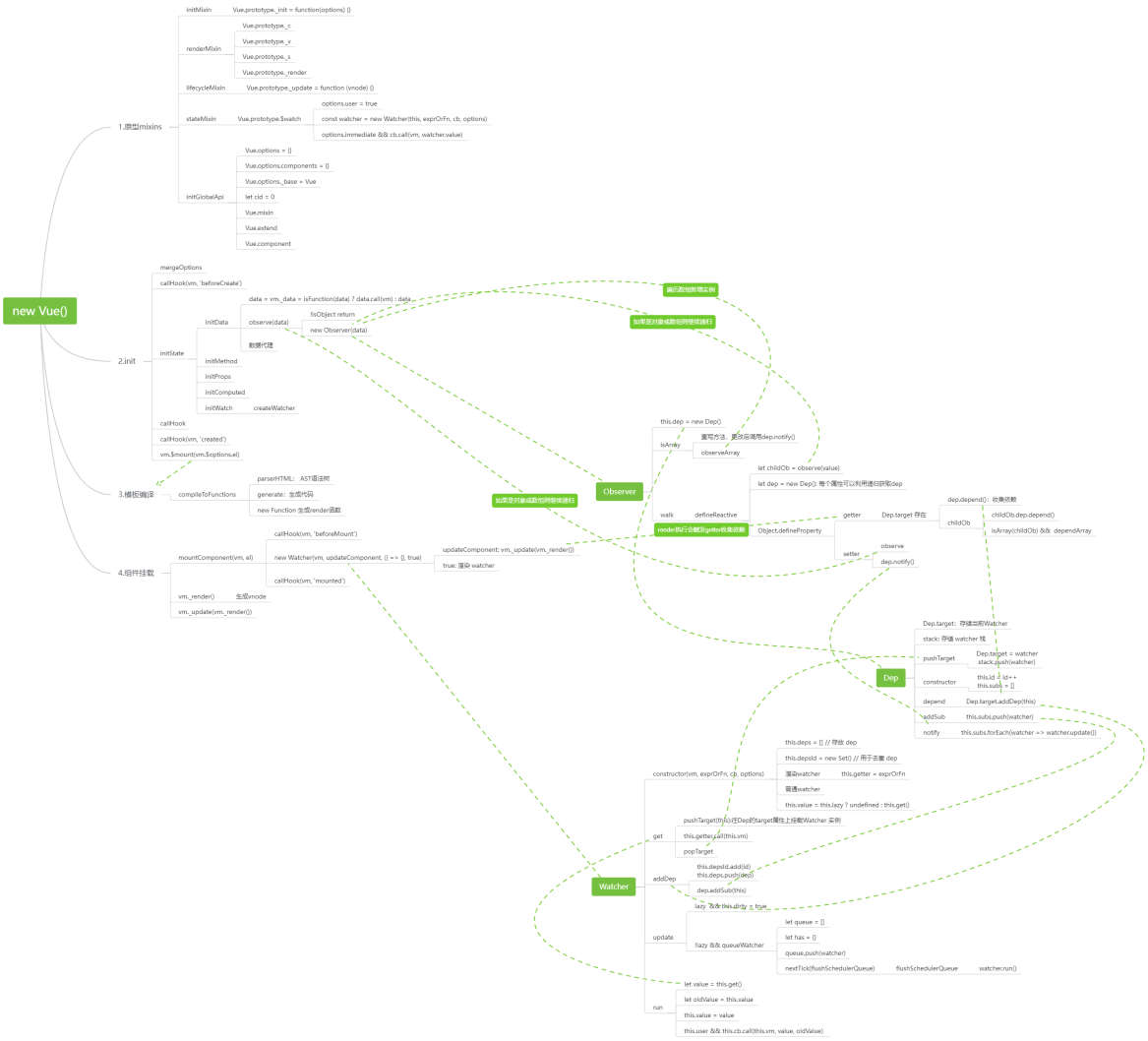
vue

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股之视图的更新，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', xxx)
```

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

// 自定义绑定属性
<MyComponent v-model:title="bookTitle" />
<MyComponent
  :title="bookTitle"
  @update:title="newValue => bookTitle = newValue"
/>
$emit('update:title', xxx)
```

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-fo 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。
```

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

生命周期

1.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

2.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

3.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"）