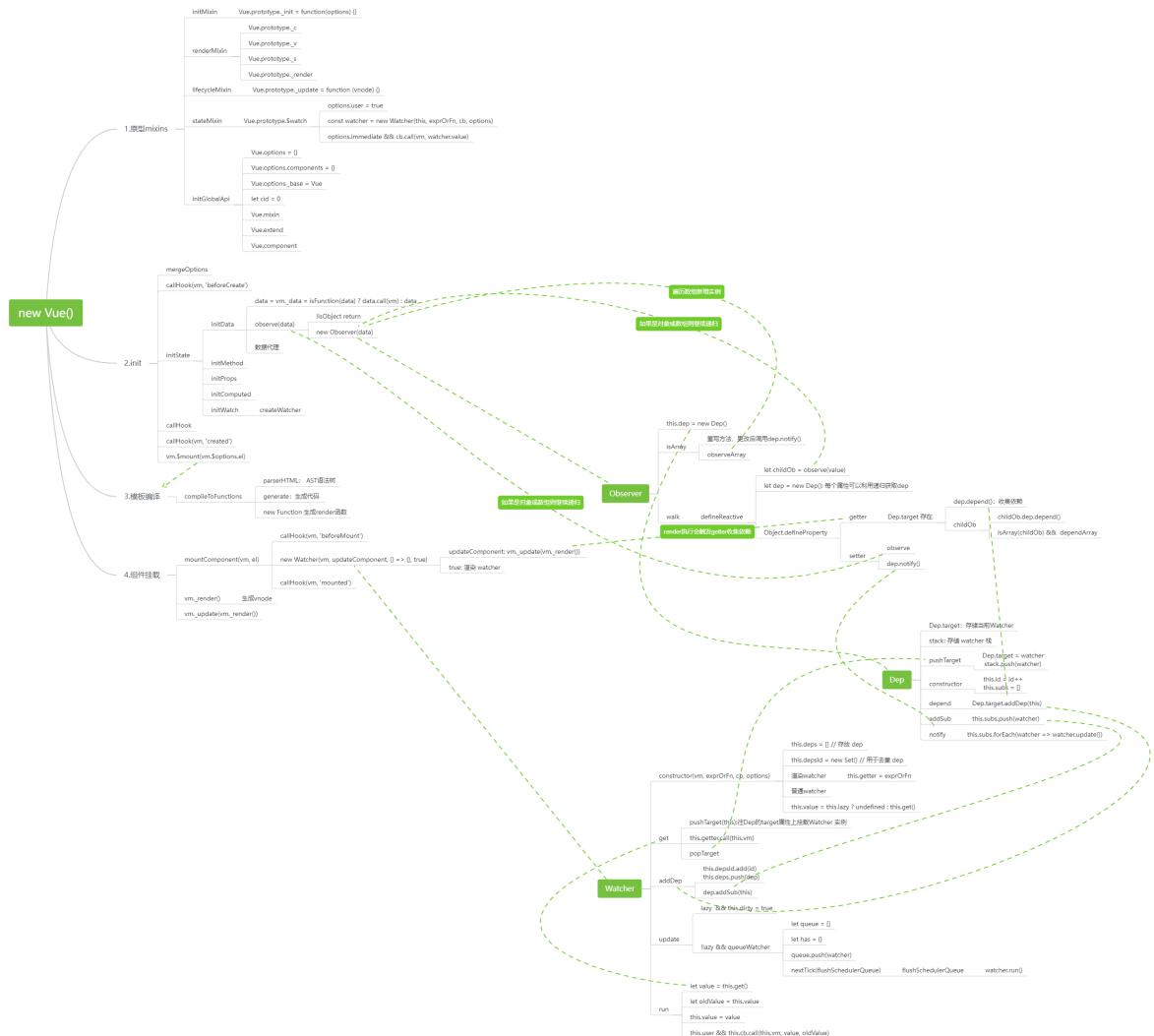


vue

1.vue2运行机制



1. `initData`: 遍历 `data` 属性，如果是数组或对象则递归遍历，对每个属性使用 `Object.defineProperty`，同时生成一个 `dep` 实例，用于收集依赖与触发更新。
2. `initWatcher`: 遍历 `new Watcher`, `pushTarget(Dep.target)` 记录当前 `watcher`, `state` 记录 `watcher` 栈, `call` 被监听属性，触发 `getter`。

 1. `dep depend`: `dep.depend => Dep.target.addDep(this)` 调用当前 `watcher` 的 `addDep`
 2. `watcher addDep`: `this.deps.push(dep)` 往 `deps` 存储 `dep`, 调用 `dep.addSub(this)`
 3. `dep addSub`: 往 `subs` 存储 `watcher`

3. 模板解析：解析 `html` 模板，生成 `AST` 语法树=> 生成渲染函数。
4. 组件挂载：`new Watcher`, 记录渲染 `watcher`, 执行渲染函数(触发 `getter` 收集依赖)生成 `vnode`, 通过 `vnode` 的 `patch` 生成元素并挂载。
5. 当第一次渲染后，如果修改了值，则会触发 `setter`=> `dep.notify => 把 subs 中的每一个 watcher 执行 update => 收集 watcher 队列, nextTick 中执行 watcher.run 触发重新 render 与 patch。`

2. 双向绑定原理

1. 使用 数据劫持 结合 发布订阅模式 实现。
2. 对 data 数据进行递归遍历，加上 `setter` 和 `getter`，生成一个 `dep` 实例用于收集依赖与触发更新。
3. `compiler` 解析模板，生成 `AST` 语法树，生成 `render` 函数，执行 `render` 函数触发 `getter` 为数据添加订阅者 `watcher`。绑定节点的更新函数，触发更改数据。
4. `watcher` 在数据更改后，`dep` 通知 `watcher` 去执行其对应的回调函数，完成页面更新。

3.Object.defineProperty 的缺点

- 对象新增属性无法劫持，必须改变整个对象重新劫持。
- 数组通过下标修改和其他操作，这里重写了内部方法，更改值后调用属性的dep.notify()发布更新通知。
- vue3.0使用 proxy 对对象数组进行代理，从而实现数据劫持。唯一的缺点就是兼容性问题。

4.MVVM

- Model：数据模型，数据和业务逻辑在此定义
- View: UI视图，负责数据的展示
- ViewModel：负责监听Model中数据的改变并且控制之视图的更新，处理用户交互操作
- Model和View并无直接关联，而是通过ViewModel来进行联系的，Model和ViewModel之间有着双向数据绑定的联系。因此当Model中的数据改变时会触发View层的刷新，View中由于用户交互操作而改变的数据也会在Model中同步。

5.computed 和 watch 的区别？

- 缓存
 - computed 支持缓存，只有依赖的数据发生变化才会重新计算。
 - watch 不支持，数据变化就会触发操作
- 异步
 - computed 不支持异步（下方直接返回一个Promise）

```
asyncComputed() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(this.counter * 2)
        }, 1000)
    });
}
```

- watch 支持（例如监听值变化发起请求）
- 场景
 - computed 用于数据计算，依赖于其他值时，利用缓存特性避免重新计算
 - watch 数据变化时异步或开销较大的操作。
- 使用
 - computed 默认getter，可以设置getter和setter
 - watch 可以设置 deep(深度监听)、immediate (加载组件立即触发)

6.slot

- Vue 内容分发机制，是子组件的一个模板标签元素，其显示和如何显示由父组件决定。
- slot分类
 - 默认插槽，没有指定name属性
 - 具名插槽

```
// 定义
<slot name='xxx'>

// 使用
<template #xxx></template>
<template v-slot:xxx></template>
```

- 作用域插槽

```
// 定义
<slot :item="item" v-for="item in list" />

// 使用
<list v-slot="props">
  <span>{{props.item}}</span>
</list>
// 使用（解构）
<list v-slot="{item}">
  <span>{{item}}</span>
</list>
```

- 原理

- 组件实例化时获取父组件传入的slot标签内容，存放在 `vm.$slot` 中
- 执行渲染函数时，遇到slot标签，用 `vm.$slot` 中的内容进行替换

7.如何保存页面当前状态

A=> B B=>A :只有从B页面返回A页面，A页面才使用原来的状态

- 前组件卸载
 - 使用 `localStorage/sessionStorage` 存储
 - 优点：兼容性好，简单
 - 缺点：数据JSON化的时候需要考虑兼容情况（Date对象等），需要增加flag判断。
 - 使用 `vuex` 存储
 - 需要记录从哪个页面而来，可以使用`VueRouter`守卫来获取

```
// main.js

Vue.prototype.$previousRoute = null
router.beforeEach((to, from, next) => {
  Vue.prototype.$previousRoute = from
  next()
})
new Vue({
  router,
  store,
  render: h => h(App)
}).$mount('#app')
```

- 前组件不卸载

- `keep-alive`
 - `activated`

- deactivated

```
<keep-alive
  :include="include"
  :exclude="exclude"
  :max="max"
>
  <router-view />
</keep-alive>
```

8.常见事件修饰符

- .stop 等同于 `event.stopPropagation()`, 防止冒泡
- .prevent 等同于 `event.preventDefault()` 阻止默认行为
- .capture: 与事件冒泡方向相反
- .self: 只触发自己范围内的事件, 不包含子元素
- .once: 只触发一次

9.v-if、v-show、v-html原理

- v-if 生成vnode的时候忽略对应节点, render的时候就不会渲染
- v-show 正常生成vnode, render时候渲染真实节点, render过程中根据show的属性值修改 `display`
- 设置节点的innerHTML 属性为 v-html 的值

10.v-if和v-show的区别

- 手段: v-if 动态添加删除DOM元素, v-show动态修改display属性
- 编译条件: v-if 只有真时才编译生成DOM, v-show无论真假都会编译DOM
- 性能消耗: v-if 更高的切换消耗 v-show 更高的初始渲染消耗
- 使用场景: v-show 适合频繁切换 v-if 相反

11.v-model 语法糖

```
<CustomInput v-model="searchText"/>
<CustomInput
  :value="searchText"
  @input="newValue => searchText = newValue"
/>
emit('input', xxxx)
```

```
// 默认
<CustomInput v-model="searchText"/>
<CustomInput
  :modelValue="searchText"
  @update:modelValue="newValue => searchText = newValue"
/>
emit('update:modelValue', xxxx)

// 自定义绑定属性
<MyComponent v-model:title="bookTitle" />
<MyComponent
  :title="bookTitle"
  @update:title="newValue => bookTitle = newValue"
```

```
/>  
$emit('update:title', xxxx)
```

12. data 为什么是一个函数而不是一个对象

- 对象是引用类型，会被所有实例化的组件所共享，导致他们引用同一个数据对象
- 采用函数的形式，再initData的时候将其作为工厂函数提供全新的对象

13.\$nextTick 的原理及作用

```
let callbacks = []  
  
function flushCallbacks () {  
    callbacks.forEach(cb => cb())  
}  
  
let timerFunc  
let waiting = false  
if (Promise) {  
    timerFunc = () => {  
        Promise.resolve().then(flushCallbacks)  
    }  
} else if (MutationObserver) {  
    let observe = new MutationObserver(flushCallbacks)  
    let textNode = document.createTextNode(1)  
    observe.observe(textNode, {  
        characterData: true  
    })  
    timerFunc = () => {  
        textNode.textContent = 2  
    }  
} else if (setImmediate) {  
    timerFunc = () => {  
        setImmediate(flushCallbacks)  
    }  
} else {  
    timerFunc = () => {  
        setTimeout(flushCallbacks, 0)  
    }  
}  
  
export function nextTick (cb) {  
    callbacks.push(cb)  
    if (!waiting) {  
        timerFunc()  
        waiting = true  
    }  
}
```

- nextTick 所在作用域维护一个回调函数的栈
- 利用 `Promise`、`MutationObserver`、`setImmediate`、`setTimeout`，这里Promise是属于微任务，可以在同一事件循环迭代当前宏任务结束之后立即执行，其他的是宏任务，在下一事件循环中执行。所以这里Promise优先级更高
- 引入这种异步更新机制的原因

- 同步更新会频繁触发DOM渲染（减少渲染）
- 由于虚拟DOM，状态的变更会导致频繁计算（减少计算）

14. 给对象添加新属性

- 视图没有更新，因为属性没有被转换为响应式数据（没有收集依赖）
- `this.$set(target, key, value)`

15. Vue template 到 render 过程

`template => ast(抽象语法树) => render 函数`

16. vue中封装的数组方法有哪些？如何实现页面更新

- 数组
 - `push`
 - `pop`
 - `shift`
 - `unshift`
 - `splice`
 - `sort`
 - `reverse`
- 调用后返回结果之前，通过`dep.notify()`触发依赖更新。

17. 自定义指令

- 对普通DOM元素进行底层操作
- 钩子函数（vue2）
 - `bind`
 - `inserted`
 - `update`
 - `ComponentUpdate`
 - `unbind`
- 钩子函数（vue3）
 - `created`
 - `beforeMount`
 - `mounted`
 - `beforeUpdate`
 - `beforeUnmount`
 - `unmounted`

18. 子组件可以直接改变父组件数据吗？

- 只可以父级 `props` 更新留向子组件，每次更新时子组件中所有 `props` 都会刷新
- 防止意外改变父组件状态，增加 degger 成本
- 只能通过 `$emit` 派发自定义事件由父组件修改

19. v-if 和 v-for 优先级

- v-for 优先级更高
- 在循环中根据v-if 来确定是否渲染该元素，如果v-if 是假，则跳过

```
<div v-for="item in items" v-if="item.isActive">
  {{ item.name }}
</div>
// 在上述示例中，v-for会遍历items数组中的每个元素，并且只有当item.isActive为真时，才会渲染对应的<div>元素。
```

- vue2 中 v-for > v-if
- vue3 中 v-if > v-for

20. vue 初始化页面闪动问题

代码还没解析好看到模板的问题

```
[v-cloak] {
  display: none;
}
```

21. SPA 优缺点

- SPA
 - 页面初始化时加载相应资源，加载完成后不会因为用户操作进行页面的重新加载和跳转，利用路由机制实现内容变化与交互。
- 优点
 - 体验好，内容变化不需要重新加载，避免不必要的跳转和重复渲染
 - 服务器压力小
 - 前后端职责分离
- 缺点
 - 白屏时间长
 - 前端需要自行建立路由管理
 - SEO难度大：所有内容在一个页面中动态替换显示

22. 虚拟DOM

- 通过js对象的方式描述DOM结构
- 优点
 - 跨平台
 - 性能优化，收集变更再对比刷新
 - 简化开发，无需操作真是DOM

23. diff 算法

1. 标签名不一样，直接替换
2. 如果标签一样比较属性
3. 比较双方儿子
 - 老的没儿子新的有儿子循环创建

- 老的有儿子新的没儿子删除老节点
- 双方都有儿子节点，双指针比较
 - oldStartIndex: 旧头下标, oldStartVnode
 - oldEndIndex: 旧尾下标, oldEndVnode
 - newStartIndex: 新头下标, newStartVnode
 - newEndIndex: 新尾下标, newEndVnode
 - 头头比较, 如果相同头开始指针后移, 更新对应的vnode
 - 尾尾比较, 如果相同结束指针前移, 更新对应的vnode
 - 旧头新尾比较, 如果相同, 旧头后移新尾前移
 - 旧尾新头比较, 如果相同, 旧尾前移新头后移
 - 乱序比较
 - 通过旧列表创建key=》index映射, 通过新头指针对应节点的key找到对应的index
 - 如果找不到, 直接创建新的节点插入老节点的头指针处
 - 如果找到, 移动该节点到旧头指针处
- 如果一方有一方没有
 - 新的还有旧的没有
 - 旧的还有新的没有

24. key 的作用

作为vnode的唯一标记, 可以使diff操作更准确与高效, 确保数据更新时进行最小化的DOM操作与正确的节点重用。

25.vue生命周期

开始创建=》初始化数据=》模板编译=》挂载DOM=>渲染更新=》渲染、卸载一系列流程

1. beforeCreate
 - 数据观察和初始化事件还没开始, 无法访问到data、computed等数据和方法
2. created
 - 实例创建完成, 已经可以访问data等数据和方法, 但是没有挂载到DOM
3. beforeMount
 - 挂载之前被调用, 已经对模板解析得到了render函数
4. mounted
 - 完成了调用render函数生成vnode, 把vnode转换成html替换内容
5. beforeUpdate
 - 响应式数据更新时调用, UI还没更新
6. updated
 - UI已经更新
7. beforeDestory
 - 实例销毁前调用, 服务端渲染不被调用
8. destoryed
 - 实例销毁后调用, 服务端渲染不被调用

26.父子组件执行顺序

- 加载渲染过程
 1. 父组件 beforeCreate
 2. 父组件 created
 3. 父组件 beforeMount
 4. 子组件 beforeCreate
 5. 子组件 created
 6. 子组件 beforeMount
 7. 子组件 mounted
 8. 父组件 mounted
- 更新过程
 1. 父组件 beforeUpdate
 2. 子组件 beforeCreate
 3. 子组件 updated
 4. 父组件 updated
- 销毁
 1. 父组件 beforeDestory
 2. 子组件 beforeDestory
 3. 子组件 destoryed
 4. 父组件 destoryed

27.组件通信

- 父子组件传值 (props/\$emit)
- eventBus事件总线 (\$emit/\$on)
 1. 创建事件中心

```
import Vue from 'vue'  
export const EventBus = new Vue()
```

2. 发送事件

```
import {EventBus} from './event-bus.js'  
EventBus.$emit('xxx', {...})
```

3. 接收事件

```
import {EventBus} from './event-bus.js'  
EventBus.$on('xxx', {...})
```

- 依赖注入 (provide/inject)
- ref / \$refs
- \$parent / \$children
- \$attrs/\$listeners (A=>B=>C 跨代传递)
 - `inheritAttrs` 默认只继承除 `pros` 外的所有属性，如果为 `false` 则只继承 `class` 属性
 - `$attrs` 继承所有的父组件属性 (除 `props`、`class`、`style`) (`v-bind="$attrs"`)
 - `$listeners` 继承所有父组件的自定义监听器 (`v-on="$listeners"`)

Vue3

vue3新特性

1. 性能提升

- 响应式性能提升
- diff算法优化（增加静态标志）
- 模板编译优化，不参与更新元素只创建一次
- 更高效组件初始化

2. 更好的ts支持，提供了更好的类型检查

3. Composition API

- 更好的代码组织形式，避免来回切换
- 更好的组件件代码复用，避免vue2 mixin的变量冲突、来源不清晰的缺点

4. 新增组件

- Fragment：不再限制 template 只有一个根节点
- Teleport：传送门，允许我们将控制的内容传送到任意的 DOM 中
- Suspense：等待异步组件时渲染一些额外的内容，让应用有更好的用户体验

响应式原理

- 初始化阶段：初始化阶段通过组件初始化方法形成对应的 proxy 对象，然后形成一个负责渲染的 effect
- get 依赖收集阶段：通过解析 template，替换真实 data 属性，来触发 get，然后通过 track 方法，通过 proxy 对象和 key 形成对应的 deps，将负责渲染的 effect 存入 deps
- set 派发更新阶段：触发 setter 的时候通过 effect 通知 effect 更新

v-if、v-for 优先级

v-if 的优先级更高。

setup

- 属性和方法无需返回，可以直接使用
- 引入组件自动注册
- 使用 defineProps 接收父组件传递值, defineEmits 获取自定义事件。
- useAttrs 获取属性，useSlots 获取插槽。
- 默认不会对外暴露任何属性，如果有需要可使用 defineExpose

通信方式

- props/\$emit
- expose/ref
- \$attrs
- provide/inject
- mitt

ref 与 reactive的区别?

- ref 在js中需要通过.value 使用
- ref 可以接收原始数据类型和引用数据类型，其判断是非原始数据类型使用reactive封装
- reactive 只能接收引用数据类型

vueRouter

路由懒加载

使用箭头函数+import

1.hash和history模式

1. 原理

- hash: hashchange
- history: pushState、 replaceState

2. 特点:

- hash值出现在url里面，改变hash值不会重新加载页面，兼容性好
- 比hash值好看，需要后端服务器配置正确的入口文件避免出现404

2.\$route 和 \$router 区别

- \$route: 路由信息对象，包括 path、 params、 hash、 name、 query等信息
- \$router: 路由实例，包括路由的跳转方法

3.动态路由

1. param

- 配置: /router/:id
- 跳转

```
<router-link :to="{name: 'users', params: {...}}">replace 用户</router-link>
this.$router.push({name: 'users', params:{}})
```

- 参数获取: this.\$route.params.userid

2. query

- 配置: /router
- 跳转

```
<router-link :to="{name: 'users', query: {...}}">replace 用户</router-link>
this.$router.push({name: 'users', query:{}})
```

- 参数获取: this.\$route.query

3. 区别: params在浏览器地址中不显示参数，刷新会丢失路由数据

4. 生命周期

- 全局路由钩子
 - router.beforeEach: 路由切换时触发，用于是否允许路由切换
 - router.beforeResolve: 路由解析完成前，用于额外逻辑处理如加载数据、修改参数
 - router.afterEach: 进入路由之后
- 路由独享钩子
 - beforeEnter: 进入某个具体路由前执行
- 组件内守卫
 - beforeRouteEnter: 进入组件前触发，不能获取this
 - beforeRouteUpdate: 当前地址改变并且组件被服用触发，如foo/:id
 - beforeRouteLeave: 离开组件被调用
- 完整的导航解析流程
 1. 导航被触发
 2. 在失活的组件里调用 beforeRouteLeave
 3. 调用全局 beforeEach
 4. 重用的组件调用 beforeRouteUpdate
 5. 路由配置调用 beforeEnter
 6. 解析异步路由组件
 7. 被激活的组件调用 beforeRouterEnter
 8. 调用全局 beforeResolve
 9. 导航被确认
 10. 调用全局的 afterEach
 11. 触发 DOM 更新
 12. 调用 beforeRouteEnter

存储

vuex

- 属性
 - state
 - getters
 - mutations
 - actions
 - modules
- 操作方法
 - commit 提交 mutation
 - dispatch 提交 action
- 思想
 - 单一数据源
 - 变化可以预测
- 辅助函数
 - mapState
 - mapGetters
 - mapMutations
 - mapActions

```
computed: {
  ...mapState(['name']),
  ...mapGetters(['myAge'])
}
methods: {
  ...mapMutations(['changeAge'])
  ...mapActions(['changeAge'])
}
```

pina

- 属性
 - state: setup: `ref()` 就是 `state` 属性
 - getters: setup: `computed()` 就是 `getters`
 - actions: setup: `function()` 就是 `actions`
- 定义 `defineStore`

```
export const useAlertsStore = defineStore('alerts', {
  state: () => {},
  getters: {},
  actions: {}
})

export const useCounterStore = defineStore('counter', () => {
  const count = ref(0)
  function increment() {
  }
  return { count, increment }
})
```

- 使用

```
// 使用
const store = useCounterStore()
// 解构 state 需要使用 storeToRefs
const { name, doubleCount } = storeToRefs(store)
// 解构 action 可以直接解构
const { increment } = store
```