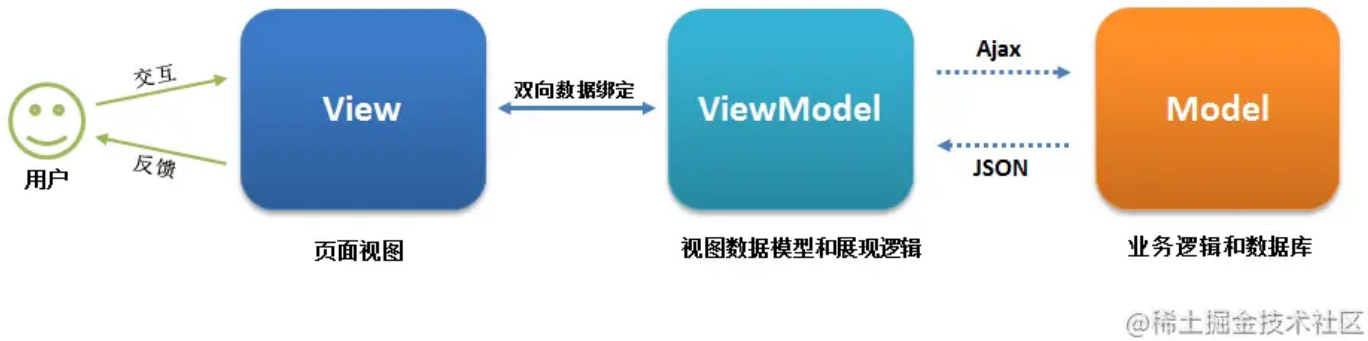
Vue 面试题

什么是 MVVM?

Model-View-ViewModel（MVVM）是一个软甲架构设计模式，由微软 WPF 和 Silverlight 的架构师 Ken Cooper 和 Ted Peters 开发，是一种简化用户界面的事件驱动编程方式。由 John Gossman（同样也是 WPF 和 Silverlight 的架构师）于2005年在他的博客上发表

MVVM 源自于经典的 Model–View–Controller（MVC）模式，MVVM 的出现促进了前端开发与后端业务逻辑的分离，极大地提高了前端开发效率，MVVM 的核心是 ViewModel 层，它就像是一个中转站（value converter），负责转换 Model 中的数据对象来让数据变得更容易管理和使用，该层向上与视图层进行双向数据绑定，向下与 Model 层通过接口请求进行数据交互，起呈上启下作用。如下图所示：



（1）View 层

View 是视图层，也就是用户界面。前端主要由 HTML 和 CSS 来构建 。

（2）Model 层

Model 是指数据模型，泛指后端进行的各种业务逻辑处理和数据操控，对于前端来说就是后端提供的 api 接口。

（3）ViewModel 层

ViewModel 是由前端开发人员组织生成和维护的视图数据层。在这一层，前端开发者对从后端获取的 Model 数据进行转换处理，做二次封装，以生成符合 View 层使用预期的视图数据模型。需要注意的是 ViewModel 所封装出来的数据模型包括视图的状态和行为两部分，而 Model 层的数据模型是只包含状态的，比如页面的这一块展示什么，而页面加载进来时发生什么，点击这一块发生什么，这一块滚动时发生什么这些都属于视图行为（交互），视图状态和行为都封装在了 ViewModel 里。这样的封装使得 ViewModel 可以完整地去描述 View 层。

MVVM 框架实现了双向绑定，这样 ViewModel 的内容会实时展现在 View 层，前端开发者再也不必低效又麻烦地通过操纵 DOM 去更新视图，MVVM 框架已经把最脏最累的一块做好了，我们开发者只需要处理和维护 ViewModel，更新数据视图就会自动得到相应更新。这样 View 层展现的不是

Model 层的数据，而是 ViewModel 的数据，由 ViewModel 负责与 Model 层交互，这就完全解耦了 View 层和 Model 层，这个解耦是至关重要的，它是前后端分离方案实施的重要一环。

说说你对SPA单页面的理解，它的优缺点分别是什么?

SPA（ single-page application ）仅在 Web 页面初始化时加载相应的 HTML、JavaScript 和 CSS。一旦页面加载完成，SPA 不会因为用户的操作而进行页面的重新加载或跳转；取而代之的是利用路由机制实现 HTML 内容的变换，UI 与用户的交互，避免页面的重新加载

优点:

- 用户体验好、快，内容的改变不需要重新加载整个页面，避免了不必要的跳转和重复渲染
- 基于上面一点，SPA 相对对服务器压力小
- 前后端职责分离，架构清晰，前端进行交互逻辑，后端负责数据处理

缺点:

- 初次加载耗时多：为实现单页 Web 应用功能及显示效果，需要在加载页面的时候将 JavaScript、CSS 统一加载，部分页面按需加载
- 前进后退路由管理：由于单页应用在一个页面中显示所有的内容，所以不能使用浏览器的前进后退功能，所有的页面切换需要自己建立堆栈管理
- SEO 难度较大：由于所有的内容都在一个页面中动态替换显示，所以在 SEO 上其有着天然的弱势

说说Vue的优缺点

优点:

- 数据驱动视图，对真实 DOM 进行抽象出 virtual dom（本质就是一个js对象），并配合 diff 算法、响应式和观察者、异步队列等手段以最小代价更新 DOM，渲染页面
- 组件化，组件用单文件的形式进行代码的组织编写，使得我们可以在一个文件里编写 html/css/js（scoped 属性配置 css 隔离）并且配合 Vue-loader 之后，支持更强大的预处理器等功能
- 强大且丰富的 API 提供一系列的 api 能满足业务开发中各类需求
- 生命周期钩子函数，选项式的代码组织方式
- 生态好，社区活跃

缺点:

- 由于底层基于 Object.defineProperty 实现响应式，而这个 api 本身不支持 IEIE8 及以下浏览器
- csr 的先天不足，首屏性能问题（白屏）
- 由于百度等搜索引擎爬虫无法爬取 js 中的内容，所以 spa 先天对 seo 优化心有余而力不足

Vue中用过哪些修饰器?

- 事件修饰符

    - .stop
    - .prevent
    - .capture
    - .self
    - .once
    - .passive
- 按键修饰符
    - .enter
    - .tab
    - .delete
    - .esc
    - .space
    - .up
    - .down
    - .left
    - .right
- 其他常用的修饰符
    - .trim
    - .number
    - .lazy
    - .sync

说一下 Vue 自带的指令

- v-text：更新元素的内容
- v-html：更新元素的 innerHTML
- v-show：切换元素的 display
- v-if：根据表达式的真假值来有条件地渲染元素
- v-else：前一兄弟元素必须有 v-if 或 v-else-if
- v-else-if：前一兄弟元素必须有 v-if 或 v-else-if
- v-for：基于源数据多次渲染元素或模板块
- v-on：绑定事件监听器
- v-bind：动态绑定一个或多个attribute，或一个组件 prop 到表达式
- v-model：在表单控件或者组件上创建双向绑定
- v-slot：提供具名插槽或需要接收 prop 的插槽
- v-pre：跳过这个元素和它的子元素的编译过程

- v-cloak：这个指令保持在元素上直到关联组件实例结束编译
- v-once：只渲染元素和组件**一次**。随后的重新渲染，元素/组件及其所有的子节点将被视为静态内容并跳过。这可以用于优化更新性能。
- v-memo：记住一个模板的子树。元素和组件上都可以使用。该指令接收一个固定长度的数组作为依赖值进行记忆比对。如果数组中的每个值都和上次渲染的时候相同，则整个该子树的更新会被跳过。

说一下 Vue 的内置组件?

- component：渲染一个“元组件”为动态组件
- transtion：作为单个元素/组件的过渡效果。<transition > 只会把过渡效果应用到其包裹的内容上，而不会额外渲染 DOM 元素，也不会出现在可被检查的组件层级中
- transtion-group：提供多个元素/组件的国度效果
- 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
- slot：作为组件模板之中的内容分发插槽。<slot> 元素自身将被替换
- teleport：允许我们控制在 DOM 中哪个父节点渲染HTML

slot 是什么？有什么作用？原理是什么?

slot 又名插槽，是 Vue 的内容分发机制，组件内部的模板引擎使用 slot 元素作为承载分发内容的出口。插槽 slot 是子组件的一个模板标签元素，而这一个标签元素是否显示，以及怎么显示是由父组件决定的。

slot 又分三类，默认插槽、具名插槽和作用域插槽

- 默认插槽：又名匿名插槽，当slot没有指定name属性值的时候一个默认显示插槽，一个组件内只有一个匿名插槽
- 具名插槽：带有具体名字的插槽，也就是带有name属性的slot，一个组件可以出现多个具名插槽
- 作用域插槽：默认插槽、具名插槽的一个辩题，可以是匿名插槽，也可以是具名插槽，该插槽的不同点是在子组件渲染作用域插槽时，可以将子组件内部的数据传递给父组件，让父组件根据子组件传递过来的数据决定如何渲染该插槽

实现原理：

当子组件vm实例化时，获取到父组件传入的slot标签的内容，存放在 vm.$slot 中，默认插槽为 vm.$slot.default，具名插槽为 vm.$slot.xxx，xxx 为插槽名，当组件执行渲染函数时，遇到 slot 标签，使用 $slot 中的内容进行替换，此时可以为插槽传递数据，若存在数据，则可称该插槽为作用域插槽

v-show 与 v-if 有什么区别?

v-if 是真正的条件渲染，因为它会确保在切换过程中条件快内的事件监听器和子组件适当地被销毁和重建；也是惰性的。如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块

v-show 就简单得多了——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于CSS的“display”属性进行切换

所以，v-if 适用于在运行时很少改变条件，不需要频繁切换的场景；v-show 则适用于需要非常频繁切换条件的场景

为什么 v-for 和 v-if 不建议用在一起?

- 当 v-for 和 v-if 处于同一个节点时，v-for 的优先级比 v-if 更高。这意味着 v-if 将分别重复运行每个 v-for 循环中。如果要遍历的数组很大，而真正要展示的数据很少时，这将造成很大的性能浪费
- 这种场景建议使用 computed，先对数据进行过滤

怎样理解 Vue 的单向数据流?

所有的 prop 都使得其父子 prop 之间形成了一个**单向下行绑定**：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解

额外的，每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。这意味着你不应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器的控制台中发出警告。子组件想修改时，只能通过 $emit 派发一个自定义事件，父组件接收到后，由父组件修改。

computed 和 watch 的区别和运用的场景?

computed：计算属性，依赖其他属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变了，下一次获取 computed 的值时才会重新计算 computed 的值

watch：更多的是“观察”的作用，类似于某些数据的监听回调，每当监听的数据变化时都会执行回调进行后续操作

运用场景：

- 当我们需要进行数值计算，并且依赖于其它数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算；
- 当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，使用 watch 选项允许我们执行异步操作 (访问一个 API)，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

注意点：

- computed 和 watch 在源码都是通过 Watcher 类创建出来的
- 初始化时，先创建 computed 再创建 watch。数据改变时，先执行 computed 再执行 watch

直接给一个数组项赋值，Vue能检测到变化吗?

由于 JavaScript 的限制，Vue 不能检测到以下数组的变动：

- 你利用索引直接设置一个数组项时，例如：vm.items[indexOfItem] = newValue
- 当你修改数组的长度时，例如：vm.items.length = newLength

为了解决第一个问题，Vue 提供了以下的操作方法：

```
// Vue.set
Vue.set(vm.items, indexOfItem, newValue)
// vm.$set, Vue.set的一个别名
vm.$set(vm.items, indexOfItem, newValue)
// Array.prototype.splice
vm.items.splice(indexOfItem, 1, newValue)
```

为了解决第二个问题，Vue提供了以下操作方法：

```
// Array.prototype.splice
vm.items.splice(newLength)
```

vm.$set 的实现原理：

- 如果目标是数组，直接使用数组的 splice 方法触发响应式
- 如果目标是对象，会先判断属性是否存在、对象是否是响应式，最终如果要对属性进行响应式处理，则是通过调用 defineReactive 方法进行响应式处理（defineReactive 方法就是 Vue 在初始化对象时，给对象属性采用 Object.defineProperty 动态添加 getter 和 setter 的功能所调用的方法）

淡淡你对 Vue 生命周期的理解?

（1）生命周期是什么?

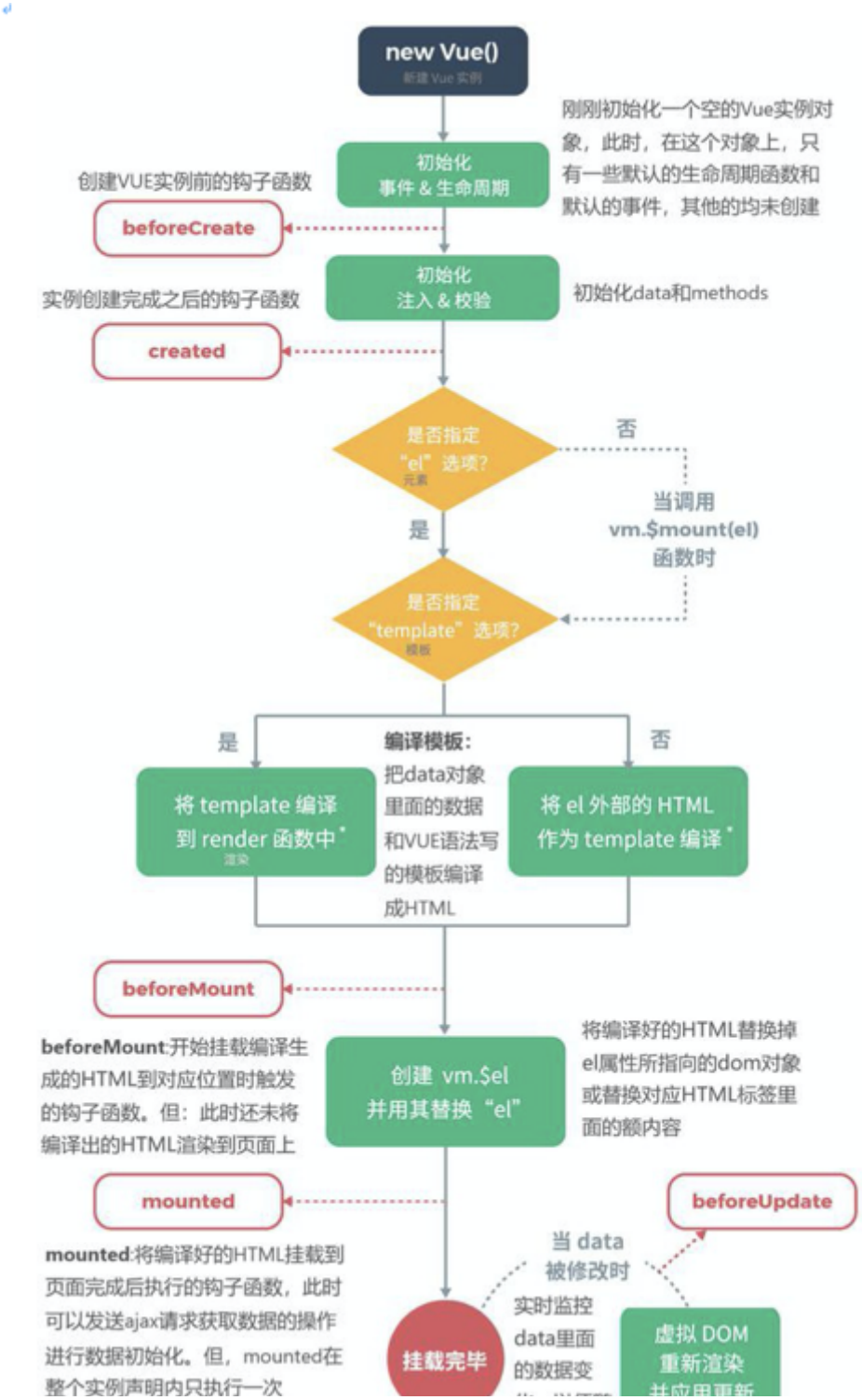
Vue 实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模板、挂载DOM->渲染、更新->渲染、卸载等一系列过程，我们称这是Vue的生命周期

（2）各个生命周期的作用

生命周期	描述
beforeCreate	在实例初始化之后、进行数据侦听和事件/侦听器

	的配置之前同步调用
created	在实例创建完成后被立即同步调用。在这一步中，实例已完成对选项的处理，意味着以下内容已被配置完毕：数据侦听、计算属性、方法、事件/侦听器的回调函数。然而，挂载节点还没开始，且 $el property 目前尚不可用
beforeMount	在挂载开始之前被调用，相关的 render 函数首次被调用，该钩子在服务端渲染期间不被调用
mounted	在实例挂载完成后被调用，这时候传递 app.mount 的元素已经被新创建的 vm.$el 替换了。如果根实例被挂载到了一个文档内的元素上，当 mounted 被调用时，vm.$el 也会在文档内。注意：mounted 不会保证所有的子组件也都被挂载完成，如果你希望等待整个视图都渲染完成。可以在 mounted 内容使用 vm.$nextTick。该钩子在服务器端渲染期间不被调用
beforeUpdate	在数据发生变化后，DOM被更新之前被调用。这里适合在现有DOM将要被更新之前访问它，比如移除手动添加的事件监听器。该钩子在服务器端渲染期间不被调用，因为只有初次渲染会在服务器端进行
updated	在数据更改后导致的虚拟DOM重新渲染和更新完毕之后被调用。当这个钩子被调用时，组件DOM已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态。如果要相应状态改变，通常最好使用 计算属性和侦听器取而代之 注意，update 不会保证所有的子组件也都被重新渲染完毕。如果你希望等待整个视图都渲染完毕，可以在 updated 内部使用 vm.$nextTick。该钩子函数在服务器端渲染期间不被调用
activited	keep-alive专属，组件被激活时调用。该钩子函数在服务器端渲染期间不被调用
deactivited	keep-alive专属，组件被销毁时调用。该钩子函数在服务器端渲染期间不被调用
beforeDestory（2.x）	组件销毁前调用
destoryed（2.x）	组件销毁后调用
beforeUnmount	在卸载组件实例之前调用。在这个阶段，实例仍然是完全正确的。该钩子函数在服务器端渲染期间不被调用
unmounted	卸载组件实例后调用。调用此钩子时，组件实例的所有指令都被解除绑定，所有事件侦听器都被

	移除，所有子组件实例都被卸载。该钩子函数在服务器端渲染期间不被调用
errorCaptured	在捕获一个来自后代组件的错误时被调用。
renderTracked	跟踪虚拟 DOM 重新渲染时调用。钩子接受 debugger event 作为参数。此事件告诉你哪个操作跟踪了组件以及该操作的目标对象和键
renderTriggered	当虚拟DOM重新渲染被触发时调用。和 renderTracked 类似，接收 debugger event 作为参数。此事件告诉你是什么操作触发了重新渲染，以及该操作的目标对象和键



Vue 父组件和子组件生命周期钩子函数执行顺序?

Vue 的父组件和子组件生命周期钩子函数执行顺序可以归类为以下 4 部分：

- 加载渲染过程： 父 beforeCreate -> 父 created -> 父 beforeMount -> 子 beforeCreate -> 子 created -> 子 beforeMount -> 子 mounted -> 父 mounted
- 子组件更新过程：父 beforeUpdate -> 子 beforeUpdate -> 子 updated -> 父 updated
- 父组件更新过程：父 beforeUpdate -> 父 updated
- 销毁过程：父 beforeDestroy -> 子 beforeDestroy -> 子 destroyed -> 父 destroyed

父组件可以监听到子组件的生命周期吗?

- 通过 $emit 触发父组件的事件
- 引用子组件时通过 @hook 来监听

淡淡你对 keep-alive 的了解?

keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，避免重新渲染，其有以下特性：

- 一般结合路由和动态组件一起使用，用于缓存组件
- 提供 include 和 exclude 属性，两者都支持字符串和正则表达式，include 表示只有名称匹配的组件会被缓存，exclude 表示任何名称匹配的组件都不会被缓存，其中 exclude 的优先级比 include 高
- 对应两个钩子函数 activated 和 deactivated，当组件被激活时，触发 activated 钩子函数，当组件被销毁时，触发 deactivated 钩子函数

组件中 data 为什么是一个函数?

因为组件是用来复用的，且 JS 里对象是引用关系，如果组件中 data 是一个对象，那么这样作用域没有隔离，子组件中的 data 属性值会相互影响，如果组件中 data 选项是一个函数，那么每个实例可以维护一份被返回对象的独立的拷贝，组件实例之间的 data 属性值不会互相影响；而 new Vue 的实例，是不会被复用的，因此不存在引用对象的问题

v-model 的原理

v-model 本质上是一种语法糖，v-model 在内部为不同的输入元素使用不同的属性并抛出不同的事件

- text 和 textarea 元素使用 value 属性和 input 事件
- checkbox 和 radio 使用 checked 属性和 change 事件
- select 字段将 value 作为 prop 并将 change 作为事件

Vue 组件间通信有哪几种?

- props / $emit（父子组件）
- ref / $parent / $children（父子组件）
- EventBus（$emit/$on）（父子、隔代、兄弟组件）
- $attrs / $listeners（隔代组件）
- provide / inject（隔代组件）
- Vuex / pinia（父子、隔代、兄弟组件）

使用过 Vue SSR 吗? 说说 SSR?

> 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。
>
> 即：SSR大致的意思就是vue在客户端将标签渲染成的整个 html 片段的工作在服务端完成，服务端形成的html 片段直接返回给客户端这个过程就叫做服务端渲染

服务端渲染 SSR 的优缺点如下：

（1）优点：

- 更好的SEO：因为 SPA 页面的内容是通过Ajax获取，而搜索引擎爬取工具并不会等待Ajax异步完成后再抓取页面内容，所以在SPA中是抓取不到页面通过Ajax获取到的内容；而 SSR 是直接由服

务端返回已经渲染好的页面（数据已经包含在页面中），所以搜索引擎爬取工具可以抓取渲染好的页面

- 更快的内容到达时间（首屏加载更快）：SPA 会等待所有 Vue 编译后的 js 文件都下载完成后，才开始进行页面的渲染，文件下载需要一定的时间，所以首屏渲染也需要一定的时间；SSR 直接由服务端渲染好页面直接返回显示，无需等待下载js文件及再去渲染等，所以SSR有更快的内容到达时间

（2）缺点：

- 更多的开发条件限制：例如服务端渲染只支持 beforeCreated 和 created 两个钩子函数，这会导致一些外部扩展库需要特殊处理，才能在服务端渲染应用程序中运行；并且与可以部署在任何静态文件服务器上的完全静态单页面应用程序 SPA 不同，服务端渲染应用程序，需要处于Node.js server 运行环境
- 更多的服务器负载：在 Node.js 中渲染完整的应用程序，显然会比仅仅提供静态文件的 server 更加大量占用 CPU 资源（CPU-intensive - CPU 密集），因此如果你预料到在高流量环境（high traffic）下使用，请准备相应的服务器负载，并明智地采用缓存策略。

介绍一下Vue响应式系统

Vue 是 MVVM 框架，当据模型 data 变化时，页面视图会得到响应更新，其原理是对 data 的 getter/setter 方法进行拦截（Object.defineProperty 或者 Proxy），利用发布订阅的设计模式，在 getter 方法中进行订阅，在 setter 方法中发布通知，让所有订阅者完成响应

在响应式系统中，Vue 会为数据模型 data 的每一个属性新建一个订阅中心作为发布者，而监听器 watch、计算属性 computed、视图渲染 template/render 三个角色同时作为订阅者，对于监听器 watch，会直接订阅观察监听的属性，对于计算属性 computed 和视图渲染 template/render，如果内容执行获取了 data 的某个属性，就会执行该属性的 getter 方法，然后自动完成对该属性的订阅，当属性被修改时，就会执行该属性的setter方法，从而完成该属性的发布通知，通知所有订阅者进行更新

Vue 3.0 里为什么要用 Proxy API替代 defineProperty API?

（1）**defineProperty API 的局限性最大原因是它只能针对单例属性做监听**

Vue 2.x 中的响应式实现正是基于 defineProperty 中的 descriptor，对 data 中的属性做了遍历 + 递归，为每个属性设置了 getter、setter

这也就是为什么 Vue 只是对 data 中预定义过的属性做出响应的原因，在 Vue 中使用下标的方式直接修改属性的值或者添加一个预先不存在的对象属性是无法做到 setter监听的，这是 defineProperty 的局限性

（2）**Proxy API 的监听是针对一个对象的，那么对这个对象的所有操作会进入监听操作，这就完全可以代理所有属性，将会带来很大的性能提升和更优的代码**

Proxy 可以理解成，在目标对象之前架设一层"拦截"，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写

（3）**响应式是惰性的**

在 Vue.js 2.x 中，对于一个深层属性嵌套的对象，要劫持它内部深层次的变化，就需要递归遍历这个对象，执行 Object.defineProperty 把每一层对象数据都变成响应式的，这无疑会有很大的性能消耗

在 Vue.js 3.0 中，使用 Proxy API 并不能监听到对象内部深层次的属性变化，因此它的处理方式是在 getter 中去递归响应式，这样的好处是真正访问到的内部属性才会变成响应式，简单地可以说按需实现响应式，减少性能消耗

（4）Proxy 只会代理对象的第一层，Vue3 是怎样处理这个问题的呢?

- 判断当前Reflect.get的返回值是否为Object，如果是则再通过reactive方法做代理，这样就实现了深度观测
- 检测数组的时候可能触发了多个get/set，那么如何防止触发多次呢？我们可以判断key是否为当前被代理兑现target自身属性，也可以判断旧值与新值是否相等，只有满足以上两个条件之一时，才有可能执行 trigger

Proxy 与 Object.defineProperty 优劣对比

Proxy 的优势如下:

- Proxy 可以直接监听对象而非属性
- Proxy 可以直接监听数组的变化
- Proxy 有多达13种拦截方法，不限于apply、ownKeys、deleteProperty、has 等等是 Object.defineProperty 不具备的
- Proxy 返回的是一个新对象，我们可以只操作新的对象达到目的，而 Object.defineProperty 只能遍历对象属性直接修改
- Proxy 作为新标准将受到浏览器厂商重点持续的性能优化，也就是传说中的新标准的性能红利

Object.defineProperty 的优势如下:

- 兼容性好，支持 IE9，而 Proxy 的存在浏览器兼容性问题,而且无法用 polyfill 磨平，因此 Vue 的作者才声明需要等到下个大版本(3.0)才能用 Proxy 重写

Vue 2.x 中如何检测数组变化?

- 使用了函数劫持的方式，重写了数组的方法，Vue将data中的数组进行了原型链重写，指向了自己定义的数组原型方法，当调用数组api时，可以通知依赖更新
- 如果数组中包含着引用类型，会对数组中的引用类型再次递归遍历进行监控。这样就实现了监测数组变化

删除数组用 delete 和 Vue.delete 有什么区别

- delete：只是被删除数组成员变为 empty/undefined，其他元素键值不变
- Vue.delete：直接删除了数组成员，并改变了数组的键值（对象是响应式的，确保删除能触发更新视图，这个方法主要用于避开 Vue 不能检测到属性被删除的限制）

说一下Vue-Router 导航守卫

参考 Vue的钩子函数[路由导航守卫、keep-alive、生命周期钩子]

全局守卫：

- router.beforeEach 全局前置守卫，进入路由之前
- router.beforeResolve 全局解析守卫（2.5.0+），在 beforeRouteEnter 调用之后调用
- router.afterEach 全局后置钩子，进入路由之后

```
import router from './router'
router.beforeEach((to, from, next) => {
    next()
})
router.beforeResolve((to, from, next) => {
    next()
})
router.afterEach((to, from) => {})
```

路由独享守卫：

```
const router = new VueRouter({
    routes: [
        {
            path: '/foo',
            component: Foo,
            boforeEnter: (to, from, next) => {
                // 调用顺序在全局前置守卫后面，所以不会被全局守卫覆盖
            }
        }
    ]
})
```

路由组件内守卫：

- beforeRouteEnter：进入路由前

- boforeRouteUpdate：路由复用同一个组件时
- beforeRouteLeave：离开当前路由时

```
beforeRouteEnter (to, from, next) {
    // 在路由独享守卫后调用，不能获取组件实例，组件实例还没被创建
}
boforeRouteUpdate (to, from, next) {
    // 在当前路由改变，但是该组件被复用时调用，可以访问组件实例
}
beforeRouteLeave (to, from, next) {
    // 导航离开该组件的对应路由时调用，可以访问组件实例
}
```

完整的路由导航解析流程（不包括其他生命周期）

1. 触发进入其他路由
2. 调用要离开路由的组件守卫 beforeRouteLeave
3. 调用全局前置守卫：beforeEach
4. 在重用的组件里调用：beforeRouteUpdate
5. 调用路由独享守卫：beforeEnter
6. 解析异步路由组件
7. 在将要进入的路由组件中调用 beforeRouteEnter
8. 调用全局解析守卫 boforeResolve
9. 导航被确认
10. 调用全局后置钩子的 afterEach 钩子
11. 触发 DOM 更新（mounted）
12. 执行 beforeRouteEnter 守卫中传给 next 的回调函数

说一下Vue路由懒加载?

懒加载简单来说就是把不同的路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，是为了给用户更好的使用体验，首屏组件加载速度更快一些，解决白屏问题

路由懒加载的方式：

- vue异步组件技术：resolve => require(['需要加载的路由的地址'], resolve)
- es提出的动态import（推荐使用这种方式）：import('需要加载的模块地址')

注：使用命名 chunk，和 webpack 中的魔法注释就可以把某个路由下的所有组件都打包在同个异步块（chunk）中

```
const Foo = () => import (/* webpackChunkNamae: 'group-foo' */ './Foo.vue')
```

Vue自定义指令的钩子函数

2.x 语法：

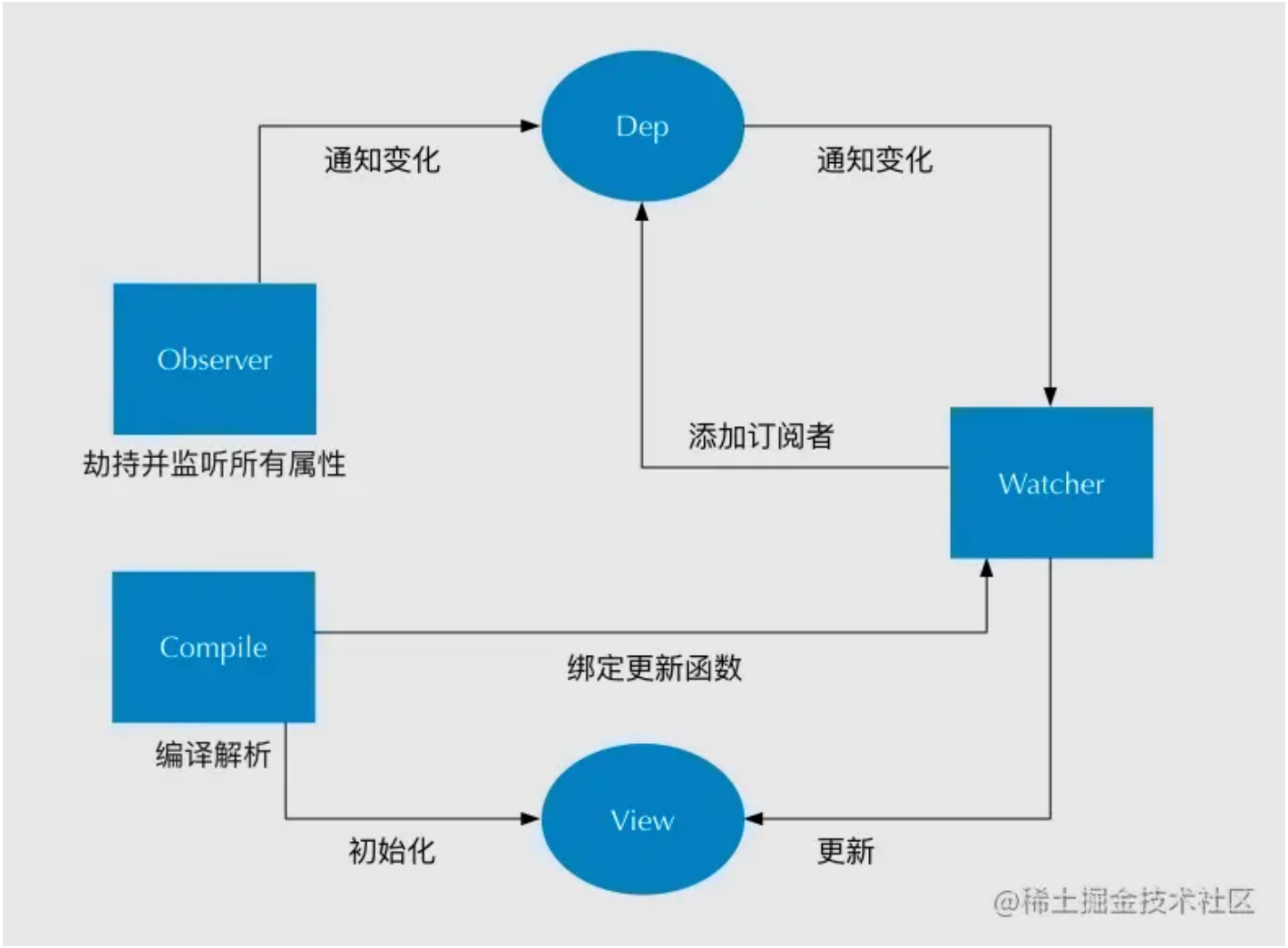
- bind：指令绑定到元素后调用，只调用一次
- inserted：元素插入父 DOM 后调用
- update：当元素更新，但子元素尚未更新时，将调用此钩子
- componentUpdated：一旦组件和子级被更新，就会调用这个钩子
- unbind：一旦指令被移除，就会调用这个钩子，也只调用一次

3.x 语法：

- created（新增）：在元素的 attribute 或事件监听器被应用之前调用
- bind -> beforeMount
- inserted -> mounted
- beforeUpdate（新增）：在元素本身被更新之前调用，与组件的生命周期钩子十分相似
- update（移除）
- componentUpdate -> updated
- beforeUnmount（新增）：将在元素被卸载之前调用
- unbind -> unmounted

Vue是如何实现数据双向绑定的?

- **监听器 Observer：**对数据对象进行遍历，包括子属性对象的属性，利用 Object.defineProperty() 对属性都加上 setter 和 getter。这样的话，给对象的某个值赋值，就会触发 setter，那么就能监听到了数据变化
- **解析器 Compile：**解析 Vue模板指令，将模板中的变量都替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，调用更新函数进行数据更新
- **订阅者 Watcher：**Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁，主要的任务是订阅 Observer 中的属性值变化的消息，当收到属性值变化的消息时，触发解析器 Compile 中对应的更新函数。每个组件实例都有相应的 watcher 实例对象，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的setter被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新——这是一个典型的观察者模式
- **订阅器 Dep：**订阅器采用发布-订阅设计模式，用来收集订阅者Watcher，对监听器 Observer 和订阅者 Watcher 进行统一管理



说一下Vue的异步更新机制和nextTick原理

参考 Vue你不得不知道的异步更新机制和nextTick原理 和 vue方法nextTick源码分析

Vue 的异步更新机制：

Vue 进行 DOM 更新内部也是调用 nextTick 来做异步队列控制。只要观察到数据变化，Vue 将开启一个队列 queue，然后将 watcher 加入到一个 queue 队列中，最后触发 queue 中所有 watcher 的 run 方法来更新，并且加入 queue 的过程中还会对 watcher 进行去重操作，因此在一个组件中 data 内定义得数据都是存储同一个“渲染 watcher”，所以数据即使更新了多次，最终也只会执行一个更新页面的逻辑。

DOM 至少会在当前事件循环的所有数据变化完成之后，再统一更新视图。而当我们自己调用 nextTick 的时候，它就在更新DOM的微任务队列后追加了我们自己的回调函数

从而确保我们的代码在DOM更新后执行，同时也避免了setTimeout可能存在的多次执行问题。 确保队列中的微任务在一次事件循环前被执行完毕。

nextTick 原理：

异步更新内部最重要的就是 nextTick 方法，它负责将异步任务加入队列和执行异步任务。Vue 也将它暴露出来提供给用户使用。数据修改完成后，立即获取相关DOM还没那么快更新，使用 nextTick 便可以解决这一问题

总结：

异步更新机制使用微任务和宏任务，基于事件循环运行，在 Vue 中对性能起着至关重要的作用，它对重复冗余的 watcher 进行过滤，而 nextTick 根据不同的环境，使用优先级更高的异步任务。这样做的好处是等待所有的状态同步更新完毕，再一次性渲染页面。用户创建的 nextTick 运行页面更新之后，因此能够回去更新后的 DOM

Vue 实现一个高阶组件

高阶组件就是一个函数，且该函数接受一个组件作为参数，并返回一个新的组件。在不改变对象自身的前提下在程序运行期间动态地给对象添加一些额外的属性或行为

```
// 高阶组件(HOC)接收到的 props 应该透传给被包装组件即直接将原组件prop传给包装组件
// 高阶组件完全可以添加、删除、修改 props
export default function Console (BaseComponent) {
    return {
        props: BaseComponent.props,
        mounted () {
            console.log('高阶组件')
        },
        render (h) {
            console.log(this)
            // 将 this.$slots 格式化为数组，因为 h 函数第三个参数是子节点，是一个数组
            const slots = Object.keys(this.$slots)
                .reduce((arr, key) => arr.concat(this.$slots[key]), [])
                .map(vnode => {
                    vnode.context = this._self // 定到高阶组件上，vm：解决具名插槽被作为默认插槽进行渲染
                    return vnode
                })
            // 透传props、透传事件、透传slots
            return h(
                BaseComponent,
                {
                    on: this.$listeners,
                    attrs: this.$attrs, // attrs 指的是那些没有被声明为 props 的属性
                    props: this.$props
                }
                slots
            )
```

```
28            }
29        }
30    }
```

你有对Vue项目进行哪些优化?

（1）代码层面的优化

- 尽量减少 data 中的数据，data 中的数据都会增加 getter 和 setter，会收集对应的 watcher
- v-if 和 v-show 区分使用场景
- computed 和 watch 区分使用场景
- v-for 遍历必须为 item 添加 key，且避免同时使用 v-if
- 如果需要使用 v-for 给每项元素绑定事件时使用事件代理
- 长列表性能优化，滚动到可视区域动态加载
- 图片资源懒加载
- 使用路由懒加载、异步组件
- 第三方插件的按需引入
- 优化无限列表性能
- 服务端渲染 SSR or 预渲染
- SPA 页面采用 keep-alive 缓存组件
- key 保证唯一
- 防抖、节流

（2） Webpack 层面的优化

- Wepack 对图片进行压缩
- 减少 ES6 转为 ES5 的冗余代码
- 提取公共代码，包括JS和CSS
- 模板预编译
- 优化 SourceMap
- 构建结果输出分析
- Vue 项目的编译优化
- Tree Shaking/Scope Hoisting
- 使用 CDN 加载第三方模块
- 多线程打包 happypack
- splitChunks 抽离公共文件
- sourceMap 优化

（3）基础的Web技术的优化

- 开启gzip压缩

- 浏览器缓存
- 使用 Chrome Performance 查找性能瓶颈

（4）SEO优化

- 预渲染
- 服务端渲染SSR

Compositon Api 和 Vue 2.x 使用的 Options Api 有什么区别?

- Options Api：包含一个描述组件选项（data、methods、props等）的对象 options；API 开发复杂组件，同一个功能逻辑的代码被拆分到不同选项；使用 mixin 重用公用代码，也有问题：命名冲突，数据来源不清晰
- composition Api：vue3 新增的一组 api，它是基于函数的api，可以灵活地组织组件的逻辑。解决 options api 在大型项目中不好拆分和重用的问题

Vue 3 编译做了哪些优化?

（1）响应式系统提升

使用Proxy代替defineProperty对数据进行过劫持，proxy的性能本来比defineproperty好，proxy可以拦截属性的访问、赋值、删除等操作，不需要初始化的时候遍历所有属性，另外有多层属性嵌套的话，只有访问某个属性的时候，才会递归处理下一级的属性

优势：

- 可以监听动态新增的属性
- 可以监听删除的属性
- 可以监听数组的索引和length属性

（2）生成了 Block tree

Vue 2.x 的数据更新并触发重新渲染的粒度是组件级的，单个组件内部需要遍历该组件的整个 vnode 树，在 2.0 里，渲染效率的快慢与组件的大小成正相关：组件越大，渲染效率越慢。并且，对于一些静态节点，无数据更新，这些遍历都是性能消耗

Vue 3.0 做到了通过编译阶段对静态模板的分析，编译生成了 Block tree。Block tree 是一个将模板基于动态节点指令切割的嵌套区块，每个区块内部的节点结构是固定的，每个区块只需要追踪自身包含的动态节点。所以，在 3.0 里，渲染效率不再与模板大小成正比，而是与模板中动态节点的数量成正相关

（3）slot编译优化

Vue 2.x 中，如果又一个组件传入了 slot，那么每次父组件更新的时候，会强制使子组件 update，造成性能的浪费

Vue 3.0 优化了 slot 的生成，使得非动态 slot 中属性的更新只会触发子组件的更新。动态 slot 指的是在 slot 上面使用 v-if、v-for、动态 slot 名字等会导致 slot 产生运动时动态变化但是又无法被子

组件 track 的操作

（4）diff 算法优化

Vue 2.x 中的虚拟DOM是进行全量的对比

Vue 3.0 中新增了静态标记（patchFlag）：在与上次虚拟节点进行对比时，值对比带有patch flag 的节点，并且可以通过 flag 的信息得知当前节点要对比的具体内容

（5）hoistStatic 静态提升

Vue 2.x 中无论元素是否参与更新，每次都会重新创建

Vue 3.0中对不参与更新的元素，只会被创建一次，之后会在每次渲染时候被不停地复用

（6）cacheHandlers 事件侦听器缓存

默认情况下 onClick 会被视为动态绑定，所以每次都会去追踪它的变化但是因为是同一个函数，所有没有追踪变化，直接缓存起来复用即可

（7）源码体积的优化

Vue3移除了一些不常用的api，例如：inline-template、filter等，使用tree-shaking

对于即将到来的 Vue3 特性你有什么了解的吗?

（1）监测机制的改变

Vue3 实现基于 Proxy 的 observer 实现，提供全语言覆盖的反应性跟踪。这消除了 Vue2 当中基于 Object.defineProperty 的实现所存在的很多限制：

- 只能监听属性，不能监测对象
- 监测属性的添加和删除
- 监测数组索引和长度的变更
- 支持 Map、Set、WeakMap 和 WeakSet

新的 observer 还提供以下特性：

- 用于创建 observable 的公开 API。这为中小规模场景提供了简单轻量级的跨组件状态管理解决方案
- 默认采用惰性观察。在 2.x 中，不管反应式数据有多大，都会在启动时被观察到。如果你的数据集很大，这可能会在应用启动时带来明显的开销。在 3.x 中，只观察用于渲染应用程序最初可见部分的数据
- 更精确的变更通知。在 2.x 中，通过 Vue.set 强制添加新属性将导致依赖于该对象的 watcher 收到变更通知。在 3.x 中，只有依赖于特定属性的 watcher 才会收到通知
- 不可变的 observable：我们可以创建值的“不可变”版本（即使是嵌套属性），除非系统在内部暂时将其“解禁”。这个机制可用于冻结 prop 传递或 Vuex 状态树以外的变化
- 更好的调试功能：我们可以使用新的 renderTracked 和 renderTriggered 钩子精确地跟踪组件在什么时候以及为什么重新渲染

（2）模板

模板方面没有大的变更，只改了作用域插槽，2.x 的机制导致作用域插槽变了，父组件会重新渲染，而 3.0 把作用域插槽改成了函数的方式，这样只会影响子组件的重新渲染，提升了渲染的性能

同时，对于 render 函数的方面，vue3.0 也会进行一系列更改来方便习惯直接使用 api 来生成 vdom

（3）对象式的组件声明方式

vue2.x 中的组件是通过声明的方式传入一系列 option，和 TypeScript 的结合需要通过一些装饰器的方式来做，虽然能实现功能，但是比较麻烦。3.0 修改了组件的声明方式，改成了类式的写法，这样使得和 TypeScript 的结合变得很容易。

此外，vue 的源码也改用了 TypeScript 来写。其实当代码的功能复杂之后，必须有一个静态类型系统来做一些辅助管理。现在 vue3.0 也全面改用 TypeScript 来重写了，更是使得对外暴露的 api 更容易结合 TypeScript。静态类型系统对于复杂代码的维护确实很有必要。

（4）其他方面的更改

vue3.0 的改变是全面的，上面只涉及到主要的 3 个方面，还有一些其他的更改：

- 支持自定义渲染器，从而使得 weex 可以通过自定义渲染器的方式来扩展，而不是直接 fork 源码来改的方式
- 支持 Fragment（多个根节点）和 Protal（在 dom 其他部分渲染组建内容）组件，针对一些特殊的场景做了处理
- 基于 treeshaking 优化，提供了更多的内置功能

Vue2和Vue3有什么区别?

参考[总结一下Vue2和Vue3都有哪些区别?]

- 生命周期的变化
- 多根节点：Vue3 支持多根节点组件fragment
- Teleport：Vue3 提供了Teleport 组件可将部分 Dom 移动到 vue app 之外的位置，比如 Dialog 组件
- 异步组件：Vue3提供 Suspense 组件
- 组合式API：Vue2是选项式API，一个逻辑会在不同的位置，导致代码可读性差，需要上下来回翻看。Vue3是组合式API，可以将同一个逻辑的内容写在一起，提高了代码的可读性、复用性
- 响应式原理
- 虚拟DOM：Vue3 相比于 Vue2 虚拟DOM 上增加 patchFlag 字段
- Diff优化：patchFlag 字段帮助 diff 时区分静态节点，以及不同类型的动态节点，一定程度地减少节点本身及其属性的比对
- 事件缓存：Vue3 的cacheHandler 可在第一次渲染后缓存我们的事件
- 打包优化：
- 自定义渲染API

- TypeScript支持

Vue 怎么用 vm.$set() 解决对象新增属性不能响应的问题?

受现代 JavaScript 的限制，Vue **无法检测到对象属性的添加或删除。**由于 Vue 会在初始化实例时对属性执行 getter/setter 转化，所以属性必须在 data 对象上存在才能让 Vue 将它转换为响应式的。但是 Vue 提供了 Vue.set (object, propertyName, value) / vm.$set (object, propertyName, value) 来实现为对象添加响应式属性，那框架本身是如何实现的呢?

```
export function set (target: Array<any> | Object, key: any, val: any): any {
    // target 为数组
    if (Array.isArray(target) && isValidArrayIndex(key)) {
        // 修改数组的长度, 避免索引>数组长度导致splcie()执行有误
        target.length = Math.max(target.length, key)
        // 利用数组的splice变异方法触发响应式
        target.splice(key, 1, val)
        return val
    }
    // key 已经存在，直接修改属性值
    if (key in target && !(key in Object.prototype)) {
        target[key] = val
        return val
    }
    const ob = (target: any).__ob__
    // target 本身就不是响应式数据, 直接赋值
    if (!ob) {
        target[key] = val
        return val
    }
    // 对属性进行响应式处理
    defineReactive(ob.value, key, val)
    ob.dep.notify()
    return val
}
```

我们阅读以上源码可知，vm.$set 的实现原理是:

- 如果目标是数组，直接使用数组的 splice 方法触发响应式
- 如果目标是对象，会先判断属性是否存在，对象是否是响应式，最终如果要对属性进行响应式处理，则是通过调用 defineReactive 方法进行响应式处理（defineReactive 方法就是 Vue 在初始化

对象时，给对象属性采用 Object.defineProperty 动态添加 getter 和setter 的功能所调用的方法）

什么是虚拟DOM？为什么要使用虚拟DOM？

虚拟DOM就是一个用来描述真是DOM的JavaScript对象

虚拟 DOM 是相对于浏览器所渲染出来的真实DOM的对象结构，在 react、vue 等技术出现之前，我们要改变页面展示的内容只能通过遍历查询 dom 树的方式找到需要修改的 dom 节点，然后修改样式行为或者结构，来达到更新UI的目的

这种方式相当消耗计算资源，因为每次查询DOM都需要遍历整棵 DOM 树，如果建立一个与 DOM 树对应的虚拟 DOM 对象（js 对象），以对象嵌套的方式来表示 DOM 树，那么每次 DOM 的更改就变成了 js 对象的属性的更改，这样一来就能查找 js 对象的属性变化要比查询DOM树的性能开销小

用 JavaScript 对象结构表示 DOM 树的结构，然后用这个树构建一个真正的 DOM 树插到文档当中。当状态变更时，重新构造一棵新的对象树，然后用新的树和旧的树进行比较，记录两棵树的差异，把所记录的差异应用到所构建的真正的DOM树上，视图旧更新来，virtual dom 本质上就是在 JS 和 DOM 之间做了一个缓存

- 虚拟DOM可以维护程序的状态，跟踪上一次的状态，通过比较前后两次状态差异更新真是DOM
- 真实DOM的属性很多，创建DOM节点开销很大，虚拟DOM只是普通的JavaScript对象，描述属性并不需要很多，创建开销小
- 复杂视图情况下提升渲染性能（操作dom性能消耗大，减少操作dom的范围可以提升性能）

虚拟DOM的优缺点?

优点:

- **保证性能下限:** 框架的虚拟DOM需要适配任何上层API可能产生的操作，它的一些DOM操作的实现必须是普适的，所以它的性能并不是最优的；但是比起粗暴的DOM操作性能要好很多，因此框架的虚拟DOM至少可以保证在你不需要手动优化的情况下，依然可以提供还不错的性能，即保证性能的下限
- **无需手动操作DOM:** 我们不再需要手动去操作DOM，只需要写好 View-Model 的代码逻辑，框架会根据虚拟DOM和数据双向绑定，帮我们以可预期的方式更新视图，极大提高我们的开发效率
- **跨平台:** 虚拟DOM本质上是 JavaScript 对象，而DOM与平台强相关，相比之下虚拟DOM可以进行更方便地跨平台操作，例如服务器渲染，weex开发等等

缺点:

- **无法进行极致优化:** 虽然虚拟DOM+合理的优化，足以应对绝大部分应用的性能需求，但在一些性能要求极高的应用中虚拟DOM无法进行针对性的极致优化

虚拟DOM实现原理?

- 用 JavaScript 对象模拟真实 DOM 树，对真实 DOM 进行抽象
- diff 算法 —— 比较两棵虚拟 DOM 树的差异
- pach 算法 —— 将两个虚拟 DOM 对象的差异应用到真正的 DOM 树

说一下 Diff 算法

参考 极为细致的Vue的Diff流程详解——以流程图表达

- init() 设置模块，创建 patch() 函数
- 使用 h() 函数创建对象 （Vnode）描述真实DOM
- patch() 比较新旧两个 Vnode（核心：递归 + 双指针）
  - 把新节点中变化的内容渲染到真实DOM，最后返回新节点作为下次处理的旧节点
  - 首先对比新旧 VNode 是否为相同节点（节点的key和sel是否相同），比对属性
  - 如果不是相同节点，删除之前的内容，重新渲染
  - 如果是相同节点，可区分以下4种情况：
    - 旧VNode有子节点，新VNode没有子节点，直接删除旧VNode的子节点
    - 旧VNode没有子节点，新VNode有子节点，则将新Vnode的子节点添加到旧节点上
    - 旧Vnode没有子节点，新Vnode也没有子节点，判断是否有文本节点，如果有文本节点的话就进行比对
    - 旧Vnode有子节点，新Vnode也有子节点，这时候就要用双指针对新旧Vnode的子节点列表进行判断。主要有以下4种情况：
      - 旧头指针和新头指针的比对
      - 旧尾指针和新尾指针的比对
      - 旧头指针和新尾指针的比对
      - 旧尾指针和新头指针的比对
- 把变化的内容更新到真实DOM树

Vue中的key有什么作用?

key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速。Vue 的 diff 过程可以概括为：oldCh 和 newCh 各有两个头尾的变量 oldStartIndex、oldEndIndex 和 newStartIndex、newEndIndex，它们会新节点和旧节点会进行两两对比，即一共有4种比较方式：newStartIndex 和oldStartIndex 、newEndIndex 和 oldEndIndex 、newStartIndex 和 oldEndIndex 、newEndIndex 和 oldStartIndex，如果以上 4 种比较都没匹配，如果设置了key，就会用 key 再进行比较，在比较的过程中，遍历会往中间靠，一旦 StartIdx > EndIdx 表明 oldCh 和

newCh 至少有一个已经遍历完了，就会结束比较。具体有无 key 的 diff 过程，可以查看作者写的另一篇详解虚拟 DOM 的文章《深入剖析：Vue核心之虚拟DOM》

所以 Vue 中 key 的作用是：key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速

更准确：因为带 key 就不是就地复用了，在 sameNode 函数 a.key === b.key 对比中可以避免就地复用的情况。所以会更加准确。

更快速：利用 key 的唯一性生成 map 对象来获取对应节点，比遍历方式更快，源码如下：

```
function createKeyToOldIndex (children, beginIdx, endIdx) {
    let i, key
    const map = {}
    for (i = beginIdx; i < endIdx; ++i) {
        key = children[i].key
        if (isDef(key)) map[key] = i
    }
    return map
}
```

设置 Key 的好处：

- 数据更新时，可以尽可能地减少DOM操作
- 列表渲染时，可以提高列表渲染的效率，提高页面的性能；

什么是 mixin?

- Mixin 使我么能够为 Vue 组件编写可插拔和可重用的功能
- 如果你希望多个组件之间重用一组组件选项，例如生命周期 hook、方法等，则可以将其编写为 mixin，并在组件中简单地引用它
- 然后将 mixin 的内容合并到组件中。如果你要在 mixin 中定义生命周期 hook，那么它在执行时将优化于组件自己的 hook

Vue 模板编译原理知道吗？能简单说一下吗?

简答说：Vue的编译模板就是将 template 转化为 render 函数的过程。会经历以下阶段（生成 AST 树/优化/codegen）

- 首先解析模板，生成AST语法树（一种用JavaScript对象的形式来描述整个模板）。使用大量的正则表达式对模板进行解析，遇到标签、文本的时候都会执行对应的钩子进行相关处理
- Vue 的数据时响应式的，但其实模板中并不是所有的数据都是响应式的。有一些数据首次渲染后就不会再变化，对应的 DOM 也不会变化。那么优化过程就是深度遍历 AST 树，按照相关条件对树

节点进行标记。这些被标记的节点（静态节点）我们就可以跳过对它们的比对，对运行时的模板起到很大的优化作用

- 编译的最后一步是将优化后的AST树转换为可执行的代码

说说 Vue 和 React 的异同?

- **相同点:**
  - 使用 Virtual DOM
  - 提供了响应式（Reactive）和组件化（Composable）的视图组件
  - 将注意力集中保持在核心库，而将其他功能如路由和全局状态管理交给相关的库
- **不同点:**
  - 在 React 应用中，当某个组件的状态发生变化时，它会以该组件为根，重新渲染整个组件字数（除非使用 PureComponents/shouldComponentUpdate），在Vue应用中，组件的依赖是在渲染过程中自动追踪的，所以系统能精确知晓哪个组件确实需要被重渲染
  - 在 React 中，一切都是 JavaScript。不仅仅是 HTML 可以用 JSX 来表达，现在的潮流也越来越多地将 CSS 也纳入 JavaScript 中来处理

Vue-Router

vue-router 实现原理

参考 [前端路由简介以及vue-router实现原理]

原理核心就是更新视图但不重新请求页面。路径之间的切换，也就是组件的切换。vue-router 实现单页面路由跳转模式：hash模式和history模式。根据设置mode参数

- hash模式：通过描点值的改变，根据不同的值，渲染指定DOM位置的不同数据。每一次改变 # 后的部分，都会在浏览器的访问历史中增加一个记录，使用“后退”按钮，就可以返回上一个位置。
- history模式：利用 window.history.pushState API来完成URL跳转而无须重新加载页面

vue-router 路由模式有几种?

vue-router 有 3 种路由模式：hash、history、abstract

```
switch (mode) {
    case 'history':
        this.history = new HTML5History(this, options.base)
        break
    case 'hash':
        this.history = new HashHistory(this, options.base, this.fallback)
```

```
            break
        case 'abstract':
            this.history = new AbstractHistory(this, options.base)
            break
        default:
            if (process.env.NODE_ENV !== 'production') {
                assert(false, `invalid mode: ${mode}`)
            }
    }
```

其中，3种路由模式的说明如下：

- hash：使用 URL hash 值来做路由。支持所有浏览器，包括不支持 HTML5 History Api 的浏览器
- history：依赖 HTML5 History API 和服务器配置。
- abstract：支持所有JavaScript运行环境，如 Node.js 服务器。如果发现没有浏览器的API，路由会自动强制进入这个模式

能说下 vue-router 中常用的 hash 和 history 路由模式实现原理吗?

（1）hash模式的实现原理

早期的前端路由的实现就是基于 location.hash 来实现。其实现原理很简单，location.hash 的值就是 URL 中 # 后面的内容。比如下面这个网站，它的 location.hash 的值就是 ‘#search’

```
https://www.word.com#search
```

hash 路由模式的实现主要是基于下面几种特性：

- URL中的 hash 值只是客户端的一种状态，也就是说当向服务器端发出请求时，hash部分不会被发送
- hash值的改变，都会在浏览器的访问历史中增加一个记录。因此我们能通过浏览器的回退、前进按钮控制hash的切换
- 可以通过a标签，并设置 href 属性，当用户点击这个标签后，URL的hash值会发生改变；或者使用 JavaScript 来对 location.hash 进行赋值，改变 URL 的 hash 值
- 我们可以使用 hashchange 事件来监听 hash 值的变化，从而对页面进行跳转（渲染）

（2）history 模式的实现原理

HTML5 提供了 History API 来实现 URL 的变化。其中最主要的API有以下两个：history.pushState() 和 history.replaceState()。这两个 API 可以在不进行刷新的情况下，操作浏览器的历史记录。唯一不同的是，前者是新增一个历史记录，后者是直接替换当前的历史记录，如下所示：

```
window.history.pushState(null, null, path);
window.history.replaceState(null, null, path);
```

history 路由模式的实现主要基于存在下面几个特性：

- pushState 和 replacState 两个API 来操作实现 URL 的变化
- 我们可以使用 popstate 事件来监听 url 的变化，从而对页面进行跳转（渲染）
- history.pushState() 或 history.replaceState 不会触发 popState 事件，这时我们需要手动触发页面跳转（渲染）

vue router 如何使用 params query 传参，以及有什么区别?

```
/router/:id  // 这里的id 在 params
/router?id=123 // 这里的id 在 query
```

区别：

- params 是路由的一部分，必须要有。query 是拼接在url后面的参数
- params、query 不设置也可以传参，但是 params 不设置的时候，刷新页面或者返回参数会丢失，query 并不会出现这种情况

Vuex

Vuex实现原理

- Vue.use(vuex) 会调用 vuex 的 install 方法
- 在 beforeCreated 钩子前混入 vuexInit 方法，VuexInit 方法实现了 store 注入 vue组件实例，并注册了 vuex store 的引用属性 $store
- Vuex 的 state 状态是响应性，是借助 vue 的 data 是响应式，将 state 存入 vue 实例组件的 data 中
- Vuex 的 getters 则是借助 vue 的计算属性 computed 实现数据实时监听

你使用过 Vuex 吗?

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。每一个 Vuex 应用的核心就是 store（仓库）。“store”基本上就是一个容器，它包含你的应用中大部分的状态（state）

（1）Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。

（2）改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样使得我们可以方便地跟踪每一个状态的变化。

主要包括以下几个模块：

- State：定义了应用状态的数据结构，可以在这里设置默认的初始状态

- Getter：允许组件从 Store 中获取数据，mapGetters 辅助函数仅仅是将 store 中的 getter 映射到局部计算属性
- Mutation：是唯一更改 store 中状态的方法，且必须是同步函数
- Action：用于提交 mutation，而不是直接变更状态，可以包含任意异步操作
- Module：允许将单一的Store拆分为多个 store 且同时保存在单一的状态树中

为什么Vuex的mutation中不能做异步操作?

每个 mutation 执行完成后都会对应到一个新的状态变更，这样 devtools 就可以打个快照存下来，然后就可以实现 time-travel 了。如果 mutation 支持异步操作，就没有办法知道状态是何时更新的，无法很好地进行状态的追踪，给调试带来困难

Vuex 和单纯的全局对象有什么区别?

- Vuex 的状态存储是响应式的。当Vue组件从store中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新
- 不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交（commit）mutation。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用

Vuex 的 action 有返回值吗？返回的是什么?

- store.dispatch 可以处理被触发的 action 的处理函数返回的 Promise， 并且 store.dispatch 仍旧返回 Promise
- Action 通常是异步的，要知道 action 什么时候结束或者组合多个 action 以处理更加复杂的异步流程，可以通过定义 action 时返回一个 promise 对象，就可以在派发 action 的时候就可以通过处理返回的 Promise 处理异步流程

> 一个 store.dispatch 在不同模块中可以触发多个 action 函数。在这种情况下，只有当所有触发函数完成后，返回的 Promise 才会执行

axios

axios 是什么？怎样使用它？怎么解决跨域的问题?

axios 是一种异步请求，用法和ajax类似，安装 npm install axios --save 即可使用，请求中包括 get、post、put、patch、delete 等五种请求方式，解决跨域可以在请求头中添加 Access-Control-

Allow-Origin，也可以在 index.js 文件中更改 proxyTable 配置等解决跨域问题