slotProps">
4     来自子组件数据: {{slotProps.testProps}}
5   </template>
6   <template #default="slotProps">
7     来自子组件数据: {{slotProps.testProps}}
8   </template>
9 </child>

```

18.3.4. 小结：

- `v-slot` 属性只能在 `<template>` 上使用，但在只有默认插槽时可以在组件标签上使用
- 默认插槽名为 `default`，可以省略 `default` 直接写 `v-slot`

- 缩写为 `#` 时不能不写参数，写成 `#default`
- 可以通过解构获取 `v-slot={user}`，还可以重命名 `v-slot="{user: newName}"` 和定义默认值 `v-slot="{user = '默认值'}"`

18.4. 原理分析

`slot` 本质上是返回 `VNode` 的函数，一般情况下，`Vue` 中的组件要渲染到页面上需要经过 `template -> render function -> VNode -> DOM` 过程，这里看看 `slot` 如何实现：

编写一个 `buttonCounter` 组件，使用匿名插槽

```
1 Vue.component('button-counter', {  
2   template: '<div> <slot>我是默认内容</slot></div>'  
3 })
```

使用该组件

```
1 new Vue({  
2   el: '#app',  
3   template: '<button-counter><span>我是slot传入内容</span></button-counter>',  
4   components:{buttonCounter}  
5 })
```

获取 `buttonCounter` 组件渲染函数

```
1 (function anonymous(  
2 ) {  
3 with(this){return _c('div',[_t("default"),[_v("我是默认内容")]],2)}  
4 })
```

`_v` 表示穿件普通文本节点，`_t` 表示渲染插槽的函数

渲染插槽函数 `renderSlot`（做了简化）

JavaScript | 复制代码

```

1  function renderSlot (
2    name,
3    fallback,
4    props,
5    bindObject
6  ) {
7    // 得到渲染插槽内容的函数
8    var scopedSlotFn = this.$scopedSlots[name];
9    var nodes;
10   // 如果存在插槽渲染函数，则执行插槽渲染函数，生成nodes节点返回
11   // 否则使用默认值
12   nodes = scopedSlotFn(props) || fallback;
13   return nodes;
14 }

```

`name` 属性表示定义插槽的名字，默认值为 `default`，`fallback` 表示子组件中的 `slot` 节点的默认值

关于 `this.$scopedSlots` 是什么，我们可以先看看 `vm.slot`

JavaScript | 复制代码

```

1  function initRender (vm) {
2    ...
3    vm.$slots = resolveSlots(options._renderChildren, renderContext);
4    ...
5  }

```

`resolveSlots` 函数会对 `children` 节点做归类和过滤处理，返回 `slots`

```
1  function resolveSlots (
2      children,
3      context
4  ) {
5      if (!children || !children.length) {
6          return {}
7      }
8      var slots = {};
9      for (var i = 0, l = children.length; i < l; i++) {
10         var child = children[i];
11         var data = child.data;
12         // remove slot attribute if the node is resolved as a Vue slot node
13         if (data && data.attrs && data.attrs.slot) {
14             delete data.attrs.slot;
15         }
16         // named slots should only be respected if the vnode was rendered in the
17         // same context.
18         if ((child.context === context || child.fnContext === context) &&
19             data && data.slot != null
20         ) {
21             // 如果slot存在(slot="header") 则拿对应的值作为key
22             var name = data.slot;
23             var slot = (slots[name] || (slots[name] = []));
24             // 如果是tempalte元素 则把template的children添加进数组中，这也就是为什么
25             // 你写的template标签并不会渲染成另一个标签到页面
26             if (child.tag === 'template') {
27                 slot.push.apply(slot, child.children || []);
28             } else {
29                 slot.push(child);
30             }
31             } else {
32                 // 如果没有就默认是default
33                 (slots.default || (slots.default = [])).push(child);
34             }
35             // ignore slots that contains only whitespace
36             for (var name$1 in slots) {
37                 if (slots[name$1].every(isWhitespace)) {
38                     delete slots[name$1];
39                 }
40             }
41             return slots
42 }
```

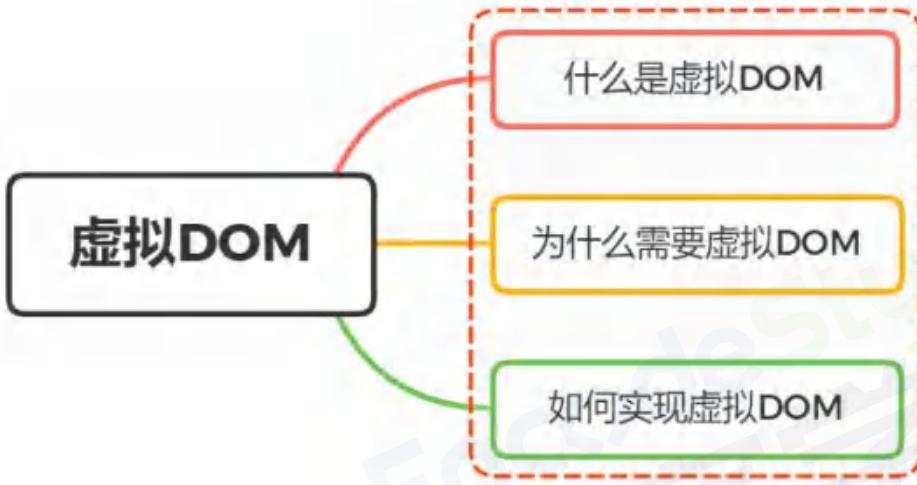
`_render` 渲染函数通过 `normalizeScopedSlots` 得到 `vm.$scopedSlots`

```
1  vm.$scopedSlots = normalizeScopedSlots(  
2    _parentVnode.data.scopedSlots,  
3    vm.$slots,  
4    vm.$scopedSlots  
5  );
```

JavaScript | 复制代码

作用域插槽中父组件能够得到子组件的值是因为在 `renderSlot` 的时候执行会传入 `props`，也就是上述 `_t` 第三个参数，父组件则能够得到子组件传递过来的值

19. 什么是虚拟DOM？如何实现一个虚拟DOM？说说你的思路



19.1. 什么是虚拟DOM

虚拟 DOM (`Virtual DOM`) 这个概念相信大家都不陌生，从 `React` 到 `Vue`，虚拟 `DOM` 为这两个框架都带来了跨平台的能力 (`React-Native` 和 `Weex`)

实际上它只是一层对真实 `DOM` 的抽象，以 `JavaScript` 对象 (`VNode` 节点) 作为基础的树，用对象的属性来描述节点，最终可以通过一系列操作使这棵树映射到真实环境上

在 `Javascript` 对象中，虚拟 `DOM` 表现为一个 `Object` 对象。并且最少包含标签名 (`tag`)、属性 (`attrs`) 和子元素对象 (`children`) 三个属性，不同框架对这三个属性的命名可能会有差别

创建虚拟 DOM 就是为了更好将虚拟的节点渲染到页面视图中，所以虚拟 DOM 对象的节点与真实 DOM 的属性一一照应

在 vue 中同样使用到了虚拟 DOM 技术

定义真实 DOM

```
▼ HTML | 复制代码  
1 <div id="app">  
2   <p class="p">节点内容</p>  
3   <h3>{{ foo }}</h3>  
4 </div>
```

实例化 vue

```
▼ JavaScript | 复制代码  
1 const app = new Vue({  
2   el:"#app",  
3   data:{  
4     foo:"foo"  
5   }  
6 })
```

观察 render 的 render，我们能得到虚拟 DOM

```
▼ JavaScript | 复制代码  
1 (function anonymous(  
2 ) {  
3   with(this){return _c('div',{attrs:{'id':'app'}},[_c('p',{staticClass:'p'}  
,  
        [_v("节点内容")],_v(" "),_c('h3',[_v(_s(foo))])]))}})
```

通过 VNode，vue 可以对这颗抽象树进行创建节点,删除节点以及修改节点的操作， 经过 diff 算法得出一些需要修改的最小单位,再更新视图，减少了 dom 操作，提高了性能

19.2. 为什么需要虚拟DOM

DOM 是很慢的，其元素非常庞大，页面的性能问题，大部分都是由 DOM 操作引起的

真实的 DOM 节点，哪怕一个最简单的 `div` 也包含着很多属性，可以打印出来直观感受一下：

```
<div> {> document.documentElement (<<div>)
</div>
  <div style="border: 1px solid black; width: 100px; height: 100px; margin: auto; position: relative; text-align: center; font-size: 14px; color: red; border-radius: 50%;> I am a div
  </div>
  <script>
    console.log(this);
  </script>
</div>
```

由此可见，操作 `DOM` 的代价仍旧是昂贵的，频繁操作还是会出现页面卡顿，影响用户的体验

举个例子：

你用传统的原生 `api` 或 `jQuery` 去操作 `DOM` 时，浏览器会从构建 `DOM` 树开始从头到尾执行一遍流程

当你在一次操作时，需要更新10个 `DOM` 节点，浏览器没这么智能，收到第一个更新 `DOM` 请求后，并不知道后续还有9次更新操作，因此会马上执行流程，最终执行10次流程

而通过 `VNode`，同样更新10个 `DOM` 节点，虚拟 `DOM` 不会立即操作 `DOM`，而是将这10次更新的 `diff` 内容保存到本地的一个 `js` 对象中，最终将这个 `js` 对象一次性 `attach` 到 `DOM` 树上，避免大量的无谓计算

很多人认为虚拟 `DOM` 最大的优势是 `diff` 算法，减少 JavaScript 操作真实 `DOM` 的带来的性能消耗。虽然这一个虚拟 `DOM` 带来的一个优势，但并不是全部。虚拟 `DOM` 最大的优势在于抽象了原本的渲染过程，实现了跨平台的能力，而不仅仅局限于浏览器的 `DOM`，可以是安卓和 IOS 的原生组件，可以是近期很火热的小程序，也可以是各种GUI

19.3. 如何实现虚拟DOM

首先可以看看 `vue` 中 `VNode` 的结构

源码位置：src/core/vdom/vnode.js

```
1 export default class VNode {
2     tag: string | void;
3     data: VNodeData | void;
4     children: ?Array<VNode>;
5     text: string | void;
6     elm: Node | void;
7     ns: string | void;
8     context: Component | void; // rendered in this component's scope
9     functionalContext: Component | void; // only for functional component root nodes
10    key: string | number | void;
11    componentOptions: VNodeComponentOptions | void;
12    componentInstance: Component | void; // component instance
13    parent: VNode | void; // component placeholder node
14    raw: boolean; // contains raw HTML? (server only)
15    isStatic: boolean; // hoisted static node
16    isRootInsert: boolean; // necessary for enter transition check
17    isComment: boolean; // empty comment placeholder?
18    isCloned: boolean; // is a cloned node?
19    isOnce: boolean; // is a v-once node?
20
21    constructor (
22        tag?: string,
23        data?: VNodeData,
24        children?: ?Array<VNode>,
25        text?: string,
26        elm?: Node,
27        context?: Component,
28        componentOptions?: VNodeComponentOptions
29    ) {
30        /*当前节点的标签名*/
31        this.tag = tag
32        /*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*/
33        this.data = data
34        /*当前节点的子节点，是一个数组*/
35        this.children = children
36        /*当前节点的文本*/
37        this.text = text
38        /*当前虚拟节点对应的真实dom节点*/
39        this.elm = elm
40        /*当前节点的名字空间*/
41        this.ns = undefined
42        /*编译作用域*/
43        this.context = context
}
```

```

44     /*函数化组件作用域*/
45     this.functionalContext = undefined
46     /*节点的key属性，被当作节点的标志，用以优化*/
47     this.key = data && data.key
48     /*组件的option选项*/
49     this.componentOptions = componentOptions
50     /*当前节点对应的组件的实例*/
51     this.componentInstance = undefined
52     /*当前节点的父节点*/
53     this.parent = undefined
54     /*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*/
55     this.raw = false
56     /*静态节点标志*/
57     this.isStatic = false
58     /*是否作为跟节点插入*/
59     this.isRootInsert = true
60     /*是否为注释节点*/
61     this.isComment = false
62     /*是否为克隆节点*/
63     this.isCloned = false
64     /*是否有v-once指令*/
65     this.isOnce = false
66   }
67
68   // DEPRECATED: alias for componentInstance for backwards compat.
69   /* istanbul ignore next https://github.com/answershuto/learnVue*/
70   get child (): Component | void {
71     return this.componentInstance
72   }
73 }

```

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

```
1  export function createElement (
2    context: Component,
3    tag: any,
4    data: any,
5    children: any,
6    normalizationType: any,
7    alwaysNormalize: boolean
8  ): VNode | Array<VNode> {
9    if (Array.isArray(data) || isPrimitive(data)) {
10      normalizationType = children
11      children = data
12      data = undefined
13    }
14    if (isTrue(alwaysNormalize)) {
15      normalizationType = ALWAYS_NORMALIZE
16    }
17    return _createElement(context, tag, data, children, normalizationType)
18 }
```

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

```

1  export function _createElement(
2      context: Component,
3      tag?: string | Class<Component> | Function | Object,
4      data?: VNodeData,
5      children?: any,
6      normalizationType?: number
7  ): VNode | Array<VNode> {
8      if (isDef(data) && isDef((data: any).__ob__)) {
9          process.env.NODE_ENV !== 'production' && warn(
10              `Avoid using observed data object as vnode data: ${JSON.stringify(
11                  data
12              )}\n` +
13              'Always create fresh vnode data objects in each render!',
14              context
15          )
16      }
17      // object syntax in v-bind
18      if (isDef(data) && isDef(data.is)) {
19          tag = data.is
20      }
21      if (!tag) {
22          // in case of component :is set to falsy value
23          return createEmptyVNode()
24      }
25      ...
26      // support single function children as default scoped slot
27      if (Array.isArray(children) &&
28          typeof children[0] === 'function'
29      ) {
30          data = data || {}
31          data.scopedSlots = { default: children[0] }
32          children.length = 0
33      }
34      if (normalizationType === ALWAYS_NORMALIZE) {
35          children = normalizeChildren(children)
36      } else if ( === SIMPLE_NORMALIZE) {
37          children = simpleNormalizeChildren(children)
38      }
39      // 创建VNode
40      ...
}

```

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

```

1 if (normalizationType === ALWAYS_NORMALIZE) {
2     children = normalizeChildren(children)
3 } else if ( === SIMPLE_NORMALIZE) {
4     children = simpleNormalizeChildren(children)
5 }

```

`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/normalize-children.js

在规范化 `children` 后，就去创建 `VNode`

JavaScript | 复制代码

```
1 let vnode, ns
2 // 对tag进行判断
3 if (typeof tag === 'string') {
4     let Ctor
5     ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
6 }
7 if (config.isReservedTag(tag)) {
8     // 如果是内置的节点，则直接创建一个普通VNode
9     vnode = new VNode(
10         config.parsePlatformTagName(tag), data, children,
11         undefined, undefined, context
12     )
13 } else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
14     // component
15     // 如果是component类型，则会通过createComponent创建VNode节点
16     vnode = createComponent(Ctor, data, context, children, tag)
17 } else {
18     vnode = new VNode(
19         tag, data, children,
20         undefined, undefined, context
21     )
22 } else {
23     // direct component options / constructor
24     vnode = createComponent(tag, data, context, children)
25 }
```

createComponent 同样是创建 VNode

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

```
1  export function createComponent (          // 构建子类构造函数
2    Ctor: Class<Component> | Function | Object | void,
3    data: ?VNodeData,
4    context: Component,
5    children: ?Array<VNode>,
6    tag?: string
7  ): VNode | Array<VNode> | void {
8    if (isUndef(Ctor)) {
9      return
10    }
11    // 构建子类构造函数
12    const baseCtor = context.$options._base
13
14    // plain options object: turn it into a constructor
15    if (isObject(Ctor)) {
16      Ctor = baseCtor.extend(Ctor)
17    }
18
19    // if at this stage it's not a constructor or an async component factor
20    y,
21    // reject.
22    if (typeof Ctor !== 'function') {
23      if (process.env.NODE_ENV !== 'production') {
24        warn(`Invalid Component definition: ${String(Ctor)}`, context)
25      }
26    }
27
28    // async component
29    let asyncFactory
30    if (isUndef(Ctor.cid)) {
31      asyncFactory = Ctor
32      Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)
33    }
34    if (Ctor === undefined) {
35      return createAsyncPlaceholder(
36        asyncFactory,
37        data,
38        context,
39        children,
40        tag
41      )
42    }
43
44    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
61        ren)
62    }
63
64    // extract listeners, since these needs to be treated as
65    // child component listeners instead of DOM listeners
66    const listeners = data.on
67    // replace with listeners with .native modifier
68    // so it gets processed during parent component patch.
69    data.on = data.nativeOn
70
71    if (isTrue(Ctor.options.abstract)) {
72        const slot = data.slot
73        data = {}
74        if (slot) {
75            data.slot = slot
76        }
77    }
78
79    // 安装组件钩子函数，把钩子函数合并到data.hook中
80    installComponentHooks(data)
81
82    //实例化一个VNode返回。组件的VNode是没有children的
83    const name = Ctor.options.name || tag
84    const vnode = new VNode(
85        `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
86        data, undefined, undefined, undefined, context,
87        { Ctor, propsData, listeners, tag, children },
88        asyncFactory
89    )
90    if (__WEEEX__ && isRecyclableComponent(vnode)) {
91        return renderRecyclableComponentTemplate(vnode)
92    }

```

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

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

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

导入

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

发送请求

```
1 - axios({
2     url:'xxx',      // 设置请求的地址
3     method:"GET", // 设置请求方法
4 -     params:{      // get请求使用params进行参数凭借,如果是post请求用data
5         type: '',
6         page: 1
7     }
8 - }).then(res => {
9     // res为后端返回的数据
10    console.log(res);
11 })
```

并发请求 `axios.all([])`

```
1 - function getUserAccount() {  
2     return axios.get('/user/12345');  
3 }  
4  
5 - function getUserPermissions() {  
6     return axios.get('/user/12345/permissions');  
7 }  
8  
9 axios.all([getUserAccount(), getUserPermissions()])  
10 - .then(axios.spread(function (res1, res2) {  
11     // res1第一个请求的返回的内容, res2第二个请求返回的内容  
12     // 两个请求都执行完成才会执行  
13 }));
```

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：

```

1  axios('http://localhost:3000/data', {
2      // 配置代码
3      method: 'GET',
4      timeout: 1000,
5      withCredentials: true,
6      headers: {
7          'Content-Type': 'application/json',
8          Authorization: 'xxx',
9      },
10     transformRequest: [function (data, headers) {
11         return data;
12     }],
13     // 其他请求配置...
14 })
15     .then((data) => {
16         // todo: 真正业务逻辑代码
17         console.log(data);
18     }, (err) => {
19         // 错误处理代码
20         if (err.response.status === 401) {
21             // handle authorization error
22         }
23         if (err.response.status === 403) {
24             // handle server forbidden error
25         }
26         // 其他错误处理.....
27         console.log(err);
28 });

```

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

```
1 if (process.env.NODE_ENV === 'development') {  
2     axios.defaults.baseURL = 'http://dev.xxx.com'  
3 } else if (process.env.NODE_ENV === 'production') {  
4     axios.defaults.baseURL = 'http://prod.xxx.com'  
5 }
```

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

```
1 devServer: {  
2     proxy: {  
3         '/proxyApi': {  
4             target: 'http://dev.xxx.com',  
5             changeOrigin: true,  
6             pathRewrite: {  
7                 '/proxyApi': ''  
8             }  
9         }  
10    }  
11 }
```

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 | 复制代码

```
1 const service = axios.create({
2   ...
3   timeout: 30000, // 请求 30s 超时
4   headers: {
5     get: {
6       'Content-Type': 'application/x-www-form-urlencoded;charset=utf-
8'
7       // 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
8     },
9     post: {
10       'Content-Type': 'application/json;charset=utf-8'
11       // 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
12     }
13   },
14 })
```

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

```
1 // get 请求
2 export function httpGet({
3   url,
4   params = {}
5 }) {
6   return new Promise((resolve, reject) => {
7     axios.get(url, {
8       params
9     }).then((res) => {
10       resolve(res.data)
11     }).catch(err => {
12       reject(err)
13     })
14   })
15 }
16
17 // post
18 // post请求
19 export function httpPost({
20   url,
21   data = {},
22   params = {}
23 }) {
24   return new Promise((resolve, reject) => {
25     axios({
26       url,
27       method: 'post',
28       transformRequest: [function (data) {
29         let ret = ''
30         for (let it in data) {
31           ret += encodeURIComponent(it) + '=' + encodeURIComponent(data[it])
32         } + '&'
33         return ret
34       }],
35       // 发送的数据
36       data,
37       // url参数
38       params
39     }).then(res => {
40       resolve(res.data)
41     })
42   })
43 }
44 }
```

把封装的方法放在一个 `api.js` 文件中

```
▼ JavaScript | ⌂ 复制代码  
1 import { httpGet, httpPost } from './http'  
2 export const getorglist = (params = {}) => httpGet({ url: 'apps/api/org/list', params })
```

页面中就能直接调用

```
▼ JavaScript | ⌂ 复制代码  
1 // .vue  
2 import { getorglist } from '@/assets/js/api'  
3  
4 getorglist({ id: 200 }).then(res => {  
5   console.log(res)  
6 })
```

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 | ⌂ 复制代码  
1 // 请求拦截器  
2 axios.interceptors.request.use(  
3   config => {  
4     // 每次发送请求之前判断是否存在token  
5     // 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  
6     // 情况，此处token一般是用户完成登录后储存到localStorage里的  
7     token && (config.headers.Authorization = token)  
8     return config  
9   },  
10   error => {  
11     return Promise.error(error)  
12   })
```

20.4.5. 响应拦截器

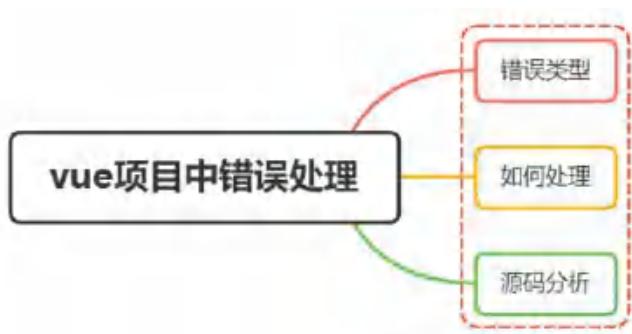
响应拦截器可以在接收到响应后先做一层操作，如根据状态码判断登录状态、授权

```
1 // 响应拦截器
2 axios.interceptors.response.use(response => {
3     // 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
4     // 否则的话抛出错误
5     if (response.status === 200) {
6         if (response.data.code === 511) {
7             // 未授权调取授权接口
8         } else if (response.data.code === 510) {
9             // 未登录跳转登录页
10        } else {
11            return Promise.resolve(response)
12        }
13    } else {
14        return Promise.reject(response)
15    }
16 }, error => {
17     // 我们可以在这里对异常状态作统一处理
18     if (error.response.status) {
19         // 处理请求失败的情况
20         // 对不同返回码对相应处理
21         return Promise.reject(error.response)
22     }
23 })
```

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

```


1  apiClient.interceptors.response.use(
2    response => {
3      return response;
4    },
5    error => {
6      if (error.response.status == 401) {
7        router.push({ name: "Login" });
8      } else {
9        message.error("出错了");
10       return Promise.reject(error);
11     }
12   }
13 );


```

JavaScript

复制代码

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 | ⌂ 复制代码

1 Vue.config.errorHandler = function (err, vm, info) {
2   // handle error
3   // `info` 是 Vue 特定的错误信息，比如错误所在的生命周期钩子
4   // 只在 2.2.0+ 可用
5 }
```

`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 | ⌂ 复制代码

1 (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`

```

1  Vue.component('cat', {
2      template: `
3          <div>
4              <h1>Cat: </h1>
5                  <slot></slot>
6          </div>`,
7      props: {
8          name: {
9              required: true,
10             type: String
11         }
12     },
13     errorCaptured(err, vm, info) {
14         console.log(`cat EC: ${err.toString()}\ninfo: ${info}`);
15         return false;
16     }
17 }
18 });

```

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

```

1  Vue.component('kitten', {
2      template: '<div><h1>Kitten: {{ dontexist() }}</h1></div>',
3      props: {
4          name: {
5              required: true,
6              type: String
7          }
8      }
9  });

```

页面中使用组件

```
1 <div id="app" v-cloak>
2   <cat name="my cat">
3     <kitten></kitten>
4   </cat>
5 </div>
```

在父组件的 `errorCaptured` 则能够捕获到信息

```
1 cat EC: TypeError: dontexist is not a function
2 info: render
```

21.2.3. 源码分析

异常处理源码

源码位置：/src/core/util/error.js

```

1 // Vue 全局配置,也就是上面的Vue.config
2 import config from '../config'
3 import { warn } from './debug'
4 // 判断环境
5 import { inBrowser, inWeex } from './env'
6 // 判断是否是Promise, 通过val.then === 'function' && val.catch === 'function', val !== null && val !== undefined
7 import { isPromise } from 'shared/util'
8 // 当错误函数处理错误时, 停用deps跟踪以避免可能出现的infinite rendering
9 // 解决以下出现的问题https://github.com/vuejs/vuex/issues/1505的问题
10 import { pushTarget, popTarget } from '../observer/dep'
11
12 export function handleError (err: Error, vm: any, info: string) {
13     // Deactivate deps tracking while processing error handler to avoid possible infinite rendering.
14     pushTarget()
15     try {
16         // vm指当前报错的组件实例
17         if (vm) {
18             let cur = vm
19             // 首先获取到报错的组件, 之后递归查找当前组件的父组件, 依次调用errorCaptured 方法。
20             // 在遍历调用完所有 errorCaptured 方法、或 errorCaptured 方法有报错时, 调用 globalHandleError 方法
21             while ((cur = cur.$parent)) {
22                 const hooks = cur.$options.errorCaptured
23                 // 判断是否存在errorCaptured钩子函数
24                 if (hooks) {
25                     // 选项合并的策略, 钩子函数会被保存在一个数组中
26                     for (let i = 0; i < hooks.length; i++) {
27                         // 如果errorCaptured 钩子执行自身抛出了错误,
28                         // 则用try{}catch{}捕获错误, 将这个新错误和原本被捕获的错误都会发送给全局的config.errorHandler
29                         // 调用globalHandleError方法
30                         try {
31                             // 当前errorCaptured执行, 根据返回是否是false值
32                             // 是false, capture = true, 阻止其它任何会被这个错误唤起的 errorCaptured 钩子和全局的 config.errorHandler
33                             // 是true capture = false, 组件的继承或父级从属链路中存在的多个 errorCaptured 钩子, 会被相同的错误逐个唤起
34                             // 调用对应的钩子函数, 处理错误
35                             const capture = hooks[i].call(cur, err, vm, i)
36                             if (capture) return
37                         } catch (e) {

```

```

38                               globalHandleError(e, cur, 'errorCaptured hook')
39           k')
40           }
41           }
42           }
43           }
44           // 除非禁止错误向上传播，否则都会调用全局的错误处理函数
45           globalHandleError(err, vm, info)
46     } finally {
47       popTarget()
48     }
49   }
50   // 异步错误处理函数
51   export function invokeWithErrorHandling (
52     handler: Function,
53     context: any,
54     args: null | any[],
55     vm: any,
56     info: string
57   ) {
58     let res
59     try {
60       // 根据参数选择不同的handle执行方式
61       res = args ? handler.apply(context, args) : handler.call(
62         context)
63       // handle返回结果存在
64       // res._isVue an flag to avoid this being observed, 如果传
65       // 入值的_isVue为ture时(即传入的值是Vue实例本身)不会新建observer实例
66       // isPromise(res) 判断val.then === 'function' && val.catc
67       h === 'function', val !=== null && val !== undefined
68       // !res._handled _handle是Promise 实例的内部变量之一， 默认是
69       false, 代表onFulfilled,onRejected是否被处理
70       if (res && !res._isVue && isPromise(res) && !res._handled)
71     ) {
72       res.catch(e => handleError(e, vm, info + ` (Promise/a
73       sync)`))
74       calls
75       }
76     } catch (e) {
77       // 处理执行错误
78       handleError(e, vm, info)
79     }
80     return res
81   }

```

```

78 //全局错误处理
79 function globalHandleError (err, vm, info) {
80     // 获取全局配置，判断是否设置处理函数，默认undefined
81     // 已配置
82     if (config.errorHandler) {
83         // try{}catch{} 住全局错误处理函数
84         try {
85             // 执行设置的全局错误处理函数，handle error 想干啥就干啥❤
86             return config.errorHandler.call(null, err, vm, info)
87         } catch (e) {
88             // 如果开发者在errorHandler函数中手动抛出同样错误信息throw err
89             // 判断err信息是否相等，避免log两次
90             // 如果抛出新的错误信息throw err Error('你好毒')，将会一起log输出
91             if (e !== err) {
92                 logError(e, null, 'config.errorHandler')
93             }
94         }
95     }
96     // 未配置常规log输出
97     logError(err, vm, info)
98 }
99
100
101 // 错误输出函数
102 function logError (err, vm, info) {
103     if (process.env.NODE_ENV !== 'production') {
104         warn(`Error in ${info}: "${err.toString()}"`, vm)
105     }
106     /* istanbul ignore else */
107     if ((inBrowser || inWeex) && typeof console !== 'undefined') {
108         console.error(err)
109     } else {
110         throw err
111     }
112 }

```

21.3. 小结

- `handleError` 在需要捕获异常的地方调用，首先获取到报错的组件，之后递归查找当前组件的父组件，依次调用 `errorCaptured` 方法，在遍历调用完所有 `errorCaptured` 方法或 `errorCaptured` 方法有报错时，调用 `globalHandleError` 方法
- `globalHandleError` 调用全局的 `errorHandler` 方法，再通过 `logError` 判断环境输出错误信息

- `invokeWithErrorHandling` 更好的处理异步错误信息
- `logError` 判断环境，选择不同的抛错方式。非生产环境下，调用 `warn` 方法处理错误

22. 你了解axios的原理吗？有看过它的源码吗？



22.1. axios的使用

关于 `axios` 的基本使用，上篇文章已经有所涉及，这里再稍微回顾下：

发送请求

```
1 import axios from 'axios';
2
3 axios(config) // 直接传入配置
4 axios(url[, config]) // 传入url和配置
5 axios[method](url[, option]) // 直接调用请求方式方法，传入url和配置
6 axios[method](url[, data[, option]]) // 直接调用请求方式方法，传入data、url和配置
7 axios.request(option) // 调用 request 方法
8
9 const axiosInstance = axios.create(config)
10 // axiosInstance 也具有以上 axios 的能力
11
12 axios.all([axiosInstance1, axiosInstance2]).then(axios.spread(response1, response2))
13 // 调用 all 和传入 spread 回调
```

请求拦截器

JavaScript | 复制代码

```

1 axios.interceptors.request.use(function (config) {
2     // 这里写发送请求前处理的代码
3     return config;
4 }, function (error) {
5     // 这里写发送请求错误相关的代码
6     return Promise.reject(error);
7 });

```

响应拦截器

JavaScript | 复制代码

```

1 axios.interceptors.response.use(function (response) {
2     // 这里写得到响应数据后处理的代码
3     return response;
4 }, function (error) {
5     // 这里写得到错误响应处理的代码
6     return Promise.reject(error);
7 });

```

取消请求

JavaScript | 复制代码

```

1 // 方式一
2 const CancelToken = axios.CancelToken;
3 const source = CancelToken.source();
4
5 axios.get('xxxx', {
6     cancelToken: source.token
7 })
8 // 取消请求（请求原因是可选的）
9 source.cancel('主动取消请求');
10
11 // 方式二
12 const CancelToken = axios.CancelToken;
13 let cancel;
14
15 axios.get('xxxx', {
16     cancelToken: new CancelToken(function executor(c) {
17         cancel = c;
18     })
19 });
20 cancel('主动取消请求');

```

22.2. 实现一个简易版axios

构建一个 `Axios` 构造函数，核心代码为 `request`

```

1  class Axios {
2    constructor() {
3    }
4
5    request(config) {
6      return new Promise(resolve => {
7        const {url = '', method = 'get', data = {}} = config;
8        // 发送ajax请求
9        const xhr = new XMLHttpRequest();
10       xhr.open(method, url, true);
11       xhr.onload = function() {
12         console.log(xhr.responseText)
13         resolve(xhr.responseText);
14       }
15       xhr.send(data);
16     })
17   }
18 }
19 }
```

导出 `axios` 实例

```

1 // 最终导出axios的方法，即实例的request方法
2 function CreateAxiosFn() {
3   let axios = new Axios();
4   let req = axios.request.bind(axios);
5   return req;
6 }
7
8 // 得到最后的全局变量axios
9 let axios = CreateAxiosFn();
```

上述就已经能够实现 `axios({ })` 这种方式的请求

下面是来实现下 `axios.method()` 这种形式的请求

```
1 // 定义get,post...方法, 挂在到Axios原型上
2 const methodsArr = ['get', 'delete', 'head', 'options', 'put', 'patch', 'post'];
3 methodsArr.forEach(met => {
4     Axios.prototype[met] = function() {
5         console.log('执行'+met+'方法');
6         // 处理单个方法
7         if(['get', 'delete', 'head', 'options'].includes(met)) { // 2个参
        数(url[, config])
8             return this.request({
9                 method: met,
10                url: arguments[0],
11                ...arguments[1] || {}
12            })
13        } else { // 3个参数(url[,data[,config]])
14            return this.request({
15                method: met,
16                url: arguments[0],
17                data: arguments[1] || {},
18                ...arguments[2] || {}
19            })
20        }
21    }
22 }
23 })
```

将 `Axios.prototype` 上的方法搬运到 `request` 上

首先实现个工具类, 实现将 `b` 方法混入到 `a`, 并且修改 `this` 指向

JavaScript | 复制代码

```
1 const utils = {
2   extend(a, b, context) {
3     for(let key in b) {
4       if (b.hasOwnProperty(key)) {
5         if (typeof b[key] === 'function') {
6           a[key] = b[key].bind(context);
7         } else {
8           a[key] = b[key];
9         }
10      }
11    }
12  }
13}
14}
```

修改导出的方法

JavaScript | 复制代码

```
1 function CreateAxiosFn() {
2   let axios = new Axios();
3
4   let req = axios.request.bind(axios);
5   // 增加代码
6   utils.extend(req, Axios.prototype, axios)
7
8   return req;
9 }
```

构建拦截器的构造函数

JavaScript | 复制代码

```

1  class InterceptorsManage {
2    constructor() {
3      this.handlers = [];
4    }
5
6    use(fullfield, rejected) {
7      this.handlers.push({
8        fullfield,
9        rejected
10     })
11   }
12 }

```

实现 `axios.interceptors.response.use` 和 `axios.interceptors.request.use`

JavaScript | 复制代码

```

1  class Axios {
2    constructor() {
3      // 新增代码
4      this.interceptors = {
5        request: new InterceptorsManage,
6        response: new InterceptorsManage
7      }
8    }
9
10   request(config) {
11     ...
12   }
13 }

```

执行语句 `axios.interceptors.response.use` 和 `axios.interceptors.request.use` 的时候，实现获取 `axios` 实例上的 `interceptors` 对象，然后再获取 `response` 或 `request` 拦截器，再执行对应的拦截器的 `use` 方法

把 `Axios` 上的方法和属性搬到 `request` 过去

JavaScript | 复制代码

```

1 function CreateAxiosFn() {
2     let axios = new Axios();
3
4     let req = axios.request.bind(axios);
5     // 混入方法， 处理axios的request方法，使之拥有get,post...方法
6     utils.extend(req, Axios.prototype, axios)
7     // 新增代码
8     utils.extend(req, axios)
9     return req;
10 }

```

现在 `request` 也有了 `interceptors` 对象，在发送请求的时候，会先获取 `request` 拦截器的 `handlers` 的方法来执行

首先将执行 `ajax` 的请求封装成一个方法

JavaScript | 复制代码

```

1 request(config) {
2     this.sendAjax(config)
3 }
4 sendAjax(config){
5     return new Promise(resolve => {
6         const {url = '', method = 'get', data = {}} = config;
7         // 发送ajax请求
8         console.log(config);
9         const xhr = new XMLHttpRequest();
10        xhr.open(method, url, true);
11        xhr.onload = function() {
12            console.log(xhr.responseText)
13            resolve(xhr.responseText);
14        };
15        xhr.send(data);
16    })
17 }

```

获得 `handlers` 中的回调

```

1  request(config) {
2      // 拦截器和请求组装队列
3      let chain = [this.sendAjax.bind(this), undefined] // 成对出现的，失败回调
        暂时不处理
4
5      // 请求拦截
6      this.interceptors.request.handlers.forEach(interceptor => {
7          chain.unshift(interceptor.fullfield, interceptor.rejected)
8      })
9
10     // 响应拦截
11     this.interceptors.response.handlers.forEach(interceptor => {
12         chain.push(interceptor.fullfield, interceptor.rejected)
13     })
14
15     // 执行队列，每次执行一对，并给promise赋最新的值
16     let promise = Promise.resolve(config);
17     while(chain.length > 0) {
18         promise = promise.then(chain.shift(), chain.shift())
19     }
20     return promise;
21 }

```

`chains` 大概是 `['fulfilled1','reject1','fulfilled2','reject2','this.sendAjax','undefined','fulfilled2','reject2','fulfilled1','reject1']` 这种形式

这样就能够成功实现一个简易版

22.3. 源码分析

首先看看目录结构

```
└── /lib/
    |   └── /adapters/          # 项目源码目
    |       └── http.js         # 定义发送请求的适配器
    |           └── xhr.js      # 浏览器环境XML对象
    |   └── /cancel/            # 定义取消功能
    |   └── /helpers/            # 一些辅助方法
    |   └── /core/               # 一些核心功能
    |       └── Axios.js        # axios实例构造函数
    |       └── createError.js   # 抛出错误
    |       └── dispatchRequest.js # 用来调用http请求适配器方法发送请求
    |       └── InterceptorManager.js # 拦截器管理器
    |       └── mergeConfig.js    # 合并参数
    |       └── settle.js         # 根据http响应状态, 改变Promise的状态
    |       └── transformData.js   # 改变数据格式
    |   └── axios.js             # 入口, 创建构造函数
    |   └── defaults.js          # 默认配置
    |   └── utils.js              # 公用工具
```

axios 发送请求有很多实现的方法，实现入口文件为 `axios.js`

```
1  function createInstance(defaultConfig) {
2      var context = new Axios(defaultConfig);
3
4      // instance指向了request方法, 且上下文指向context, 所以可以直接以 instance(option) 方式调用
5      // Axios.prototype.request 内对第一个参数的数据类型判断, 使我们能够以 instance(url, option) 方式调用
6      var instance = bind(Axios.prototype.request, context);
7
8      // 把Axios.prototype上的方法扩展到instance对象上,
9      // 并指定上下文为context, 这样执行Axios原型链上的方法时, this会指向context
10     utils.extend(instance, Axios.prototype, context);
11
12     // Copy context to instance
13     // 把context对象上的自身属性和方法扩展到instance上
14     // 注: 因为extend内部使用的forEach方法对对象做for in 遍历时, 只遍历对象本身的属性,
15     // 而不会遍历原型链上的属性
16     // 这样, instance 就有了 defaults、interceptors 属性。
17     utils.extend(instance, context);
18     return instance;
19 }
20
21 // Create the default instance to be exported 创建一个由默认配置生成的axios实例
22 var axios = createInstance(defaults);
23
24 // Factory for creating new instances 扩展axios.create工厂函数, 内部也是 createInstance
25 axios.create = function create(instanceConfig) {
26     return createInstance(mergeConfig(axios.defaults, instanceConfig));
27 };
28
29 // Expose all/spread
30 axios.all = function all(promises) {
31     return Promise.all(promises);
32 };
33
34 axios.spread = function spread(callback) {
35     return function wrap(arr) {
36         return callback.apply(null, arr);
37     };
38 }
39 module.exports = axios;
```

印客学院独家整理，盗版必究

主要核心是 `Axios.prototype.request`，各种请求方式的调用实现都是在 `request` 内部实现的，简单看下 `request` 的逻辑

JavaScript | 复制代码

```
1 Axios.prototype.request = function request(config) {
2     // Allow for axios('example/url'[, config]) a la fetch API
3     // 判断 config 参数是否是 字符串, 如果是则认为第一个参数是 URL, 第二个参数是真正的config
4     if (typeof config === 'string') {
5         config = arguments[1] || {};
6         // 把 url 放置到 config 对象中, 便于之后的 mergeConfig
7         config.url = arguments[0];
8     } else {
9         // 如果 config 参数是否是 字符串, 则整体都当做config
10        config = config || {};
11    }
12    // 合并默认配置和传入的配置
13    config = mergeConfig(this.defaults, config);
14    // 设置请求方法
15    config.method = config.method ? config.method.toLowerCase() : 'get';
16    /*
17        something... 此部分会在后续拦截器单独讲述
18    */
19 };
20
21 // 在 Axios 原型上挂载 'delete', 'get', 'head', 'options' 且不传参的请求方法,
22 // 实现内部也是 request
22 utils.forEach(['delete', 'get', 'head', 'options'], function forEachMethodNoData(method) {
23     Axios.prototype[method] = function(url, config) {
24         return this.request(utils.merge(config || {}, {
25             method: method,
26             url: url
27         }));
28     };
29 });
30
31 // 在 Axios 原型上挂载 'post', 'put', 'patch' 且传参的请求方法, 实现内部同样也是
32 // request
32 utils.forEach(['post', 'put', 'patch'], function forEachMethodWithData(method) {
33     Axios.prototype[method] = function(url, data, config) {
34         return this.request(utils.merge(config || {}, {
35             method: method,
36             url: url,
37             data: data
38         )));
39     };
40 });


```

`request` 入口参数为 `config`，可以说 `config` 贯彻了 `axios` 的一生

`axios` 中的 `config` 主要分布在这几个地方：

- 默认配置 `defaults.js`
- `config.method` 默认为 `get`
- 调用 `createInstance` 方法创建 `axios` 实例，传入的 `config`
- 直接或间接调用 `request` 方法，传入的 `config`

```
▼ JavaScript | 复制代码

1 // axios.js
2 // 创建一个由默认配置生成的axios实例
3 var axios = createInstance(defaults);
4
5 // 扩展axios.create工厂函数，内部也是 createInstance
6 - axios.create = function create(instanceConfig) {
7     return createInstance(mergeConfig(axios.defaults, instanceConfig));
8 }
9
10 // Axios.js
11 // 合并默认配置和传入的配置
12 config = mergeConfig(this.defaults, config);
13 // 设置请求方法
14 config.method = config.method ? config.method.toLowerCase() : 'get';
```

从源码中，可以看到优先级：默认配置对象 `default` < `method:get` < `Axios` 的实例属性 `this.default` < `request` 参数

下面重点看看 `request` 方法

JavaScript | 复制代码

```
1 Axios.prototype.request = function request(config) {
2  /*
3   先是 mergeConfig ... 等, 不再阐述
4  */
5  // Hook up interceptors middleware 创建拦截器链. dispatchRequest 是重中之重, 后续重点
6  var chain = [dispatchRequest, undefined];
7
8  // push各个拦截器方法 注意: interceptor.fulfilled 或 interceptor.rejected 是可能为undefined
9  this.interceptors.request.forEach(function unshiftRequestInterceptors(interceptor) {
10    // 请求拦截器逆序 注意此处的 forEach 是自定义的拦截器的forEach方法
11    chain.unshift(interceptor.fulfilled, interceptor.rejected);
12  });
13
14  this.interceptors.response.forEach(function pushResponseInterceptors(interceptor) {
15    // 响应拦截器顺序 注意此处的 forEach 是自定义的拦截器的forEach方法
16    chain.push(interceptor.fulfilled, interceptor.rejected);
17  });
18
19  // 初始化一个promise对象, 状态为resolved, 接收到的参数为已经处理合并过的config对象
20  var promise = Promise.resolve(config);
21
22  // 循环拦截器的链
23  while (chain.length) {
24    promise = promise.then(chain.shift(), chain.shift()); // 每一次向外弹出拦截器
25  }
26  // 返回 promise
27  return promise;
28};
```

拦截器 `interceptors` 是在构建 `axios` 实例化的属性

JavaScript | 复制代码

```

1 function Axios(instanceConfig) {
2     this.defaults = instanceConfig;
3     this.interceptors = {
4         request: new InterceptorManager(), // 请求拦截
5         response: new InterceptorManager() // 响应拦截
6     };
7 }

```

InterceptorManager 构造函数

JavaScript | 复制代码

```

1 // 拦截器的初始化 其实就是一组钩子函数
2 function InterceptorManager() {
3     this.handlers = [];
4 }
5
6 // 调用拦截器实例的use时就是往钩子函数中push方法
7 InterceptorManager.prototype.use = function use(fulfilled, rejected) {
8     this.handlers.push({
9         fulfilled: fulfilled,
10        rejected: rejected
11    });
12    return this.handlers.length - 1;
13 };
14
15 // 拦截器是可以取消的，根据use的时候返回的ID，把某一个拦截器方法置为null
16 // 不能用 splice 或者 slice 的原因是 删除之后 id 就会变化，导致之后的顺序或者是操作
17 // 不可控
18 InterceptorManager.prototype.eject = function eject(id) {
19     if (this.handlers[id]) {
20         this.handlers[id] = null;
21     }
22 };
23
24 // 这就是在 Axios的request方法中 中循环拦截器的方法 forEach 循环执行钩子函数
25 InterceptorManager.prototype.forEach = function forEach(fn) {
26     utils.forEach(this.handlers, function forEachHandler(h) {
27         if (h !== null) {
28             fn(h);
29         }
30     });

```

印客学院独家整理，盗版必究

请求拦截器方法是被 `unshift` 到拦截器中，响应拦截器是被 `push` 到拦截器中的。最终它们会拼接上一个叫 `dispatchRequest` 的方法被后续的 `promise` 顺序执行

JavaScript | 复制代码

```
1 var utils = require('../utils');
2 var transformData = require('./transformData');
3 var isCancel = require('../cancel/isCancel');
4 var defaults = require('../defaults');
5 var isAbsoluteURL = require('../helpers/isAbsoluteURL');
6 var combineURLs = require('../helpers/combineURLs');
7
8 // 判断请求是否已被取消，如果已经被取消，抛出已取消
9 function throwIfCancellationRequested(config) {
10 if (config.cancelToken) {
11     config.cancelToken.throwIfRequested();
12 }
13 }
14
15 module.exports = function dispatchRequest(config) {
16     throwIfCancellationRequested(config);
17
18     // 如果包含baseUrl，并且不是config.url绝对路径，组合baseUrl以及config.url
19     if (config.baseUrl && !isAbsoluteURL(config.url)) {
20         // 组合baseUrl与url形成完整的请求路径
21         config.url = combineURLs(config.baseUrl, config.url);
22     }
23
24     config.headers = config.headers || {};
25
26     // 使用/lib/defaults.js中的transformRequest方法，对config.headers和config.data进行格式化
27     // 比如将headers中的Accept, Content-Type统一处理成大写
28     // 比如如果请求正文是一个Object会格式化为JSON字符串，并添加application/json;charset=utf-8的Content-Type
29     // 等一系列操作
30     config.data = transformData(
31         config.data,
32         config.headers,
33         config.transformRequest
34     );
35
36     // 合并不同配置的headers，config.headers的配置优先级更高
37     config.headers = utils.merge(
38         config.headers.common || {},
39         config.headers[config.method] || {},
40         config.headers || {}
41     );
42
43     // 删除headers中的method属性
```

```

44     utils.forEach(
45       ['delete', 'get', 'head', 'post', 'put', 'patch', 'common'],
46       function cleanHeaderConfig(method) {
47         delete config.headers[method];
48       }
49     );
50
51     // 如果config配置了adapter, 使用config中配置adapter的替代默认的请求方法
52     var adapter = config.adapter || defaults.adapter;
53
54     // 使用adapter方法发起请求 (adapter根据浏览器环境或者Node环境会有不同)
55     return adapter(config).then(
56       // 请求正确返回的回调
57       function onAdapterResolution(response) {
58         // 判断是否以及取消了请求, 如果取消了请求抛出以取消
59         throwIfCancellationRequested(config);
60
61         // 使用/lib/defaults.js中的transformResponse方法, 对服务器返回的数据进行格
62         // 式化
63         // 例如, 使用JSON.parse对响应正文进行解析
64         response.data = transformData(
65           response.data,
66           response.headers,
67           config.transformResponse
68         );
69
70         return response;
71       },
72       // 请求失败的回调
73       function onAdapterRejection(reason) {
74         if (!isCancel(reason)) {
75           throwIfCancellationRequested(config);
76
77           if (reason && reason.response) {
78             reason.response.data = transformData(
79               reason.response.data,
80               reason.response.headers,
81               config.transformResponse
82             );
83           }
84         }
85       }
86     );
87   };

```

再来看看 `axios` 是如何实现取消请求的, 实现文件在 `CancelToken.js`

```

1  function CancelToken(executor) {
2    if (typeof executor !== 'function') {
3      throw new TypeError('executor must be a function.');
4    }
5    // 在 CancelToken 上定义一个 pending 状态的 promise , 将 resolve 回调赋值给外部变量 resolvePromise
6    var resolvePromise;
7    this.promise = new Promise(function promiseExecutor(resolve) {
8      resolvePromise = resolve;
9    });
10
11   var token = this;
12   // 立即执行 传入的 executor函数, 将真实的 cancel 方法通过参数传递出去。
13   // 一旦调用就执行 resolvePromise 即前面的 promise 的 resolve, 就更改promise的状态为 resolve。
14   // 那么xhr中定义的 CancelToken.promise.then方法就会执行, 从而xhr内部会取消请求
15   executor(function cancel(message) {
16     // 判断请求是否已经取消过, 避免多次执行
17     if (token.reason) {
18       return;
19     }
20     token.reason = new Cancel(message);
21     resolvePromise(token.reason);
22   });
23 }
24
25 CancelToken.source = function source() {
26   // source 方法就是返回了一个 CancelToken 实例, 与直接使用 new CancelToken 是一样的操作
27   var cancel;
28   var token = new CancelToken(function executor(c) {
29     cancel = c;
30   });
31   // 返回创建的 CancelToken 实例以及取消方法
32   return {
33     token: token,
34     cancel: cancel
35   };
36 };

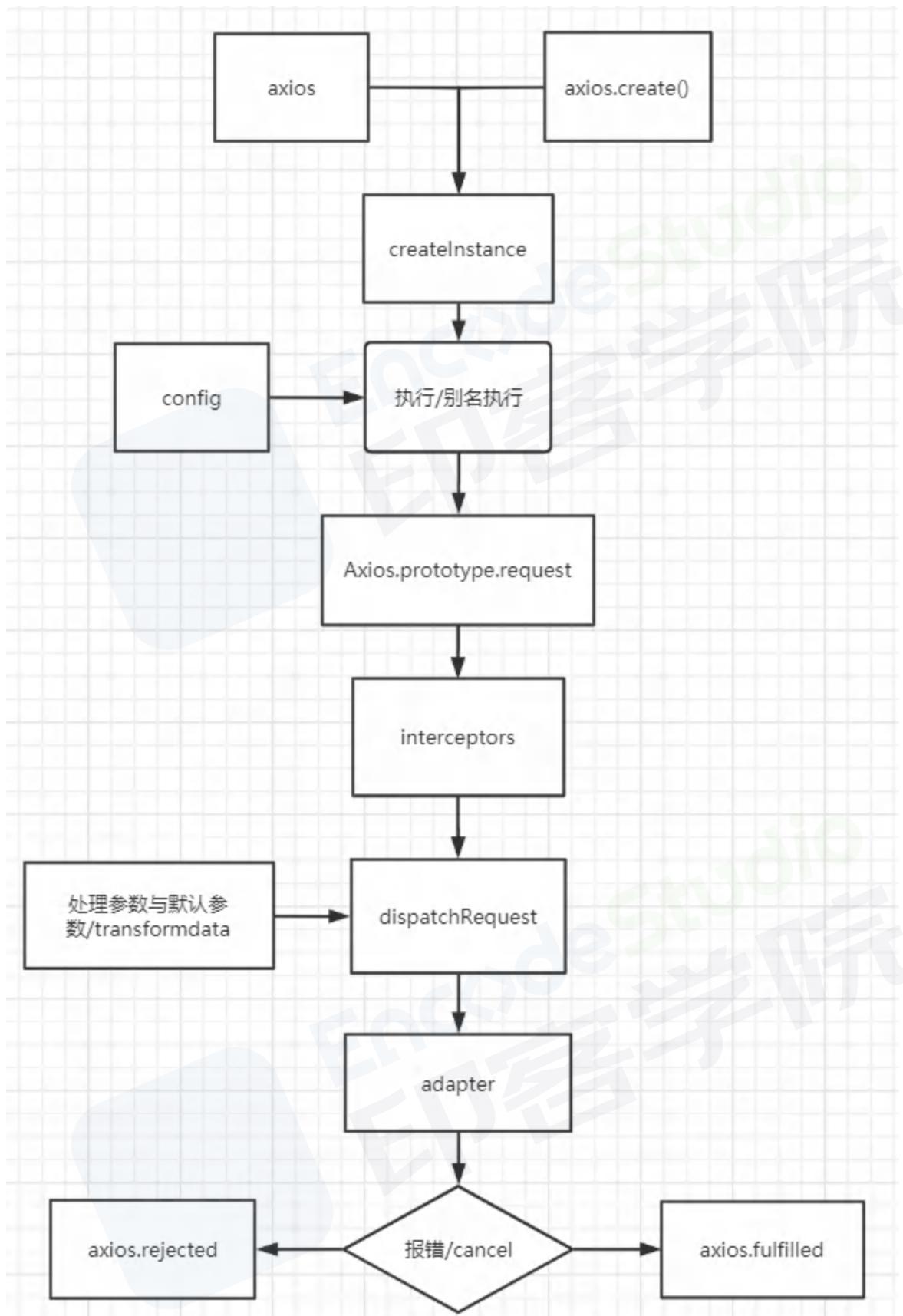
```

实际上取消请求的操作是在 `xhr.js` 中也有响应的配合的

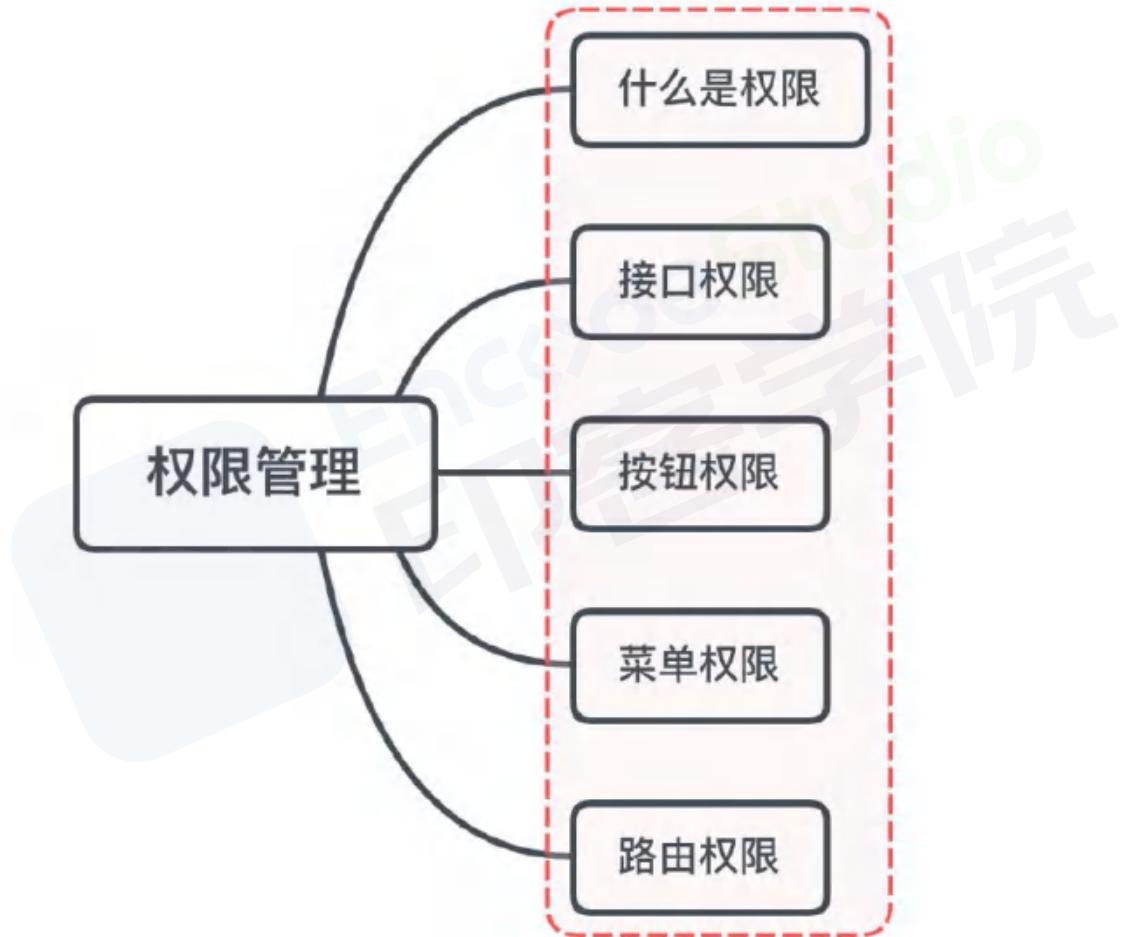
```
1 if (config.cancelToken) {
2   config.cancelToken.promise.then(function onCanceled(cancel) {
3     if (!request) {
4       return;
5     }
6     // 取消请求
7     request.abort();
8     reject(cancel);
9   });
10 }
```

巧妙的地方在 `CancelToken` 中 `executor` 函数，通过 `resolve` 函数的传递与执行，控制 `promise` 的状态

22.4. 小结



23. vue要做权限管理该怎么做？



23.1. 是什么

权限是对特定资源的访问许可，所谓权限控制，也就是确保用户只能访问到被分配的资源

而前端权限归根结底是请求的发起权，请求的发起可能有下面两种形式触发

- 页面加载触发
- 页面上的按钮点击触发

总的来说，所有的请求发起都触发自前端路由或视图

所以我们可以从这两方面入手，对触发权限的源头进行控制，最终要实现的目标是：

- 路由方面，用户登录后只能看到自己有权访问的导航菜单，也只能访问自己有权访问的路由地址，否则将跳转 `4xx` 提示页
- 视图方面，用户只能看到自己有权浏览的内容和有权操作的控件
- 最后再加上请求控制作为最后一道防线，路由可能配置失误，按钮可能忘了加权限，这种时候请求

控制可以用来兜底，越权请求将在前端被拦截

23.2. 如何做

前端权限控制可以分为四个方面：

- 接口权限
- 按钮权限
- 菜单权限
- 路由权限

23.2.1. 接口权限

接口权限目前一般采用 `jwt` 的形式来验证，没有通过的话一般返回 `401`，跳转到登录页面重新进行登录

登录完拿到 `token`，将 `token` 存起来，通过 `axios` 请求拦截器进行拦截，每次请求的时候头部携带 `token`

```
1 axios.interceptors.request.use(config => {
2     config.headers['token'] = cookie.get('token')
3     return config
4 }
5 axios.interceptors.response.use(res=>{}, {response}=>{
6     if (response.data.code === 40099 || response.data.code === 40098) { //token过期或者错误
7         router.push('/login')
8     }
9 })
```

23.2.2. 路由权限控制

方案一

初始化即挂载全部路由，并且在路由上标记相应的权限信息，每次路由跳转前做校验

```

1  const routerMap = [
2    {
3      path: '/permission',
4      component: Layout,
5      redirect: '/permission/index',
6      alwaysShow: true, // will always show the root menu
7      meta: {
8        title: 'permission',
9        icon: 'lock',
10       roles: ['admin', 'editor'] // you can set roles in root nav
11     },
12    children: [{ 
13      path: 'page',
14      component: () => import('@/views/permission/page'),
15      name: 'pagePermission',
16      meta: {
17        title: 'pagePermission',
18        roles: ['admin'] // or you can only set roles in sub nav
19      }
20    }, { 
21      path: 'directive',
22      component: () => import('@/views/permission/directive'),
23      name: 'directivePermission',
24      meta: {
25        title: 'directivePermission'
26        // if do not set roles, means: this page does not require permission
27        on
28      }
29    }]
30  ]

```

这种方式存在以下四种缺点：

- 加载所有的路由，如果路由很多，而用户并不是所有的路由都有权限访问，对性能会有影响。
- 全局路由守卫里，每次路由跳转都要做权限判断。
- 菜单信息写死在前端，要改个显示文字或权限信息，需要重新编译
- 菜单跟路由耦合在一起，定义路由的时候还有添加菜单显示标题，图标之类的信息，而且路由不一定作为菜单显示，还要多加字段进行标识

方案二

初始化的时候先挂载不需要权限控制的路由，比如登录页，404等错误页。如果用户通过URL进行强制访问，则会直接进入404，相当于从源头上做了控制

印客学院独家整理，盗版必究

登录后，获取用户的权限信息，然后筛选有权限访问的路由，在全局路由守卫里进行调用

`addRoutes` 添加路由

```
1 import router from './router'
2 import store from './store'
3 import { Message } from 'element-ui'
4 import NProgress from 'nprogress' // progress bar
5 import 'nprogress/nprogress.css'// progress bar style
6 import { getToken } from '@/utils/auth' // getToken from cookie
7
8 NProgress.configure({ showSpinner: false })// NProgress Configuration
9
10 // permission judge function
11 - function hasPermission(roles, permissionRoles) {
12     if (roles.indexOf('admin') >= 0) return true // admin permission passed directly
13     if (!permissionRoles) return true
14     return roles.some(role => permissionRoles.indexOf(role) >= 0)
15 }
16
17 const whiteList = ['/login', '/authredirect']// no redirect whitelist
18
19 - router.beforeEach((to, from, next) => {
20     NProgress.start() // start progress bar
21     if (getToken()) { // determine if there has token
22         /* has token*/
23         if (to.path === '/login') {
24             next({ path: '/' })
25             NProgress.done() // if current page is dashboard will not trigger afterEach hook, so manually handle it
26         } else {
27             if (store.getters.roles.length === 0) { // 判断当前用户是否已拉取完user_info信息
28                 store.dispatch('GetUserInfo').then(res => { // 拉取user_info
29                     const roles = res.data.roles // note: roles must be a array! such as: ['editor','develop']
30                     store.dispatch('GenerateRoutes', { roles }).then(() => { // 根据roles权限生成可访问的路由表
31                         router.addRoutes(store.getters.addRouters) // 动态添加可访问路由表
32                         next({ ...to, replace: true }) // hack方法 确保addRoutes已完成，set the replace: true so the navigation will not leave a history record
33                     })
34                 }).catch((err) => {
35                     store.dispatch('FedLogOut').then(() => {
36                         Message.error(err || 'Verification failed, please login again')
37                     })
38                     next({ path: '/' })
39                 })
40             }
41         }
42     }
43 }
44
45 // catch error
46 - router.onError(error => {
47     if (error.message === 'Navigation failed: the previous route ended with success, but its success callback never fired') {
48         router.push('/login')
49     }
50 }
51 )
52
53 // 4. 路由懒加载
54 // 引入懒加载
55 - import('./views/Home') // 引入懒加载
56
57 // 定义路由
58 - const routes = [
59     {
60         path: '/',
61         component: Home // 引入懒加载
62     }
63 ]
64
65 // 定义路由
66 - const routes = [
67     {
68         path: '/',
69         component: Home // 引入懒加载
70     }
71 ]
72
73 // 定义路由
74 - const routes = [
75     {
76         path: '/',
77         component: Home // 引入懒加载
78     }
79 ]
80
81 // 定义路由
82 - const routes = [
83     {
84         path: '/',
85         component: Home // 引入懒加载
86     }
87 ]
88
89 // 定义路由
90 - const routes = [
91     {
92         path: '/',
93         component: Home // 引入懒加载
94     }
95 ]
96
97 // 定义路由
98 - const routes = [
99     {
100        path: '/',
101        component: Home // 引入懒加载
102    }
103 ]
104
105 // 定义路由
106 - const routes = [
107     {
108         path: '/',
109         component: Home // 引入懒加载
110     }
111 ]
112
113 // 定义路由
114 - const routes = [
115     {
116         path: '/',
117         component: Home // 引入懒加载
118     }
119 ]
120
121 // 定义路由
122 - const routes = [
123     {
124         path: '/',
125         component: Home // 引入懒加载
126     }
127 ]
128
129 // 定义路由
130 - const routes = [
131     {
132         path: '/',
133         component: Home // 引入懒加载
134     }
135 ]
136
137 // 定义路由
138 - const routes = [
139     {
140         path: '/',
141         component: Home // 引入懒加载
142     }
143 ]
144
145 // 定义路由
146 - const routes = [
147     {
148         path: '/',
149         component: Home // 引入懒加载
150     }
151 ]
152
153 // 定义路由
154 - const routes = [
155     {
156         path: '/',
157         component: Home // 引入懒加载
158     }
159 ]
160
161 // 定义路由
162 - const routes = [
163     {
164         path: '/',
165         component: Home // 引入懒加载
166     }
167 ]
168
169 // 定义路由
170 - const routes = [
171     {
172         path: '/',
173         component: Home // 引入懒加载
174     }
175 ]
176
177 // 定义路由
178 - const routes = [
179     {
180         path: '/',
181         component: Home // 引入懒加载
182     }
183 ]
184
185 // 定义路由
186 - const routes = [
187     {
188         path: '/',
189         component: Home // 引入懒加载
190     }
191 ]
192
193 // 定义路由
194 - const routes = [
195     {
196         path: '/',
197         component: Home // 引入懒加载
198     }
199 ]
200
201 // 定义路由
202 - const routes = [
203     {
204         path: '/',
205         component: Home // 引入懒加载
206     }
207 ]
208
209 // 定义路由
210 - const routes = [
211     {
212         path: '/',
213         component: Home // 引入懒加载
214     }
215 ]
216
217 // 定义路由
218 - const routes = [
219     {
220         path: '/',
221         component: Home // 引入懒加载
222     }
223 ]
224
225 // 定义路由
226 - const routes = [
227     {
228         path: '/',
229         component: Home // 引入懒加载
230     }
231 ]
232
233 // 定义路由
234 - const routes = [
235     {
236         path: '/',
237         component: Home // 引入懒加载
238     }
239 ]
240
241 // 定义路由
242 - const routes = [
243     {
244         path: '/',
245         component: Home // 引入懒加载
246     }
247 ]
248
249 // 定义路由
250 - const routes = [
251     {
252         path: '/',
253         component: Home // 引入懒加载
254     }
255 ]
256
257 // 定义路由
258 - const routes = [
259     {
260         path: '/',
261         component: Home // 引入懒加载
262     }
263 ]
264
265 // 定义路由
266 - const routes = [
267     {
268         path: '/',
269         component: Home // 引入懒加载
270     }
271 ]
272
273 // 定义路由
274 - const routes = [
275     {
276         path: '/',
277         component: Home // 引入懒加载
278     }
279 ]
280
281 // 定义路由
282 - const routes = [
283     {
284         path: '/',
285         component: Home // 引入懒加载
286     }
287 ]
288
289 // 定义路由
290 - const routes = [
291     {
292         path: '/',
293         component: Home // 引入懒加载
294     }
295 ]
296
297 // 定义路由
298 - const routes = [
299     {
300         path: '/',
301         component: Home // 引入懒加载
302     }
303 ]
304
305 // 定义路由
306 - const routes = [
307     {
308         path: '/',
309         component: Home // 引入懒加载
310     }
311 ]
312
313 // 定义路由
314 - const routes = [
315     {
316         path: '/',
317         component: Home // 引入懒加载
318     }
319 ]
320
321 // 定义路由
322 - const routes = [
323     {
324         path: '/',
325         component: Home // 引入懒加载
326     }
327 ]
328
329 // 定义路由
330 - const routes = [
331     {
332         path: '/',
333         component: Home // 引入懒加载
334     }
335 ]
336
337 // 定义路由
338 - const routes = [
339     {
340         path: '/',
341         component: Home // 引入懒加载
342     }
343 ]
344
345 // 定义路由
346 - const routes = [
347     {
348         path: '/',
349         component: Home // 引入懒加载
350     }
351 ]
352
353 // 定义路由
354 - const routes = [
355     {
356         path: '/',
357         component: Home // 引入懒加载
358     }
359 ]
360
361 // 定义路由
362 - const routes = [
363     {
364         path: '/',
365         component: Home // 引入懒加载
366     }
367 ]
368
369 // 定义路由
370 - const routes = [
371     {
372         path: '/',
373         component: Home // 引入懒加载
374     }
375 ]
376
377 // 定义路由
378 - const routes = [
379     {
380         path: '/',
381         component: Home // 引入懒加载
382     }
383 ]
384
385 // 定义路由
386 - const routes = [
387     {
388         path: '/',
389         component: Home // 引入懒加载
390     }
391 ]
392
393 // 定义路由
394 - const routes = [
395     {
396         path: '/',
397         component: Home // 引入懒加载
398     }
399 ]
400
401 // 定义路由
402 - const routes = [
403     {
404         path: '/',
405         component: Home // 引入懒加载
406     }
407 ]
408
409 // 定义路由
410 - const routes = [
411     {
412         path: '/',
413         component: Home // 引入懒加载
414     }
415 ]
416
417 // 定义路由
418 - const routes = [
419     {
420         path: '/',
421         component: Home // 引入懒加载
422     }
423 ]
424
425 // 定义路由
426 - const routes = [
427     {
428         path: '/',
429         component: Home // 引入懒加载
430     }
431 ]
432
433 // 定义路由
434 - const routes = [
435     {
436         path: '/',
437         component: Home // 引入懒加载
438     }
439 ]
440
441 // 定义路由
442 - const routes = [
443     {
444         path: '/',
445         component: Home // 引入懒加载
446     }
447 ]
448
449 // 定义路由
450 - const routes = [
451     {
452         path: '/',
453         component: Home // 引入懒加载
454     }
455 ]
456
457 // 定义路由
458 - const routes = [
459     {
460         path: '/',
461         component: Home // 引入懒加载
462     }
463 ]
464
465 // 定义路由
466 - const routes = [
467     {
468         path: '/',
469         component: Home // 引入懒加载
470     }
471 ]
472
473 // 定义路由
474 - const routes = [
475     {
476         path: '/',
477         component: Home // 引入懒加载
478     }
479 ]
480
481 // 定义路由
482 - const routes = [
483     {
484         path: '/',
485         component: Home // 引入懒加载
486     }
487 ]
488
489 // 定义路由
490 - const routes = [
491     {
492         path: '/',
493         component: Home // 引入懒加载
494     }
495 ]
496
497 // 定义路由
498 - const routes = [
499     {
500         path: '/',
501         component: Home // 引入懒加载
502     }
503 ]
504
505 // 定义路由
506 - const routes = [
507     {
508         path: '/',
509         component: Home // 引入懒加载
510     }
511 ]
512
513 // 定义路由
514 - const routes = [
515     {
516         path: '/',
517         component: Home // 引入懒加载
518     }
519 ]
520
521 // 定义路由
522 - const routes = [
523     {
524         path: '/',
525         component: Home // 引入懒加载
526     }
527 ]
528
529 // 定义路由
530 - const routes = [
531     {
532         path: '/',
533         component: Home // 引入懒加载
534     }
535 ]
536
537 // 定义路由
538 - const routes = [
539     {
540         path: '/',
541         component: Home // 引入懒加载
542     }
543 ]
544
545 // 定义路由
546 - const routes = [
547     {
548         path: '/',
549         component: Home // 引入懒加载
550     }
551 ]
552
553 // 定义路由
554 - const routes = [
555     {
556         path: '/',
557         component: Home // 引入懒加载
558     }
559 ]
560
561 // 定义路由
562 - const routes = [
563     {
564         path: '/',
565         component: Home // 引入懒加载
566     }
567 ]
568
569 // 定义路由
570 - const routes = [
571     {
572         path: '/',
573         component: Home // 引入懒加载
574     }
575 ]
576
577 // 定义路由
578 - const routes = [
579     {
580         path: '/',
581         component: Home // 引入懒加载
582     }
583 ]
584
585 // 定义路由
586 - const routes = [
587     {
588         path: '/',
589         component: Home // 引入懒加载
590     }
591 ]
592
593 // 定义路由
594 - const routes = [
595     {
596         path: '/',
597         component: Home // 引入懒加载
598     }
599 ]
599
600 // 定义路由
601 - const routes = [
602     {
603         path: '/',
604         component: Home // 引入懒加载
605     }
606 ]
606
607 // 定义路由
608 - const routes = [
609     {
610         path: '/',
611         component: Home // 引入懒加载
612     }
613 ]
613
614 // 定义路由
615 - const routes = [
616     {
617         path: '/',
618         component: Home // 引入懒加载
619     }
620 ]
620
621 // 定义路由
622 - const routes = [
623     {
624         path: '/',
625         component: Home // 引入懒加载
626     }
627 ]
627
628 // 定义路由
629 - const routes = [
630     {
631         path: '/',
632         component: Home // 引入懒加载
633     }
634 ]
634
635 // 定义路由
636 - const routes = [
637     {
638         path: '/',
639         component: Home // 引入懒加载
640     }
641 ]
641
642 // 定义路由
643 - const routes = [
644     {
645         path: '/',
646         component: Home // 引入懒加载
647     }
648 ]
648
649 // 定义路由
650 - const routes = [
651     {
652         path: '/',
653         component: Home // 引入懒加载
654     }
655 ]
655
656 // 定义路由
657 - const routes = [
658     {
659         path: '/',
660         component: Home // 引入懒加载
661     }
662 ]
662
663 // 定义路由
664 - const routes = [
665     {
666         path: '/',
667         component: Home // 引入懒加载
668     }
669 ]
669
670 // 定义路由
671 - const routes = [
672     {
673         path: '/',
674         component: Home // 引入懒加载
675     }
676 ]
676
677 // 定义路由
678 - const routes = [
679     {
680         path: '/',
681         component: Home // 引入懒加载
682     }
683 ]
683
684 // 定义路由
685 - const routes = [
686     {
687         path: '/',
688         component: Home // 引入懒加载
689     }
690 ]
690
691 // 定义路由
692 - const routes = [
693     {
694         path: '/',
695         component: Home // 引入懒加载
696     }
697 ]
697
698 // 定义路由
699 - const routes = [
700     {
701         path: '/',
702         component: Home // 引入懒加载
703     }
704 ]
704
705 // 定义路由
706 - const routes = [
707     {
708         path: '/',
709         component: Home // 引入懒加载
710     }
711 ]
711
712 // 定义路由
713 - const routes = [
714     {
715         path: '/',
716         component: Home // 引入懒加载
717     }
718 ]
718
719 // 定义路由
720 - const routes = [
721     {
722         path: '/',
723         component: Home // 引入懒加载
724     }
725 ]
725
726 // 定义路由
727 - const routes = [
728     {
729         path: '/',
730         component: Home // 引入懒加载
731     }
732 ]
732
733 // 定义路由
734 - const routes = [
735     {
736         path: '/',
737         component: Home // 引入懒加载
738     }
739 ]
739
740 // 定义路由
741 - const routes = [
742     {
743         path: '/',
744         component: Home // 引入懒加载
745     }
746 ]
746
747 // 定义路由
748 - const routes = [
749     {
750         path: '/',
751         component: Home // 引入懒加载
752     }
753 ]
753
754 // 定义路由
755 - const routes = [
756     {
757         path: '/',
758         component: Home // 引入懒加载
759     }
760 ]
760
761 // 定义路由
762 - const routes = [
763     {
764         path: '/',
765         component: Home // 引入懒加载
766     }
767 ]
767
768 // 定义路由
769 - const routes = [
770     {
771         path: '/',
772         component: Home // 引入懒加载
773     }
774 ]
774
775 // 定义路由
776 - const routes = [
777     {
778         path: '/',
779         component: Home // 引入懒加载
780     }
781 ]
781
782 // 定义路由
783 - const routes = [
784     {
785         path: '/',
786         component: Home // 引入懒加载
787     }
788 ]
788
789 // 定义路由
790 - const routes = [
791     {
792         path: '/',
793         component: Home // 引入懒加载
794     }
795 ]
795
796 // 定义路由
797 - const routes = [
798     {
799         path: '/',
800         component: Home // 引入懒加载
801     }
802 ]
802
803 // 定义路由
804 - const routes = [
805     {
806         path: '/',
807         component: Home // 引入懒加载
808     }
809 ]
809
810 // 定义路由
811 - const routes = [
812     {
813         path: '/',
814         component: Home // 引入懒加载
815     }
816 ]
816
817 // 定义路由
818 - const routes = [
819     {
820         path: '/',
821         component: Home // 引入懒加载
822     }
823 ]
823
824 // 定义路由
825 - const routes = [
826     {
827         path: '/',
828         component: Home // 引入懒加载
829     }
830 ]
830
831 // 定义路由
832 - const routes = [
833     {
834         path: '/',
835         component: Home // 引入懒加载
836     }
837 ]
837
838 // 定义路由
839 - const routes = [
840     {
841         path: '/',
842         component: Home // 引入懒加载
843     }
844 ]
844
845 // 定义路由
846 - const routes = [
847     {
848         path: '/',
849         component: Home // 引入懒加载
850     }
851 ]
851
852 // 定义路由
853 - const routes = [
854     {
855         path: '/',
856         component: Home // 引入懒加载
857     }
858 ]
858
859 // 定义路由
860 - const routes = [
861     {
862         path: '/',
863         component: Home // 引入懒加载
864     }
865 ]
865
866 // 定义路由
867 - const routes = [
868     {
869         path: '/',
870         component: Home // 引入懒加载
871     }
872 ]
872
873 // 定义路由
874 - const routes = [
875     {
876         path: '/',
877         component: Home // 引入懒加载
878     }
879 ]
879
880 // 定义路由
881 - const routes = [
882     {
883         path: '/',
884         component: Home // 引入懒加载
885     }
886 ]
886
887 // 定义路由
888 - const routes = [
889     {
890         path: '/',
891         component: Home // 引入懒加载
892     }
893 ]
893
894 // 定义路由
895 - const routes = [
896     {
897         path: '/',
898         component: Home // 引入懒加载
899     }
900 ]
900
901 // 定义路由
902 - const routes = [
903     {
904         path: '/',
905         component: Home // 引入懒加载
906     }
907 ]
907
908 // 定义路由
909 - const routes = [
910     {
911         path: '/',
912         component: Home // 引入懒加载
913     }
914 ]
914
915 // 定义路由
916 - const routes = [
917     {
918         path: '/',
919         component: Home // 引入懒加载
920     }
921 ]
921
922 // 定义路由
923 - const routes = [
924     {
925         path: '/',
926         component: Home // 引入懒加载
927     }
928 ]
928
929 // 定义路由
930 - const routes = [
931     {
932         path: '/',
933         component: Home // 引入懒加载
934     }
935 ]
935
936 // 定义路由
937 - const routes = [
938     {
939         path: '/',
940         component: Home // 引入懒加载
941     }
942 ]
942
943 // 定义路由
944 - const routes = [
945     {
946         path: '/',
947         component: Home // 引入懒加载
948     }
949 ]
949
950 // 定义路由
951 - const routes = [
952     {
953         path: '/',
954         component: Home // 引入懒加载
955     }
956 ]
956
957 // 定义路由
958 - const routes = [
959     {
960         path: '/',
961         component: Home // 引入懒加载
962     }
963 ]
963
964 // 定义路由
965 - const routes = [
966     {
967         path: '/',
968         component: Home // 引入懒加载
969     }
970 ]
970
971 // 定义路由
972 - const routes = [
973     {
974         path: '/',
975         component: Home // 引入懒加载
976     }
977 ]
977
978 // 定义路由
979 - const routes = [
980     {
981         path: '/',
982         component: Home // 引入懒加载
983     }
984 ]
984
985 // 定义路由
986 - const routes = [
987     {
988         path: '/',
989         component: Home // 引入懒加载
990     }
991 ]
991
992 // 定义路由
993 - const routes = [
994     {
995         path: '/',
996         component: Home // 引入懒加载
997     }
998 ]
998
999 // 定义路由
1000 - const routes = [
1001     {
1002         path: '/',
1003         component: Home // 引入懒加载
1004     }
1005 ]
1005
1006 // 定义路由
1007 - const routes = [
1008     {
1009         path: '/',
1010         component: Home // 引入懒加载
1011     }
1012 ]
1012
1013 // 定义路由
1014 - const routes = [
1015     {
1016         path: '/',
1017         component: Home // 引入懒加载
1018     }
1019 ]
1019
1020 // 定义路由
1021 - const routes = [
1022     {
1023         path: '/',
1024         component: Home // 引入懒加载
1025     }
1026 ]
1026
1027 // 定义路由
1028 - const routes = [
1029     {
1030         path: '/',
1031         component: Home // 引入懒加载
1032     }
1033 ]
1033
1034 // 定义路由
1035 - const routes = [
1036     {
1037         path: '/',
1038         component: Home // 引入懒加载
1039     }
1040 ]
1040
1041 // 定义路由
1042 - const routes = [
1043     {
1044         path: '/',
1045         component: Home // 引入懒加载
1046     }
1047 ]
1047
1048 // 定义路由
1049 - const routes = [
1050     {
1051         path: '/',
1052         component: Home // 引入懒加载
1053     }
1054 ]
1054
1055 // 定义路由
1056 - const routes = [
1057     {
1058         path: '/',
1059         component: Home // 引入懒加载
1060     }
1061 ]
1061
1062 // 定义路由
1063 - const routes = [
1064     {
1065         path: '/',
1066         component: Home // 引入懒加载
1067     }
1068 ]
1068
1069 // 定义路由
1070 - const routes = [
1071     {
1072         path: '/',
1073         component: Home // 引入懒加载
1074     }
1075 ]
1075
1076 // 定义路由
1077 - const routes = [
1078     {
1079         path: '/',
1080         component: Home // 引入懒加载
1081     }
1082 ]
1082
1083 // 定义路由
1084 - const routes = [
1085     {
1086         path: '/',
1087         component: Home // 引入懒加载
1088     }
1089 ]
1089
1090 // 定义路由
1091 - const routes = [
1092     {
1093         path: '/',
1094         component: Home // 引入懒加载
1095     }
1096 ]
1096
1097 // 定义路由
1098 - const routes = [
1099     {
1100         path: '/',
1101         component: Home // 引入懒加载
1102     }
1103 ]
1103
1104 // 定义路由
1105 - const routes = [
1106     {
1107         path: '/',
1108         component: Home // 引入懒加载
1109     }
1110 ]
1110
1111 // 定义路由
1112 - const routes = [
1113     {
1114         path: '/',
1115         component: Home // 引入懒加载
1116     }
1117 ]
1117
1118 // 定义路由
1119 - const routes = [
1120     {
1121         path: '/',
1122         component: Home // 引入懒加载
1123     }
1124 ]
1124
1125 // 定义路由
1126 - const routes = [
1127     {
1128         path: '/',
1129         component: Home // 引入懒加载
1130     }
1131 ]
1131
1132 // 定义路由
1133 - const routes = [
1134     {
1135         path: '/',
1136         component: Home // 引入懒加载
1137     }
1138 ]
1138
1139 // 定义路由
1140 - const routes = [
1141     {
1142         path: '/',
1143         component: Home // 引入懒加载
1144     }
1145 ]
1145
1146 // 定义路由
1147 - const routes = [
1148     {
1149         path: '/',
1150         component: Home // 引入懒加载
1151     }
1152 ]
1152
1153 // 定义路由
1154 - const routes = [
1155     {
1156         path: '/',
1157         component: Home // 引入懒加载
1158     }
1159 ]
1159
1160 // 定义路由
1161 - const routes = [
1162     {
1163         path: '/',
1164         component: Home // 引入懒加载
1165     }
1166 ]
1166
1167 // 定义路由
1168 - const routes = [
1169     {
1170         path: '/',
1171         component: Home // 引入懒加载
1172     }
1173 ]
1173
1174 // 定义路由
1175 - const routes = [
1176     {
1177         path: '/',
1178         component: Home // 引入懒加载
1179     }
1180 ]
1180
1181 // 定义路由
1182 - const routes = [
1183     {
1184         path: '/',
1185         component: Home // 引入懒加载
1186     }
1187 ]
1187
1188 // 定义路由
1189 - const routes = [
1190     {
1191         path: '/',
1192         component: Home // 引入懒加载
1193     }
1194 ]
1194
1195 // 定义路由
1196 - const routes = [
1197     {
1198         path: '/',
1199         component: Home // 引入懒加载
1200     }
1201 ]
1201
1202 // 定义路由
1203 - const routes = [
1204     {
1205         path: '/',
1206         component: Home // 引入懒加载
1207     }
1208 ]
1208
1209 // 定义路由
1210 - const routes = [
1211     {
1212         path: '/',
1213         component: Home // 引入懒加载
1214     }
1215 ]
1215
1216 // 定义路由
1217 - const routes = [
1218     {
1219         path: '/',
1220         component: Home // 引入懒加载
1221     }
1222 ]
1222
1223 // 定义路由
1224 - const routes = [
1225     {
1226         path: '/',
1227         component: Home // 引入懒加载
1228     }
1229 ]
1229
1230 // 定义路由
1231 - const routes = [
1232     {
1233         path: '/',
1234         component: Home // 引入懒加载
1235     }
1236 ]
1236
1237 // 定义路由
1238 - const routes = [
1239     {
1240         path: '/',
1241         component: Home // 引入懒加载
1242     }
1243 ]
1243
1244 // 定义路由
1245 - const routes = [
1246     {
1247         path: '/',
1248         component: Home // 引入懒加载
1249     }
1250 ]
1250
1251 // 定义路由
1252 - const routes = [
1253     {
1254         path: '/',
1255         component: Home // 引入懒加载
1256     }
1257 ]
1257
1258 // 定义路由
1259 - const routes = [
1260     {
1261         path: '/',
1262         component: Home // 引入懒加载
1263     }
1264 ]
1264
1265 // 定义路由
1266 - const routes = [
1267     {
1268         path: '/',
1269         component: Home // 引入懒加载
1270     }
1271 ]
1271
1272 // 定义路由
1273 - const routes = [
1274     {
1275         path: '/',
1276         component: Home // 引入懒加载
1277     }
1278 ]
1278
1279 // 定义路由
1280 - const routes = [
1281     {
1282         path: '/',
1283         component: Home // 引入懒加载
1284     }
1285 ]
1285
1286 // 定义路由
1287 - const routes = [
1288     {
1289         path: '/',
1290         component: Home // 引入懒加载
1291     }
1292 ]
1292
1293 // 定义路由
1294 - const routes = [
1295     {
1296         path: '/',
1297         component: Home // 引入懒加载
1298     }
1299 ]
1299
1300 // 定义路由
1301 - const routes = [
1302     {
1303         path: '/',
1304         component: Home // 引入懒加载
1305     }
1306 ]
1306
1307 // 定义路由
1308 - const routes = [
1309     {
1310         path: '/',
1311         component: Home // 引入懒加载
1312     }
1313 ]
1313
1314 // 定义路由
1315 - const routes = [
1316     {
1317         path: '/',
1318         component: Home // 引入懒加载
1319     }
1320 ]
1320
1321 // 定义路由
1322 - const routes = [
1323     {
1324         path: '/',
1325         component: Home // 引入懒加载
1326     }
1327 ]
1327
1328 // 定义路由
1329 - const routes = [
1330     {
1331         path: '/',
1332         component: Home // 引入懒加载
1333     }
1334 ]
1334
1335 // 定义路由
1336 - const routes = [
1337     {
1338         path: '/',
1339         component: Home // 引入懒加载
1340     }
1341 ]
1341
1342 // 定义路由
1343 - const routes = [
1344     {
1345         path: '/',
1346         component: Home // 引入懒加载
1347     }
1348 ]
1348
1349 // 定义路由
1350 - const routes = [
1351     {
1352         path: '/',
1353         component: Home // 引入懒加载
1354     }
1355 ]
1355
1356 // 定义路由
1357 - const routes = [
1358     {
1359         path: '/',
1360         component: Home // 引入懒加载
1361     }
1362 ]
1362
1363 // 定义路由
1364 - const routes = [
1365     {
1366         path: '/',
1367         component: Home // 引入懒加载
1368     }
1369 ]
1369
1370 // 定义路由
1371 - const routes = [
1372     {
1373         path: '/',
1374         component: Home // 引入懒加载
1375     }
1376 ]
1376
1377 // 定义路由
1378 - const routes = [
1379     {
1380         path: '/',
1381         component: Home // 引入懒加载
1382     }
1383 ]
1383
1384 // 定义路由
1385 - const routes = [
1386     {
1387         path: '/',
1388         component: Home // 引入懒加载
1389     }
1390 ]
1390
1391 // 定义路由
1392 - const routes = [
1393     {
1394         path: '/',
1395         component: Home // 引入懒加载
1396     }
1397 ]
1397
1398 // 定义路由
1399 - const routes = [
1400     {
1401         path: '/',
1402         component:
```

```

38         })
39     })
40   } else {
41     // 没有动态改变权限的需求可直接next() 删除下方权限判断 ↓
42     if (hasPermission(store.getters.roles, to.meta.roles)) {
43       next()//
44     } else {
45       next({ path: '/401', replace: true, query: { noGoBack: true }})
46     }
47     // 可删 ↑
48   }
49 }
50 } else {
51   /* has no token*/
52   if (whiteList.indexOf(to.path) !== -1) { // 在免登录白名单, 直接进入
53     next()
54   } else {
55     next('/login') // 否则全部重定向到登录页
56     NProgress.done() // if current page is login will not trigger afterE
ach hook, so manually handle it
57   }
58 }
59 })
60 )
61 router.afterEach(() => {
62   NProgress.done() // finish progress bar
63 })

```

按需挂载，路由就需要知道用户的路由权限，也就是在用户登录进来的时候就要知道当前用户拥有哪些路由权限

这种方式也存在了以下的缺点：

- 全局路由守卫里，每次路由跳转都要做判断
- 菜单信息写死在前端，要改个显示文字或权限信息，需要重新编译
- 菜单跟路由耦合在一起，定义路由的时候还有添加菜单显示标题，图标之类的信息，而且路由不一定作为菜单显示，还要多加字段进行标识

23.2.3. 菜单权限

菜单权限可以理解成将页面与理由进行解耦

23.2.3.1. 方案一

菜单与路由分离，菜单由后端返回

前端定义路由信息

```
1  {
2      name: "login",
3      path: "/login",
4      component: () => import("@/pages/Login.vue")
5 }
```

JavaScript

复制代码

`name` 字段都不为空，需要根据此字段与后端返回菜单做关联，后端返回的菜单信息中必须要有 `name` 对应的字段，并且做唯一性校验

全局路由守卫里做判断

JavaScript | 复制代码

```
1 function hasPermission(router, accessMenu) {
2   if (whiteList.indexOf(router.path) !== -1) {
3     return true;
4   }
5   let menu = Util.getMenuByName(router.name, accessMenu);
6   if (menu.name) {
7     return true;
8   }
9   return false;
10}
11
12
13 Router.beforeEach(async (to, from, next) => {
14   if (getToken()) {
15     let userInfo = store.state.user.userInfo;
16   if (!userInfo.name) {
17     try {
18       await store.dispatch("GetUserInfo")
19       await store.dispatch('updateAccessMenu')
20     if (to.path === '/login') {
21       next({ name: 'home_index' })
22     } else {
23       //Util.toDefaultPage([...routers], to.name, router, next);
24       next({ ...to, replace: true })//菜单权限更新完成,重新进一次当前路由
25     }
26   }
27   catch (e) {
28     if (whiteList.indexOf(to.path) !== -1) { // 在免登录白名单, 直接进入
29       next()
30     } else {
31       next('/login')
32     }
33   }
34 } else {
35   if (to.path === '/login') {
36     next({ name: 'home_index' })
37   } else {
38     if (hasPermission(to, store.getters.accessMenu)) {
39       Util.toDefaultPage(store.getters.accessMenu,to, routes, next);
40     } else {
41       next({ path: '/403', replace:true })
42     }
43   }
44 }
45 } else {
```

```

46      if (whiteList.indexOf(to.path) !== -1) { // 在免登录白名单，直接进入
47          next()
48      } else {
49          next('/login')
50      }
51  }
52  let menu = Util.getMenuByName(to.name, store.getters.accessMenu);
53  Util.title(menu.title);
54 });
55
56 Router.afterEach((to) => {
57   window.scrollTo(0, 0);
58 });

```

每次路由跳转的时候都要判断权限，这里的判断也很简单，因为菜单的 `name` 与路由的 `name` 是一一对应的，而后端返回的菜单就已经是经过权限过滤的

如果根据路由 `name` 找不到对应的菜单，就表示用户有没权限访问

如果路由很多，可以在应用初始化的时候，只挂载不需要权限控制的路由。取得后端返回的菜单后，根据菜单与路由的对应关系，筛选出可访问的路由，通过 `addRoutes` 动态挂载

这种方式的缺点：

- 菜单需要与路由做一一对应，前端添加了新功能，需要通过菜单管理功能添加新的菜单，如果菜单配置的不对会导致应用不能正常使用
- 全局路由守卫里，每次路由跳转都要做判断

23.2.3.2. 方案二

菜单和路由都由后端返回

前端统一定义路由组件

▼
JavaScript
 复制代码

```

1 const Home = () => import("../pages/Home.vue");
2 const UserInfo = () => import("../pages/UserInfo.vue");
3 export default {
4   home: Home,
5   userInfo: UserInfo
6 };

```

后端路由组件返回以下格式

```

1  [
2   {
3     name: "home",
4     path: "/",
5     component: "home"
6   },
7   {
8     name: "home",
9     path: "/userinfo",
10    component: "userInfo"
11  }
12 ]

```

在将后端返回路由通过 `addRoutes` 动态挂载之间，需要将数据处理一下，将 `component` 字段换为真正的组件

如果有嵌套路由，后端功能设计的时候，要注意添加相应的字段，前端拿到数据也要做相应的处理
这种方法也会存在缺点：

- 全局路由守卫里，每次路由跳转都要做判断
- 前后端的配合要求更高

23.2.4. 按钮权限

23.2.4.1. 方案一

按钮权限也可以用 `v-if` 判断

但是如果页面过多，每个页面都要获取用户权限 `role` 和路由表里的 `meta.btnPermissions`，然后再做判断

这种方式就不展开举例了

23.2.4.2. 方案二

通过自定义指令进行按钮权限的判断

首先配置路由

JavaScript | 复制代码

```
1  {
2      path: '/permission',
3      component: Layout,
4      name: '权限测试',
5      meta: {
6          btnPermissions: ['admin', 'supper', 'normal']
7      },
8      //页面需要的权限
9      children: [
10         {
11             path: 'supper',
12             component: _import('system/supper'),
13             name: '权限测试页',
14             meta: {
15                 btnPermissions: ['admin', 'supper']
16             }
17         },
18         {
19             path: 'normal',
20             component: _import('system/normal'),
21             name: '权限测试页',
22             meta: {
23                 btnPermissions: ['admin']
24             }
25     }]
}
```

自定义权限鉴定指令

JavaScript | 复制代码

```

1 import Vue from 'vue'
2 /**权限指令*/
3 const has = Vue.directive('has', {
4   bind: function (el, binding, vnode) {
5     // 获取页面按钮权限
6     let btnPermissionsArr = [];
7     if(binding.value){
8       // 如果指令传值, 获取指令参数, 根据指令参数和当前登录人按钮权限做比较。
9       btnPermissionsArr = Array.of(binding.value);
10    }else{
11      // 否则获取路由中的参数, 根据路由的btnPermissionsArr和当前登录人按钮权
12     限做比较。
13      btnPermissionsArr = vnode.context.$route.meta.btnPermissions;
14    }
15    if (!Vue.prototype.$_has(btnPermissionsArr)) {
16      el.parentNode.removeChild(el);
17    }
18  });
19 // 权限检查方法
20 Vue.prototype.$_has = function (value) {
21   let isExist = false;
22   // 获取用户按钮权限
23   let btnPermissionsStr = sessionStorage.getItem("btnPermissions");
24   if (btnPermissionsStr == undefined || btnPermissionsStr == null) {
25     return false;
26   }
27   if (value.indexOf(btnPermissionsStr) > -1) {
28     isExist = true;
29   }
30   return isExist;
31 };
32 export {has}

```

在使用的按钮中只需要引用 `v-has` 指令

JavaScript | 复制代码

```

1 <el-button @click='editClick' type="primary" v-has>编辑</el-button>

```

23.3. 小结

关于权限如何选择哪种合适的方案，可以根据自己项目的方案项目，如考虑路由与菜单是否分离

权限需要前后端结合，前端尽可能的去控制，更多的需要后台判断

24. 说说你对keep-alive的理解是什么？



24.1. Keep-alive 是什么

`keep-alive` 是 `vue` 中的内置组件，能在组件切换过程中将状态保留在内存中，防止重复渲染 `DOM`

`keep-alive` 包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们

`keep-alive` 可以设置以下 `props` 属性：

- `include` – 字符串或正则表达式。只有名称匹配的组件会被缓存
- `exclude` – 字符串或正则表达式。任何名称匹配的组件都不会被缓存
- `max` – 数字。最多可以缓存多少组件实例

关于 `keep-alive` 的基本用法：

```
1 <keep-alive>
2   <component :is="view"></component>
3 </keep-alive>
```

使用 `includes` 和 `exclude`：

Go | 复制代码

```

1 <keep-alive include="a,b">
2   <component :is="view"></component>
3 </keep-alive>
4
5 <!-- 正则表达式（使用 `v-bind`）-->
6 <keep-alive :include="/a|b/">
7   <component :is="view"></component>
8 </keep-alive>
9
10 <!-- 数组（使用 `v-bind`）-->
11 <keep-alive :include="['a', 'b']">
12   <component :is="view"></component>
13 </keep-alive>

```

匹配首先检查组件自身的 `name` 选项，如果 `name` 选项不可用，则匹配它的局部注册名称（父组件 `components` 选项的键值），匿名组件不能被匹配

设置了 `keep-alive` 缓存的组件，会多出两个生命周期钩子（`activated` 与 `deactivated`）：

- 首次进入组件时：`beforeRouteEnter` > `beforeCreate` > `created` > `mounted` > `activated` > > `beforeRouteLeave` > `deactivated`
- 再次进入组件时：`beforeRouteEnter` > `activated` > > `beforeRouteLeave` > `deactivated`

24.2. 使用场景

使用原则：当我们在某些场景下不需要让页面重新加载时我们可以使用 `keepalive`

举个栗子：

当我们从 `首页` -> `列表页` -> `商详页` -> `再返回`，这时候列表页应该是需要 `keep-alive`

从 `首页` -> `列表页` -> `商详页` -> `返回到列表页(需要缓存)` -> `返回到首页(需要缓存)` -> `再次进入列表页(不需要缓存)`，这时候可以按需来控制页面的 `keep-alive`

在路由中设置 `keepAlive` 属性判断是否需要缓存

Go | 复制代码

```
1  {
2    path: 'list',
3    name: 'itemList', // 列表页
4    component (resolve) {
5      require(['@/pages/item/list'], resolve)
6    },
7    meta: {
8      keepAlive: true,
9      title: '列表页'
10 }
11 }
```

使用 `<keep-alive>`

Go | 复制代码

```
1 <div id="app" class='wrapper'>
2   <keep-alive>
3     <!-- 需要缓存的视图组件 -->
4     <router-view v-if="$route.meta.keepAlive"></router-view>
5   </keep-alive>
6     <!-- 不需要缓存的视图组件 -->
7     <router-view v-if="!$route.meta.keepAlive"></router-view>
8 </div>
```

24.3. 原理分析

`keep-alive` 是 `vue` 中内置的一个组件

源码位置：src/core/components/keep-alive.js

[Go](#) | [复制代码](#)

```
1 export default {
2   name: 'keep-alive',
3   abstract: true,
4
5   props: {
6     include: [String, RegExp, Array],
7     exclude: [String, RegExp, Array],
8     max: [String, Number]
9   },
10
11  created () {
12    this.cache = Object.create(null)
13    this.keys = []
14  },
15
16  destroyed () {
17    for (const key in this.cache) {
18      pruneCacheEntry(this.cache, key, this.keys)
19    }
20  },
21
22  mounted () {
23    this.$watch('include', val => {
24      pruneCache(this, name => matches(val, name))
25    })
26    this.$watch('exclude', val => {
27      pruneCache(this, name => !matches(val, name))
28    })
29  },
30
31  render() {
32    /* 获取默认插槽中的第一个组件节点 */
33    const slot = this.$slots.default
34    const vnode = getFirstComponentChild(slot)
35    /* 获取该组件节点的componentOptions */
36    const componentOptions = vnode && vnode.componentOptions
37
38    if (componentOptions) {
39      /* 获取该组件节点的名称，优先获取组件的name字段，如果name不存在则获取组件的tag */
40      const name = getComponentName(componentOptions)
41
42      const { include, exclude } = this
43      /* 如果name不在inlcude中或者存在于exlude中则表示不缓存，直接返回vnode */
44      if (
```

```

45         (include && (!name || !matches(include, name))) ||
46         // excluded
47         (exclude && name && matches(exclude, name))
48     ) {
49     return vnode
50 }
51
52     const { cache, keys } = this
53     /* 获取组件的key值 */
54     const key = vnode.key == null
55         // same constructor may get registered as different local component
56     ts
57         // so cid alone is not enough (#3269)
58         ? componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions.tag}` : '')
59         : vnode.key
60     /* 拿到key值后去this.cache对象中去寻找是否有该值，如果有则表示该组件有缓存，即命中缓存 */
61     if (cache[key]) {
62         vnode.componentInstance = cache[key].componentInstance
63         // make current key freshest
64         remove(keys, key)
65         keys.push(key)
66     }
67     /* 如果没有命中缓存，则将其设置进缓存 */
68     else {
69         cache[key] = vnode
70         keys.push(key)
71         // prune oldest entry
72         /* 如果配置了max并且缓存的长度超过了this.max，则从缓存中删除第一个 */
73         if (this.max && keys.length > parseInt(this.max)) {
74             pruneCacheEntry(cache, keys[0], keys, this._vnode)
75         }
76     }
77     vnode.data.keepAlive = true
78 }
79 return vnode || (slot && slot[0])
80 }
81 }

```

可以看到该组件没有 `template`，而是用了 `render`，在组件渲染的时候会自动执行 `render` 函数

`this.cache` 是一个对象，用来存储需要缓存的组件，它将以如下形式存储：

[Go](#) | [复制代码](#)

```

1 this.cache = {
2   'key1': '组件1',
3   'key2': '组件2',
4   // ...
5 }
```

在组件销毁的时候执行 `pruneCacheEntry` 函数

[Go](#) | [复制代码](#)

```

1 function pruneCacheEntry (
2   cache: VNodeCache,
3   key: string,
4   keys: Array<string>,
5   current?: VNode
6 ) {
7   const cached = cache[key]
8   /* 判断当前没有处于被渲染状态的组件，将其销毁*/
9   if (cached && (!current || cached.tag !== current.tag)) {
10     cached.componentInstance.$destroy()
11   }
12   cache[key] = null
13   remove(keys, key)
14 }
```

在 `mounted` 钩子函数中观测 `include` 和 `exclude` 的变化，如下：

[Go](#) | [复制代码](#)

```

1 mounted () {
2   this.$watch('include', val => {
3     pruneCache(this, name => matches(val, name))
4   })
5   this.$watch('exclude', val => {
6     pruneCache(this, name => !matches(val, name))
7   })
8 }
```

如果 `include` 或 `exclude` 发生了变化，即表示定义需要缓存的组件的规则或者不需要缓存的组件的规则发生了变化，那么就执行 `pruneCache` 函数，函数如下：

[Go](#) | [复制代码](#)

```

1 function pruneCache (keepAliveInstance, filter) {
2   const { cache, keys, _vnode } = keepAliveInstance
3   for (const key in cache) {
4     const cachedNode = cache[key]
5     if (cachedNode) {
6       const name = getComponentName(cachedNode.componentOptions)
7       if (name && !filter(name)) {
8         pruneCacheEntry(cache, key, keys, _vnode)
9       }
10    }
11  }
12}

```

在该函数内对 `this.cache` 对象进行遍历，取出每一项的 `name` 值，用其与新的缓存规则进行匹配，如果匹配不上，则表示在新的缓存规则下该组件已经不需要被缓存，则调用 `pruneCacheEntry` 函数将其从 `this.cache` 对象剔除即可

关于 `keep-alive` 的最强大缓存功能是在 `render` 函数中实现

首先获取组件的 `key` 值：

```

1 const key = vnode.key == null?
2   componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions.tag}` : '')
3   : vnode.key

```

拿到 `key` 值后去 `this.cache` 对象中去寻找是否有该值，如果有则表示该组件有缓存，即命中缓存，如下：

```

1 /* 如果命中缓存，则直接从缓存中拿 vnode 的组件实例 */
2 if (cache[key]) {
3   vnode.componentInstance = cache[key].componentInstance
4   /* 调整该组件key的顺序，将其从原来的地方删掉并重新放在最后一个 */
5   remove(keys, key)
6   keys.push(key)
7 }

```

直接从缓存中拿 `vnode` 的组件实例，此时重新调整该组件 `key` 的顺序，将其从原来的地方删掉并重新放在 `this.keys` 中最后一个

`this.cache` 对象中没有该 `key` 值的情况，如下：

```

1  /* 如果没有命中缓存，则将其设置进缓存 */
2  else {
3      cache[key] = vnode
4      keys.push(key)
5      /* 如果配置了max并且缓存的长度超过了this.max，则从缓存中删除第一个 */
6      if (this.max && keys.length > parseInt(this.max)) {
7          pruneCacheEntry(cache, keys[0], keys, this._vnode)
8      }
9  }

```

表明该组件还没有被缓存过，则以该组件的 `key` 为键，组件 `vnode` 为值，将其存入 `this.cache` 中，并且把 `key` 存入 `this.keys` 中

此时再判断 `this.keys` 中缓存组件的数量是否超过了设置的最大缓存数量值 `this.max`，如果超过了，则把第一个缓存组件删掉

24.4. 思考题：缓存后如何获取数据

解决方案可以有以下两种：

- `beforeRouteEnter`
- `activated`

24.4.1. `beforeRouteEnter`

每次组件渲染的时候，都会执行 `beforeRouteEnter`

```

1 beforeRouteEnter(to, from, next){
2     next(vm=>{
3         console.log(vm)
4         // 每次进入路由执行
5         vm.getData() // 获取数据
6     })
7 },

```

24.4.2. `activated`

在 `keep-alive` 缓存的组件被激活的时候，都会执行 `activated` 钩子

```
1 ▾ activated(){
2     this.getData() // 获取数据
3 },
```

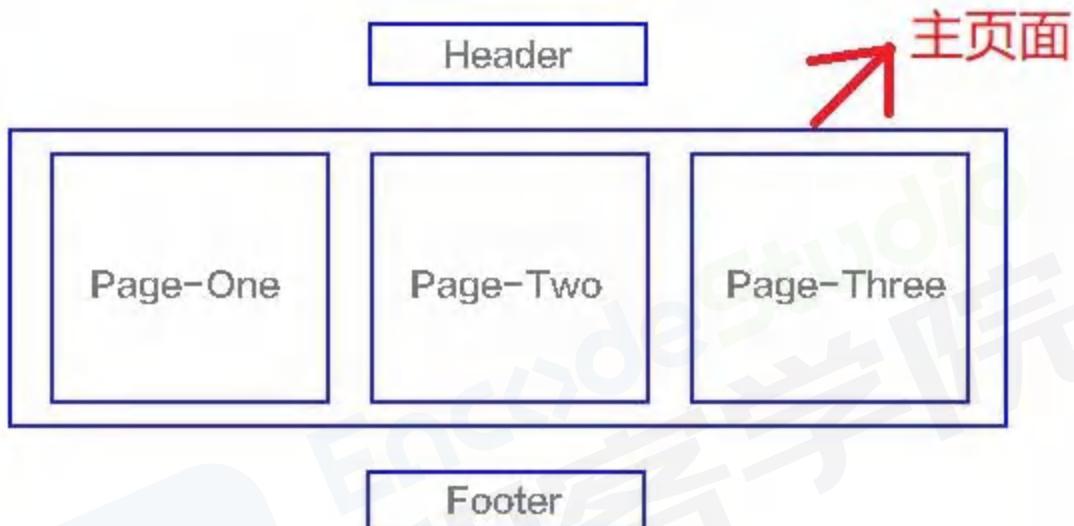
注意：服务器端渲染期间 `activated` 不被调用

25. 你对SPA单页面的理解，它的优缺点分别是什么？如何实现SPA应用呢



25.1. 什么是SPA

SPA (single-page application)，翻译过来就是单页应用。SPA 是一种网络应用程序或网站的模型，它通过动态重写当前页面来与用户交互，这种方法避免了页面之间切换打断用户体验。在单页应用中，所有必要的代码（HTML、JavaScript 和 CSS）都通过单个页面的加载而检索，或者根据需要（通常是响应用户操作）动态装载适当的资源并添加到页面。页面在任何时间点都不会重新加载，也不会将控制转移到其他页面。举个例子来讲就是一个杯子，早上装的牛奶，中午装的是开水，晚上装的是茶，我们发现，变的始终是杯子里的内容，而杯子始终是那个杯子。结构如下图。



我们熟知的JS框架如 react , vue , angular , ember 都属于 SPA

25.2. SPA和MPA的区别

上面大家已经对单页面有所了解了，下面来讲讲多页应用MPA (MultiPage-page application)，翻译过来就是多页应用在 MPA 中，每个页面都是一个主页面，都是独立的当我们在访问另一个页面的时候，都需要重新加载 html 、 css 、 js 文件，公共文件则根据需求按需加载如下图



25.2.1.1. 单页应用与多页应用的区别

	单页面应用 (SPA)	多页面应用 (MPA)
组成	一个主页面和多个页面片段	多个主页面

刷新方式	局部刷新	整页刷新
url模式	哈希模式	历史模式
SEO搜索引擎优化	难实现，可使用SSR方式改善	容易实现
数据传递	容易	通过url、cookie、localStorage等传递
页面切换	速度快，用户体验良好	切换加载资源，速度慢，用户体验差
维护成本	相对容易	相对复杂

25.2.1.2. 单页应用优缺点

优点：

- 具有桌面应用的即时性、网站的可移植性和可访问性
- 用户体验好、快，内容的改变不需要重新加载整个页面
- 良好的前后端分离，分工更明确

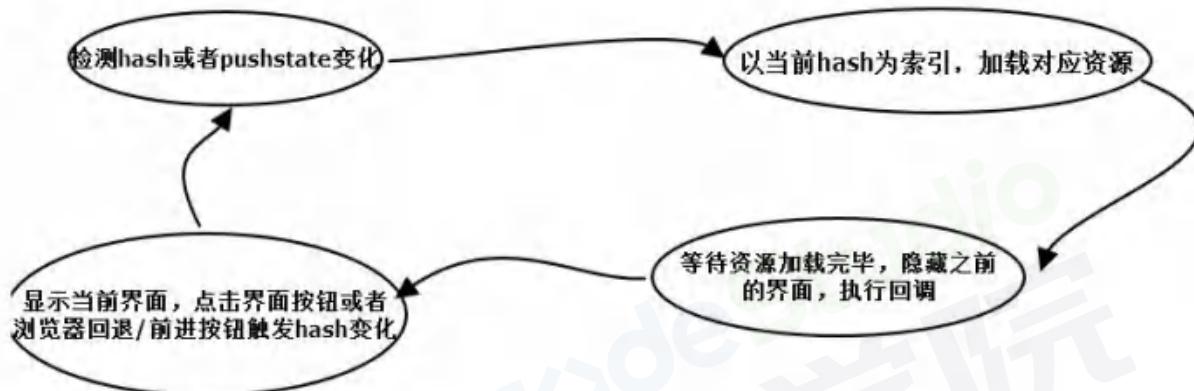
缺点：

- 不利于搜索引擎的抓取
- 首次渲染速度相对较慢

25.3. 实现一个SPA

25.3.1.1. 原理

1. 监听地址栏中 `hash` 变化驱动界面变化
2. 用 `pushstate` 记录浏览器的历史，驱动界面发送变化



25.3.1.2. 实现

25.3.1.2.1. hash 模式

核心通过监听 `url` 中的 `hash` 来进行路由跳转

```

1 // 定义 Router
2 class Router {
3   constructor () {
4     this.routes = {};// 存放路由path及callback
5     this.currentUrl = '';
6
7     // 监听路由change调用相对应的路由回调
8     window.addEventListener('load', this.refresh, false);
9     window.addEventListener('hashchange', this.refresh, false);
10  }
11
12  route(path, callback){
13    this.routes[path] = callback;
14  }
15
16  push(path) {
17    this.routes[path] && this.routes[path]()
18  }
19 }
20
21 // 使用 router
22 window.miniRouter = new Router();
23 miniRouter.route('/', () => console.log('page1'))
24 miniRouter.route('/page2', () => console.log('page2'))
25
26 miniRouter.push('/') // page1
27 miniRouter.push('/page2') // page2

```

25.3.1.2.2. history模式

`history` 模式核心借用 `HTML5 history api`，`api` 提供了丰富的 `router` 相关属性先了解一个几个相关的api

- `history.pushState` 浏览器历史纪录添加记录
- `history.replaceState` 修改浏览器历史纪录中当前纪录
- `history.popState` 当 `history` 发生变化时触发

```

1 // 定义 Router
2 class Router {
3   constructor () {
4     this.routes = {};
5     this.listerPopState()
6   }
7
8   init(path) {
9     history.replaceState({path: path}, null, path);
10    this.routes[path] && this.routes[path]()
11  }
12
13  route(path, callback){
14    this.routes[path] = callback;
15  }
16
17  push(path) {
18    history.pushState({path: path}, null, path);
19    this.routes[path] && this.routes[path]()
20  }
21
22  listerPopState () {
23    window.addEventListener('popstate' , e => {
24      const path = e.state && e.state.path;
25      this.routers[path] && this.routers[path]()
26    })
27  }
28 }
29
30 // 使用 Router
31
32 window.miniRouter = new Router();
33 miniRouter.route('/', ()=> console.log('page1'))
34 miniRouter.route('/page2', ()=> console.log('page2'))
35
36 // 跳转
37 miniRouter.push('/page2') // page2

```

25.3.1. 如何给SPA做SEO

下面给出基于 Vue 的 SPA 如何实现 SEO 的三种方式

1. SSR服务端渲染

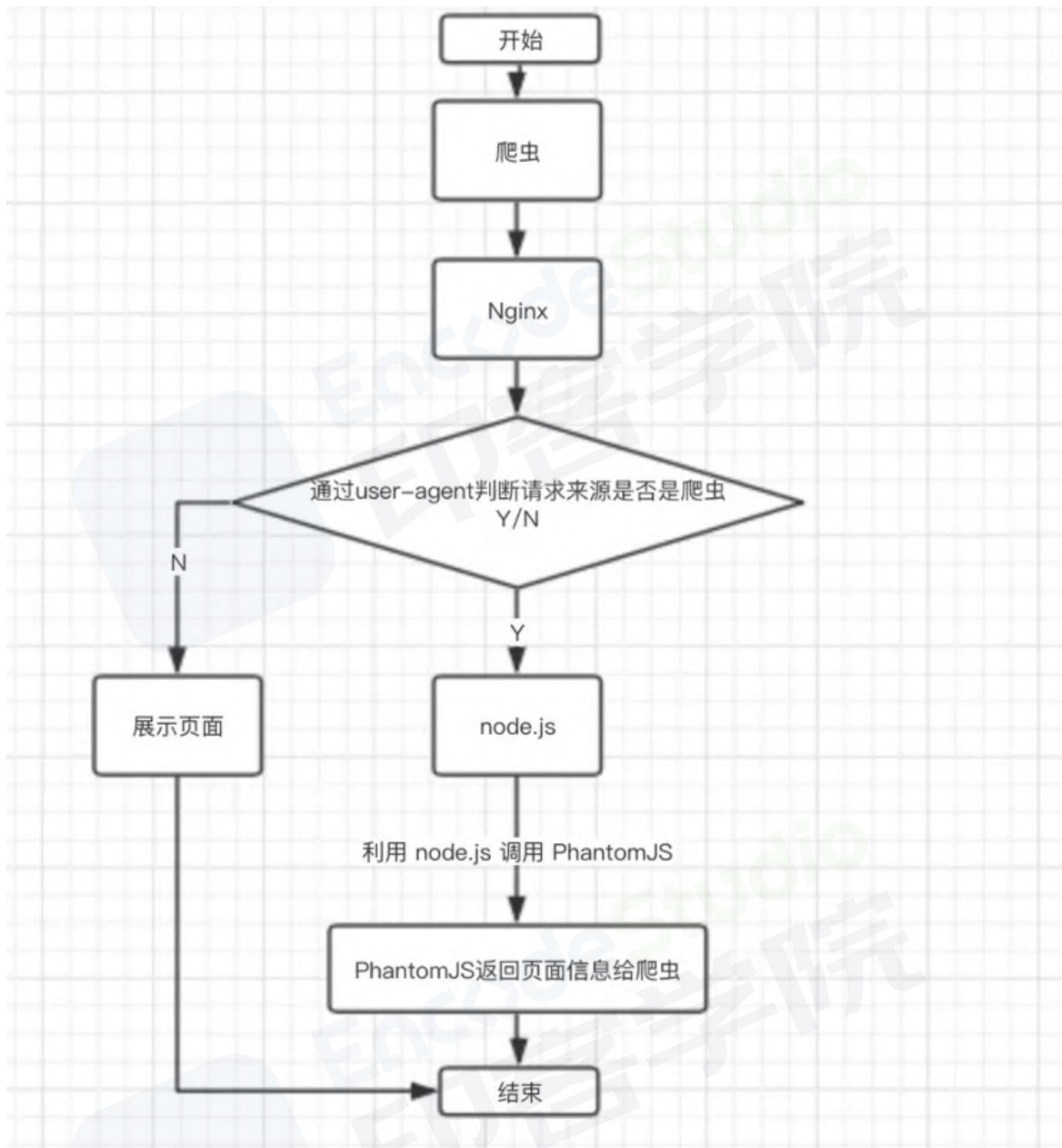
将组件或页面通过服务器生成html，再返回给浏览器，如 `nuxt.js`

2. 静态化

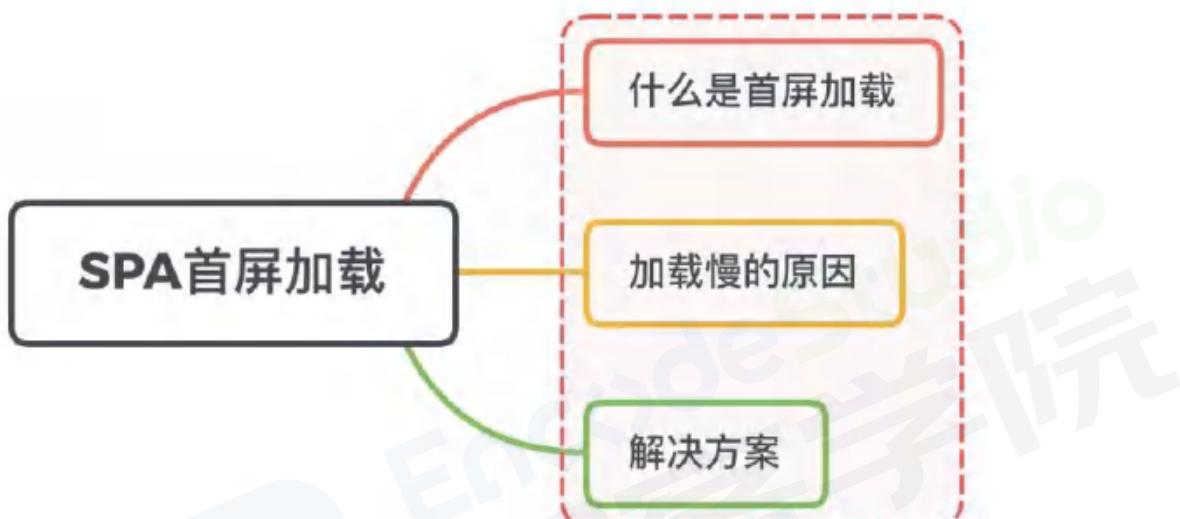
目前主流的静态化主要有两种：（1）一种是通过程序将动态页面抓取并保存为静态页面，这样的页面的实际存在于服务器的硬盘中（2）另外一种是通过WEB服务器的 `URL Rewrite` 的方式，它的原理是通过web服务器内部模块按一定规则将外部的URL请求转化为内部的文件地址，一句话来说就是把外部请求的静态地址转化为实际的动态页面地址，而静态页面实际是不存在的。这两种方法都达到了实现URL静态化的效果

3. 使用 `Phantomjs` 针对爬虫处理

原理是通过 `Nginx` 配置，判断访问来源是否为爬虫，如果是则搜索引擎的爬虫请求会转发到一个 `node server`，再通过 `PhantomJS` 来解析完整的 `HTML`，返回给爬虫。下面是大致流程图



26. SPA首屏加载速度慢的怎么解决？



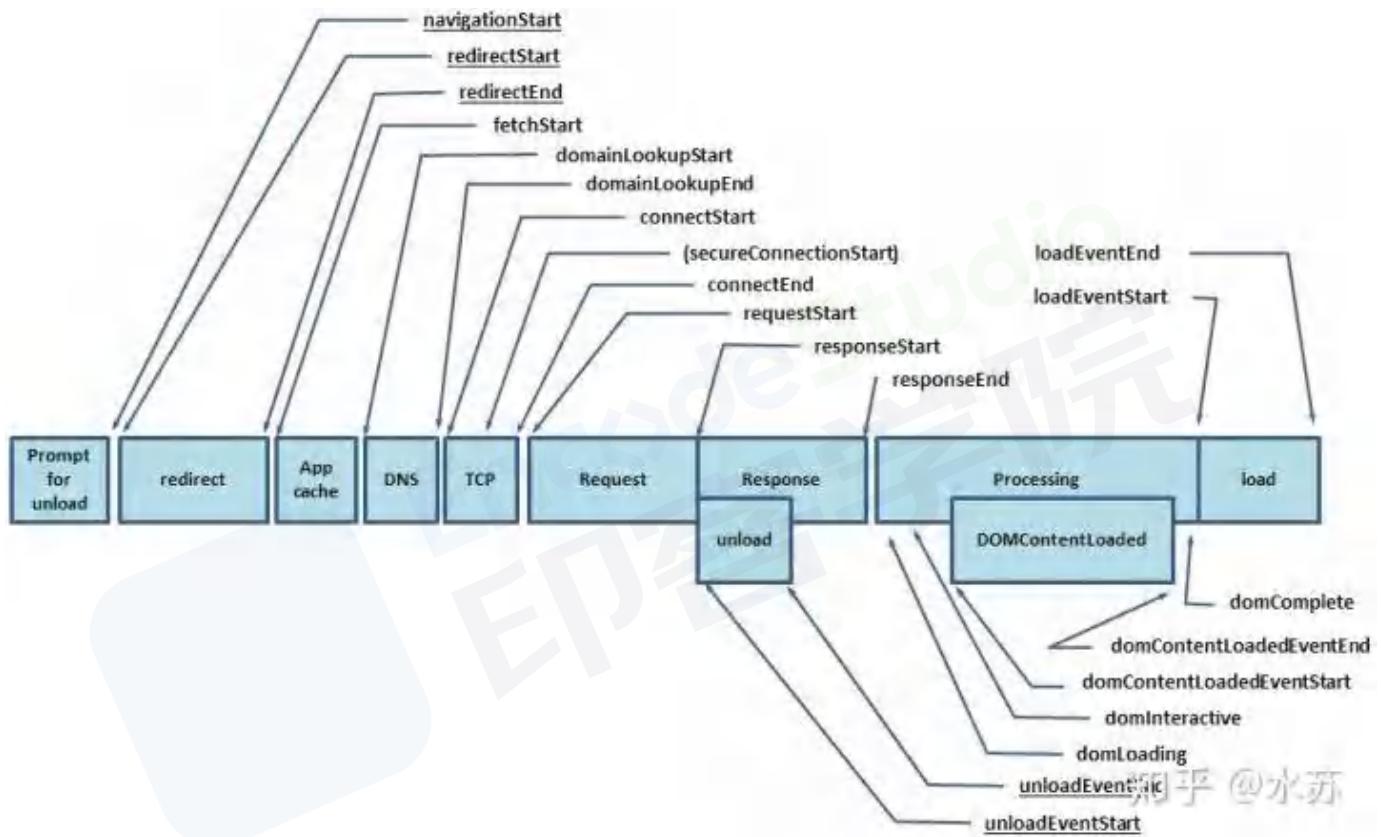
26.1. 什么是首屏加载

首屏时间 (First Contentful Paint) , 指的是浏览器从响应用户输入网址地址, 到首屏内容渲染完成的时间, 此时整个网页不一定要全部渲染完成, 但需要展示当前视窗需要的内容

首屏加载可以说是用户体验中最重要的环节

26.1.1. 关于计算首屏时间

利用 `performance.timing` 提供的数据:



通过 `DOMContentLoaded` 或者 `performance` 来计算出首屏时间

```

1 // 方案一:
2 document.addEventListener('DOMContentLoaded', (event) => {
3     console.log('first contentful painting');
4 });
5 // 方案二:
6 performance.getEntriesByName("first-contentful-paint")[0].startTime
7
8 // performance.getEntriesByName("first-contentful-paint")[0]
9 // 会返回一个 PerformancePaintTiming的实例, 结构如下:
10 {
11     name: "first-contentful-paint",
12     entryType: "paint",
13     startTime: 507.80000002123415,
14     duration: 0,
15 };

```

26.2. 加载慢的原因

在页面渲染的过程，导致加载速度慢的因素可能如下：

- 网络延时问题
- 资源文件体积是否过大
- 资源是否重复发送请求去加载了
- 加载脚本的时候，渲染内容堵塞了

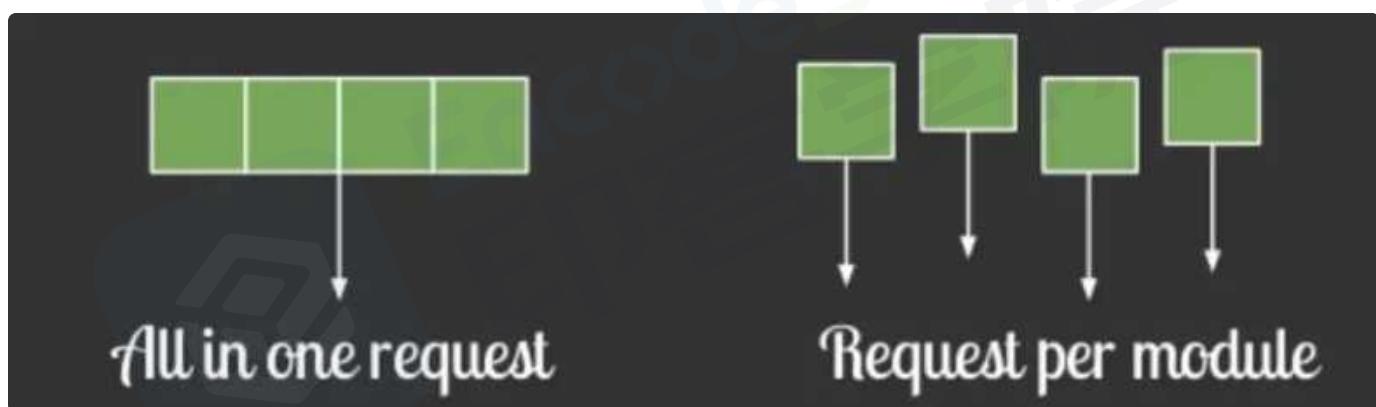
26.3. 解决方案

常见的几种SPA首屏优化方式

- 减小入口文件积
- 静态资源本地缓存
- UI框架按需加载
- 图片资源的压缩
- 组件重复打包
- 开启GZip压缩
- 使用SSR

26.3.1. 减小入口文件体积

常用的手段是路由懒加载，把不同路由对应的组件分割成不同的代码块，待路由被请求的时候会单独打包路由，使得入口文件变小，加载速度大大增加



在 `vue-router` 配置路由的时候，采用动态加载路由的形式

JavaScript | 复制代码

```

1 routes: [
2     path: 'Blogs',
3     name: 'ShowBlogs',
4     component: () => import('./components>ShowBlogs.vue')
5 ]

```

以函数的形式加载路由，这样就可以把各自的路由文件分别打包，只有在解析给定的路由时，才会加载路由组件

26.3.2. 静态资源本地缓存

后端返回资源问题：

- 采用 `HTTP` 缓存，设置 `Cache-Control`，`Last-Modified`，`Etag` 等响应头
- 采用 `Service Worker` 离线缓存

前端合理利用 `localStorage`

26.3.3. UI框架按需加载

在日常使用 `UI` 框架，例如 `element-UI`、或者 `antd`，我们经常性直接引用整个 `UI` 库

JavaScript | 复制代码

```

1 import ElementUI from 'element-ui'
2 Vue.use(ElementUI)

```

但实际上我用到的组件只有按钮，分页，表格，输入与警告 所以我们要按需引用

JavaScript | 复制代码

```

1 import { Button, Input, Pagination, Table, TableColumn, MessageBox } from
  'element-ui';
2 Vue.use(Button)
3 Vue.use(Input)
4 Vue.use(Pagination)

```

26.3.4. 组件重复打包

假设 `A.js` 文件是一个常用的库，现在有多个路由使用了 `A.js` 文件，这就造成了重复下载

解决方案：在 `webpack` 的 `config` 文件中，修改 `CommonsChunkPlugin` 的配置

```
1 minChunks: 3
```

JavaScript | 复制代码

`minChunks` 为3表示会把使用3次及以上的包抽离出来，放进公共依赖文件，避免了重复加载组件

26.3.5. 图片资源的压缩

图片资源虽然不在编码过程中，但它却是对页面性能影响最大的因素

对于所有的图片资源，我们可以进行适当的压缩

对页面上使用到的 `icon`，可以使用在线字体图标，或者雪碧图，将众多小图标合并到同一张图上，用以减轻 `http` 请求压力。

26.3.6. 开启GZip压缩

拆完包之后，我们再用 `gzip` 做一下压缩 安装 `compression-webpack-plugin`

```
1 cnpm i compression-webpack-plugin -D
```

JavaScript | 复制代码

在 `vue.config.js` 中引入并修改 `webpack` 配置

```
1 const CompressionPlugin = require('compression-webpack-plugin')
2
3 configureWebpack: (config) => {
4   if (process.env.NODE_ENV === 'production') {
5     // 为生产环境修改配置...
6     config.mode = 'production'
7     return {
8       plugins: [new CompressionPlugin({
9         test: /\.js$|\.html$|\.css/, // 匹配文件名
10        threshold: 10240, // 对超过10k的数据进行压缩
11        deleteOriginalAssets: false // 是否删除原文件
12      })
13    }
14 }
```

JavaScript | 复制代码

在服务器我们也要做相应的配置 如果发送请求的浏览器支持 `gzip`，就发送给它 `gzip` 格式的文件 我的服务器是用 `express` 框架搭建的 只要安装一下 `compression` 就能使用

```
1 const compression = require('compression')
2 app.use(compression()) // 在其他中间件使用之前调用
```

26.3.7. 使用SSR

SSR (Server side)，也就是服务端渲染，组件或页面通过服务器生成html字符串，再发送到浏览器
从头搭建一个服务端渲染是很复杂的，`vue` 应用建议使用 `Nuxt.js` 实现服务端渲染

26.4. 小结

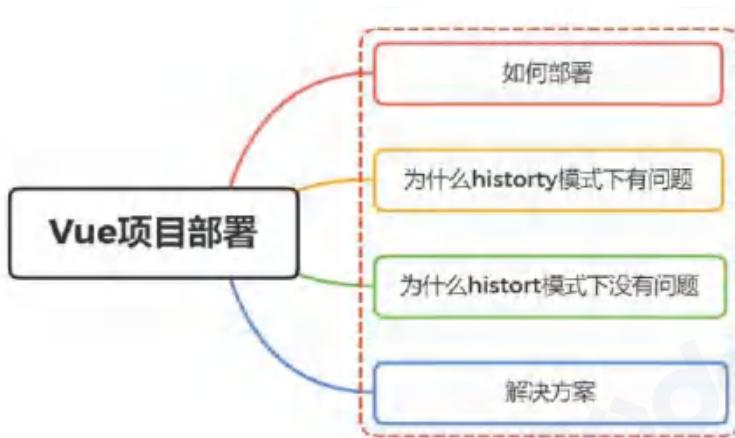
减少首屏渲染时间的方法有很多，总的来讲可以分成两大部分：资源加载优化 和 页面渲染优化

下图是更为全面的首屏优化的方案



大家可以根据自己项目的情况选择各种方式进行首屏渲染的优化

27. vue项目本地开发完成后部署到服务器后报404是什么原因呢？



27.1. 如何部署

前后端分离开发模式下，前端是独立部署的，前端只需要将最后的构建物上传至目标服务器的 web 容器指定的静态目录下即可

我们知道 vue 项目在构建后，是生成一系列的静态文件

常规部署我们只需要将这个目录上传至目标服务器即可

```
Bash | 复制代码
```

```
1 // scp 上传 user为主机登录用户, host为主机外网ip, xx为web容器静态资源路径
2 scp dist.zip user@host:/xx/xx/xx
```

让 web 容器跑起来，以 nginx 为例

```
Bash | 复制代码
```

```
1 server {
2   listen 80;
3   server_name www.xxx.com;
4
5   location / {
6     index /data/dist/index.html;
7   }
8 }
```

配置完成记得重启 nginx

Bash | 复制代码

```
1 // 检查配置是否正确
2 nginx -t
3
4 // 平滑重启
5 nginx -s reload
```

操作完后就可以在浏览器输入域名进行访问了

当然上面只是提到最简单也是最直接的一种布署方式

什么自动化，镜像，容器，流水线布署，本质也是将这套逻辑抽象，隔离，用程序来代替重复性的劳动，本文不展开

27.2. 404问题

这是一个经典的问题，相信很多同学都有遇到过，那么你知道其真正的原因吗？

我们先还原一下场景：

- vue 项目在本地时运行正常，但部署到服务器中，刷新页面，出现了404错误

先定位一下，HTTP 404 错误意味着链接指向的资源不存在

问题在于为什么不存在？且为什么只有 history 模式下会出现这个问题？

27.2.1. 为什么history模式下有问题

Vue 是属于单页应用 (single-page application)

而 SPA 是一种网络应用程序或网站的模型，所有用户交互是通过动态重写当前页面，前面我们也看到了，不管我们应用有多少页面，构建物都只会产出一个 index.html

现在，我们回头来看一下我们的 nginx 配置

JavaScript | 复制代码

```

1  server {
2      listen 80;
3      server_name www.xxx.com;
4
5      location / {
6          index /data/dist/index.html;
7      }
8  }

```

可以根据 `nginx` 配置得出，当我们在地址栏输入 `www.xxx.com` 时，这时会打开我们 `dist` 目录下的 `index.html` 文件，然后我们在跳转路由进入到 `www.xxx.com/login`

关键在这里，当我们在 `website.com/login` 页执行刷新操作，`nginx location` 是没有相关配置的，所以就会出现 404 的情况

27.2.2. 为什么hash模式下没有问题

`router hash` 模式我们都知道是用符号#表示的，如 `website.com/#/login`，`hash` 的值为 `#/login`

它的特点在于：`hash` 虽然出现在 `URL` 中，但不会被包括在 `HTTP` 请求中，对服务端完全没有影响，因此改变 `hash` 不会重新加载页面

`hash` 模式下，仅 `hash` 符号之前的内容会被包含在请求中，如 `website.com/#/login` 只有 `website.com` 会被包含在请求中，因此对于服务端来说，即使没有配置 `location`，也不会返回404错误

27.3. 解决方案

看到这里我相信大部分同学都能想到怎么解决问题了，

产生问题的本质是因为我们的路由是通过JS来执行视图切换的，

当我们进入到子路由时刷新页面，`web` 容器没有相对应的页面此时会出现404

所以我们只需要配置将任意页面都重定向到 `index.html`，把路由交由前端处理

对 `nginx` 配置文件 `.conf` 修改，添加 `try_files $uri $uri/ /index.html;`

Bash | 复制代码

```

1  server {
2      listen 80;
3      server_name www.xxx.com;
4
5      location / {
6          index /data/dist/index.html;
7          try_files $uri $uri/ /index.html;
8      }
9  }

```

修改完配置文件后记得配置的更新

Bash | 复制代码

```
1 nginx -s reload
```

这么做以后，你的服务器就不再返回 404 错误页面，因为对于所有路径都会返回 `index.html` 文件

为了避免这种情况，你应该在 `Vue` 应用里面覆盖所有的路由情况，然后在给出一个 404 页面

JavaScript | 复制代码

```

1 const router = new VueRouter({
2     mode: 'history',
3     routes: [
4         { path: '*', component: NotFoundComponent }
5     ]
6 })

```

28. SSR解决了什么问题？有做过SSR吗？你是怎么做的？



28.1. 是什么

`Server-Side Rendering` 我们称其为 `SSR`，意为服务端渲染

指由服务侧完成页面的 `HTML` 结构拼接的页面处理技术，发送到浏览器，然后为其绑定状态与事件，成为完全可交互页面的过程

先来看看 `Web` 3个阶段的发展史：

- 传统服务端渲染SSR
- 单页面应用SPA
- 服务端渲染SSR

28.1.1. 传统web开发

网页内容在服务端渲染完成，一次性传输到浏览器

打开页面查看源码，浏览器拿到的是全部的 `dom` 结构

28.1.2. 单页应用SPA

单页应用优秀的用户体验，使其逐渐成为主流，页面内容由 `JS` 渲染出来，这种方式称为客户端渲

打开页面查看源码，浏览器拿到的仅有宿主元素 `#app`，并没有内容

28.1.3. 服务端渲染SSR

SSR 解决方案，后端渲染出完整的首屏的 dom 结构返回，前端拿到的内容包括首屏及完整 spa 结构，应用激活后依然按照 spa 方式运行

看完前端发展，我们再看看 Vue 官方对 SSR 的解释：

Vue.js 是构建客户端应用程序的框架。默认情况下，可以在浏览器中输出 Vue 组件，进行生成 DOM 和操作 DOM。然而，也可以将同一个组件渲染为服务器端的 HTML 字符串，将它们直接发送到浏览器，最后将这些静态标记“激活”为客户端上完全可交互的应用程序

服务器渲染的 Vue.js 应用程序也可以被认为是“同构”或“通用”，因为应用程序的大部分代码都可以在服务器和客户端上运行

我们从上门解释得到以下结论：

- Vue SSR 是一个在 SPA 上进行改良的服务端渲染
- 通过 Vue SSR 渲染的页面，需要在客户端激活才能实现交互
- Vue SSR 将包含两部分：服务端渲染的首屏，包含交互的 SPA

28.2. 解决了什么

SSR主要解决了以下两种问题：

- seo：搜索引擎优先爬取页面 HTML 结构，使用 ssr 时，服务端已经生成了和业务想关联的 HTML，有利于 seo
- 首屏呈现渲染：用户无需等待页面所有 js 加载完成就可以看到页面视图（压力来到了服务器，所以需要权衡哪些用服务端渲染，哪些交给客户端）

但是使用 SSR 同样存在以下的缺点：

- 复杂度：整个项目的复杂度
- 库的支持性，代码兼容
- 性能问题
 - 每个请求都是 n 个实例的创建，不然会污染，消耗会变得很大
 - 缓存 node serve、nginx 判断当前用户有没有过期，如果没过期的话就缓存，用刚刚的结果。
 - 降级：监控 cpu、内存占用过多，就 spa，返回单个的壳
- 服务器负载变大，相对于前后端分离服务器只需要提供静态资源来说，服务器负载更大，所以要慎

重使用

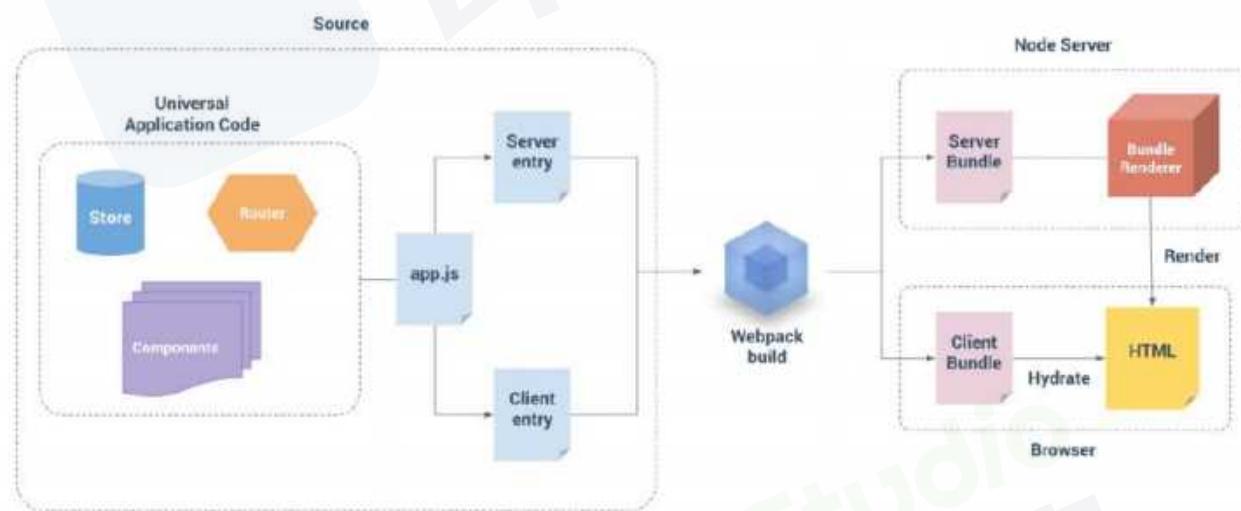
所以在我们选择是否使用 `SSR` 前，我们需要慎重问问自己这些问题：

1. 需要 `SEO` 的页面是否只是少数几个，这些是否可以使用预渲染（Prerender SPA Plugin）实现
2. 首屏的请求响应逻辑是否复杂，数据返回是否大量且缓慢

28.3. 如何实现

对于同构开发，我们依然使用 `webpack` 打包，我们要解决两个问题：服务端首屏渲染和客户端激活

这里需要生成一个服务器 `bundle` 文件用于服务端首屏渲染和一个客户端 `bundle` 文件用于客户端激活



代码结构 除了两个不同入口之外，其他结构和之前 `vue` 应用完全相同

```

1  src
2  └── router
3      └── index.js # 路由声明
4  └── store
5      └── index.js # 全局状态
6  └── main.js # 用于创建vue实例
7  └── entry-client.js # 客户端入口，用于静态内容“激活”
8  └── entry-server.js # 服务端入口，用于首屏内容渲染

```

路由配置

JavaScript | 复制代码

```

1 import Vue from "vue";
2 import Router from "vue-router";
3
4 Vue.use(Router);
5 //导出工厂函数
6
7 export function createRouter() {
8   return new Router({
9     mode: 'history',
10    routes: [
11      // 客户端没有编译器，这里要写成渲染函数
12      { path: "/", component: { render: h => h('div', 'index page') } },
13      { path: "/detail", component: { render: h => h('div', 'detail page') } }
14    ]
15  });
16}

```

主文件main.js

跟之前不同，主文件是负责创建 `Vue` 实例的工厂，每次请求均会有独立的 `Vue` 实例创建

JavaScript | 复制代码

```

1 import Vue from "vue";
2 import App from "./App.vue";
3 import { createRouter } from "./router";
4 // 导出Vue实例工厂函数，为每次请求创建独立实例
5 // 上下文用于给Vue实例传递参数
6 export function createApp(context) {
7   const router = createRouter();
8   const app = new Vue({
9     router,
10    context,
11    render: h => h(App)
12  });
13  return { app, router };
14}

```

编写服务端入口 `src/entry-server.js`

它的任务是创建 `Vue` 实例并根据传入 `url` 指定首屏

JavaScript | 复制代码

```

1 import { createApp } from "./main";
2 // 返回一个函数，接收请求上下文，返回创建的vue实例
3 export default context => {
4     // 这里返回一个Promise，确保路由或组件准备就绪
5     return new Promise((resolve, reject) => {
6         const { app, router } = createApp(context);
7         // 跳转到首屏的地址
8         router.push(context.url);
9         // 路由就绪，返回结果
10        router.onReady(() => {
11            resolve(app);
12        }, reject);
13    });
14 };

```

编写客户端入口 `entry-client.js`客户端入口只需创建 `vue` 实例并执行挂载，这一步称为激活

JavaScript | 复制代码

```

1 import { createApp } from "./main";
2 // 创建vue、router实例
3 const { app, router } = createApp();
4 // 路由就绪，执行挂载
5 router.onReady(() => {
6     app.$mount("#app");
7 });

```

对 `webpack` 进行配置

安装依赖

JavaScript | 复制代码

```
1 npm install webpack-node-externals lodash.merge -D
```

对 `vue.config.js` 进行配置

JavaScript | 复制代码

```
1 // 两个插件分别负责打包客户端和服务端
2 const VueSSRServerPlugin = require("vue-server-renderer/server-plugin");
3 const VueSSRClientPlugin = require("vue-server-renderer/client-plugin");
4 const nodeExternals = require("webpack-node-externals");
5 const merge = require("lodash.merge");
6 // 根据传入环境变量决定入口文件和相应配置项
7 const TARGET_NODE = process.env.WEBPACK_TARGET === "node";
8 const target = TARGET_NODE ? "server" : "client";
9 module.exports = {
10   css: {
11     extract: false
12   },
13   outputDir: './dist/'+target,
14   configureWebpack: () => ({
15     // 将 entry 指向应用程序的 server / client 文件
16     entry: `./src/entry-${target}.js`,
17     // 对 bundle renderer 提供 source map 支持
18     devtool: 'source-map',
19     // target设置为node使webpack以Node适用的方式处理动态导入,
20     // 并且还会在编译Vue组件时告知`vue-loader`输出面向服务器代码。
21     target: TARGET_NODE ? "node" : "web",
22     // 是否模拟node全局变量
23     node: TARGET_NODE ? undefined : false,
24     output: {
25       // 此处使用Node风格导出模块
26       libraryTarget: TARGET_NODE ? "commonjs2" : undefined
27     },
28     // https://webpack.js.org/configuration/externals/#function
29     // https://github.com/liady/webpack-node-externals
30     // 外置化应用程序依赖模块。可以使服务器构建速度更快，并生成较小的打包文件。
31     externals: TARGET_NODE
32     ? nodeExternals({
33       // 不要外置化webpack需要处理的依赖模块。
34       // 可以在这里添加更多的文件类型。例如，未处理 *.vue 原始文件,
35       // 还应该将修改`global`（例如polyfill）的依赖模块列入白名单
36       whitelist: [/\.css$/]
37     })
38     : undefined,
39     optimization: {
40       splitChunks: undefined
41     },
42     // 这是将服务器的整个输出构建为单个 JSON 文件的插件。
43     // 服务端默认文件名为 `vue-ssr-server-bundle.json`
44     // 客户端默认文件名为 `vue-ssr-client-manifest.json`。
45     plugins: [TARGET_NODE ? new VueSSRServerPlugin() : new
```

```
46          VueSSRClientPlugin()])
47      }),
48      chainWebpack: config => {
49          // cli4项目添加
50          if (TARGET_NODE) {
51              config.optimization.delete('splitChunks')
52          }
53
54          config.module
55              .rule("vue")
56              .use("vue-loader")
57              .tap(options => {
58                  merge(options, {
59                      optimizeSSR: false
60                  });
61              });
62      }
63  };
```

对脚本进行配置，安装依赖

```
1  npm i cross-env -D
```

定义创建脚本 `package.json`

```
1  "scripts": {
2      "build:client": "vue-cli-service build",
3      "build:server": "cross-env WEBPACK_TARGET=node vue-cli-service build",
4      "build": "npm run build:server && npm run build:client"
5  }
```

执行打包: `npm run build`

最后修改宿主文件 `/public/index.html`

HTML | 复制代码

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width,initial-scale=1.
0">
7      <title>Document</title>
8  </head>
9  <body>
10     <!--vue-ssr-outlet-->
11  </body>
12 </html>

```

是服务端渲染入口位置，注意不能为了好看而在前后加空格

安装 vuex

JavaScript | 复制代码

```
1  npm install -S vuex
```

创建 vuex 工厂函数

JavaScript | 复制代码

```

1  import Vue from 'vue'
2  import Vuex from 'vuex'
3  Vue.use(Vuex)
4  export function createStore () {
5      return new Vuex.Store({
6          state: {
7              count:108
8          },
9          mutations: {
10             add(state){
11                 state.count += 1;
12             }
13         }
14     })
15 }

```

在 `main.js` 文件中挂载 `store`

JavaScript | 复制代码

```

1 import { createStore } from './store'
2 export function createApp (context) {
3     // 创建实例
4     const store = createStore()
5     const app = new Vue({
6         store, // 挂载
7         render: h => h(App)
8     })
9     return { app, router, store }
10 }

```

服务器端渲染的是应用程序的“快照”，如果应用依赖于一些异步数据，那么在开始渲染之前，需要先预取和解析好这些数据

在 `store` 进行一步数据获取

JavaScript | 复制代码

```

1 export function createStore() {
2     return new Vuex.Store({
3         mutations: {
4             // 加一个初始化
5             init(state, count) {
6                 state.count = count;
7             },
8         },
9         actions: {
10             // 加一个异步请求count的动作
11             getCount({ commit }) {
12                 return new Promise(resolve => {
13                     setTimeout(() => {
14                         commit("init", Math.random() * 100);
15                         resolve();
16                     }, 1000);
17                 });
18             },
19         },
20     });
21 }

```

组件中的数据预取逻辑

JavaScript | 复制代码

```
1 export default {
2   asyncData({ store, route }) { // 约定预取逻辑编写在预取钩子asyncData中
3     // 触发 action 后, 返回 Promise 以便确定请求结果
4     return store.dispatch("getCount");
5   }
6 };
```

服务端数据预取, `entry-server.js`

```

1 import { createApp } from "./app";
2 export default context => {
3   return new Promise((resolve, reject) => {
4     // 拿出store和router实例
5     const { app, router, store } = createApp(context);
6     router.push(context.url);
7     router.onReady(() => {
8       // 获取匹配的路由组件数组
9       const matchedComponents = router.getMatchedComponents();
10
11      // 若无匹配则抛出异常
12      if (!matchedComponents.length) {
13        return reject({ code: 404 });
14      }
15
16      // 对所有匹配的路由组件调用可能存在的`asyncData()`
17      Promise.all(
18        matchedComponents.map(Component => {
19          if (Component.asyncData) {
20            return Component.asyncData({
21              store,
22              route: router.currentRoute,
23            });
24          }
25        }),
26      )
27        .then(() => {
28          // 所有预取钩子 resolve 后,
29          // store 已经填充入渲染应用所需状态
30          // 将状态附加到上下文, 且 `template` 选项用于 renderer 时,
31          // 状态将自动序列化为 `window.__INITIAL_STATE__`, 并注入 HTML
32          context.state = store.state;
33
34          resolve(app);
35        })
36        .catch(reject);
37      }, reject);
38    );
39  };

```

客户端在挂载到应用程序之前, `store` 就应该获取到状态, `entry-client.js`

JavaScript | 复制代码

```

1 // 导出store
2 const { app, router, store } = createApp();
3 // 当使用 template 时, context.state 将作为 window.__INITIAL_STATE__ 状态自动嵌入到最终的 HTML
4 // 在客户端挂载到应用程序之前, store 就应该获取到状态:
5 if (window.__INITIAL_STATE__) {
6     store.replaceState(window.__INITIAL_STATE__);
7 }

```

客户端数据预取处理, `main.js`

JavaScript | 复制代码

```

1 Vue.mixin({
2     beforeMount() {
3         const { asyncData } = this.$options;
4         if (asyncData) {
5             // 将获取数据操作分配给 promise
6             // 以便在组件中, 我们可以在数据准备就绪后
7             // 通过运行 `this.dataPromise.then(...)` 来执行其他任务
8             this.dataPromise = asyncData({
9                 store: this.$store,
10                route: this.$route,
11            });
12        }
13    },
14 });

```

修改服务器启动文件

```

1 // 获取文件路径
2 const resolve = dir => require('path').resolve(__dirname, dir)
3 // 第 1 步：开放dist/client目录，关闭默认下载index页的选项，不然到不了后面路由
4 app.use(express.static(resolve('../dist/client'), {index: false}))
5 // 第 2 步：获得一个createBundleRenderer
6 const { createBundleRenderer } = require("vue-server-renderer");
7 // 第 3 步：服务端打包文件地址
8 const bundle = resolve("../dist/server/vue-ssr-server-bundle.json");
9 // 第 4 步：创建渲染器
10 const renderer = createBundleRenderer(bundle, {
11   runInNewContext: false, // https://ssr.vuejs.org/zh/api/#runinnewcontext
12   template: require('fs').readFileSync(resolve("../public/index.html"),
13 "utf8"), // 宿主文件
14   clientManifest: require(resolve("../dist/client/vue-ssr-clientmanifest.json")) // 客户端清单
15 });
16 app.get('*', async (req, res) =>{
17   // 设置url和title两个重要参数
18   const context = {
19     title:'ssr test',
20     url:req.url
21   }
22   const html = await renderer.renderToString(context);
23   res.send(html)
24 })

```

28.4. 小结

- 使用 `ssr` 不存在单例模式，每次用户请求都会创建一个新的 `vue` 实例
- 实现 `ssr` 需要实现服务端首屏渲染和客户端激活
- 服务端异步获取数据 `asyncData` 可以分为首屏异步获取和切换组件获取
 - 首屏异步获取数据，在服务端预渲染的时候就应该已经完成
 - 切换组件通过 `.mixin` 混入，在 `beforeMount` 钩子完成数据获取

29. vue3有了解过吗？能说说跟vue2的区别吗？



29.1. Vue3介绍

关于 vue3 的重构背景，尤大是这样说的：

「Vue 新版本的理念成型于 2018 年末，当时 Vue 2 的代码库已经有两岁半了。比起通用软件的生命周期来这好像也没那么久，但在这段时期，前端世界已经今昔非比了」

在我们更新（和重写）Vue 的主要版本时，主要考虑两点因素：首先是新的 JavaScript 语言特性在主流浏览器中的受支持水平；其次是当前代码库中随时间推移而逐渐暴露出来的一些设计和架构问题」

简要就是：

- 利用新的语言特性(es6)
- 解决架构问题

29.2. 哪些变化

What's coming in Vue 3.0

- Make it faster
- Make it smaller
- Make it more maintainable
- Make it easier to target native
- Make your life easier

从上图中，我们可以概览 Vue3 的新特性，如下：

- 速度更快
- 体积减少
- 更易维护
- 更接近原生
- 更易使用

29.2.1. 速度更快

vue3 相比 vue2

- 重写了虚拟 Dom 实现
- 编译模板的优化
- 更高效的组件初始化
- `undate` 性能提高1.3~2倍
- `SSR` 速度提高了2~3倍

Performance

- Rewritten virtual dom implementation
- Compiler-informed fast paths
- More efficient component initialization
- 1.3~2x better update performance*
- 2~3x faster SSR*

29.2.2. 体积更小

通过 `webpack` 的 `tree-shaking` 功能，可以将无用模块“剪辑”，仅打包需要的能够 `tree-shaking`，有两大好处：

- 对开发人员，能够对 `vue` 实现更多其他的功能，而不必担忧整体体积过大
- 对使用者，打包出来的包体积变小了

`vue` 可以开发出更多其他的功能，而不必担忧 `vue` 打包出来的整体体积过多

Tree-shaking

- Most optional features (e.g. v-model, <transition>) are now tree-shakable
- Bare-bone HelloWorld size: **13.5kb**
 - 11.75kb with only Composition API support
- All runtime features included: **22.5kb**
 - More features but still lighter than Vue 2

29.2.3. 更易维护

29.2.3.1. compositon Api

- 可与现有的 Options API 一起使用
- 灵活的逻辑组合与复用
- Vue3 模块可以和其他框架搭配使用

Composition API

- Usable alongside existing Options API
- Flexible logic composition and reuse
- Reactivity module can be used as a standalone library

29.2.3.2. 更好的TypeScript支持

VUE3 是基于 typescript 编写的，可以享受到自动的类型定义提示

Better TypeScript Support

- Codebase written in TS w/ auto-generated type definitions
- API is the same in JS and TS
 - In fact, code will also be largely the same
- TSX support
- Class component is still supported ([vue-class-component@next](#) is currently in alpha)

29.2.3.3. 编译器重写

Compiler Rewrite

- Pluggable architecture
- Parser w/ location info (source maps!)
- Serve as infrastructure for more robust IDE support

29.2.4. 更接近原生

可以自定义渲染 API

Custom Renderer API

```
import { createRenderer } from '@vue/runtime-core'

const { render } = createRenderer({
  nodeOps,
  patchData
})
```

29.2.5. 更易使用

响应式 `Api` 暴露出来

Exposed reactivity API

```
import { observable, effect } from 'vue'

const state = observable({
  count: 0
})

effect(() => {
  console.log(`count is: ${state.count}`)
}) // count is: 0

state.count++ // count is: 1
```

轻松识别组件重新渲染原因

Easily identify
why a component is re-rendering

```
const Comp = {
  render(props) {
    return h('div', props.count)
  },
  renderTriggered(event) {
    debugger
  }
}
```

29.3. Vue3新增特性

Vue 3 中需要关注的一些新功能包括：

- framents
- Teleport
- composition Api
- createRenderer

29.3.1. framents

在 `Vue3.x` 中，组件现在支持有多个根节点

```

1  <!-- Layout.vue -->
2  <template>
3  <header>...</header>
4  <main v-bind="$attrs">...</main>
5  <footer>...</footer>
6  </template>

```

29.3.2. Teleport

`Teleport` 是一种能够将我们的模板移动到 `DOM` 中 `Vue app` 之外的其他位置的技术，就有点像哆啦A梦的“任意门”

在 `vue2` 中，像 `modals`，`toast` 等这样的元素，如果我们嵌套在 `Vue` 的某个组件内部，那么处理嵌套组件的定位、`z-index` 和样式就会变得很困难

通过 `Teleport`，我们可以在组件的逻辑位置写模板代码，然后在 `Vue` 应用范围之外渲染它

```

1  <button @click="showToast" class="btn">打开 toast</button>
2  <!-- to 属性就是目标位置 -->
3  <teleport to="#teleport-target">
4    <div v-if="visible" class="toast-wrap">
5      <div class="toast-msg">我是一个 Toast 文案</div>
6    </div>
7  </teleport>

```

29.3.3. createRenderer

通过 `createRenderer`，我们能够构建自定义渲染器，我们能够将 `vue` 的开发模型扩展到其他平台

我们可以将其生成在 `canvas` 画布上

关于 `createRenderer`，我们了解下基本使用，就不展开讲述了

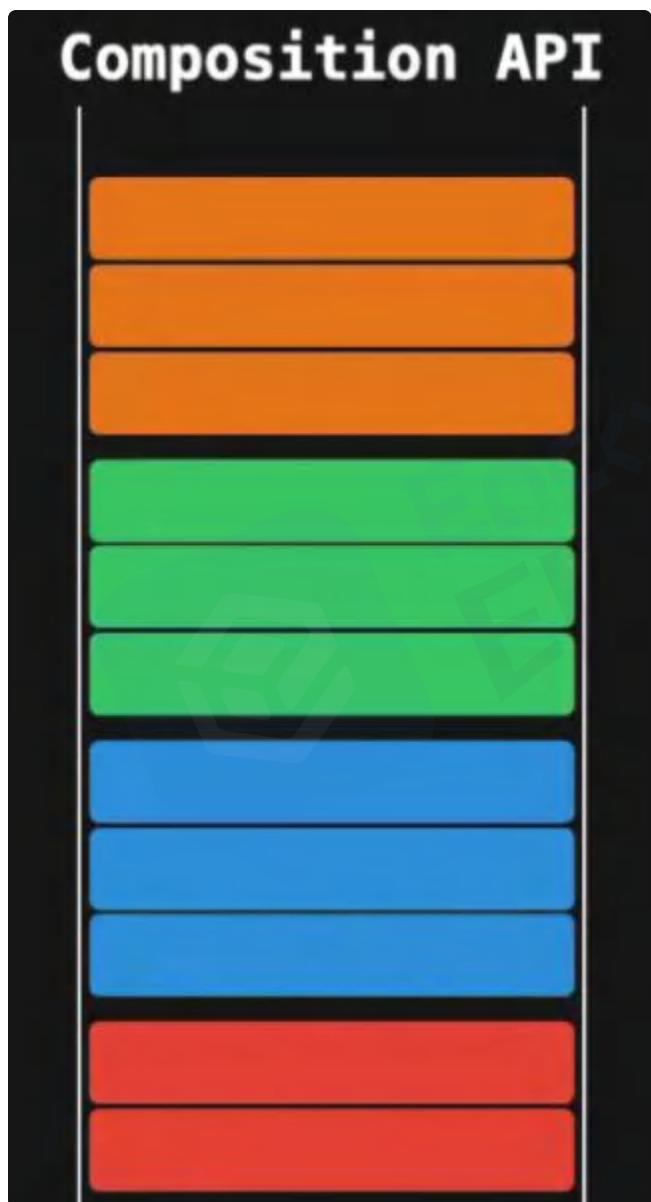
```
1 import { createRenderer } from '@vue/runtime-core'
2
3 const { render, createApp } = createRenderer({
4   patchProp,
5   insert,
6   remove,
7   createElement,
8   // ...
9 })
10
11 export { render, createApp }
12
13 export * from '@vue/runtime-core'
```

JavaScript

复制代码

29.3.4. composition Api

composition Api，也就是组合式 `api`，通过这种形式，我们能够更加容易维护我们的代码，将相同功能的变量进行一个集中式的管理



关于 `compositon api` 的使用，这里以下图展开



简单使用:

```

1  export default {
2    setup() {
3      const count = ref(0)
4      const double = computed(() => count.value * 2)
5      function increment() {
6        count.value++
7      }
8      onMounted(() => console.log('component mounted!'))
9      return {
10        count,
11        double,
12        increment
13      }
14    }
15  }

```

29.3.5. 非兼容变更

29.3.6. Global API

- 全局 `Vue API` 已更改为使用应用程序实例
- 全局和内部 `API` 已经被重构为可 `tree-shakable`

29.3.7. 模板指令

- 组件上 `v-model` 用法已更改
- `<template v-for>` 和非 `v-for` 节点上 `key` 用法已更改
- 在同一元素上使用的 `v-if` 和 `v-for` 优先级已更改
- `v-bind="object"` 现在排序敏感
- `v-for` 中的 `ref` 不再注册 `ref` 数组

29.3.8. 组件

- 只能使用普通函数创建功能组件
- `functional` 属性在单文件组件 (SFC)
- 异步组件现在需要 `defineAsyncComponent` 方法来创建

29.3.9. 渲染函数

- 渲染函数 API 改变
- `$scopedSlots` property 已删除，所有插槽都通过 `$slots` 作为函数暴露
- 自定义指令 API 已更改为与组件生命周期一致
- 一些转换 `class` 被重命名了：
 - `v-enter` -> `v-enter-from`
 - `v-leave` -> `v-leave-from`
- 组件 `watch` 选项和实例方法 `$watch` 不再支持点分隔字符串路径，请改用计算函数作为参数
- 在 `Vue 2.x` 中，应用根容器的 `outerHTML` 将替换为根组件模板（如果根组件没有模板/渲染选项，则最终编译为模板）。`Vue 3.x` 现在使用应用程序容器的 `innerHTML`。

29.3.10. 其他改变

- `destroyed` 生命周期选项被重命名为 `unmounted`
- `beforeDestroy` 生命周期选项被重命名为 `beforeUnmount`
- `[prop default]` 工厂函数不再有权访问 `this` 是上下文
- 自定义指令 API 已更改为与组件生命周期一致
- `data` 应始终声明为函数
- 来自 `mixin` 的 `data` 选项现在可简单地合并
- `attribute` 强制策略已更改
- 一些过渡 `class` 被重命名
- 组建 `watch` 选项和实例方法 `$watch` 不再支持以点分隔的字符串路径。请改用计算属性函数作为参数。
- `<template>` 没有特殊指令的标记 (`v-if/else-if/else`、`v-for` 或 `v-slot`) 现在被视为普通元素，并将生成原生的 `<template>` 元素，而不是渲染其内部内容。
- 在 `Vue 2.x` 中，应用根容器的 `outerHTML` 将替换为根组件模板（如果根组件没有模板/渲染选项，则最终编译为模板）。`Vue 3.x` 现在使用应用程序容器的 `innerHTML`，这意味着容器本身不再被视为模板的一部分。

29.3.11. 移除 API

- `keyCode` 支持作为 `v-on` 的修饰符
- `$on`，`$off` 和 `$once` 实例方法

- 过滤 `filter`
- 内联模板 `attribute`
- `$destroy` 实例方法。用户不应再手动管理单个 `Vue` 组件的生命周期。