1、vue开发中常用的指令有哪些?

- 条件渲染：v-if、 v-else、 v-else-if、 v-show（注意，v-show 不支持 `<template>` 元素，也不支持 v-else）

- 列表渲染：v-for
- 监听事件：v-on
- 表单输入绑定：v-model
- 其他：v-bind、v-html、v-text

v-if vs v-show: 以下是vue官方文档的描述。#17

v-if 是“真正”的条件渲染，因为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。 v-if 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。 相比之下，v-show 就简单得多——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 进行切换。 一般来说，v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，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。

- **手段：v-if是动态的向DOM树内添加或者删除DOM元素；v-show是通过设置DOM元素的display样式属性控制显隐；**
- **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是惰性的，如果初始条件为假，则什么也不做；只有在条件第一次变为真时才开始局部编译; v-show是在任何条件下，无论首次条件是否为真，都被编译，然后被缓存，而且DOM元素保留；**
- **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；**
- **使用场景：v-if适合条件不大可能改变；v-show适合频繁切换。**

2、vue diff算法的原理

https://juejin.cn/post/7010594233253888013 https://juejin.cn/post/7045976871116210213

点题收敛

Vue的Virtual DOM实现了一种高效的diff算法，能够快速比较两个虚拟DOM树的差异，然后只更新有必要更新的部分，从而提高渲染性能。

原理大概总结

dff算法的实现原理是从根节点开始逐层遍历新旧两个虚拟DOM树，比较节点的类型、属性和子节点等内容，如果节点有差异，则记录该差异，并将其添加到一个差异队列中。在比较完成后，对于存在差异的节点，根据其类型进行相应的更新操作，比如替换节点、修改属性、移动节点位置等等。

需要注意的是，为了提高diff算法的性能，现代前端框架往往采用一些优化技巧，比如只比较同层级的节点、使用key值进行优化等，从而进一步提高渲染性能。

Diff 算法的优化

1.「分层diff」 只比较同一层级，不跨级比较

2.「组件的Type(Tagname)不一致」 比较标签名

3.「列表组件的Key不一致」 比较 key

key的作用

在Vue的虚拟DOM算法中，key的作用主要有两个：

1. 优化组件的渲染，减少DOM操作

当列表中的数据发生变化时，Vue会使用diff算法来比较旧的虚拟DOM树和新的虚拟DOM树，找出需要更新的节点，然后进行DOM操作。如果列表中每一项都没有key，那么默认情况下使用节点的索引作为key，这样做会导致在对列表中的数据进行删除、添加等操作时，所有后面的节点都需要重新创建和渲染，这样会导致性能上的浪费。

当我们使用key时，Vue会通过比较新旧节点的key值来判断哪些节点需要更新，哪些节点需要删除和添加。这样可以大大减少DOM操作的次数，提高页面的性能。

2. 标识每个节点的唯一性，提高虚拟DOM算法的效率

当有节点需要被移动时，Vue会根据节点的key值来判断是否是同一个节点。如果节点没有key，那么Vue会根据节点的标签名称、属性和子节点来判断是否为同一个节点，这样会导致效率上的低下。

使用key可以唯一标识每个节点，提高虚拟DOM算法的效率，从而减少页面的渲染时间。

```
export function isSameVNodeType(n1: VNode, n2: VNode): boolean {
    // 比较类型和key是否一致（）
    return n1.type === n2.type && n1.key === n2.key
}
```

详细回答对比过程

Vue的diff算法具体实现如下：

首先比较新旧节点的标签名，如果不同，直接替换成新节点。如果标签名相同，则比较节点的属性和事件，如果不同，直接修改成新节点。

如果节点相同，则比较节点的子节点，如果子节点有不同，则继续比较子节点，直到所有子节点比较完成。

如果子节点也相同，则不需要更新，直接退出比较。

追问继续答，不追问就不用答了 --- 这是vue2的

在进行比较时，Vue.js使用了两个指针来遍历新旧节点树，分别为新节点的开始指针和旧节点的开始指针，分别向右移动比较。如果新节点的开始指针在旧节点的开始指针的左侧，则新节点需要插入到旧节点的前面。如果新节点的开始指针在旧节点的开始指针的右侧，则说明旧节点需要删除。如果两个指针指向的节点相同，则说明节点不需要更新，直接跳过比较。在比较完成后，Vue.js会生成一份差异表，记录哪些节点需要更新、删除、插入等操作，然后根据差异表进行DOM更新。

- 新的头和老的头对比
- 新的尾和老的尾对比
- 新的头和老的尾对比
- 新的尾和老的头对比

vue3---双端对比算法以及静态标记

- 头和头比
- 尾和尾比
- 基于最长递增子序列进行移动/添加/删除

看个例子，比如

- 老的 children：[a, b, c, d, e, f, g]
- 新的 children：[a, b, f, c, d, e, h, g]

1. 先进行头和头比，发现不同就结束循环，得到 [a, b]
2. 再进行尾和尾比，发现不同就结束循环，得到 [g]
3. 再保存没有比较过的节点 [f, c, d, e, h]，并通过 newIndexToOldIndexMap 拿到在数组里对应的下标，生成数组 [5, 2, 3, 4, -1]，-1 是老数组里没有的就说明是新增
4. 然后再拿取出数组里的最长递增子序列，也就是 [2, 3, 4] 对应的节点 [c, d, e]
5. 然后只需要把其他剩余的节点，基于 [c, d, e] 的位置进行移动/新增/删除就可以了

使用最长递增子序列可以最大程度的减少 DOM 的移动，达到最少的 DOM 操作

静态标记

Vue 3对静态标记做了重大升级，使用基于ES6 Proxy 的新API来实现静态标记，以提高性能和可维护性。以下是Vue 3和Vue 2静态标记之间的主要区别

1. 新的模板编译器：Vue 3中的模板编译器支持将所有静态节点提取到一个单独的渲染函数中，这样可以避免在每次渲染中重复读取静态节点的开销。
2. 静态提升：Vue 3中新的编译器还提供了一种静态提升技术，可以自动将动态节点转换为静态节点。静态提升意味着只要生成渲染函数一次，就可以在整个组件的生命周期内重复使用该渲染函数，从而显著减少渲染时间和内存占用。
3. 固化：Vue 3中的静态标记被称为“固化”，因为它使用ES6 Proxy是实现，而不是Vue 2中使用的Object.defineProperty。这种技术可以避免由于属性添加和删除而造成的运行时性能问题，而且因为ES6 Proxy是全新编写的，所以它可以避免一些Vue 2中静态标记的限制。

3、vue mixin解决了什么问题，原理以及缺点?

1. mixin和 mixins 区别

```
/* minxin 用于全局混入，会影响到每个组件实例，通常插件都是这样做初始化的
* mixin 使我们能够为 Vue 组件编写可插拔和可重用的功能
* 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.mixin({
    beforeCreate() {
        // ...逻辑
        // 这种方式会影响到每个组件的 beforeCreate 钩子函数
    }
})

// mixins
/* 是一种分发 Vue 组件中可复用功能的非常灵活的方式
* mixins 是一个 js 对象
* 它可以包含我们组件中 script 项中的任意功能选项
* mixins 混入的钩子函数会先于组件内的钩子函数执行，并且在遇到同名选项的时候也会有选择性的进行合并。
*/
import { myMixins } from "@/mixins/myMixins.js";
export default {
  mixins: [myMixins],

/* mixins: {
    components:{},
    data() {
      return {}
    },
    created() {
      console.log('xxx from mixins')
    }
        } */
}
```

2. 简述 mixin、extends 的覆盖逻辑

（1）mixin 和 extends

相同点：mixin 和 extends均是用于合并、拓展组件的，两者均通过 **mergeOptions** 方法实现合并。

不同点：

a. mixins 接收一个混入对象的数组，其中混入对象可以像正常的实例对象一样包含实例选项，这些选项会被合并到最终的选项中。Mixin 钩子按照传入顺序依次调用，并在调用组件自身的钩子之前被调用。

b. extends 类似于mixin,相当于继承,但是只是继承 options Api 中的内容，不继承 template 模板。主要是为了便于扩展单文件组件，接收一个对象或构造函数。

属性名称	合并策略	对应合并函数
data	mixins/extends 只会将自己有的但是组件上没有内容混合到组件上，重复定义默认使用组件上的 如果data里的值是对象，将递归内部对象继续按照该策略合并	mergeDataOrFn, mergeData
provide	同上	mergeDataOrFn, mergeData
props	mixins/extends 只会将自己有的但是组件上没有内容混合到组件上	extend
methods	同上	extend
inject	同上	extend
computed	同上	extend
组件，过滤器，指令属性	同上	extend
el	同上	defaultStrat
propsData	同上	defaultStrat
watch	合并watch监控的回调方法 执行顺序是先mixins/extends里watch定义的回调，然后是组件的回掉	strats.watch
HOOKS 生命周期钩子	同一种钩子的回调函数会被合并成数组 执行顺序是先mixins/extends里定义的钩子函数，然后才是组件里定义的	mergeHook

(2) mergeOptions 的执行过程

该方法的作用是合并 options，options 除了存在于构造函数中，我们在 new Vue({}) 时传递的对象、Vue.mixin({}) 传递的对象、Vue.extend({}) 传递的对象也都是 options。

- 规范化选项（normalizeProps、normalizeInject、normalizeDirectives)
- 对未合并的选项，进行判断

```
if(!child._base) {
   if(child.extends) {
      parent = mergeOptions(parent, child.extends, vm)
   }
   if(child.mixins) {
      for(let i = 0, l = child.mixins.length; i < l; i++){
        parent = mergeOptions(parent, child.mixins[i], vm)
     }
   }
}
```

- 合并处理：根据一个通用 Vue 实例所包含的选项进行分类逐一判断合并，如 props、data、methods、watch、computed、生命周期等，将合并结果存储在新定义的 options 对象里
- 返回合并结果 options

缺点：

1. 命名冲突：使用 mixin 时，可能会出现命名冲突，特别是当不同 mixin 中具有相同名称的数据、方法或计算属性时。这可能导致意外的行为和难以调试的问题。
2. 耦合性增加：使用 mixin 可能会增加组件之间的耦合性，因为它们引入了共享的逻辑和状态。这使得组件的依赖关系变得更加复杂，并且难以跟踪和理解组件之间的关系。
3. 难以维护：随着项目的增长和需求的变化，mixin 可能会变得难以维护。当多个组件依赖相同的 mixin 时，对 mixin 的修改可能会对整个应用程序产生意想不到的影响，导致维护困难。
4. 命名空间污染：使用 mixin 可能会导致全局作用域的污染，因为 mixin 中的数据和方法会被混入到组件中，使得全局命名空间变得混乱，增加了命名冲突的可能性。

4、vue3有哪些改变?

Composition API、 TypeScript 支持、响应式 API、Async 异步组件、Teleport、Proxy-Based Reactivity System

1. 响应式系统改变：Vue2采用Object.defineProperty实现响应式系统，Vue3使用了更高效的Proxy代理对象实现响应式系统，提高了性能和稳定性。
2. 数据改变检测方式改变：Vue2采用递归的方式进行数据改变检测，Vue3使用了基于Proxy的观测机制和内部追踪之间的关系，有效解决了Vue2数据检测的性能瓶颈。
    - 检测属性的添加和删除；
    - 检测数组索引和长度的变更；
    - 支持 Map、Set、WeakMap 和 WeakSet。
3. 生命周期改变：Vue3废除了Vue2中的beforeDestroy和Destroyed钩子函数，并新增了两个钩子函数：beforeUnmount 和unmounted；此外，activated和deactivated这两个钩子在Vue3中被废弃，使用setup()返回的对象的onActivated和onDeactivated属性来替代。。
4. 异步组件改变：Vue3使用全局函数defineAsyncComponent来定义异步组件，从而方便了异步组件的使用。
5. Teleport改变：Vue3在Teleport中添加了两个插槽，分别是空插槽和to插槽，用于在传输过程中处理内容的变化。
6. 编译器改变：Vue3的编译器已被拆分为单独的包（@vue/compiler-sfc），可以进行独立地安装和使用
7. 更好的TypeScript支持：Vue3对TypeScript的支持更加友好，引入了完全支持类型推断的API和更好的类型定义。
8. Composition API改变：Vue3引入了Composition API，可以更好地组织和管理组件中的代码，提高开发效率。

5、说一下generator的原理

https://juejin.cn/post/7111347194904444958 Generator是ES6引入的一种新的函数类型，它可以让函数在执行时暂停，后续又可在需要时恢复执行。Generator是一种特殊的迭代器，用于生成一系列的值。Generator的原理可以分为以下几个步骤：

1. 当调用一个Generator函数时，它并不会立即执行，而是返回一个迭代器对象（Iterator）。
2. 当不断调用迭代器的next()方法时，Generator函数内部的代码会逐行执行，直到执行到第一个yield关键字时，代码会暂停，并将yield后面的表达式的值作为Generator函数返回对象的value属性值返回，此时yield表达式本身并没有执行。
3. 在下一次调用next()方法时，由于上一次暂停时保存的上下文（Context）信息仍然存在，所以Generator函数内部的代码会从上一次暂停的地方继续执行，直到再次执行到yield关键字，代码会再次暂停，并将最新的yield后面的表达式的值作为Generator函数返回对象的value属性值返回。
4. 通过对迭代器的不断调用next()方法，可以一步步地取出所有Generator函数中yield关键字后的表达式的值，直到函数执行结束，Generator函数返回的迭代器的done属性值变为true。

需要注意的是，Generator函数可以通过yield关键字返回任意次数的值，此外，Generator函数内部任意一处抛出异常都会导致迭代器的done属性变为true，并且抛出的异常会在外部代码中被捕获。除此之外，Generator函数还可以通过yield*关键字委托给其他Generator函数，从而实现协程的功能

只记下面的 当调用一个 Generator 函数时，它返回一个迭代器 (iterator)，并不会立即执行

```
function* generatorFn() {
  // 代码1
  yield 'Iteration 1' // 生成第1个值

  // 代码2
  yield 'Iteration 2' // 生成第2个值

  // 代码3
  yield 'Iteration 3' // 生成第3个值

  // 代码4
  return 'Completed' // 返回结果
}
```

```
const iterator = generatorFn()

console.log(iterator) // -> Generator {}

// 调用迭代器的 next 方法时，Generator 内部的代码会逐行执行，直到执行到第一个 yield 关键字时
// 此时代码会暂停，并将 yield 关键字后面的表达式的值作为 Generator 函数返回对象的 value 属性值返回
console.log(iterator.next()) // -> { value: 'Iteration 1', done: false }

// 通过对迭代器不断调用 next 方法，可以依次取出所有 Generator 函数中的值
console.log(iterator.next()) // -> { value: 'Iteration 2', done: false }
console.log(iterator.next()) // -> { value: 'Iteration 3', done: false }
console.log(iterator.next()) // -> { value: 'Completed', done: true }
console.log(iterator.next()) // -> { value: undefined, done: true }
```

当 Generator 函数内部执行到 yield 语句时，函数执行的状态会保存下来，并在 yield 语句后返回一个对象，其中 value 属性就是 yield 后面表达式的值

在下一次调用 `next()` 方法时，Generator 函数从上一次暂停的地方继续执行，直到再次执行到 yield 语句，代码会再次暂停，并将最新的 yield 后面的表达式的值作为 Generator 函数返回对象的 value 属性值返回：

```
console.log(iterator.next()) // -> { value: 'Iteration 1', done: false }
console.log(iterator.next()) // -> { value: 'Iteration 2', done: false }
console.log(iterator.next()) // -> { value: 'Iteration 3', done: false }
console.log(iterator.next()) // -> { value: 'Completed', done: true }
```

通过对迭代器的不断调用 `next()` 方法，可以一步步地取出所有 Generator 函数中 yield 后面的表达式的值，直到函数执行结束，Generator 函数返回的迭代器的 `done` 属性值变为 true。

6. 谈谈你对vue的理解

点题收敛

Vue是一个渐进式JavaScript框架，它专注于构建用户界面：Vue的核心思想是数据驱动和组件化。通过将页面拆分成独立的组件，可以更好地管理代码，提高代码的复用性和可维护性。

追问拓展

Vue的优势在于其简单易用、灵活性高、性能卓越和扩展性强。Vue的模板语法易于理解和学习，可以快速构建交互式的Web应用程序。同时，Vue的生命周期钩子和自定义指令等功能，使得Vue可以满足各种复杂的需求。另外，Vue还提供了vuex、VueRouter等官方插件，可以进一步扩展Vue的功能。

Vue的**响应式数据绑定机制**是Vue最核心的特性之一。通过对数据进行劫持和监听，可以实现数据的双向绑定，即数据变化会自动更新视图，同时视图的变化也会反映到数据上。这种机制使得u的数据流非常清晰和可预测，同时也减少了开发的工作量。

总之，我认为Vue是一个优秀的JavaScript框架，它简单易用、功能强大、扩展性好，并且有着极佳的性能表现。对于前端开发人员来说，Vue是一个值得深入学习和使用的框架。

Vue.js 是一款渐进式、轻量级的前端框架，它的核心思想是【数据驱动视图】。它通过数据劫持的方式实现了响应式编程，具有高度灵活性和可组合性。它采用了虚拟DOM和异步渲染等技术，使得应用程序在性能、可维护性和开发效率方面都得到了很大的实际提升。

Vue.js 最大的特点是简洁易学、易于上手。其组件化的设计使得页面整体的结构更加清晰明了，配合指令和事件可以大大简化前端开发的复杂性。在模板语法、组件化、数据双向绑定、计算属性、虚拟DOM、自定义指令、过滤器等方面，Vue.js都提供了方便的API支持，并且与其他主流前端库和框架兼容性良好。

此外，Vue.js在可维护性和易用性方面也非常强，有良好的文档和社区支持，为开发人员提供了强有力的工具支持，同时还提供了插件机制和可扩展性的API，以满足不同开发场景下的需求。

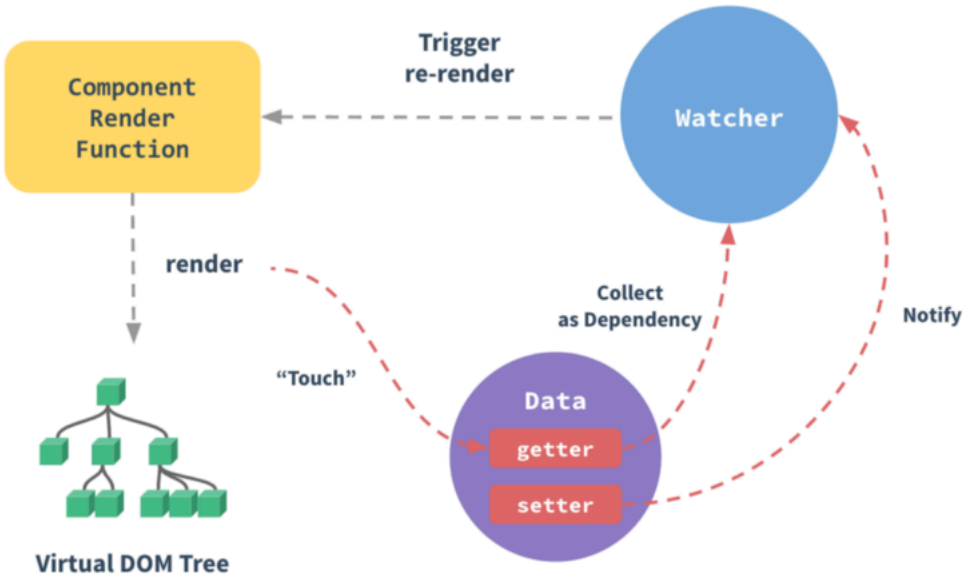
综上，Vue.js是一款非常优秀的前端框架，具有易学易用、高效灵活、易于维护扩展等优点，被广泛应用于实际的前端开发中，Vue是一个值得深入学习和使用的框架。

7. 双向数据绑定的原理

点题收敛

vue的数据绑定机制是通过**数据劫持和发布/订阅模式**实现的。当数据发生变化时，会自动更新视图，并通过虚拟DOM对比算法来提高性能。这个机制可以有效地简化开发过程，提高代码的可维护性和可读性。

如果追问深入来说



在Vue中，每个组件实例都有一个对应的响应式数据对象。当数据发生变化时，会自动更新视图。这个响应式数据对象通过数据劫持的方式实现，即通过Object.defineProperty方法来劫持数据的getter和setter方法，当数据被读取或修改时，就会触发getter或setter方法，从而实现数据的监控和响应。

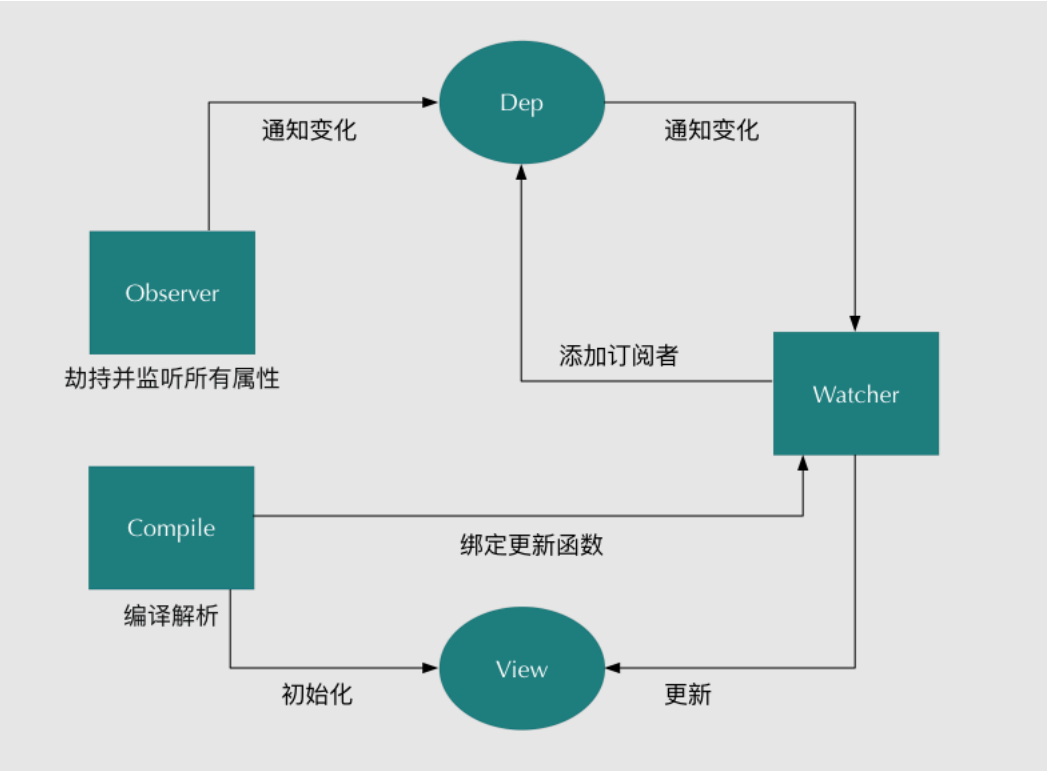
在Vue中，每个组件实例都有一个对应的Watcher对象Watcher对象订阅响应式数据对象的变化。当响应式数据对象发生变化时，Watcher>对象会接收到通知，并更新组件的视图。

Watcher对象通过发布/订阅模式实现，即Vatcher对象订阅响应式数据对象的变化，响应式数据对象在变化时发布通知，通知订阅者(Watcher>对象)进行更新。

当一个组件的数据发生变化时，Vue会通过虚拟DOM对比算法来找到变化的部分，然后更新对应的DOM节点。由于虚拟DOM对比算法只会更新变化的部分，因此可以有效地提高性能。

Vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过Object.defineProperty()来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。主要分为以下几个步骤：

1. 需要observe的数据对象进行递归遍历，包括子属性对象的属性，都加上setter和getter这样的话，给这个对象的某个值赋值，就会触发setter，那么就能监听到了数据变化
2. compile解析模板指令，将模板中的变量替换成数据，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，更新视图
3. Watcher订阅者是Observer和Compile之间通信的桥梁，主要做的事情是: ①在自身实例化时往属性订阅器(dep)里面添加自己 ②自身必须有一个update()方法 ③待属性变动dep.notice()通知时，能调用自身的update()方法，并触发Compile中绑定的回调，则功成身退。
4. MVVM作为数据绑定的入口，整合Observer、Compile和Watcher三者，通过Observer来监听自己的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据model变更的双向绑定效果。



8. 使用 Object.defineProperty() 跟Proxy进行数据劫持分别有什么缺点?

1. **Object.defineProperty()在对一些属性进行操作时，使用这种方法无法拦截**

比如通过下标方式修改数组数据或者给对象新增属性，这都不能触发组件的重新渲染，因为 Object.defineProperty 不能拦截到这些操作。更精确的来说，对于数组而言，大部分操作都是拦截不到的，只是 Vue 内部通过重写函数的方式解决了这个问题。

2. **通过使用 Proxy 对对象进行代理，从而实现数据劫持**

在 Vue3.0 中已经不使用这种方式了，而是通过使用 Proxy 对对象进行代理，从而实现数据劫持。使用Proxy的好处是它可以完美的监听到任何方式的数据改变，唯一的缺点是兼容性的问题，因为 Proxy 是 ES6 的语法。使用 1.Object.defineProperty() 进行数据劫持的缺点：

1. 只能监听属性的读取、修改操作，无法监听数组的 push、pop、shift、unshift、splice 等方法。
2. 无法监听对象的新增属性和删除操作。
3. 对象的属性必须已经存在，才能使用 Object.defineProperty() 监听它们，这样会导致新增的属性无法被监听。
4. 对象的每个属性都需要分别设置 getter 和 setter，如果对象属性过多，会导致代码冗长、不易维护。

2.使用 Proxy 进行数据劫持的缺点：

1. 不支持低版本浏览器。
2. 劫持整个对象的操作，无法针对单个属性进行监听。
3. 可能会影响性能，因为每次操作都会触发代理的拦截器方法，而这些方法的执行效率较低。
4. 程序员需要对代理的拦截器方法有一定的理解和掌握，才能使用 Proxy 进行数据劫持。

9. Computed 和 Watch 的区别

点题收敛

Vue中的computed和watch都是用来监听数据变化并做出相应的操作的，但它们的使用场景和功能有所不同。

分开介绍

computed,属性是通过计算已有的属性值得出的一个新值。computed,属性可以依赖于其他的响应式数据，当这些数据发生变化时，computed属性会自动更新。computed,属性的值会被缓存，只有在依赖数据发生变化时才会重新计算，这样可以避免重复计算和提高性能。computed,属性可以看做是一个缓存的属性，它不会直接修改数据，只是对已有数据的计算和处理。

watch属性用于监听数据的变化，并在数据变化时执行一些逻辑。watch,属性可以监听单个数据或者一个数据数组，当数据发生变化时，watch属性会执行对应的回调函数。watch属性可以用来处理一些异步操作或者需要对数据进行复杂处理的逻辑。

与computed.属性不同的是，watch属性不会缓存计算结果，它会在每次数据变化时都执行回调函数。

最后总结

总的来说，computed,属性适用于需要根据已有数据进行计算和处理的场景，而watch属性适用于需要对数据变化做出响应或者执行异步操作的场景。当需要根据已有数据计算一个新值时，使用computed属性可以提高性能。而当需要监听数据变化并执行一些逻辑时，使用watch属性可以更加灵活和方便。我们需要根据具体的业务场景选择合适的方式来监听数据变化并做出相应的处理。

对于Computed：

- 它支持缓存，只有依赖的数据发生了变化，才会重新计算
- 不支持异步，当Computed中有异步操作时，无法监听数据的变化
- computed的值会默认走缓存，计算属性是基于它们的响应式依赖进行缓存的，也就是基于data声明过，或者父组件传递过来的props中的数据进行计算的。
- 如果一个属性是由其他属性计算而来的，这个属性依赖其他的属性，一般会使用computed
- 如果computed属性的属性值是函数，那么默认使用get方法，函数的返回值就是属性的属性值；在computed中，属性有一个get方法和一个set方法，当
- 。

对于Watch：

- 它不支持缓存，数据变化时，它就会触发相应的操作
- 支持异步监听
- 监听的函数接收两个参数，第一个参数是最新的值，第二个是变化之前的值
- 当一个属性发生变化时，就需要执行相应的操作
  - 监听数据必须是data中声明的或者父组件传递过来的props中的数据，当发生变化时，会触发其他操作，函数有两个的参数：
  - immediate：组件加载立即触发回调函数
  - deep：深度监听，发现数据内部的变化，在复杂数据类型中使用，例如数组中的对象发生变化。需要注意的是，deep无法监听到数组和对象内部的变化。

当想要执行异步或者昂贵的操作以响应不断的变化时，就需要使用watch。

总结：

- computed 计算属性 : 依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值。
- watch 侦听器 : 更多的是观察的作用，无缓存性，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 选项允许执行异步操作 (访问一个 API)，限制执行该操作的频率，并在得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

10. Computed 和 Methods 的区别

可以将同一函数定义为一个 method 或者一个计算属性。对于最终的结果，两种方式是相同的

不同点：

- computed: 计算属性是基于它们的依赖进行缓存的，只有在它的相关依赖发生改变时才会重新求值；
- method 调用总会执行该函数。

11. 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、具名插槽的一个变体，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的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候，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。

12. 说一下mvc、mvp以及mvvm的区别和使用场景

MVC、MVP以及MVVM均是一种架构模式，其主要的区别在于视图层如何与模型层交互以及数据流的流向。

1. MVC（Model-View-Controller）模式：

MVC模式将应用程序分成三个主要的部分:模型、视图和控制器。模型是应用程序的核心，模型表示某些数据以及如何操作数据。视图是显示数据的地方。控制器向模型发出指令，更新模型，然后通过视图展示模型的数据。 优点：MVC模式可以将要处理的数据、数据的呈现方式和用户交互分离出来，极大地降低了应用程序各个模块间的耦合度，提高了代码的可维护性以及可扩展性。MVC模式的每个部分各司其职，代码更加清晰。 缺点：MVC模式存在的问题在于视图和控制器之间的紧密耦合，使得一个部分的改变需要牵扯到其他部分的修改，这样会导致代码难以维护。 使用场景：适用于简单的应用程序，如简单的Web应用程序、桌面应用程序等。

2. MVP（Model-View-Presenter）模式：

MVP模式是MVC模式的变体，它将控制器拆分为Presenter和View，然后通过Presenter和View之间的接口进行通信。Presenter负责从模型层读取数据，并将其提供给视图层展示，视图层向Presenter发出指令并传递用户事件，Presenter对这些事件做出反应并更新模型。 优点：MVP模式的Presenter起到了连接视图层和模型层的桥梁，可以非常有效的降低视图和模型层之间的耦合度，使得代码更容易扩展和维护。 缺点：MVP模式的缺点在于它的实现难度较高，因为需要实现观察者模式并在交互的过程中处理复杂的事件流。 使用场景：适用于大型的GUI应用程序，如WPF图形化应用程序、Android应用程序等。

3. MVVM（Model-View-ViewModel）模式：

MVVM模式是一种由Microsoft提出的模式，它将视图层和模型层分离，并引入了ViewModel来管理UI控件。ViewModel是一种特殊的控制器，它使视图和模型紧密连接在一起，ViewModel负责视图层的更新和绑定，处理用户事件并更新模型，从而实现数据双向绑定。 优点：MVVM模式的最大优点就是实现了双向绑定，并且可以将界面的数据与业务逻辑分离，有效解决了MVC和MVP模式的缺点。MVVM模式可以提高代码的可维护性以及可测试性。 缺点：MVVM模式的最大缺点就是它的实现较为复杂，需要依赖于处理数据的庞大框架，需要学习额外的语法和学习曲线。 使用场景：适用于Web应用程序和大型单页应用程序，如AngularJS和Vue.js。 总结：综合来看，MVC，MVP，MVVM三种模式各有优劣，具体应该按照实际的应用

场景选择适合的模式。MVC适合于小型的应用程序，MVP适合于大型的GUI程序，MVVM适合于Web底层框架。

13. 如何保存页面的当前的状态

既然是要保持页面的状态（其实也就是组件的状态），那么会出现以下两种情况：

- **组件会被卸载**
- **组件不会被卸载**

那么可以按照这两种情况分别得到以下方法：

组件会被卸载：

（1）将状态存储在LocalStorage / SessionStorage 只需要在组件即将被销毁的生命周期 componentWillUnmount （react）中在 LocalStorage / SessionStorage 中把当前组件的 state 通过 JSON.stringify() 储存下来就可以了。在这里面需要注意的是组件更新状态的时机。 比如从 B 组件跳转到 A 组件的时候，A 组件需要更新自身的状态。但是如果从别的组件跳转到 B 组件的时候，实际上是希望 B 组件重新渲染的，也就是不要从 Storage 中读取信息。所以需要在 Storage 中的状态加入一个 flag 属性，用来控制 A 组件是否读取 Storage 中的状态。

优点

- 兼容性好，不需要额外库或工具。
- 简单快捷，基本可以满足大部分需求。

缺点

- 状态通过 JSON 方法储存（相当于深拷贝），如果状态中有特殊情况（比如 Date 对象、Regexp 对象等）的时候会得到字符串而不是原来的值。（具体参考用 JSON 深拷贝的缺点）
- 如果 B 组件后退或者下一页跳转并不是前组件，那么 flag 判断会失效，导致从其他页面进入 A 组件页面时 A 组件会重新读取 Storage，会造成很奇怪的现象

（2）路由传值 通过 react-router 的 Link 组件的 prop —— to 可以实现路由间传递参数的效果。 在这里需要用到 state 参数，在 B 组件中通过 history.location.state 就可以拿到 state 值，保存它。返回 A 组件时再次携带 state 达到路由状态保持的效果。

优点

- 简单快捷，不会污染 LocalStorage / SessionStorage。
- 可以传递 Date、RegExp 等特殊对象（不用担心 JSON.stringify / parse 的不足）

缺点

- 如果 A 组件可以跳转至多个组件，那么在每一个跳转组件内都要写相同的逻辑。

组件不会被卸载：

（1）单页面渲染 要切换的组件作为子组件全屏渲染，父组件中正常储存页面状态。

优点

- 代码量少
- 不需要考虑状态传递过程中的错误

缺点

- 增加 A 组件维护成本
- 需要传入额外的 prop 到 B 组件
- 无法利用路由定位页面

除此之外，在Vue中，还可以是用keep-alive来缓存页面，当组件在keep-alive内被切换时组件的activated、deactivated这两个生命周期钩子函数会被执行 **被包裹在keep-alive中的组件的状态将会被保留：**

```
<keep-alive>
    <router-view v-if="$route.meta.keepAlive"></router-view>
</kepp-alive>
```

router.js

```
{
  path: '/',
  name: 'xxx',
  component: ()=>import('../src/views/xxx.vue'),
  meta:{
    keepAlive: true // 需要被缓存
  }
},
```

14. 常见的事件修饰符及其作用

- .stop：等同于 JavaScript 中的 event.stopPropagation()，防止事件冒泡；
- .prevent：等同于 JavaScript 中的 event.preventDefault()，防止执行预设的行为（如果事件可取消，则取消该事件，而不停止事件的进一步传播）；
- .capture：与事件冒泡的方向相反，事件捕获由外到内；
- .self：只会触发自己范围内的事件，不包含子元素；
- .once：只会触发一次。

15. v-if、v-show、v-html 的原理

- v-if会调用**addIfCondition**方法，生成vnode的时候会忽略对应节点，render的时候就不会渲染；
- v-show会生成vnode，render的时候也会渲染成真实节点，只是在render过程中会在节点的属性中修改show属性值，也就是常说的display；
- v-html会先移除节点下的所有节点，调用html方法，通过addProp添加innerHTML属性，归根结底还是设置innerHTML为v-html的值。

16. v-model 是如何实现的，语法糖实际是什么?

`v-model` 是 Vue 中常用的指令之一，用于双向绑定表单元素的值和数据对象的值。它的语法糖实际上是将 `v-bind` 和 `v-on` 指令结合在一起使用，简化了模板中数据绑定和事件绑定的书写方式。

例如，下面的代码：

```
<input v-model="message">
```

等价于：

```
<input :value="message" @input="message = $event.target.value">
```

这里的 `v-model` 本质上就是一个语法糖，它将属性绑定和事件绑定结合在一起，使得输入框的值随着数据对象属性的改变而改变，同时数据对象随着输入框的值的改变而改变。具体实现如下：

1. 对表单元素进行双向数据绑定

v-model 对于不同的表单元素，其实现方式略有不同，但是基本思路都是将表单元素的值与数据对象的属性进行数据绑定。

- 对于 input 和 textarea 元素，v-model 通过监听 input 事件实现数据绑定。
- 对于 checkbox 和 radio 元素，v-model 通过监听 change 事件实现数据绑定。
- 对于 select 元素，v-model 通过监听 change 事件实现数据绑定，同时可以通过 multiple 属性实现多选数据绑定。

2. 监听输入框的值变化

v-model 通过引入 oninput 或 onchange 等事件来监听表单元素的值的变化。在监听到输入框的值改变时，通过 v-bind 指令将数据对象中的属性值绑定到表单元素的 value 属性上。

3. 将输入框的值改变时的事件与数据对象绑定起来

在表单元素的值改变时，通过 v-on 指令绑定一个事件，将输入框的值改变时触发的事件与数据对象的属性绑定起来，保证数据对象属性的值与表单元素的值同步更新。 综上所述，v-model 本质上是对多个指令进行了封装和优化，使用起来更加方便和简洁。 v-model 除了可以用在表单元素中，也可以用在自定义组件中，并且在两种场景下都有不同的作用。

适用于表单元素

在表单元素中，v-model 用于将表单元素的值与数据对象的属性进行双向数据绑定，方便开发者更快速地完成表单的处理。

举例来说，下面这个简单的例子展示了如何使用 v-model 将 input 元素的值和 message 数据对象的 msg 属性绑定在一起：

```
<template>
  <div>
    <input v-model="message">
    <p>Message: {{ message }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: ''
    }
  }
}
</script>
```

在这个例子中，v-model 将 input 元素的值与 message 数据对象中的 msg 属性绑定，这样每当表单元素的值改变时，数据对象的属性值都会同步更新。

适用于组件

在自定义组件中，v-model 可以轻松地实现父子组件之间的双向数据绑定，让父组件能够更加方便地控制子组件的状态。

举例来说，在下面的组件中，我们可以将 value 属性作为父组件的数据传递给子组件 my-input，然后通过使用 v-model 来监听用户的输入，并将输出值绑定到 value 上，实现父子组件之间的数据传递。

父组件代码：

```
<template>
  <div>
    <my-input v-model="message"></my-input>
    <p>Message: {{ message }}</p>
  </div>
</template>

<script>
import MyInput from './MyInput.vue'
```

```
export default {
  components: {
    MyInput
  },
  data() {
    return {
      message: ''
    }
  }
}
</script>
```

子组件代码：

```
<template>
  <input :value="value" @input="$emit('input', $event.target.value)">
</template>

<script>
export default {
  props: ['value']
}
</script>
```

在这个例子中，父组件传递给子组件的 `message` 数据对象，可以通过使用 `v-model` 指令来在父子组件之间实现双向数据绑定。而在子组件中，需要定义一个名为 `value` 的 prop 来接收父组件传递过来的数据。同时，子组件在监听到表单元素的值变化时，需要使用 `$emit` 方法触发一个名为 `input` 的事件，并将值传递给父组件，让父组件能够自动同步更新。

17. v-model 可以被用在自定义组件上吗？如果可以，如何使用？

可以。v-model 实际上是一个语法糖，如：

实际上相当于：

```
<input
  v-bind:value="searchText"
  v-on:input="searchText = $event.target.value"
>
```

用在自定义组件上也是同理：

相当于：

```
<custom-input
  v-bind:value="searchText"
  v-on:input="searchText = $event"
></custom-input>
```

显然，custom-input 与父组件的交互如下：

1. **父组件将 searchText变量传入custom-input 组件，使用的 prop 名为 value；**
2. **custom-input 组件向父组件传出名为 input的事件，父组件将接收到的值赋值给 searchText；**

所以，custom-input 组件的实现应该类似于这样：

```
Vue.component('custom-input', {
  props: ['value'],
  template: `
    <input
      v-bind:value="value"
```

```
        v-on:input="$emit('input', $event.target.value)"
      >
    `
})
```

18. data为什么是一个函数而不是对象

JavaScript中的对象是引用类型的数据，当多个实例引用同一个对象时，只要一个实例对这个对象进行操作，其他实例中的数据也会发生变化。 而在Vue中，更多的是想要复用组件，那就需要每个组件都有自己的数据，这样组件之间才不会相互干扰。 所以组件的数据不能写成对象的形式，而是要写成函数的形式。数据以函数返回值的形式定义，这样当每次复用组件的时候，就会返回一个新的data，也就是说每个组件都有自己的私有数据空间，它们各自维护自己的数据，不会干扰其他组件的正常运行。

19. $nextTick 原理及作用

点题收敛

在Vue.js中，当我们对数据进行修改时，Vue.js会异步执行DOM更新。在某些情况下（比如数据更新后，input框重新聚焦），我们需要在DOM更新完成后执行些操作，这时就需要使用Vue.nextTick()方法。

详细拓展

Vue.nextTick()方法的实现原理是基于浏览器的异步任务队列，采用微任务优先的方式。当我们修改数据时，Vue.js会将DOM更新操作放到一个异步任务队列中，等待下一次事件循环时执行。而Vue.nextTick)方法则是将一个回调函数推入到异步任务队列中，等待DOM更新完成后执行。

具体实现方式有以下几种：

使用原生的setTimeout方法：在Vue.js2.x中，如果浏览器支持Promise,则会优先使用Promise.then()方法。如果不支持Promise,则会使用原生的setTimeout方法模拟异步操作。

使用MutationObserver:如果浏览器支持MutationObserver,Vue.js会使用MutationObserver监听DOM更新，并在DOM更新完成后执行回调函数。

使用setlmmediate:在IE中，setlmmediate方法可以用来延迟异步执行任务。在Vue.js2.x中，如果浏览器支持

setlmmediate,则会优先使用setlmmediate,否则会使用setTimeout。

最后收敛

总之，Vue.nextTick0的实现原理是利用浏览器的异步任务队列，在DOM更新完成后执行回调函数。不同浏览器支持的异步任务方法不同，Vue.js会根据浏览器的支持情况选择合适的异步任务方法。

Vue.nextTick()的意义在哪里

(理解一下)

关键点：确保我们操作的是更新后的DOM;这样做可以避免频繁的DOM操作，提高性能。

Vue.nextTick()的意义在于它可以让我们在下次DOM更新循环结束后执行回调函数，确保我们操作的是更新后的DOM。

Vue.js采用异步更新机制来提高渲染效率，当我们修改数据时，Vue.js不会立即更新DOM,而是将DOM更新操作放到一个异步队列中，等到下一次事件循环时再执行。这样做可以避免频繁的DOM操作，提高性能。

但是，由于Vue.js的异步更新机制，当我们修改数据后，如果想要立即获取更新后的DOM,可能会出现获取到的是更新前的DOM的情况。这时就需要使用Vue.nextTick)方法。

Vue.nextTick(0方法可以将回调函数推入到异步队列中，在DOM更新完成后执行。这样就可以确保我们操作的是更新后的DOM,而不是更新前的DOM。比如在某些情况下需要获取某个元素的尺寸、位置等属性时，如果不使用Vue.nextTick(0,可能会获取到错误的结果。

因此，Vue.nextTick()是一个非常实用的方法，能够确保我们在操作DOM时获取到更新后的结果，提高代码的可靠性。

智能AI

在 Vue.js 中，$nextTick 是一个非常重要的方法，可以帮助我们在 DOM 更新后执行某些操作，以实现对页面的操作更加精确和准确。

*nextTick*的原理是，通过将回调函数放入一个异步队列中，等到*DOM*更新完成后执行该回调函数。具体来说，nextTick 就是在该队列清空时执行相应的回调函数。

$nextTick 的作用主要有两个：

1. 保证在 DOM 更新后立刻获取正确的 DOM 信息。因为在修改数据后，Vue 可能需要一些时间才能更新 DOM。此时，如果我们需要获取 DOM 元素的信息，就需要等待 DOM 更新完成后再获取，否则可能会得到错误的信息。而 $nextTick 可以帮助我们在 DOM 更新后立刻获取正确的 DOM 信息。
2. 保证在 DOM 更新后立即执行某些操作。有时候，我们需要在操作 DOM 后执行一些相关的操作，如计算 DOM 元素的尺寸或位置。如果在 DOM 更新之前就执行这些操作，会得到错误的结果。而 $nextTick 可以帮助我们在 DOM 更新后立即执行这些操作，从而避免错误或不准确的结果。

举例来说，在以下情况下，我们需要使用 $nextTick：

1. 在修改数据后立即操作 DOM。
2. 在修改数据后需要使用 computed 或 watch 观察者获取更新后的数据。
3. 在修改数据后需要等待一些异步操作，如 API 请求或图片加载。这时我们需要等待异步操作完成，再更新 DOM。

综上所述，$nextTick 的作用很重要，可以保证我们在操作 DOM 时获取或使用正确的数据信息，也可以方便地执行一些异步操作，从而提高应用程序的性能和稳定性。

$nextTick 方法的实现过程是比较简单的。

首先，$nextTick 方法拥有一个异步队列，用于存放所有的回调函数。当第一个回调函数被添加进来时，就会创建一个异步的宏任务，并且记录任务的 ID。接下来，如果有其他回调函数添加到队列中，就会直接向队列中添加回调函数，而不是再创建一个宏任务。

当一个宏任务执行时，会执行所有保存在队列中的回调函数，并在执行之后，清空所有的队列。

而每当数据发生变化时，就会将 $nextTick 回调函数添加到队列中。当 Vue 更新 DOM 完成之后，就会执行这些回调函数。

下面是 $nextTick 方法的大致实现过程：

```
let callbacks = []
let pending = false

function nextTick (callback) {
  callbacks.push(callback)
  if (!pending) {
    pending = true
    Promise.resolve().then(flushCallbacks)
  }
}

function flushCallbacks () {
  pending = false
  const copies = callbacks.slice(0)
  callbacks = []
  for (let i = 0; i < copies.length; i++) {
    copies[i]()
  }
}
```

以上代码中，我们创建了一个 callbacks 数组来保存所有的回调函数，同时设置一个 pending 标志位来表示异步队列当前状态是否被 pend，如果 pend 为 false，则表示当前 async（宏任务）队列中已没有任务执行，这时就会创建一个 Promise 实例，并在 then 方法中执行 flushCallbacks 方法。当 flushCallbacks 被

调用时，会将 pending 标志位设为 false，并使用 Array.slice() 方法来深拷贝 callbacks 数组，以遍历拷贝后的数组并执行所有的回调函数。最后再将 callbacks 数组清空以便保存下一个异步队列任务。

总之，$nextTick 方法通过利用 JavaScript 异步队列机制来实现异步回调函数的延迟执行，从而保证回调函数执行时 DOM 已经完成了更新。

20. Vue template 到 render 的过程

Vue.js 在解析和编译模板时，会经过以下过程：

1. 解析模板：Vue.js 使用自定义的 HTML 解析器将模板解析成抽象语法树（AST）。解析器会分析模板中的 HTML 标记、指令、表达式和事件等内容，并构建出一颗表示模板结构的 AST。
2. 静态优化：在生成 AST 的过程中，Vue.js 会对静态内容进行优化。静态内容是指在编译过程中不会发生变化的部分，例如纯文本内容。Vue.js 会将静态内容标记为静态节点，以在后续更新过程中跳过对其的处理，提高性能。
3. 编译为渲染函数：Vue.js 将 AST 编译为渲染函数。渲染函数是一个 JavaScript 函数，它接收数据作为参数，并返回一个虚拟 DOM（VNode）树，用于渲染组件的视图。
4. 渲染视图：当组件的数据发生变化时，渲染函数会被调用，生成新的虚拟 DOM 树。Vue.js 会通过比较新旧虚拟 DOM 树的差异，计算出需要更新的部分，并将其应用到实际的 DOM 上，从而更新组件的视图。

在上述过程中，模板会被转换成一个渲染函数。渲染函数可以是以下几种形式之一：

- 使用 `render` 方法编写的渲染函数：在组件中定义了一个 `render` 方法，该方法返回一个虚拟 DOM 树。
- 使用单文件组件（.vue 文件）：Vue.js 提供了单文件组件的支持，其中的 `<template>` 部分就是模板，通过编译转换为渲染函数。
- 使用 Vue.js 的模板语法：在 Vue 组件的 `template` 选项中使用 Vue.js 的模板语法，也会被编译为渲染函数。

总之，Vue.js 将模板解析为 AST，然后根据 AST 生成渲染函数，最终通过渲染函数来渲染组件的视图。这个过程使得 Vue.js 能够高效地根据数据动态更新视图。

21. Vue data 中某一个属性的值发生改变后，视图会立即同步执行重新渲染吗?

不会立即同步执行重新渲染。Vue 实现响应式并不是数据发生变化之后 DOM 立即变化，而是按一定的策略进行 DOM 的更新。Vue 在更新 DOM 时是异步执行的。只要侦听到数据变化， Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。

如果同一个watcher被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环tick中，Vue 刷新队列并执行实际（已去重的）工作。

22. Vue 中给 data 中的对象属性添加一个新的属性时会发生什么？如何解决?

```
<template>
   <div>
      <ul>
         <li v-for="value in obj" :key="value"> {{value}} </li>
      </ul>
      <button @click="addObjB">添加 obj.b</button>
   </div>
</template>

<script>
    export default {
```

```
        data () {
           return {
               obj: {
                   a: 'obj.a'
               }
           }
        },
        methods: {
           addObjB () {
               this.obj.b = 'obj.b'
               console.log(this.obj)
           }
        }
    }
</script>
```

点击 button 会发现，obj.b 已经成功添加，但是视图并未刷新。这是因为在Vue实例创建时，obj.b并未声明，因此就没有被Vue转换为响应式的属性，自然就不会触发视图的更新，这时就需要使用Vue的全局 api $set()：

```
addObjB () (
   this.$set(this.obj, 'b', 'obj.b')
   console.log(this.obj)
}
```

$set()方法相当于手动的去把obj.b处理成一个响应式的属性，此时视图也会跟着改变了。

23. 描述下Vue自定义指令

在 Vue2.0 中，代码复用和抽象的主要形式是组件。然而，有的情况下，你仍然需要对普通 DOM 元素进行底层操作，这时候就会用到自定义指令。

一般需要对DOM元素进行底层操作时使用，尽量只用来操作 DOM展示，不修改内部的值。当使用自定义指令直接修改 value 值时绑定v-model的值也不会同步更新；如必须修改可以在自定义指令中使用keydown事件，在vue组件中使用 change事件，回调中修改vue数据;

- **（1）自定义指令基本内容**
  - **全局定义：Vue.directive("focus",{})**
  - **局部定义：directives:{focus:{}}**
  - **钩子函数：指令定义对象提供钩子函数**

```
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更新之前调用。指令的值可能发生了改变，也可能没有。但是可以通过比较更新前后的值来忽略不必要的模板更新。
ComponentUpdate：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。
```

- **钩子函数参数**

```
el：绑定元素
bing：  指令核心对象，描述指令全部信息属性
name
value
oldValue
expression
arg
modifers
vnode   虚拟节点
oldVnode：上一个虚拟节点（更新钩子函数中才有用）
```

（2）使用场景

- **普通DOM元素进行底层操作的时候，可以使用自定义指令**

- **自定义指令是用来操作DOM的。尽管Vue推崇数据驱动视图的理念，但并非所有情况都适合数据驱动。自定义指令就是一种有效的补充和扩展，不仅可用于定义任何的DOM操作，并且是可复用的。**

（3）使用案例

初级应用：

- **鼠标聚焦**
- **下拉菜单**
- **相对时间转换**
- **滚动动画**

高级应用：

- **自定义指令实现图片懒加载**
- **自定义指令集成第三方插件**

24. 子组件可以直接改变父组件的数据吗？

子组件不可以直接改变父组件的数据。这样做主要是为了维护父子组件的单向数据流。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。如果这样做了，Vue 会在浏览器的控制台中发出警告。 Vue提倡单向数据流，即父级 props 的更新会流向子组件，但是反过来则不行。这是为了防止意外的改变父组件状态，使得应用的数据流变得难以理解，导致数据流混乱。如果破坏了单向数据流，当应用复杂时，debug 的成本会非常高。 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。

25. Vue是如何收集依赖的？

在初始化 Vue 的每个组件时，会对组件的 data 进行初始化，就会将由普通对象变成响应式对象，在这个过程中便会进行依赖收集的相关逻辑，如下所示：

```
function defineReactive (obj, key, val){
  const dep = new Dep();
  ...
  Object.defineProperty(obj, key, {
    ...
    get: function reactiveGetter () {
      if(Dep.target){
        dep.depend();
        ...
      }
      return val
    }
    ...
  })
}
```

以上只保留了关键代码，主要就是 const dep = new Dep()实例化一个 Dep 的实例，然后在 get 函数中通过 dep.depend() 进行依赖收集。 （1）Dep Dep是整个依赖收集的核心，其关键代码如下：

```
class Dep {
  static target;
  subs;

  constructor () {
    ...
    this.subs = [];
  }
  addSub (sub) {
    this.subs.push(sub)
  }
  removeSub (sub) {
    remove(this.sub, sub)
  }
  depend () {
    if(Dep.target){
      Dep.target.addDep(this)
```

```
    }
  }
  notify () {
    const subs = this.subds.slice();
    for(let i = 0;i < subs.length; i++){
      subs[i].update()
    }
  }
}
```

Dep 是一个 class，其中有一个关 键的静态属性 static，它指向了一个全局唯一 Watcher，保证了同一时间全局只有一个 watcher 被计算，另一个属性 subs 则是一个 Watcher 的数组，所以 Dep 实际上就是对 Watcher 的管理，再看看 Watcher 的相关代码：

（2）Watcher

```
class Watcher {
  getter;
  ...
  constructor (vm, expression){
    ...
    this.getter = expression;
    this.get();
  }
  get () {
    pushTarget(this);
    value = this.getter.call(vm, vm)
    ...
    return value
  }
  addDep (dep){
        ...
    dep.addSub(this)
  }
  ...
}
function pushTarget (_target) {
  Dep.target = _target
}
```

Watcher 是一个 class，它定义了一些方法，其中和依赖收集相关的主要有 get、addDep 等。

（3）过程

在实例化 Vue 时，依赖收集的相关过程如下：

初始化状态 initState，这中间便会通过 defineReactive 将数据变成响应式对象，其中的 getter 部分便是用来依赖收集的。

初始化最终会走 mount 过程，其中会实例化 Watcher，进入 Watcher 中，便会执行 this.get() 方法，

```
updateComponent = () => {
  vm._update(vm._render())
}
new Watcher(vm, updateComponent)
```

get 方法中的 pushTarget 实际上就是把 Dep.target 赋值为当前的 watcher。this.getter.call（vm，vm），这里的 getter 会执行 vm.__render() 方法，在这个过程中便会触发数据对象的 getter。那么每个对象值的 getter 都持有一个 dep，在触发 getter 的时候会调用 dep.depend() 方法，也就会执行 Dep.target.addDep(this)。刚才 Dep.target 已经被赋值为 watcher，于是便会执行 addDep 方法，然后走到 dep.addSub() 方法，便将当前的 watcher 订阅到这个数据持有的 dep 的 subs 中，这个目的是为后续数据变化时候能通知到哪些 subs 做准备。所以在 vm.__render() 过程中，会触发所有数据的 getter，这样便已经完成了一个依赖收集的过程。

26、对 React 和 Vue 的理解，它们的异同

- **相似之处：**

- **都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；（Ren）**
- **都有自己的构建工具，能让你得到一个根据最佳实践设置的项目模板；**
- **都使用了Virtual DOM（虚拟DOM）提高重绘性能；（Ren）**
- **都有props的概念，允许组件间的数据传递；（Ren）**
- **都鼓励组件化应用，将应用分拆成一个个功能明确的模块，提高复用性。（Ren）**
- **都支持native源生的方案，react的RN（reactNative）和vue的weex；（Ren）**

不同之处： 1）数据流 Vue默认支持数据双向绑定，而React一直提倡单向数据流 2）虚拟DOM Vue2.x开始引入"Virtual DOM"，消除了和React在这方面的差异，但是在具体的细节还是有各自的特点。

- **Vue宣称可以更快地计算出Virtual DOM的差异，这是由于它在渲染过程中，会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。**
- **对于React而言，每当应用的状态被改变时，全部子组件都会重新渲染。当然，这可以通过PureComponent/shouldComponentUpdate这个生命周期方法来进行控制，但Vue将此视为默认的优化。**

3）组件化 React与Vue最大的不同是模板的编写。

- **Vue鼓励写近似常规HTML的模板。写起来很接近标准 HTML元素，只是多了一些属性。**
- **React推荐你所有的模板通用JavaScript的语法扩展——JSX书写。**

具体来讲：React中render函数是支持闭包特性的，所以import的组件在render中可以直接调用。但是在Vue中，由于模板中使用的数据都必须挂在 this 上进行一次中转，所以import 一个组件完了之后，还需要在 components 中再声明下。 4）监听数据变化的实现原理不同

- **Vue 通过 getter/setter 以及一些函数的劫持，能精确知道数据变化，不需要特别的优化就能达到很好的性能**
- **React 默认是通过比较引用的方式进行的，如果不优化（PureComponent/shouldComponentUpdate）可能导致大量不必要的vDOM的重新渲染。这是因为 Vue 使用的是可变数据，而React更强调数据的不可变。**

5）高阶组件 react可以通过高阶组件（HOC）来扩展，而Vue需要通过mixins来扩展。 高阶组件就是高阶函数，而React的组件本身就是纯粹的函数，所以高阶函数对React来说易如反掌。相反Vue.js使用HTML模板创建视图组件，这时模板无法有效的编译，因此Vue不能采用HOC来实现。 6）构建工具 两者都有自己的构建工具：

- **React ==> Create React APP**
- **Vue ==> vue-cli**

7）跨平台

- **React ==> React Native**
- **Vue ==> Weex**

27. Vue的优点

Vue是一个渐进式JavaScript框架，它专注于构建用户界面。Vue的核心思想是数据驱动和组件化。通过将页面拆分成独立的组件，可以更好地管理代码，提高代码的复用性和可维护性。

Vue的优势在于其简单易用、灵活性高、性能卓越和扩展性强。Vue的模板语法易于理解和学习，可以快速构建交互式的Web应用程序。同时，Vue的生命周期钩子和自定义指令等功能，使得Vue可以满足各种复杂的需求。另外，Vue还提供了Vuex、Vue Router等官方插件，可以进一步扩展Vue的功能。

Vue的响应式数据绑定机制是Vue最核心的特性之一。通过对数据进行劫持和监听，可以实现数据的双向绑定，即数据变化会自动更新视图，同时视图的变化也会反映到数据上。这种机制使得Vue的数据流非常清晰和可预测，同时也减少了开发的工作量。

总之，我认为Vue是一个优秀的JavaScript框架，它简单易用、功能强大、扩展性好，并且有着极佳的性能表现。对于前端开发人员来说，Vue是一个值得深入学习和使用的框架。

- 轻量级框架：只关注视图层，是一个构建数据的视图集合，大小只有几十 kb；
- 简单易学：国人开发，中文文档，不存在语言障碍 ，易于理解和学习；
- 双向数据绑定：保留了 angular的特点，在数据操作方面更为简单；
- 组件化：保留了 react的优点，实现了 html的封装和重用，在构建单页面应用方面有着独特的优势；
- 视图，数据，结构分离：使数据的更改更为简单，不需要进行逻辑代码的修改，只需要操作数据就能完成相关操作；
- 虚拟DOM：dom 操作是非常耗费性能的，不再使用原生的 dom操作节点，极大解放 dom操作，但具体操作的还是 dom不过是换了另一种方式；
- 运行速度更快：相比较于 react而言，同样是操作虚拟 dom，就性能而言， vue存在很大的优势。

28. assets和static的区别

相同点：assets 和 static 两个都是存放静态资源文件。项目中所需要的资源文件图片，字体图标，样式文件等都可以放在这两个文件下，这是相同点 不相同点：assets 中存放的静态资源文件在项目打包时，也就是运行 npm run build 时会将 assets 中放置的静态资源文件进行打包上传，所谓打包简单点可以理解为压缩体积，代码格式化。而压缩后的静态资源文件最终也都会放置在 static 文件中跟着 index.html 一同上传至服务器。static 中放置的静态资源文件就不会要走打包压缩格式化等流程，而是直接进入打包好的目录，直接上传至服务器。因为避免了压缩直接进行上传，在打包时会提高一定的效率，但是 static 中的资源文件由于没有进行压缩等操作，所以文件的体积也就相对于 assets 中打包后的文件提交较大点。在服务器中就会占据更大的空间。 建议： 将项目中 template需要的样式文件js文件等都可以放置在 assets 中，走打包这一流程。减少体积。而项目中引入的第三方的资源文件如 iconfoont.css 等文件可以放置在 static 中，因为这些引入的第三方文件已经经过处理，不再需要处理，直接上传。

29. delete和Vue.delete删除数组的区别

- delete只是被删除的元素变成了 empty/undefined 其他的元素的键值还是不变。
- Vue.delete 直接删除了数组 改变了数组的键值。

30. vue如何监听对象或者数组某个属性的变化

- 当在项目中直接设置数组的某一项的值，或者直接设置对象的某个属性值，这个时候，你会发现页面并没有更新。这是因为Object.defineProperty()限制，监听不到变化。 解决方式：
  - this.$set(你要改变的数组/对象，你要改变的位置/key，你要改成什么value)

```
this.$set(this.arr, 0, "OBKoro1"); // 改变数组
this.$set(this.obj, "c", "OBKoro1"); // 改变对象
```

- 调用以下几个数组的方法

splice()、 push()、pop()、shift()、unshift()、sort()、reverse() vue源码里缓存了array的原型链，然后重写了这几个方法，触发这几个方法的时候会observer数据，意思是使用这些方法不用再进行额外的操作，视图自动进行更新。 推荐使用splice方法会比较好自定义,因为splice可以在数组的任何位置进行删除/添加操作 vm.$set 的实现原理是：

- 如果目标是数组，直接使用数组的 splice 方法触发相应式；
- 如果目标是对象，会先判读属性是否存在、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）

31. Vue模版编译原理

- vue中的模板template无法被浏览器解析并渲染，因为这不属于浏览器的标准，不是正确的HTML语法，所有需要将template转化成一个JavaScript函数，这样浏览器就可以执行这一个函数并渲染出对

应的HTML元素，就可以让视图跑起来了，这一个转化的过程，就成为模板编译。模板编译又分三个阶段，解析parse，优化optimize，生成generate，最终生成可执行函数render。

- **解析阶段：使用大量的正则表达式对template字符串进行解析，将标签、指令、属性等转化为抽象语法树AST。**
- **优化阶段：遍历AST，找到其中的一些静态节点并进行标记，方便在页面重渲染的时候进行diff比较时，直接跳过这一些静态节点，优化runtime的性能。**
- **生成阶段：将最终的AST转化为render函数字符串。**

32. 对keep-alive的理解，它是如何实现的，具体缓存的是什么?

如果需要在组件切换的时候，保存一些组件的状态防止多次渲染，就可以使用 keep-alive 组件包裹需要保存的组件。 **（1）keep-alive keep-alive有以下三个属性：**

- **include 字符串或正则表达式，只有名称匹配的组件会被匹配；**
- **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。**

注意：keep-alive 包裹动态组件时，会缓存不活动的组件实例。 主要流程

1. **判断组件 name ，不在 include 或者在 exclude 中，直接返回 vnode，说明该组件不被缓存。**
2. **获取组件实例 key ，如果有获取实例的 key，否则重新生成。**
3. **key生成规则，cid +"：："+ tag ，仅靠cid是不够的，因为相同的构造函数可以注册为不同的本地组件。**
4. **如果缓存对象内存在，则直接从缓存对象中获取组件实例给 vnode ，不存在则添加到缓存对象中。**
5. **最大缓存数量，当缓存组件数量超过 max 值时，清除 keys 数组内第一个组件。**

（2）keep-alive 的实现

```
const patternTypes: Array<Function> = [String, RegExp, Array] // 接收：字符串，正则，数组

export default {
  name: 'keep-alive',
  abstract: true, // 抽象组件，是一个抽象组件：它自身不会渲染一个 DOM 元素，也不会出现在父组件链中。

  props: {
    include: patternTypes, // 匹配的组件，缓存
    exclude: patternTypes, // 不去匹配的组件，不缓存
    max: [String, Number], // 缓存组件的最大实例数量，由于缓存的是组件实例（vnode），数量过多的时候，会占用过多的内存，可以用max指定上限
  },

  created() {
    // 用于初始化缓存虚拟DOM数组和vnode的key
    this.cache = Object.create(null)
    this.keys = []
  },

  destroyed() {
    // 销毁缓存cache的组件实例
    for (const key in this.cache) {
      pruneCacheEntry(this.cache, key, this.keys)
    }
  },

  mounted() {
    // prune 削减精简[v.]
    // 去监控include和exclude的改变，根据最新的include和exclude的内容，来实时削减缓存的组件的内容
    this.$watch('include', (val) => {
      pruneCache(this, (name) => matches(val, name))
    })
    this.$watch('exclude', (val) => {
      pruneCache(this, (name) => !matches(val, name))
    })
  },
}
```

render函数：

1. **会在 keep-alive 组件内部去写自己的内容，所以可以去获取默认 slot 的内容，然后根据这个去获取组件**
2. **keep-alive 只对第一个组件有效，所以获取第一个子组件。**
3. **和 keep-alive 搭配使用的一般有：动态组件 和router-view**

```
render () {
  //
  function getFirstComponentChild (children: ?Array<VNode>): ?VNode {
    if (Array.isArray(children)) {
  for (let i = 0; i < children.length; i++) {
    const c = children[i]
    if (isDef(c) && (isDef(c.componentOptions) || isAsyncPlaceholder(c))) {
      return c
    }
  }
  }
  }
  const slot = this.$slots.default // 获取默认插槽
  const vnode: VNode = getFirstComponentChild(slot)// 获取第一个子组件
  const componentOptions: ?VNodeComponentOptions = vnode && vnode.componentOptions // 组件参数
  if (componentOptions) { // 是否有组件参数
    // check pattern
    const name: ?string = getComponentName(componentOptions) // 获取组件名
    const { include, exclude } = this
    if (
      // not included
      (include && (!name || !matches(include, name))) ||
      // excluded
      (exclude && name && matches(exclude, name))
    ) {
      // 如果不匹配当前组件的名字和include以及exclude
      // 那么直接返回组件的实例
      return vnode
    }

    const { cache, keys } = this

    // 获取这个组件的key
    const key: ?string = vnode.key == null
      // same constructor may get registered as different local components
      // so cid alone is not enough (#3269)
      ? componentOptions.Ctor.cid + (componentOptions.tag ? `::${componentOptions.tag}` : '')
      : vnode.key

    if (cache[key]) {
      // LRU缓存策略执行
      vnode.componentInstance = cache[key].componentInstance // 组件初次渲染的时候componentInstance为
undefined

      // make current key freshest
      remove(keys, key)
      keys.push(key)
      // 根据LRU缓存策略执行，将key从原来的位置移除，然后将这个key值放到最后面
    } else {
      // 在缓存列表里面没有的话，则加入，同时判断当前加入之后，是否超过了max所设定的范围，如果是，则去除
      // 使用时间间隔最长的一个
      cache[key] = vnode
      keys.push(key)
      // prune oldest entry
      if (this.max && keys.length > parseInt(this.max)) {
        pruneCacheEntry(cache, keys[0], keys, this._vnode)
      }
    }
    // 将组件的keepAlive属性设置为true
    vnode.data.keepAlive = true // 作用：判断是否要执行组件的created、mounted生命周期函数
  }
  return vnode || (slot && slot[0])
}
```

keep-alive 具体是通过 cache 数组缓存所有组件的 vnode 实例。当 cache 内原有组件被使用时会将该组件 key 从 keys 数组中删除，然后 push 到 keys数组最后，以便清除最不常用组件。

实现步骤：

1. **获取 keep-alive 下第一个子组件的实例对象，通过他去获取这个组件的组件名**

2. **通过当前组件名去匹配原来 include 和 exclude，判断当前组件是否需要缓存，不需要缓存，直接返回当前组件的实例vNode**
3. **需要缓存，判断他当前是否在缓存数组里面：**
   - **存在，则将他原来位置上的 key 给移除，同时将这个组件的 key 放到数组最后面（LRU）**
   - **不存在，将组件 key 放入数组，然后判断当前 key数组是否超过 max 所设置的范围，超过，那么削减未使用时间最长的一个组件的 key**
4. **最后将这个组件的 keepAlive 设置为 true**

（3）keep-alive 本身的创建过程和 patch 过程 缓存渲染的时候，会根据 vnode.componentInstance（首次渲染 vnode.componentInstance 为 undefined） 和 keepAlive 属性判断不会执行组件的 created、mounted 等钩子函数，而是对缓存的组件执行 patch 过程：直接把缓存的 DOM 对象直接插入到目标元素中，完成了数据更新的情况下的渲染过程。 首次渲染

- **组件的首次渲染：判断组件的 abstract 属性，才往父组件里面挂载 DOM**

```
// core/instance/lifecycle
function initLifecycle (vm: Component) {
  const options = vm.$options

  // locate first non-abstract parent
  let parent = options.parent
  if (parent && !options.abstract) { // 判断组件的abstract属性，才往父组件里面挂载DOM
    while (parent.$options.abstract && parent.$parent) {
      parent = parent.$parent
    }
    parent.$children.push(vm)
  }

  vm.$parent = parent
  vm.$root = parent ? parent.$root : vm

  vm.$children = []
  vm.$refs = {}

  vm._watcher = null
  vm._inactive = null
  vm._directInactive = false
  vm._isMounted = false
  vm._isDestroyed = false
  vm._isBeingDestroyed = false
}
```

- **判断当前 keepAlive 和 componentInstance 是否存在来判断是否要执行组件 prepatch 还是执行创建 componentInstance**

```
// core/vdom/create-component
init (vnode: VNodeWithData, hydrating: boolean): ?boolean {
    if (
      vnode.componentInstance &&
      !vnode.componentInstance._isDestroyed &&
      vnode.data.keepAlive
    ) { // componentInstance在初次是undefined!!!
      // kept-alive components, treat as a patch
      const mountedNode: any = vnode // work around flow
      componentVNodeHooks.prepatch(mountedNode, mountedNode) // prepatch函数执行的是组件更新的过程
    } else {
      const child = vnode.componentInstance = createComponentInstanceForVnode(
        vnode,
        activeInstance
      )
      child.$mount(hydrating ? vnode.elm : undefined, hydrating)
    }
  },
```

- **prepatch 操作就不会在执行组件的 mounted 和 created 生命周期函数，而是直接将 DOM 插入**
  （4）LRU （least recently used）缓存策略 LRU 缓存策略：从内存中找出最久未使用的数据并置换新的数据。 LRU（Least rencently used）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是"如果数据最近被访问过，那么将来被访问的几率也更高"。 最常见的实现是使用一个链表保存缓存数据，详细算法实现如下：
  - **新数据插入到链表头部**

- 每当缓存命中（即缓存数据被访问），则将数据移到链表头部
- 链表满的时候，将链表尾部的数据丢弃。

33.Vue中封装的数组方法有哪些，其如何实现页面更新

在Vue中，对响应式处理利用的是Object.defineProperty对数据进行拦截，而这个方法并不能监听到数组内部变化，数组长度变化，数组的截取变化等，所以需要对这些操作进行hack，让Vue能监听到其中的变化。

Vue 将被侦听的数组的变更方法进行了包裹，所以它们也将会触发视图更新。这些被包裹过的方法包括：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

那Vue是如何实现让这些数组方法实现元素的实时更新的呢，下面是Vue中对这些方法的封装：

```
// 缓存数组原型
const arrayProto = Array.prototype;
// 实现 arrayMethods.__proto__ === Array.prototype
export const arrayMethods = Object.create(arrayProto);
// 需要进行功能拓展的方法
const methodsToPatch = [
  "push",
  "pop",
  "shift",
  "unshift",
  "splice",
  "sort",
  "reverse"
];

/**
 * Intercept mutating methods and emit events
 */
methodsToPatch.forEach(function(method) {
  // 缓存原生数组方法
  const original = arrayProto[method];
  def(arrayMethods, method, function mutator(...args) {
    // 执行并缓存原生数组功能
    const result = original.apply(this, args);
    // 响应式处理
    const ob = this.__ob__;
    let inserted;
    switch (method) {
    // push、unshift会新增索引，所以要手动observer
      case "push":
      case "unshift":
        inserted = args;
        break;
      // splice方法，如果传入了第三个参数，也会有索引加入，也要手动observer。
      case "splice":
        inserted = args.slice(2);
        break;
    }
    //
    if (inserted) ob.observeArray(inserted);// 获取插入的值，并设置响应式监听
    // notify change
    ob.dep.notify();// 通知依赖更新
    // 返回原生数组方法的执行结果
    return result;
  });
});
```

简单来说就是，重写了数组中的那些原生方法，首先获取到这个数组的ob，也就是它的Observer对象，如果有新的值，就调用observeArray继续对新的值观察变化（也就是通过 target__proto__ == arrayMethods来改变了数组实例的型），然后手动调用notify，通知渲染watcher，执行update。

34.说一下Vue的生命周期

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、编译模版、挂载Dom -> 渲染、更新 -> 渲染、卸载 等一系列过程，称这是Vue的生命周期。

1. **beforeCreate（创建前）：数据观测和初始化事件还未开始，此时 data 的响应式追踪、event/watcher 都还没有被设置，也就是说不能访问到data、computed、watch、methods上的方法和数据。**
2. **created（创建后）：实例创建完成，实例上配置的 options 包括 data、computed、watch、methods 等都配置完成，但是此时渲染得节点还未挂载到 DOM，所以不能访问到 $el 属性。**
3. **beforeMount（挂载前）：在挂载开始之前被调用，相关的render函数首次被调用。实例已完成以下的配置：编译模板，把data里面的数据和模板生成html。此时还没有挂载html到页面上。**
4. **mounted（挂载后）：在el被新创建的 vm.$el 替换，并挂载到实例上去之后调用。实例已完成以下的配置：用上面编译好的html内容替换el属性指向的DOM对象。完成模板中的html渲染到html 页面中。此过程中进行ajax交互。**
5. **beforeUpdate（更新前）：响应式数据更新时调用，此时虽然响应式数据更新了，但是对应的真实 DOM 还没有被渲染。**
6. **updated（更新后）：在由于数据更改导致的虚拟DOM重新渲染和打补丁之后调用。此时 DOM 已经根据响应式数据的变化更新了。调用时，组件 DOM已经更新，所以可以执行依赖于DOM的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。**
7. **beforeDestroy（销毁前）：实例销毁之前调用。这一步，实例仍然完全可用，this仍能获取到实例。**
8. **destroyed（销毁后）：实例销毁后调用，调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务端渲染期间不被调用。**

另外还有 keep-alive 独有的生命周期，分别为 activated 和 deactivated 。用 keep-alive 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 deactivated 钩子函数，命中缓存渲染后会执行 activated 钩子函数。

35.Vue 子组件和父组件执行顺序

vue的组件加载和渲染顺序

点题收敛

首先我们来说组件的加载顺序是自上而下的，也就是先加载父组件，再加载子组件。当父组件被加载时，它会递归地加载其所有子组件，并按照顺序依次渲染它们。

组件的渲染顺序是由组件的深度优先遍历决定的，也就是先渲染最深层的子组件，再依次向上渲染其父组件。

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 beforeUpdate

3.子组件 updated

4.父组件 updated

销毁过程：

1. 父组件 beforeDestroy

2.子组件 beforeDestroy

3.子组件 destroyed

4.父组件 destoryed

36.created和mounted的区别

- **created:在模板渲染成html前调用，即通常初始化某些属性值，然后再渲染成视图。**
- **mounted:在模板渲染成html后调用，通常是初始化页面完成后，再对html的dom节点进行一些需要的操作。**

37.一般在哪个生命周期请求异步数据

我们可以在钩子函数 created、beforeMount、mounted 中进行调用，因为在这三个钩子函数中，data 已经创建，可以将服务端端返回的数据进行赋值。

推荐在 created 钩子函数中调用异步请求，因为在 created 钩子函数中调用异步请求有以下优点：

- **能更快获取到服务端数据，减少页面加载时间，用户体验更好；**
- **SSR不支持 beforeMount 、mounted 钩子函数，放在 created 中有助于一致性。**

keep-alive是 Vue 提供的一个内置组件，用来对组件进行缓存——在组件切换过程中将状态保留在内存中，防止重复渲染DOM。

如果为一个组件包裹了 keep-alive，那么它会多出两个生命周期：deactivated、activated。同时，beforeDestroy 和 destroyed 就不会再被触发了，因为组件不会被真正销毁。

当组件被换掉时，会被缓存到内存中、触发 deactivated 生命周期；当组件被切回来时，再去缓存里找这个组件、触发 activated钩子函数。

38.keep-alive 中的生命周期哪些

说下vue的keep alive

点题收敛

在Vue中，keep-alive是一个抽象组件，它可以将其包裹的组件进行缓存，从而在切换组件时可以避免重复创建和销毁组件，提高页面性能和用户体验。

面试追问深入来说

当一个组件被包裹在keep-alive中时，该组件会被缓存起来，而不是销毁。当这个组件再次被使用时，它会被从缓存中取出来并重新挂载到页面上。keep-alive提供了两个钩子函数：activated和deactivated,用来在组件被激活或停用时执行一些逻辑，比如在组件被激活时执行一些数据初始化或者异步操作。

keep-alive提供了一些配置属性，包括include、exclude、max和min等。其中，include和exclude用于指定需要缓存或排除的组件名称；max和min用于指定缓存的最大和最小数量。

最后收敛

使用keep-alive可以有效地提高页面性能和用户体验，特别是在页面中包含大量组件的情况下。但是，需要注意的是，由于keep-alive会缓存组件，因此在使用keep-alivel时需要注意数据的更新和组件的生命周期，以免出现不必要的问题。

keep-alive是 Vue 提供的一个内置组件，用来对组件进行缓存——在组件切换过程中将状态保留在内存中，防止重复渲染DOM。

如果为一个组件包裹了 keep-alive，那么它会多出两个生命周期：deactivated、activated。同时，beforeDestroy 和 destroyed 就不会再被触发了，因为组件不会被真正销毁。

当组件被换掉时，会被缓存到内存中、触发 deactivated 生命周期；当组件被切回来时，再去缓存里找这个组件、触发 activated钩子函数。

如果需要在组件切换的时候，保存一些组件的状态防止多次渲染，就可以使用 keep-alive 组件包裹需要保存的组件。

39.讲一下v-if和v-for的优先级

https://juejin.cn/post/7217810344696594488

40.Vue-Router 的懒加载如何实现

非懒加载：

```
import List from '@/components/list.vue'
const router = new VueRouter({
  routes: [
    { path: '/list', component: List }
  ]
})
```

（1）方案一(常用)：使用箭头函数+import动态加载

```
const List = () => import('@/components/list.vue')
const router = new VueRouter({
  routes: [
    { path: '/list', component: List }
  ]
})
```

（2）方案二：使用箭头函数+require动态加载

```
const router = new Router({
  routes: [
   {
     path: '/list',
     component: resolve => require(['@/components/list'], resolve)
   }
  ]
})
```

（3）方案三：使用webpack的require.ensure技术，也可以实现按需加载。 这种情况下，多个路由指定相同的chunkName，会合并打包成一个js文件。

```
// r就是resolve
const List = r => require.ensure([], () => r(require('@/components/list')), 'list');
```

```
// 路由也是正常的写法  这种是官方推荐的写的 按模块划分懒加载
const router = new Router({
  routes: [
  {
    path: '/list',
    component: List,
    name: 'list'
  }
 ]
}))
```

41.路由的hash和history模式的区别

https://juejin.cn/post/7116336664540086286

Vue-Router有两种模式：hash模式和history模式。默认的路由模式是hash模式。

1. hash模式

简介： hash模式是开发中默认的模式，它的URL带着一个#，例如：http://www.abc.com/#/vue，它的hash值就是 #/vue。 特点：hash值会出现在URL里面，但是不会出现在HTTP请求中，对后端完全没有影响。所以改变hash值，不会重新加载页面。这种模式的浏览器支持度很好，低版本的IE浏览器也支持这种模式。hash路由被称为是前端路由，已经成为SPA（单页面应用）的标配。 原理： hash模式的主要原理就是onhashchange()事件：

```
window.onhashchange = function(event){
    console.log(event.oldURL, event.newURL);
    let hash = location.hash.slice(1);
}
```

使用onhashchange()事件的好处就是，在页面的hash值发生变化时，无需向后端发起请求，window就可以监听事件的改变，并按规则加载相应的代码。除此之外，hash值变化对应的URL都会被浏览器记录下来，这样浏览器就能实现页面的前进和后退。虽然是没有请求后端服务器，但是页面的hash值和对应的URL关联起来了。

2. history模式

简介： history模式的URL中没有#，它使用的是传统的路由分发模式，即用户在输入一个URL时，服务器会接收这个请求，并解析这个URL，然后做出相应的逻辑处理。 特点： 当使用history模式时，URL就像这样：http://abc.com/user/id。相比hash模式更加好看。但是，history模式需要后台配置支持。如果后台没有正确配置，访问时会返回404。 API： history api可以分为两大部分，切换历史状态和修改历史状态：

- **修改历史状态：包括了 HTML5 History Interface 中新增的 pushState() 和 replaceState() 方法，这两个方法应用于浏览器的历史记录栈，提供了对历史记录进行修改的功能。只是当他们进行修改时，虽然修改了url，但浏览器不会立即向后端发送请求。如果要做到改变url但又不刷新页面的效果，就需要前端用上这两个API。**
- **切换历史状态： 包括 forward()、back()、go()三个方法，对应浏览器的前进，后退，跳转操作。** 是当它们执行修改时，虽然改变了当前的 URL，但浏览器不会向后端发送请求。

虽然history模式丢弃了丑陋的#。但是，它也有自己的缺点，就是在刷新页面的时候，如果没有相应的路由或资源，就会刷出404来。 如果想要切换到history模式，就要进行以下配置（后端也要进行配置）：

```
const router = new VueRouter({
  mode: 'history',
  routes: [...]
})
```

3. 两种模式对比

调用 history.pushState() 相比于直接修改 hash，存在以下优势:

- **pushState() 设置的新 URL 可以是与当前 URL 同源的任意 URL；而 hash 只可修改 # 后面的部分，因此只能设置与当前 URL 同文档的 URL；**
- **pushState() 设置的新 URL 可以与当前 URL 一模一样，这样也会把记录添加到栈中；而 hash 设置的新值必须与原来不一样才会触发动作将记录添加到栈中；**
- **pushState() 通过 stateObject 参数可以添加任意类型的数据到记录中；而 hash 只可添加短字符串；**
- **pushState() 可额外设置 title 属性供后续使用。**
- **hash模式下，仅hash符号之前的url会被包含在请求中，后端如果没有做到对路由的全覆盖，也不会返回404错误；history模式下，前端的url必须和实际向后端发起请求的url一致，如果没有对用的路由处理，将返回404错误。**

hash模式和history模式都有各自的优势和缺陷，还是要根据实际情况选择性的使用。

解决问题

生产环境 刷新 404 的解决办法可以在 `nginx` 做代理转发，在 `nginx` 中配置按顺序检查参数中的资源是否存在，如果都没有找到，让 `nginx` 内部重定向到项目首页。

```
location /test/project1 {
    root  /data/tes;
    index index.html  index.htm;
    try_files $uri $uri/ /test/project1/index.html;
}
```


Vue Router history模式上线需要注意什么

Vue Router的History模式相比于默认的Hash模式来说，能够更好地模拟传统的多页面应用的URL地址，让用户体验更加自然。但是，使用History模式需要注意以下几点事项：

后端配置：使用History模式需要后端对所有可能的路由路径都进行处理，以避免在刷新或直接输入URL时出现404错误。后端配置的方式取决于后端服务器的类型，如Apache、Nginx等，需要在服务器上进行相关配置。

安全性：使用History模式会暴露出服务器上的文件路径，因此需要特别注意安全性。在部署时需要仔细检查服务器配置，确保不会因为恶意请求而导致安全问题。

兼容性：History模式需要支持HTML5的history..oushState API,因此在一些较老的浏览器上可能会存在兼容性问题。需要在开发时做好相关的测试和兼容性处理。

打包发布：在使用ebpack等工具打包发布时，需要配置正确的publicPath,保证HTML中引用的资源路径正确。同时需要注意，如果项目使用了多个子路由，需要在打包时将所有的子路由都配置到publicPath中。

总之，使用History模式需要对后端进行相关配置，并且需要特别注意安全性和兼容性问题，同时在打包发布时需要正确配置publicPath,确保页面资源路径正确。

2. 在开发过程中，一般会使用 `location.origin` 或者 `/` 来表示网站的根目录，但是在生产环境中，应该使用实际的根目录，避免路由出现问题。可以在 `vue.config.js` 中配置 `publicPath` 来设置根目录，例如：

   ```
   module.exports = {
     publicPath: process.env.NODE_ENV === 'production' ? '/myapp/' : '/'
   }
   ```

   这样在生产环境中，路由就可以正确地使用 `/myapp/` 作为根目录了。

3. 当使用 history 模式时，需要确保没有与前端路由相冲突的 URL 路径。例如，在使用 `/about` 作为前端路由路径时，不能在其它地方重复使用 `/about`，否则可能会出现路由冲突的问题。

Vue Router history模式为什么刷新出现404

点题收敛

原因是因为浏览器在刷新页面时会向服务器发送GET请求，但此时服务器并没有相应的资源来匹配这个请求，因为在History模式下，所有路由都是在前端路由中实现的，并没有对应的后端资源文件。

解决方案

为了解决这个问题，我们需要在服务器端进行相关配置，让所有的路由都指向同一个入口文件（比如index.html），由前端路由来处理URL请求，返回对应的页面内容。具体的配置方式取决于服务器类型，常见的有Apache、Nginx等。

以Nginx为例，可以在Nginx的配置文件中加入如下代码：

```
server {
        listen 80;
        server_name yourdomain.com;

        location / {
                root /usr/share/nginx/html;
                index index.html;
                try_files $uri $uri/ /index.html;
        }
}
```

这段代码会将所有请求都指向根目录下的index.html文件，让前端路由来处理URL请求。同时需要注意，在使用History模式时需要保证所有路由的访问路径都指向index.html，否则仍然会出现404错误。

用vue-router hash模式实现锚点

使用Vue Router的Hash模式可以实现锚点跳转。

首先，在Vue Router的路由配置中，需要将mode设置为hash：

```
const router = new VueRouter({
    mode: 'hash',
    routes: [
        // 路由配置
    ]
})
```

然后，在需要跳转的地方，使用router.push方法进行路由跳转，设置目标URL的hash部分为锚点的名称：

```
this.$router.push({ path: '/yourpath#youranchor' })
```

其中，yourpath为目标路由路径，youranchor为目标锚点名称。

接着，在目标组件中，可以使用Vue的生命周期函数mounted来获取目标锚点的DOM元素，并使用scrollIntoView方法将其滚动到视图中：

```
mounted () {
    const anchor = document.getElementById('youranchor
    if (anchor) {
        anchor.scrollIntoView()
    }
}
```

其中，youranchor为目标锚点的名称，可以在模板中使用id属性设置。

这样，当路由跳转到目标页面时，页面会自动滚动到指定的锚点位置。

42.如何获取页面的hash变化

（1）监听$route的变化

```
// 监听,当路由发生变化的时候执行
watch: {
  $route: {
    handler: function(val, oldVal){
      console.log(val);
    },
    // 深度观察监听
    deep: true
  }
},
```

（2）window.location.hash读取#值

window.location.hash 的值可读可写，读取来判断状态是否改变，写入时可以在不重载网页的前提下，添加一条历史访问记录。

43.*route*和router 的区别

- **$route 是“路由信息对象”，包括 path，params，hash，query，fullPath，matched，name 等路由信息参数**
- **$router 是“路由实例”对象包括了路由的跳转方法，钩子函数等。**

44.如何定义动态路由？如何获取传过来的动态参数?

（1）param方式

- **配置路由格式：/router/:id**
- **传递的方式：在path后面跟上对应的值**
- **传递后形成的路径：/router/123**

1）路由定义

```
//在APP.vue中
<router-link :to="'/user/'+userId" replace>用户</router-link>

//在index.js
{
   path: '/user/:userid',
   component: User,
},
```

2）路由跳转

```
// 方法1:
<router-link :to="{ name: 'users', params: { uname: wade }}">按钮</router-link

// 方法2:
this.$router.push({name:'users',params:{uname:wade}})

// 方法3:
this.$router.push('/user/' + wade)
```

3）参数获取 通过 $route.params.userid 获取传递的值 （2）query方式

- **配置路由格式：/router，也就是普通配置**
- **传递的方式：对象中使用query的key作为传递方式**
- **传递后形成的路径：/route?id=123**

1）路由定义

```
//方式1：直接在router-link 标签上以对象的形式
<router-link :to="{path:'/profile',query:{name:'why',age:28,height:188}}">档案</router-link>

// 方式2：写成按钮以点击事件形式
<button @click='profileClick'>我的</button>

profileClick(){
  this.$router.push({
    path: "/profile",
    query: {
        name: "kobi",
        age: "28",
        height: 198
    }
  });
}
```

2）跳转方法

```
// 方法1：
<router-link :to="{ name: 'users', query: { uname: james }}">按钮</router-link>

// 方法2：
this.$router.push({ name: 'users', query:{ uname:james }})

// 方法3：
<router-link :to="{ path: '/user', query: { uname:james }}">按钮</router-link>

// 方法4：
this.$router.push({ path: '/user', query:{ uname:james }})

// 方法5：
this.$router.push('/user?uname=' + jsmes)
```

3）获取参数

通过$route.query 获取传递的值

45.Vue-router 路由钩子在生命周期的体现

一、Vue-Router导航守卫 有的时候，需要通过路由来进行一些操作，比如最常见的登录权限验证，当用户满足条件时，才让其进入导航，否则就取消跳转，并跳到登录页面让其登录。 为此有很多种方法可以植入路由的导航过程：全局的，单个路由独享的，或者组件级的

1. **全局路由钩子**

vue-router全局有三个路由钩子;

- **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**

具体使用：

- **beforeEach（判断是否登录了，没登录就跳转到登录页）**

```
router.beforeEach((to, from, next) => {
    let ifInfo = Vue.prototype.$common.getSession('userData');  // 判断是否登录的存储信息
    if (!ifInfo) {
        // sessionStorage里没有储存user信息
        if (to.path == '/') {
            //如果是登录页面路径，就直接next()
            next();
        } else {
            //不然就跳转到登录
            Message.warning("请重新登录！");
            window.location.href = Vue.prototype.$loginUrl;
        }
```

```
    } else {
        return next();
    }
})
```

- **afterEach （跳转之后滚动条回到顶部）**

```
router.afterEach((to, from) => {
    // 跳转之后滚动条回到顶部
    window.scrollTo(0,0);
});
```

1. **单个路由独享钩子**

beforeEnter

如果不想全局配置守卫的话，可以为某些路由单独配置守卫，有三个参数：to、from、next

```
export default [
    {
        path: '/',
        name: 'login',
        component: login,
        beforeEnter: (to, from, next) => {
            console.log('即将进入登录页面')
            next()
        }
    }
]
```

1. **组件内钩子**

beforeRouteUpdate、beforeRouteEnter、beforeRouteLeave

这三个钩子都有三个参数：to、from、next

- **beforeRouteEnter：进入组件前触发**
- **beforeRouteUpdate：当前地址改变并且改组件被复用时触发，举例来说，带有动态参数的路径foo/：id，在 /foo/1 和 /foo/2 之间跳转的时候，由于会渲染同样的foa组件，这个钩子在这种情况下就会被调用**
- **beforeRouteLeave：离开组件被调用**

注意点，beforeRouteEnter组件内还访问不到this，因为该守卫执行前组件实例还没有被创建，需要传一个回调给 next来访问，例如：

```
beforeRouteEnter(to, from, next) {
    next(target => {
        if (from.path == '/classProcess') {
            target.isFromProcess = true
        }
    })
}
```

二、Vue路由钩子在生命周期函数的体现

1. **完整的路由导航解析流程（不包括其他生命周期）**

- **触发进入其他路由。**
- **调用要离开路由的组件守卫beforeRouteLeave**
- **调用局前置守卫： beforeEach**
- **在重用的组件里调用 beforeRouteUpdate**
- **调用路由独享守卫 beforeEnter。**
- **解析异步路由组件。**
- **在将要进入的路由组件中调用 beforeRouteEnter**
- **调用全局解析守卫 beforeResolve**

- 导航被确认。
- 调用全局后置钩子的 afterEach 钩子。
- 触发DOM更新（mounted）。
- 执行beforeRouteEnter 守卫中传给 next 的回调函数

1. 触发钩子的完整顺序

路由导航、keep-alive、和组件生命周期钩子结合起来的，触发顺序，假设是从a组件离开，第一次进入b组件：

- beforeRouteLeave：路由组件的组件离开路由前钩子，可取消路由离开。
- beforeEach：路由全局前置守卫，可用于登录验证、全局路由loading等。
- beforeEnter：路由独享守卫
- beforeRouteEnter：路由组件的组件进入路由前钩子。
- beforeResolve：路由全局解析守卫
- afterEach：路由全局后置钩子
- beforeCreate：组件生命周期，不能访问tAis。
- created;组件生命周期，可以访问tAis，不能访问dom。
- beforeMount：组件生命周期
- deactivated：离开缓存组件a，或者触发a的beforeDestroy和destroyed组件销毁钩子。
- mounted：访问/操作dom。
- activated：进入缓存组件，进入a的嵌套子组件（如果有的话）。
- 执行beforeRouteEnter回调函数next。

1. 导航行为被触发到导航完成的整个过程

- 导航行为被触发，此时导航未被确认。
- 在失活的组件里调用离开守卫 beforeRouteLeave。
- 调用全局的 beforeEach守卫。
- 在重用的组件里调用 beforeRouteUpdate 守卫(2.2+)。
- 在路由配置里调用 beforeEnteY。
- 解析异步路由组件（如果有）。
- 在被激活的组件里调用 beforeRouteEnter。
- 调用全局的 beforeResolve 守卫（2.5+），标示解析阶段完成。
- 导航被确认。
- 调用全局的 afterEach 钩子。
- 非重用组件，开始组件实例的生命周期：beforeCreate&created、beforeMount&mounted
- 触发 DOM 更新。
- 用创建好的实例调用 beforeRouteEnter守卫中传给 next 的回调函数。
- 导航完成

46.Vue-router跳转和location.href有什么区别

- 使用 location.href= /url 来跳转，简单方便，但是刷新了页面；
- 使用 history.pushState(/url)，无刷新页面，静态跳转；
- 引进 router，然后使用 router.push(/url) 来跳转，使用了 diff算法，实现了按需加载，减少了 dom 的消耗。其实使用 router 跳转和使用 history.pushState() 没什么差别的，因为vue-router就是用了 history.pushState()，尤其是在history模式下。

47.params和query的区别

用法：query要用path来引入，params要用name来引入，接收参数都是类似的，分别是 this.*route.query.name*和*this.*route.params.name 。 url地址显示：query更加类似于ajax中get传参，params则类似于post，说的再简单一点，前者在浏览器地址栏中显示参数，后者则不显示 注意：query刷新不会丢失query里面的数据 params刷新会丢失 params里面的数据。

48.Vue-router 导航守卫有哪些

Vue Router提供了多种导航钩子，可以在导航过程中执行相应的操作。下面是Vue Routert中常用的导航钩子：

beforeEach:在每次路由跳转之前执行，可以用来进行用户身份验证、路由拦截等操作。

beforeResolve:在导航被确认之前，同时在所有组件内守卫和异步路由组件被解析之后执行。

afterEach:在每次路由跳转之后执行，可以用来进行路由跳转后的操作，比如页面滚动、统计PV等操作。

beforeEnter::在进入路由之前执行，与全局beforeEach的区别是它可以针对某个具体路由进行设置。

beforeRouteUpdate:在路由更新时执行，比如路由参数发生变化时。

beforeRouteLeave:在离开当前路由时执行，可以用来进行页面数据的保存或弹出提示等操作。

这些导航钩子提供了灵活的路由跳转控制机制，可以方便地实现各种复杂的路由跳转需求。同时，Vue Router还提供了一些其他的导航钩子和高级特性，比如路由元信息、动态路由、命名路由等，可以进一步提高开发效率和应用的可维护性。

- **全局前置/钩子：beforeEach、beforeResolve、afterEach**
- **路由独享的守卫：beforeEnter**
- **组件内的守卫：beforeRouteEnter、beforeRouteUpdate、beforeRouteLeave**

vue-router的核心原理

收敛点题

Vue Router是Vue.js官方提供的一款路由管理器，它通过监听URL变化，匹配路由规则，展示对应的组件内容，从而实现单页应用的路由控制。

详细深入

Vue Router的核心原理包括以下几个方面（这玩意记不住就先说下面的总结收敛的，然后再回忆，相对低频）

路由匹配：Vue Router通过定义路由规则来匹配URL路径，并根据匹配结果展示对应的组件内容。路由规则可以使用路径、参数、查询参数等多种方式进行定义，同时支持嵌套路由和命名路由等高级特性。

路由模式：Vue Router3支持两种路由模式，分别是Hash模式和History模式。在Hash模式下，路由信息会被保存在URL的Hash部分，通过监听Hash变化来进行路由控制；在History模式下，

路由信息会被保存在浏览器的History API中，通过修改浏览器历史记录来进行路由控制。

路由导航：Vue Router中的导航钩子可以监听路由变化，进行路由拦截、身份验证等操作。导航钩子包括全局导航钩子和组件内导航钩子，可以在路由跳转前、跳转后、路由更新等不同阶段执行相应的逻辑。

路由组件：Vue Router通过组件的动态加载来实现异步路由组件，可以根据需要动态加载路由组件，从而提高应用的性能和用户体验。同时，Vue Router还支持路由懒加载、路由元信息等高级特性，可以进一步提高应用的灵活性和可维护性。

总结收敛

总之，Vue Router是实现Vue.js单页应用路由控制的核心组件之它通过路由匹配、路由模式、路由导航、路由组件等多个方面实现了完整的路由控制逻辑，为开发者提供了强大的路由控制能力。

49.Vuex 的原理

回答：vuex是什么？ vuex里的各个属性是什么意思？vuex是单项数据流的好处？总结

点题收敛

Vuex是一个专门为Vue.js开发的状态管理库，它提供了一个集中式的状态管理机制，用于管理Vue应用中的所有组件的共享状态。Vuex的核心思想是将组件的共享状态抽离出来，以单独的状态树的形式存储，然后通过定义一系列的mutations、actions、getters:来操作这个状态树.

详细回答

Vuex的核心概念包括：state、mutations、actionsi和getters。.其中，state是应用的状态，而mutations用于修改state中的状态。actions则用于处理异步操作或批量的同步操作，最终通过mutations来改变state。.getters!则用于对state中的数据进行计算或过滤。

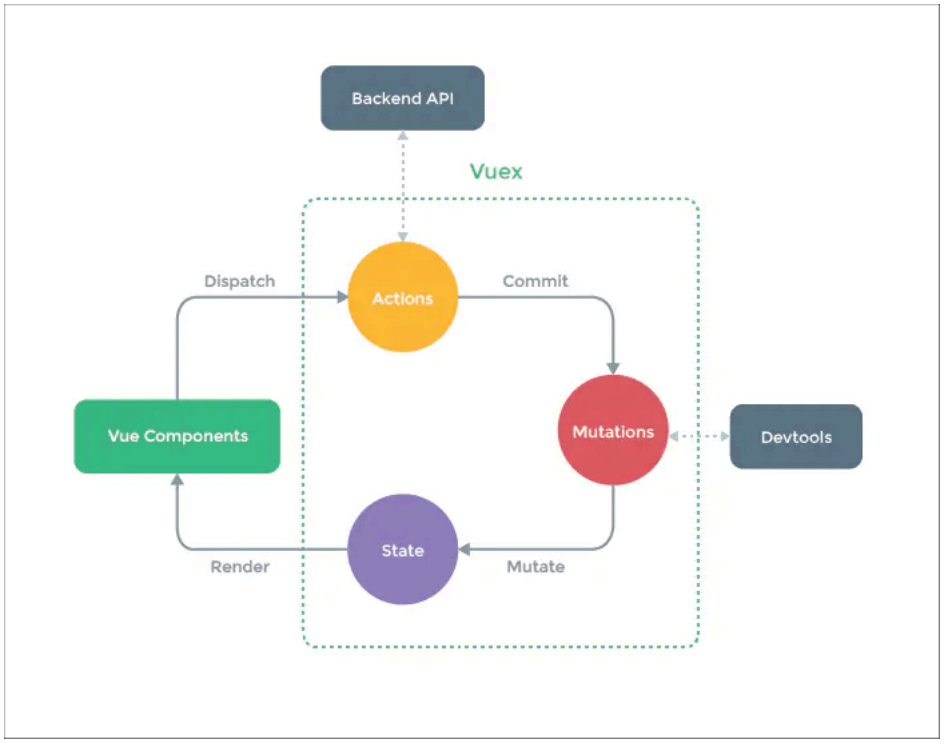
在Vuex中，数据流的流向是单向的，即从state到组件，再从组件到mutations/.actions。.这种单向数据流的机制使得数据的流动更加清晰，同时也更容易进行调试和维护。而Vuex还提供了一些辅助函数，比如napState、.mapGetters、mapActions和mapMutations等，用于方便地访问和操作状态树。

总结收敛

总之，Vuex是Vue.js生态中的一个非常重要的插件，适用于中大型的Vue.js)应用，它通过提供集中式的状态管理机制，帮助我们更好地管理数据流，提高应用的可维护性和可扩展性。同时，Vuex还有一些高级特性，比如模块化的状态管理和插件机制，能够进一步提高我们的开发效率。

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，它包含着你的应用中大部分的状态 (state)。

- **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。**
- **改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样可以方便地跟踪每一个状态的变化。**



Vuex为Vue Components建立起了一个完整的生态圈，包括开发中的API调用一环。

（1）核心流程中的主要功能：

- **Vue Components 是 vue 组件，组件会触发（dispatch）一些事件或动作，也就是图中的 Actions;**
- **在组件中发出的动作，肯定是想获取或者改变数据的，但是在 vuex 中，数据是集中管理的，不能直接去更改数据，所以会把这个动作提交（Commit）到 Mutations 中;**
- **然后 Mutations 就去改变（Mutate）State 中的数据;**
- **当 State 中的数据被改变之后，就会重新渲染（Render）到 Vue Components 中去，组件展示更新后的数据，完成一个流程。**

（2）各模块在核心流程中的主要功能：

- **Vue Components：Vue组件。HTML页面上，负责接收用户操作等交互行为，执行dispatch方法触发对应action进行回应。**
- **dispatch：操作行为触发方法，是唯一能执行action的方法。**
- **actions：操作行为处理模块。负责处理Vue Components接收到的所有交互行为。包含同步/异步操作，支持多个同名方法，按照注册的顺序依次触发。向后台API请求的操作就在这个模块中进行，包括触发其他action以及提交mutation的操作。该模块提供了Promise的封装，以支持action的链式触发。**
- **commit：状态改变提交操作方法。对mutation进行提交，是唯一能执行mutation的方法。**
- **mutations：状态改变操作方法。是Vuex修改state的唯一推荐方法，其他修改方式在严格模式下将会报错。该方法只能进行同步操作，且方法名只能全局唯一。操作之中会有一些hook暴露出来，以进行state的监控等。**
- **state：页面状态管理容器对象。集中存储Vuecomponents中data对象的零散数据，全局唯一，以进行统一的状态管理。页面显示所需的数据从该对象中进行读取，利用Vue的细粒度数据响应机制来进行高效的状态更新。**
- **getters：state对象读取方法。图中没有单独列出该模块，应该被包含在了render中，Vue Components通过该方法读取全局state对象。**

50.Vuex中action和mutation的区别

mutation中的操作是一系列的同步函数，用于修改state中的变量的的状态。当使用vuex时需要通过commit来提交需要操作的内容。mutation 非常类似于事件：每个 mutation 都有一个字符串的 事件类型 (type) 和 一个 回调函数 (handler)。这个回调函数就是实际进行状态更改的地方，并且它会接受 state 作为第一个参数：

```
const store = new Vuex.Store({
  state: {
    count: 1
  },
  mutations: {
    increment (state) {
      state.count++      // 变更状态
    }
  }
})
```

当触发一个类型为 increment 的 mutation 时，需要调用此函数：

store.commit('increment')

而Action类似于mutation，不同点在于：

- **Action 可以包含任意异步操作。**
- **Action 提交的是 mutation，而不是直接变更状态。**

```
const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      state.count++
    }
  },
  actions: {
    increment (context) {
      context.commit('increment')
    }
  }
})
```

- **Action 函数接受一个与 store 实例具有相同方法和属性的 context 对象，因此你可以调用 context.commit 提交一个 mutation，或者通过 context.state 和 context.getters 来获取 state 和 getters。 所以，两者的不同点如下：**

- Mutation专注于修改State，理论上是修改State的唯一途径；Action业务代码、异步请求。
- Mutation：必须同步执行；Action：可以异步，但不能直接操作State。
- 在视图更新时，先触发actions，actions再触发mutation
- mutation的参数是state，它包含store中的数据；actions的参数是context，它是 state 的父级，包含 state、getters

51.Vuex 和 localStorage 的区别

（1）最重要的区别

- vuex存储在内存中
- localstorage 则以文件的方式存储在本地，只能存储字符串类型的数据，存储对象需要 JSON的stringify和parse方法进行处理。 读取内存比读取硬盘速度要快

（2）应用场景

- Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。vuex用于组件之间的传值。
- localstorage是本地存储，是将数据存储到浏览器的方法，一般是在跨页面传递数据时使用 。
- Vuex能做到数据的响应式，localstorage不能

（3）永久性

刷新页面时vuex存储的值会丢失，localstorage不会。 注意：对于不变的数据确实可以用localstorage可以代替vuex，但是当两个组件共用一个数据源（对象或数组）时，如果其中一个组件改变了该数据源，希望另一个组件响应该变化时，localstorage无法做到，原因就是区别1。

52.Redux 和 Vuex 有什么区别，它们的共同思想

（1）Redux 和 Vuex区别

- Vuex改进了Redux中的Action和Reducer函数，以mutations变化函数取代Reducer，无需switch，只需在对应的mutation函数里改变state值即可
- Vuex由于Vue自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的State即可
- Vuex数据流的顺序是：View调用store.commit提交对应的请求到Store中对应的mutation函数->store改变（vue检测到数据变化自动渲染）

通俗点理解就是，vuex 弱化 dispatch，通过commit进行 store状态的一次更变;取消了action概念，不必传入特定的 action形式进行指定变更;弱化reducer，基于commit参数直接对数据进行转变，使得框架更加简易;

（2）共同思想

- 单—的数据源
- 变化可以预测

本质上：redux与vuex都是对mvvm思想的服务，将数据从视图中抽离的一种方案;

形式上：vuex借鉴了redux，将store作为全局的数据中心，进行mode管理;

53.为什么要用 Vuex 或者 Redux

由于传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致代码无法维护。

所以需要把组件的共享状态抽取出来，以一个全局单例模式管理。在这种模式下，组件树构成了一个巨大的"视图"，不管在树的哪个位置，任何组件都能获取状态或者触发行为。

另外，通过定义和隔离状态管理中的各种概念并强制遵守一定的规则，代码将会变得更结构化且易维护。

54.Vuex有哪几种属性?

有五种，分别是 State、 Getter、Mutation 、Action、 Module

- state => 基本数据(数据源存放地)
- getters => 从基本数据派生出来的数据
- mutations => 提交更改数据的方法，同步
- actions => 像一个装饰器，包裹mutations，使之可以异步。
- modules => 模块化Vuex

55.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。这样可以方便地跟踪每一个状态的变化，有利于我们调试和维护。

56.为什么 Vuex 的 mutation 中不能做异步操作?

- Vuex中所有的状态更新的唯一途径都是mutation，异步操作通过 Action 来提交 mutation实现，这样可以方便地跟踪每一个状态的变化，有利于我们调试和维护。
- 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，无法很好的进行状态的追踪，给调试带来困难。

57.Vuex的严格模式是什么,有什么作用，如何开启

在严格模式下，无论何时发生了状态变更且不是由mutation函数引起的，将会抛出错误。这能保证所有的状态变更都能被调试工具跟踪到。

在Vuex.Store 构造器选项中开启,如下

```
const store = new Vuex.Store({
    strict:true,
})
```

58.如何在组件中批量使用Vuex的getter属性

使用mapGetters辅助函数, 利用对象展开运算符将getter混入computed 对象中

```
import {mapGetters} from 'vuex'
export default{
    computed:{
        ...mapGetters(['total','discountTotal'])
    }
}
```

59.如何在组件中重复使用Vuex的mutation

使用mapMutations辅助函数,在组件中这么使用

```
import { mapMutations } from 'vuex'
methods:{
    ...mapMutations({
        setNumber:'SET_NUMBER',
    })
}
```

然后调用 this.setNumber(10)相当调用 this.$store.commit('SET_NUMBER',10)

60.defineProperty和proxy的区别

Vue 在实例初始化时遍历 data 中的所有属性，并使用 Object.defineProperty 把这些属性全部转为 getter/setter。这样当追踪数据发生变化时，setter 会被自动调用。 Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是 Vue 不支持 IE8 以及更低版本浏览器的原因。 但是这样做有以下问题：

1. **添加或删除对象的属性时，Vue 检测不到。因为添加或删除的对象没有在初始化进行响应式处理，只能通过 $set 来调用 Object.defineProperty()处理。**
    1. 属性的新增和删除
2. **无法监控到数组下标和长度的变化。**
    1. 直接通过下标赋值 arr[i] = value
    2. 直接修改数组长度 arr.length = newLen

Vue3 使用 Proxy 来监控数据的变化。Proxy 是 ES6 中提供的功能，其作用为：用于定义基本操作的自定义行为（如属性查找，赋值，枚举，函数调用等）。相对于 Object.defineProperty()，其有以下特点：

1. **Proxy 直接代理整个对象而非对象属性，这样只需做一层代理就可以监听同级结构下的所有属性变化，包括新增属性和删除属性。**
2. **Proxy 可以监听数组的变化。**

61.Vue3.0 为什么要用 proxy

1. Proxy 是 JavaScript 的标准规范（不支持低版本浏览器）

Proxy 是 JavaScript 语言的标准规范，而 Object.defineProperty 只是浏览器对于 JavaScript 标准的一个实现，不同的浏览器对于这个实现的支持存在一些小差异，例如 IE 8 及以下版本不支持。而 Proxy 作为 JavaScript 的标准规范，在所有支持 JavaScript 的平台上都可以使用，具有更广泛的兼容性。

2. Proxy 可以监听整个对象

对于 Object.defineProperty，只能在对象的每个属性上进行监听，当对象属性数量很多时，需要进行大量的监听，会比较耗费性能。而 Proxy 可以监听整个对象，不需要多次设置监听器，从而减少监听的数量，提高性能。

3. Proxy 支持对于数组的监听

Object.defineProperty 对于数组只能在特定情况下监听，例如 push、pop 等操作，而对于数组的其他操作无法监听。而 Proxy 可以完全监听数组的变化，对于数组的所有操作都可以进行监听，例如 splice、sort 等操作，使得数组的监听更加准确、完整。

4. Proxy 可以减少对于嵌套对象的递归监听

对于嵌套对象，如果使用 Object.defineProperty 进行监听，则需要递归遍历每个对象属性进行监听，这样的性能开销较大。而 Proxy 可以通过递归监听整个对象，从而减少了对于嵌套对象的递归监听，提高了效率。
综上所述，Vue3.0 选择使用 Proxy 作为其响应式系统的实现，可以充分发挥 Proxy 的优势，在性能、规范性、灵活性等方面进行提升，使得 Vue3.0 的响应式系统更加强大、高效。

在 Vue2 中， 0bject.defineProperty 会改变原始数据，而 Proxy 是创建对象的虚拟表示，并提供 set 、get 和 deleteProperty 等处理器，这些处理器可在访问或修改原始对象上的属性时进行拦截，有以下特点

：

- **不需用使用 Vue.*set*****或*****V ue*.delete 触发响应式。**
- **全方位的数组变化检测，消除了Vue2 无效的边界情况。**
- **支持 Map，Set，WeakMap 和 WeakSet。**

Proxy 实现的响应式原理与 Vue2的实现原理相同，实现方式大同小异：

- **get 收集依赖**
- **Set、delete 等触发依赖**
- **对于集合类型，就是对集合对象的方法做一层包装：原方法执行后执行依赖相关的收集或触发逻辑。**

62.Vue 3.0 中的 Vue Composition API

在 Vue2 中，代码是 Options API 风格的，也就是通过填充 (option) data、methods、computed 等属性来完成一个 Vue 组件。这种风格使得 Vue 相对于 React极为容易上手，同时也造成了几个问题：

1. **由于 Options API 不够灵活的开发方式，使得Vue开发缺乏优雅的方法来在组件间共用代码。**
2. **Vue 组件过于依赖 this上下文，Vue 背后的一些小技巧使得 Vue 组件的开发看起来与 JavaScript 的开发原则相悖，比如在 methods中的 this竟然指向组件实例来不指向 methods所在的对象。这也使得 TypeScript 在Vue2 中很不好用。**

于是在 Vue3 中，舍弃了 Options API，转而投向 Composition API。Composition API本质上是将 Options API 背后的机制暴露给用户直接使用，这样用户就拥有了更多的灵活性，也使得 Vue3 更适合于 TypeScript 结合。

如下，是一个使用了 Vue Composition API 的 Vue3 组件：

```
<template>
  <button @click="increment">
    Count: {{ count }}
  </button>
</template>

<script>
// Composition API 将组件属性暴露为函数，因此第一步是导入所需的函数
import { ref, computed, onMounted } from 'vue'

export default {
  setup() {
// 使用 ref 函数声明了称为 count 的响应属性，对应于Vue2中的data函数
    const count = ref(0)

// Vue2中需要在methods option中声明的函数，现在直接声明
    function increment() {
      count.value++
    }
 // 对应于Vue2中的mounted声明周期
    onMounted(() => console.log('component mounted!'))

    return {
      count,
      increment
    }
  }
}
</script>
```

显而易见，Vue Composition API 使得 Vue3 的开发风格更接近于原生 JavaScript，带给开发者更多地灵活性

63.Composition API与React Hook很像，区别是什么

从React Hook的实现角度看，React Hook是根据useState调用的顺序来确定下一次重渲染时的state是来源于哪个useState，所以出现了以下限制

- **不能在循环、条件、嵌套函数中调用Hook**
- **必须确保总是在你的React函数的顶层调用Hook**
- **useEffect、useMemo等函数必须手动确定依赖关系**

而Composition API是基于Vue的响应式系统实现的，与React Hook的相比

- **声明在setup函数内，一次组件实例化只调用一次setup，而React Hook每次重渲染都需要调用Hook，使得React的GC比Vue更有压力，性能也相对于Vue来说也较慢**
- **Compositon API的调用不需要顾虑调用顺序，也可以在循环、条件、嵌套函数中使用**
- **响应式系统自动实现了依赖收集，进而组件的部分的性能优化由Vue内部自己完成，而React Hook需要手动传入依赖，而且必须必须保证依赖的顺序，让useEffect、useMemo等函数正确的捕获依赖变量，否则会由于依赖不正确使得组件性能下降。**

虽然Compositon API看起来比React Hook好用，但是其设计思想也是借鉴React Hook的。

64.vue和react的区别

1. 响应式原理不同：Vue 的响应式原理是基于 Object.defineProperty 实现的数据双向绑定，而 React 则是使用单向数据流，通过 JSX 实现 UI 的渲染。
2. 组件之间通讯方式有所不同：Vue 使用 props 和 $emit 实现子组件向祖先组件传递数据，而 React 则是通过 props 和回调函数实现传递数据。
3. 所需的学习曲线不同：Vue 比 React 更容易学习和使用，因为 Vue 拥有更简单的模板语法和 Options API。而 React 则需要掌握 JSX、Hooks 等较新的概念。
4. 后续维护成本不同：Vue 的模板语法和 Options API 技巧稍多，但模板语法存在较强的限制，比如无法使用大多数 JavaScript 特性和方法和 JSX。而 React 是使用 JavaScript 编写，灵活性高，但它需要更深的理解和掌握。
5. 工具链不同：Vue 比 React 更注重完整的工具链。Vue 拥有 CLI 和官方的 Vuex、Vue Router 等库，它们可以方便地完成各项任务，而 React 则需要通过第三方工具或库来完成。

总的来说，Vue 和 React 的区别是基于响应式原理不同，组件通讯不同，学习曲线不同，后续维护成本不同，工具链不同等方面。选择 Vue 还是 React，取决于项目的需求和开发团队的技能水平。

65.对虚拟DOM的理解?

从本质上来说，Virtual Dom是一个JavaScript对象，通过对象的方式来表示DOM结构。将页面的状态抽象为JS对象的形式，配合不同的渲染工具，使跨平台渲染成为可能。通过事务处理机制，将多次DOM修改的结果一次性的更新到页面上，从而有效的减少页面渲染的次数，减少修改DOM的重绘重排次数，提高渲染性能。 虚拟DOM是对DOM的抽象，这个对象是更加轻量级的对 DOM的描述。它设计的最初目的，就是更好的跨平台，比如Node.js就没有DOM，如果想实现SSR，那么一个方式就是借助虚拟DOM，因为虚拟DOM本身是js对象。 在代码渲染到页面之前，vue会把代码转换成一个对象（虚拟 DOM）。以对象的形式来描述真实DOM结构，最终渲染到页面。在每次数据发生变化前，虚拟DOM都会缓存一份，变化之时，现在的虚拟DOM会与缓存的虚拟DOM进行比较。在vue内部封装了diff算法，通过这个算法来进行比较，渲染时修改改变的变化，原先没有发生改变的通过原先的数据进行渲染。 另外现代前端框架的一个基本要求就是无须手动操作DOM，一方面是因为手动操作DOM无法保证程序性能，多人协作的项目中如果review不严格，可能会有开发者写出性能较低的代码，另一方面更重要的是省略手动DOM操作可以大大提高开发效率。

66.虚拟DOM的解析过程

虚拟DOM的解析过程：

- **首先对将要插入到文档中的 DOM 树结构进行分析，使用 js 对象将其表示出来，比如一个元素对象，包含 TagName、props 和 Children 这些属性。然后将这个 js 对象树给保存下来，最后再将 DOM 片段插入到文档中。**

- **当页面的状态发生改变，需要对页面的 DOM 的结构进行调整的时候，首先根据变更的状态，重新构建起一棵对象树，然后将这棵新的对象树和旧的对象树进行比较，记录下两棵树的的差异。**
- **最后将记录的有差异的地方应用到真正的 DOM 树中去，这样视图就更新了。**

67.为什么要用虚拟DOM

（1）保证性能下限，在不进行手动优化的情况下，提供过得去的性能 看一下页面渲染的流程：解析HTML -> 生成DOM->生成 CSSOM->Layout->Paint->Compiler 下面对比一下修改DOM时真实DOM操作和Virtual DOM的过程，来看一下它们重排重绘的性能消耗：

- 真实DOM： 生成HTML字符串 + 重建所有的DOM元素
- 虚拟DOM： 生成vNode+ DOMDiff + 必要的dom更新

Virtual DOM的更新DOM的准备工作耗费更多的时间，也就是JS层面，相比于更多的DOM操作它的消费是极其便宜的。尤雨溪在社区论坛中说道： 框架给你的保证是，你不需要手动优化的情况下，依然可以给你提供过得去的性能。 **（2）跨平台** Virtual DOM本质上是JavaScript的对象，它可以很方便的跨平台操作，比如服务端渲染、uniapp等。 （3）开发效率： 使用虚拟 DOM，**更重要的是省略手动DOM操作可以大大提高开发效率。**

68.虚拟DOM真的比真实DOM性能好吗

- 虚拟 DOM 虽然是一种抽象的技术，但它的确可以提高程序的性能。虚拟 DOM 通过在 JavaScript 内存中创建一个以 JavaScript 对象为基础的模拟 DOM 树来实现。使用虚拟 DOM 的优点在于，虚拟 DOM 可以在 DOM 操作前建立一套完整的操作流程，然后一次性地将这些修改应用到真实的 DOM 上，避免了频繁地访问和操作真实 DOM 对页面性能的影响。

具体来说，虚拟 DOM 做到了以下几点：

1. 减少了对 DOM 的操作次数。通过 diff 算法和 vdom 某些特性的优化，可以使得对 DOM 的操作次数大大减少，从而缓解了页面操作时的性能问题。
2. 减少了对 DOM 的访问次数。虚拟 DOM 可以将多次 DOM 操作合并为一次更新，减少了访问 DOM 的次数，从而避免了频繁地访问多个 DOM 节点对页面性能的影响。
3. **保证性能下限。**使用虚拟 DOM 对操作流程建立了一套完整的抽象过程，使得性能问题可以被精准地监测，而不是在真实 DOM 上软件堆栈中被淹没。

虽然虚拟 DOM 确实有一些优势，但是在被过度使用时，也可能出现性能问题。特别是对于那些只有简单操作和少量数据，在其他方面并没有什么性能瓶颈的应用，使用虚拟 DOM 并不一定带来任何性能的提升，甚至可能比直接操作原生的 DOM 更慢。所以，是否使用虚拟 DOM，还需要根据具体情况灵活抉择，不能一概而论。

69.Vue Router history 模式为什么刷新出现404

原因是因为浏览器在刷新页面时会向服务器发送GET请求，但此时服务器并没有相应的资源来匹配这个请求，因为在History模式下，所有路由都是在前端路由中实现的，并没有对应的后端资源文件。

解决方案 为了解决这个问题，我们需要在服务器端进行相关配置，让所有的路由都指向同一个入口文件（比如index.html），由前端路由来处理URL请求，返回对应的页面内容。具体的配置方式取决于服务器类型，常见的有Apache、Nginx等。 以Nginx为例，可以在Nginx的配置文件中加入如下代码

```
server {
    listen 80;
    server_name yourdomain.com;

    location / {
        root /usr/share/nginx/html;
        index index.html;
        try_files $uri $uri/ /index.html;
```

```
    }
}
```

这段代码会将所有请求都指向根目录下的index.html文件，让前端路由来处理URL请求。同时需要注意，在使用History模式时需要保证所有路由的访问路径都指向index.html，否则仍然会出现404错误。

70.Vue Router history 模式上线需要注意什么事项

Vue Router的History模式相比于默认的Hash模式来说，能够更好地模拟传统的多页面应用的URL地址，让用户体验更加自然。但是，使用History模式需要注意以下几点事项： **后端配置**：使用History模式需要后端对所有可能的路由路径都进行处理，以避免在刷新或直接输入URL时出现404错误。后端配置的方式取决于后端服务器的类型，如Apache、Nginx等，需要在服务器上进行相关配置。 **安全性**：使用History模式会暴露出服务器上的文件路径，因此需要特别注意安全性。在部署时需要仔细检查服务器配置，确保不会因为恶意请求而导致安全问题。 **兼容性**：History模式需要支持HTML5的history.pushState API，因此在一些较老的浏览器上可能会存在兼容性问题。需要在开发时做好相关的测试和兼容性处理。 **打包发布**：在使用Webpack等工具打包发布时，需要配置正确的publicPath，保证HTML中引用的资源路径正确。同时需要注意，如果项目使用了多个子路由，需要在打包时将所有的子路由都配置到publicPath中。 总之，使用History模式需要对后端进行相关配置，并且需要特别注意安全性和兼容性问题，同时在打包发布时需要正确配置publicPath，确保页面资源路径正确。

71.用vue-router hash模式实现锚点

使用Vue Router的Hash模式可以实现锚点跳转。 首先，在Vue Router的路由配置中，需要将mode设置为hash：

```
const router = new VueRouter({
  mode: 'hash',
  routes: [
    // 路由配置
  ]
})
```

然后，在需要跳转的地方，使用router.push方法进行路由跳转，设置目标URL的hash部分为锚点的名称：

```
this.$router.push({ path: '/yourpath#youranchor' })
```

其中，yourpath为目标路由路径，youranchor为目标锚点名称。

接着，在目标组件中，可以使用Vue的生命周期函数mounted来获取目标锚点的DOM元素，并使用scrollIntoView方法将其滚动到视图中：

```
mounted () {
  const anchor = document.getElementById('youranchor')
  if (anchor) {
    anchor.scrollIntoView()
  }
}
```

其中，youranchor为目标锚点的名称，可以在模板中使用id属性设置。

这样，当路由跳转到目标页面时，页面会自动滚动到指定的锚点位置。

72.说下虚拟DOM和diff算法，key的作用

虚拟DOM和diff算法是React\vue中的两个核心概念。 虚拟DOM是指用JavaScript对象模拟DOM树结构，包括节点的类型、属性和子节点等信息。当状态发生变化时，React会使用新的状态生成一个新的虚拟DOM

树，并通过对比新旧虚拟DOM树的差异（diff算法），计算出需要更新的节点，最终只更新需要更新的节点，从而提高性能。 diff算法是指在两个树形结构之间找出差异的算法。在React中，通过对比新旧虚拟DOM树节点的不同，分为以下三种情况：

1. 替换节点：节点的类型发生了变化，例如从div变成了p。
2. 更新属性：节点的属性发生了变化，例如class、style等。
3. 更新子节点：节点的子节点发生了变化。

在diff算法的过程中，key的作用是给每个虚拟DOM节点添加一个唯一的标识符。这样在进行新旧虚拟DOM对比时，可以通过key值的对比快速判断是否是同一个节点，避免不必要的DOM操作。如果不添加key，diff算法只能通过遍历子节点的方式查找，效率较低。

Vue 也采用了虚拟DOM和diff算法来提高渲染性能。 在 Vue 中，虚拟DOM的概念被称为“VNode”，它是一个轻量级的JavaScript对象，用于描述DOM节点。当数据发生变化时，Vue 会创建一个新的VNode树，并通过diff算法来对比新旧VNode树，找到最小变更并进行渲染。这样可以避免对整个DOM树进行重绘，提高性能。 Vue中的key属性用于标识节点的唯一性，当节点需要移动时，key可以帮助Vue更准确地定位节点，避免不必要的操作。如果没有使用key，Vue会尝试通过就地复用和移动算法来尽可能减少DOM操作，但这可能会导致一些意外的行为，例如，数据不一致、输入框内容丢失等。 总之，Vue的虚拟DOM和diff算法与React类似，但具体实现和一些细节可能有所不同。

73.vue2和vue3有哪些区别?

Vue.js是一款流行的前端框架，其版本迭代也较为频繁。Vue.js 3 是 Vue.js 的最新版本，相较于 Vue.js 2，有以下主要的区别： 1、性能提升：Vue.js 3 在内部实现上进行了大量的优化，使得渲染速度更快，内存占用更少。 2、Composition API：Vue.js 3 引入了 Composition API，可以更好地组织和复用逻辑代码，提高代码的可维护性。 3、更好的TypeScript支持：Vue.js 3 对 TypeScript 的支持更加友好，提供了完整的类型定义。 4、更好的Tree Shaking支持：Vue.js 3 支持更好的 Tree Shaking，可以更加精确地按需引入需要的模块。 5、更少的依赖：Vue.js 3 的核心库的依赖更少，可以减小打包体积。 6、更多的特性：Vue.js 3 支持更多的特性，如片段和Teleport等。 总的来说，Vue.js 3 在性能、可维护性和特性上都有所提升。但是需要注意的是，由于 API 发生了较大的变化，因此 Vue.js 3 与 Vue.js 2 之间并不完全兼容，需要进行相应的迁移工作

74.vue项目中style样式中为什么要添加 scoped

在Vue中使用 scoped 属性可以让样式作用域仅限于当前组件中，不影响全局，避免了样式污染和样式冲突的问题。 scoped 会为每个组件的 style 标签添加一个唯一的属性作为标记，这样每个组件的样式规则只作用于当前组件的元素，不影响其他组件的样式。 在组件中使用 scoped 的方式如下：

```
<template>
  <div class="example">
    <p>Example Component</p>
  </div>
</template>

<style scoped>
.example {
  color: red;
}
</style>
```

在这个例子中，.example 样式规则只会作用于当前组件内部的元素，而不会影响全局的样式。

75.mounted生命周期和keep-alive中activated的优先级

在 Vue 中，mounted 生命周期是指一个组件被挂载到 DOM 中后触发的钩子函数。而 keep-alive 是一个用来缓存组件的抽象组件，它自身没有任何展示效果，只是将内部包含的组件缓存起来，从而能够在需要时快

速地切换到缓存的组件。 当一个组件第一次被挂载时，mounted 生命周期会被触发，同时 keep-alive 中的缓存组件还没有被渲染，因此 activated 生命周期并不会被触发。只有当一个被缓存的组件被激活后（比如从其他页面返回到该组件所在的页面），activated 生命周期才会被触发。因此，优先级上 mounted 生命周期高于 activated 生命周期。

76.Vue 3.0 使用的 diff 算法相比 Vue 2.0 中的双端比对有以下优势

1、最长递增子序列算法：Vue 3.0 的 diff 算法采用了最长递增子序列算法，能够减少不必要的 DOM 操作，提升性能。 2、静态标记：Vue 3.0 中，编译器会对静态节点进行标记，在更新时可以直接跳过这些静态节点，减少 DOM 操作，提升性能。 3、缓存数组：Vue 3.0 中，每次更新时会将新旧 VNode 数组缓存起来，只对数组中不同的 VNode 进行比对，减少比对次数，提升性能。 4、动态删除操作：Vue 3.0 中，对于动态删除操作，采用了异步队列的方式进行，能够将多个删除操作合并为一个，减少 DOM 操作，提升性能。 总的来说，Vue 3.0 的 diff 算法相比 Vue 2.0 更加高效，能够减少不必要的 DOM 操作，提升应用的性能。

77.vue 父子组件传值有哪些方式

Vue父子组件之间传递数据的方式有以下几种： Props：通过向子组件传递属性的方式实现数据传递。在父组件中通过v-bind绑定子组件的属性，子组件中通过props接收父组件传递的数据。这是一种单向数据流的方式，父组件可以向子组件传递数据，但是子组件不能直接修改传递过来的数据，需要通过触发事件的方式通知父组件进行修改。 事件：父组件通过*emit*方法触发子组件的自定义事件，子组件中通过on监听事件并接收参数，从而实现数据的传递。这也是一种单向数据流的方式，父组件通过事件向子组件传递数据，子组件可以通过触发事件的方式通知父组件进行修改。 *parent*/children：通过访问父组件或子组件的实例属性来实现数据的传递。但是这种方式不够直观，且容易出现问题，因为父组件或子组件的实例属性可能会在不同的组件结构中发生变化。
refs：通过在父组件中使用*ref*属性来获取子组件的实例，从而可以直接访问子组件的属性和方法。这种方式refs可能会变得混乱。

其中，Props是最常用的一种方式，因为它不仅可以实现数据的传递，还可以进行数据类型检查和默认值设置，使得数据的传递更加稳定和安全。

78.Vue2.x 和 Vue3 响应式上的区别？Vue 数据绑定是怎么实现的

总体回答

Vue 2.x 版本使用的是基于 Object.defineProperty 实现的响应式系统，而 Vue 3.x 版本使用的是基于 ES6 Proxy 实现的响应式系统，两者在实现上有很大的区别。

在 Vue 2.x 中，当一个对象被设置为响应式对象时，会通过 Object.defineProperty() 方法把每个属性都转换成 getter 和 setter，当属性值发生变化时，会触发 setter，进而通知所有引用该属性的组件更新视图。

而在 Vue 3.x 中，通过 ES6 Proxy 对象代理实现了对对象的监听和拦截，可以更加细粒度地控制对象属性的读取和赋值行为，也提供了更好的性能表现。

1、Vue 3.x 中对于新增属性和删除属性的响应式处理更加完善和高效，无需使用 Vue.set()方法，而 Vue 2.x 中需要手动使用这些方法才能保证新增或删除属性的响应式效果。

2、Vue 3.x 中使用了递归遍历 Proxy 对象的属性，因此在访问嵌套属性时会更加方便和高效，而 Vue 2.x 则需要通过 watch 或 computed 等方式才能实现嵌套属性的响应式。

3、Vue 3.x 中的响应式系统支持了 reactive() 和 readonly() 等 API，方便开发者创建只读或可变的响应式对象。而在 Vue 2.x 中则没有这些 API。

总的来说，Vue 3.x 中的响应式系统在使用上更加方便、高效和完善。

Object.defineProperty 实现数据响应式的一些问题

只能劫持对象属性

Object.defineProperty 只能劫持对象属性的 getter 和 setter 方法，无法对对象本身进行劫持，且需要对每个属性单独设置，导致代码冗余和效率低下。

譬如下述代码，实现了对对象单个属性进行 set 和 get 监听：

```
let person = {}
let personName = 'lihua'

//在person对象上添加属性namep,值为personName
Object.defineProperty(person, 'namep', {
  //但是默认是不可枚举的(for in打印打印不出来)，可：enumerable: true
  //默认不可以修改，可：wirtable: true
  //默认不可以删除，可：configurable: true
  get: function () {
    console.log('触发了get方法')
    return personName
  },
  set: function (val) {
    console.log('触发了set方法')
    personName = val
  }
})

//当读取person对象的namp属性时，触发get方法
console.log(person.namep)

//当修改personName时，重新访问person.namep发现修改成功
personName = 'liming'
console.log(person.namep)

// 对person.namep进行修改，触发set方法
person.namep = 'huahua'
console.log(person.namep)
```

所以，在 Vue2 使用 Object.defineProperty 进行数据响应式处理的时候有几个弊端： **1**、只能劫持对象属性 **2、**所以如果需要对整个对象进行劫持代理， 或者需要监听对象上的多个属性，则需要额外需要配合 Object.keys(obj) 进行遍历。 **3、**如果对象的层级不止一层，需要深度监听一个对象，则在上述的遍历操作下，还需要叠加递归处理的思想。因此，整个代码量和复杂度都是非常高的

无法监听数组变化 那么如果对象的属性是一个数组呢？我们要如何实现监听？请看下面一段代码：

```
let arr = [1, 2, 3]
let obj = {}
//把arr作为obj的属性监听
Object.defineProperty(obj, 'arr', {
  get() {
    console.log('get arr')
    return arr
  },
  set(newVal) {
    console.log('set', newVal)
    arr = newVal
  }
})
console.log(obj.arr) //输出get arr [1,2,3]  正常
obj.arr = [1, 2, 3, 4] //输出set [1,2,3,4] 正常
obj.arr.push(3) //输出get arr 不正常，监听不到push
```

由于数组的 push、pop、splice 等方法不会触发长度属性的 setter 方法，不能被 Object.defineProperty 监听到，因此需要使用额外的方法进行监听。 我们发现，通过 push 方法给数组增加的元素，set方法是监听不到的。 事实上，通过索引访问或者修改数组中已经存在的元素，是可以出发 get 和 set 的，但是对于通过 push、unshift 增加的元素，会增加一个索引，这种情况需要手动初始化，新增加的元素才能被监听到。另外， 通过 pop 或 shift 删除元素，会删除并更新索引，也会触发 setter 和 getter 方法。 在 Vue2.x 中，通过重写 Array 原型上的方法解决了这些问题：

```
// 重写数组原型上的方法
const arrayProto = Array.prototype;
const arrayMethods = Object.create(arrayProto);
```

```
['push', 'pop', 'splice', ...].forEach(function (method) {
  // 缓存原始方法
  const original = arrayProto[method];

  // 在重写的方法中添加响应式处理逻辑
  def(arrayMethods, method, function mutator() {
    // 调用原始方法
    const result = original.apply(this, arguments);

    // 触发响应式更新操作
    // ...

    return result;
  });
});

// 设置数组的 __proto__ 属性为重写后的 arrayMethods
const arr = [];
arr.__proto__ = arrayMethods;

// 对 arr 进行赋值操作时，就会触发响应式更新
```

上述代码中，我们通过 Object.create 方法创建了一个具有 Array 原型的对象 arrayMethods，并循环遍历需要重写的数组方法，然后在重写的方法中添加了响应式处理逻辑。最后，将数组的 **proto** 属性指向重写后的 arrayMethods 对象。 通过这种方式，当我们对数组进行 push、pop、splice 等操作时，就能触发响应式系统的相应更新，从而实现了数组的响应式效果。 需要注意的是，这种方式仅解决了数组方法的响应式问题，而不会处理数组元素自身的响应式。 如果需要对数组元素进行响应式处理，还需要使用 Vue 提供的 $set 或 Vue.set 方法进行操作。 举个例子：

```
// 假设有如下数据对象
data: {
  obj: {
    name: 'Alice',
    age: 20
  },
  arr: [1, 2, 3]
}

// 添加或修改对象属性
this.$set(this.obj, 'gender', 'female');
// 或者使用 Vue.set 方法
Vue.set(this.obj, 'gender', 'female');

// 修改数组元素
this.$set(this.arr, 1, 4);
// 或者使用 Vue.set 方法
Vue.set(this.arr, 1, 4);

// 删除对象属性
this.$delete(this.obj, 'age');
// 或者使用 Vue.delete 方法
Vue.delete(this.obj, 'age');

// 删除数组元素
this.$delete(this.arr, 0);
// 或者使用 Vue.delete 方法
Vue.delete(this.arr, 0);
```

在上述代码中，我们可以看到使用 $set 或 Vue.set 方法时需要传入两个参数：目标对象和要操作的属性名（或数组索引）。如果是对对象属性进行添加或修改操作，还需要提供新的属性值。而对于数组，第二个参数是数组索引，第三个参数是要设置的新值。

当我们调用 $set 或 Vue.set 方法时，它会在内部使用 Object.defineProperty 的方式对目标对象进行劫持，并触发相应的更新操作。这样，即使是新添加的属性或数组元素，也能正常触发视图更新，保证了数据的响应式特性。

同样地，对于删除对象属性或数组元素的操作，Vue 也提供了 $delete 或 Vue.delete 方法。它们用于从目标对象中删除指定的属性或数组元素，并触发相应的更新操作。

总之，使用 *set*、delete 或 Vue.set、Vue.delete 方法能够确保对已有对象和数组以及新增的属性或元素进行响应式的操作，从而实现数据的动态变化和视图的更新。

初始化性能开销大

基于上述的两个特点的描述，不难看出，使用 Object.defineProperty 的方式由于需要对每个属性都进行 setter 和 getter 的定义，因此在对象较大时，初始化的性能开销较大，影响用户体验。

Proxy 实现数据响应式 在上面使用 Object.defineProperty 的时候，我们遇到的问题有

1. 一次只能对一个属性进行监听，需要遍历来对所有属性监听
2. 在遇到一个对象的属性还是一个对象的情况下，需要递归监听
3. 对于对象的新增属性，需要手动监听
4. 对于数组通过 push、unshift 方法增加的元素，也无法监听
5. 性能开销大

与 Object.defineProperty 比起来，Proxy 有非常明显的优势：

1. 支持监听整个对象以及数组变化：通过 Proxy，可以对整个对象或数组进行劫持，不需要对每个属性单独设置，同时可以自动处理数组变化。
2. 性能开销更小：在初始化时，Proxy 只需要在对象上设置一个代理即可，避免了 Object.defineProperty 额外的属性设置操作和中间层的缓存数组，因此性能更加高效。
3. 支持隐藏属性：使用 Proxy 对象可以隐藏一些不需要观察的属性，从而保护数据的安全性。

在 Vue 中体现最为明显的一点就是：Proxy 代理对象之后不仅可以拦截对象属性的读取、更新、方法调用之外，对整个对象的新增、删除、枚举等也能直接拦截，而 Object.defineProperty 只能针对对象的已知属性进行读取和更新的操作拦截。 看一个最简单的例子学会原理：

```
const obj = { name: 'MiyueFE', age: 28 };
const proxyObj = new Proxy(obj, {
  get(target, property) {
    console.log(`Getting ${property} value: ${target[property]}`);
    return target[property];
  },
  set(target, property, value) {
    console.log(`Setting ${property} value: ${value}`);
    target[property] = value;
  },
  deleteProperty(target, property) {
    console.log(`Deleting ${property} property`);
    delete target[property];
  },
});

console.log(proxyObj.name); // Getting name value: MiyueFE, 输出 "MiyueFE"
proxyObj.name = 'MY'; // Setting name value: MY
console.log(proxyObj.name); // Getting name value: MY, 输出 "MY"
delete proxyObj.age; // Deleting age property
console.log(proxyObj.age); // undefined
```

**Vue 怎么用 vm.*set*()解决对象新增属性不能响应的问题？ **vm.set 是一个非常实用的API，因为 Vue2 实现响应式的核心是利用了 ES5 的Object.defineProperty，当我们通过直接修改数组下标更改数组或者给对象添加新的属性，这个时候 Object.defineproperty 是监听不到数据的变化的，这时候就可以使用 *set*，让修改的操作也实现进行响应式响应。通过*vm*.set() 方法可以解决对象新增属性不能响应的问题。vm.*set*()方法是*Vue*实例的一个方法，用于向响应式对象添加新属性，并确保这个新属性是响应式的。使用*vm*.set()的语法如下：

```
vm.$set(object, key, value)
```

其中，object是要添加属性的对象，key是要添加的属性名，value是要添加的属性值。 以下是一个示例：

```
// 定义一个响应式对象
data: {
  user: {
    name: 'John',
    age: 25
  }
},
```

```
// 向user对象添加新属性
methods: {
  addUserProperty() {
    this.$set(this.user, 'gender', 'male');
  }
}
```

在上述示例中，this.*set*(*this.user*,'*gender*','*male*')将向*user*对象添加一个名为*gender*的新属性，并将其值设置为'*male*'。由于使用 set() 方法，这个新增的属性将会是响应式的，当该属性发生改变时，相关的组件也会进行更新。 需要注意的是，只有在 Vue 实例创建之后，才能使用 vm.*set*()方法。当我们调用set 或 Vue.set 方法时，它会在内部使用 Object.defineProperty 的方式对目标对象进行劫持，并触发相应的更新操作。这样，即使是新添加的属性或数组元素，也能正常触发视图更新，保证了数据的响应式特性。 同样地，对于删除对象属性或数组元素的操作，Vue 也提供了*delete*或*Vue.delete*方法。它们用于从目标对象中删除指定的属性或数组元素，并触发相应的更新操作。在 set() 来解决对象新增属性不能响应的问题。 Proxy 能够自动追踪对象属性的访问和修改，并实现响应式更新。当我们直接给一个对象添加新属性时，新属性会被 Proxy 自动转换为响应式属性，无需额外调用 $set() 方法。 例如，在 Vue3 中可以这样操作：

```
const obj = reactive({ name: 'John', age: 25 });
obj.gender = 'male'; // 新增属性
```

上述代码中，通过 reactive 函数将 obj 对象转换为响应式对象，当我们直接给 obj 添加 gender 属性时，Proxy 会自动将其转换为响应式属性，并触发更新。 尽管如此，仍然有一些特殊情况下可能需要使用*set*()。例如，当我们需要在*reactive*或*ref*创建的响应式对象中，给嵌套对象添加新属性时，仍然需要使用 set() 进行手动响应式包裹。 总之，在绝大多数情况下，Vue3 中不再强制需要使用 $set() 来处理对象新增属性的响应问题。

79.详解函数式组件

是什么？ 如果一个组件无状态 (没有响应式数据)，也没有实例 (没有 this 上下文)，我们可以将这种组件标记为 functional，这意味它是一个函数式组件，这个组件只接受一些 prop。 并且函数式组件不能调用 Vue 实例方法。函数式组件在性能方面有一定的优势，因为它们渲染时不需要创建实例，也不需要进行组件的生命周期钩子函数操作。 举例子： 下面是一个简单的函数式组件的示例代码，在 Vue 2.x 版本中，可以使用 functional: true 来声明一个函数式组件：

```
Vue.component('my-functional-component', {
  functional: true,
  render: function (createElement, context) {
    var props = context.props;
    return createElement('div', props.message);
  },
  props: {
    message: String
  }
});
```

上面的代码创建了一个名为 my-functional-component 的函数式组件，它接收一个 message 属性，并将其显示在一个 元素中。 render 函数接收两个参数：第一个参数是 createElement 函数，用来创建 VNode；第二个参数是上下文对象，其中包含了组件的一些属性，比如 props，listeners 等等。 最后，要注意在函数式组件中，由于没有实例，所以不能使用 this 关键字访问组件的属性或方法，而是需要从上下文对象中获取相应的属性和方法。

函数式组件解决了什么问题?

在 Vue 2 中，函数式组件有两个主要用例：

- 作为性能优化，因为它们的初始化速度比有状态组件快得多
- 返回多个根节点

然而，在 Vue 3 中，有状态组件的性能已经提高到可以忽略不计的程度。此外，有状态组件现在还包括返回多个根节点的能力。 因此，函数式组件剩下的唯一用例就是简单组件，比如创建动态标题的组件。否则，建议你像平常一样使用有状态组件。 综上，在 Vue2，我们更多的使用函数式组件的目的在于性能优化。而如果升级到了 Vue3，函数式组件剩下的唯一用例就是简单组件，比如创建动态标题的组件。

函数式组件在 Vue2.x 的实用场景

在 Vue 2 中，函数式组件的主要优势在于其性能优化和无状态的特性。下面是一些函数式组件在 Vue 2 中可能实用的场景： 1、简单的展示组件：函数式组件适用于那些没有复杂交互逻辑、只负责渲染静态内容的组件。比如按钮、图标、标签等简单的 UI 组件。

```
Vue.component('my-button', {
  functional: true,
  render: function (createElement, context) {
    var props = context.props;
    return createElement('button', props.message);
  },
  props: {
    message: String
  }
});
```

1.高性能列表渲染：函数式组件通常比有状态组件具有更好的性能，特别是在需要渲染大量重复的子组件时。使用函数式组件作为列表项组件，可以提高列表的整体渲染性能。

```
Vue.component('list-item', {
  functional: true,
  render: function (createElement, context) {
    var props = context.props;
    return createElement('li', props.item.text);
  },
  props: {
    item: Object
  }
});
```

1.插槽（Slot）组件：函数式组件也适合用作插槽组件，尤其是那些只负责将内容插入到指定位置而不进行其他逻辑处理的组件。

```
javascriptCopy CodeVue.component('my-slot-component', {
  functional: true,
  render: function (createElement, context) {
    return createElement('div', context.children);
  }
});
```

1.高阶组件（Higher-order Components）：函数式组件可以作为高阶组件的一种实现方式，用于将某些共享逻辑应用于多个组件。

```
function withPerformance(Component) {
  return Vue.extend({
    functional: true,
    render: function (createElement, context) {
      var start = performance.now();
      var componentVNode = createElement(Component, context.data, context.children);
      var end = performance.now();
      console.log('Performance:', Component.name, 'took', (end - start).toFixed(4), 'ms to render');
      return componentVNode;
    }
  });
}

Vue.component('my-component', withPerformance({
  name: 'MyComponent',
  props: {
    message: String
  },
  render: function (createElement, context) {
    return createElement('div', context.props.message);
```

```
  }
}));
```

上面的示例代码创建了一个名为 withPerformance 的高阶组件，它接收一个普通组件作为参数，然后返回一个新的组件。新组件是一个函数式组件，它记录了被包装组件的渲染耗时，并输出到控制台。 可以通过调用 withPerformance 函数并传入一个组件来生成一个新的组件，如上面的代码所示。新组件会自动记录被包装组件的渲染耗时，以便进行性能监控和优化。 需要注意的是，由于函数式组件没有实例，无法使用 Vue 实例的生命周期钩子函数和实例方法。因此，在高阶组件中需要使用 Vue.extend 方法来创建一个新的包含生命周期钩子函数和实例方法的“虚拟”组件。

函数式组件在 Vue2.x 与 Vue3 的区别

写法上的差异： 在 Vue 3 中，所有的函数式组件都是用普通函数创建的，换句话说，不需要定义 { functional: true } 组件选项。 他们将接收两个参数：props 和 context。context 参数是一个对象，包含组件的 attrs，slots，和 emit property。 此外，现在不是在 render 函数中隐式提供 h，而是全局导入 h。

```
import { h } from 'vue';

const DynamicHeading = (props, context) => {
  return h(`h${props.level}`, context.attrs, context.slots);
}

DynamicHeading.props = ['level'];

export default DynamicHeading;
```

另外，函数式组件在 Vue3.x 相对 2.x 的变化：

- 在 3.x 中，函数式组件的性能提升可以忽略不计，因此我们建议只使用有状态的组件
- 函数式组件只能使用接收 props 和 context 的普通函数创建 (即：slots，attrs，emit)。
- 非兼容变更：functional attribute 在单文件组件 (SFC) `<template>` 已被移除
- 非兼容变更：{ functional: true } 选项在通过函数创建组件已被移除

在 Vue 2 中，函数式组件有两个主要用例：

- 作为性能优化，因为它们的初始化速度比有状态组件快得多
- 返回多个根节点

然而，在 Vue 3 中，有状态组件的性能已经提高到可以忽略不计的程度。此外，有状态组件现在还包括返回多个根节点的能力。 因此，函数式组件剩下的唯一用例就是简单组件，比如创建动态标题的组件。否则，建议你像平常一样使用有状态组件。

80.详解 Vue 单向数据流

是什么：在 Vue 中，单向数据流是指数据在组件之间的传递是单向的，即从父组件传递给子组件。 解决了什么问题：这种单向数据流的模式有助于提高代码的可维护性和可预测性。 在 Vue 中，父组件可以通过 props 将数据传递给子组件。子组件接收到父组件传递的数据后，可以在自己的作用域内使用这些数据。子组件不能直接修改父组件传递的数据，只能通过触发事件（event）的方式通知父组件进行修改。 这种单向数据流的设计思想有以下优点：

1. 数据流清晰：单向数据流让数据的流向变得明确，可以清楚地追踪数据的来源和变化。只要关注父组件传递的数据和子组件发出的事件，就能了解数据的整个流动过程，方便排查问题和调试。
2. 数据可维护性：数据的单向流动使得组件的状态更易于理解和维护。组件的状态只受到父组件传递的 props 影响，不会受到其他组件的直接修改。这样可以更好地控制和管理状态的变化，减少意外错误和副作用。
3. 可重用性和解耦：通过单向数据流，可以将组件解耦，使得组件更加独立和可重用。父组件通过 props 传递数据给子组件时，并不需要关心子组件内部是如何处理这些数据的，只需要关注数据的传递即可。这样可以提高组件的复用性和可拓展性。
4. 可预测性：由于数据的流动是单向的，组件的状态变化是可预测的。只要了解了数据的来源和影响，就可以精确预测组件的行为和渲染结果。这对于调试和维护非常重要，可以快速定位问题和优化代码。

需要注意的是，虽然 Vue 的单向数据流模式是默认规则，但在开发中也可以通过特定的方式实现双向绑定，例如使用 v-model 指令。 但在大多数情况下，遵循单向数据流的原则能够使代码更易于理解、调试和维护。

保持单向数据流的重要性

在父组件中，一个变量往往关联着多个变量或动作

- 如果在子组件中修改了传进来的prop对象，并且没有在父组件中watch该prop对象，不会主动触发其他变量的修改或动作的发生，导致数据局部展示错误。
- 如果主动watch该prop对象，由于无法准确预知修改的来源和方式，从而大大增加了父组件的维护难度。

Vue 是如何实现单向数据流的?

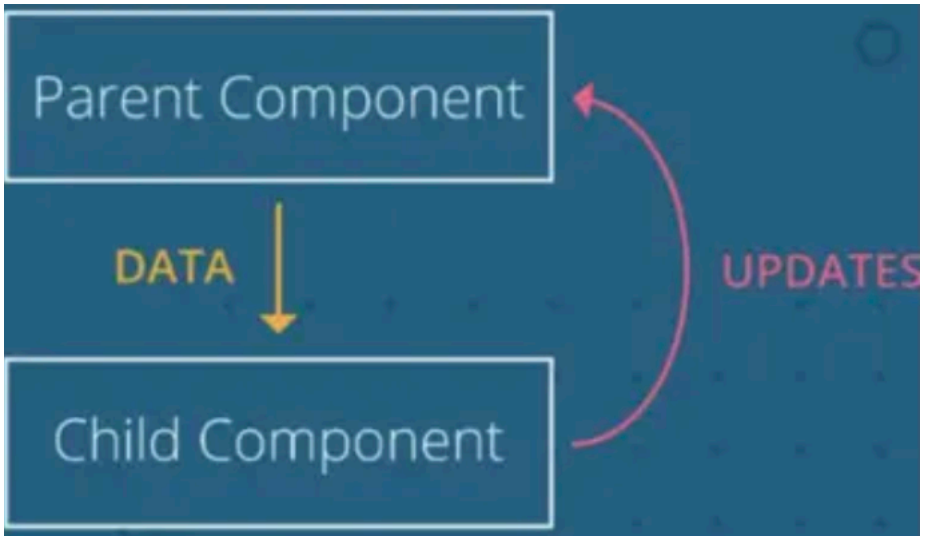
Vue 实现单向数据流的方式主要依靠 Vue 实例和组件之间的数据绑定机制。 1、父组件向子组件传递数据 在 Vue 中，父组件可以通过 props 向子组件传递数据。父组件中通过 v-bind 绑定子组件的 props 属性，将数据传递给子组件。子组件在接收到 props 之后，就可以在自己的作用域中使用这些数据。 2、但是，子组件不能直接修改 props 中的值，子组件可以通过 $emit 方法向父组件发送事件，同时携带数据。父组件通过在子组件上注册事件监听器，并通过回调函数接收子组件发送的数据。 这种单向数据流模式的实现可以保证父组件和子组件之间的数据传递和状态变化是可控的。每个组件都只关心自己的状态和数据，不需要过多地关注其他组件的实现细节。

Vue 的单向数据流和双向数据绑定是否冲突?

不冲突。

所谓的单向数据流，指的是组件之间的数据流动。

虽然 Vue 有双向绑定 v-model，但是 Vue 和 React 类似，父子组件之间数据传递，仍然还是遵循单向数据流的，父组件可以向子组件传递 Props，但是子组件不能修改父组件传递来的 Props，子组件只能通过事件通知父组件进行数据更改，如图所示：



优点是所有状态的改变可记录、可跟踪，源头易追溯，所有数据只有一份，组件数据只有唯一的入口和出口，使得程序更直观更容易理解，有利于应用的可维护性。

v-model 虽然是双向数据绑定，但是 v-model 本质上也是单向数据流，只是多了一层用户输入触发事件，更新数据的封装。

81.详解 Vue template 模板编译

是什么？解决了什么问题?

Vue 的模板编译是将 Vue 的模板代码转化为可执行的 JavaScript 代码的过程。 Vue 模板（template）在运行前会被编译成渲染函数，避免了每次渲染时重新解析模板的开销。渲染函数具有更高效的执行速度，并且可以在组件初始化时预先生成，从而减少了每次更新时的性能消耗。 它的优点如下：

- 提高性能：模板编译将模板转换成更高效的代码，避免了运行时解析模板的开销，提升了应用的性能。
- 简化开发：通过模板编译，开发者可以使用类似 HTML 的标记语法编写组件模板，而无需直接操作 JavaScript 对象和函数，降低了开发的难度。
- 实现响应式更新：模板编译会将模板中的指令和数据转化成一组渲染函数，这些渲染函数可以与 Vue 的响应式系统协同工作，实现数据的自动更新和视图的重渲染。
- 基于模板的性能优化
    - 静态节点提升：Vue 在模板编译阶段会检测出那些静态节点（不依赖响应式数据的节点），并将其优化为常量，避免了在每次重新渲染时对这些节点进行重复的创建和比对操作。
    - 列表渲染优化：Vue 提供了 v-for 指令用于列表渲染，而且在编译时会自动为每个列表项生成唯一的 key 值。这样在更新列表时，Vue 可以精确地检测到每个列表项的变化，减少了不必要的 DOM 操作，提高了性能。
    - 条件渲染优化：Vue 的模板支持使用 v-if 和 v-show 指令进行条件渲染。在编译时，Vue 会根据指令的条件进行静态分析，如果条件是确定的（即不依赖响应式数据），则会进行静态提升，优化渲染性能。
    - 缓存事件处理函数：Vue 的模板编译会自动为事件处理函数生成缓存版本，在渲染过程中复用同一个处理函数，避免了重复创建匿名函数的开销。
    - 内置指令：Vue 内置了一些常用的指令，如 v-model、v-bind、v-on 等。这些指令在模板编译时会被转换为相应的渲染函数，能够更高效地更新视图，并且可以方便地处理用户输入、属性绑定和事件监听等操作。

Runtime + Compiler 版本以及 Runtime only 版本

Vue.js 提供了 2 个版本，一个是 Runtime + Compiler 版本，一个是 Runtime only 版本。

- Runtime + Compiler 版本：包含完整的 Vue 运行时（runtime）和模板编译器（compiler）。在浏览器中运行时，会在运行前将模板编译成渲染函数并执行。这个版本的 Vue 可以直接接收模板字符串作为组件的 template，并在浏览器中编译执行。这个版本的 Vue 体积较大，但可以在开发过程中实时编译模板，通常用于在浏览器环境中开发单文件组件。
- Runtime only 版本：只包含了 Vue 的运行时（runtime），没有模板编译器。这个版本的 Vue 不能接收模板字符串作为组件的 template，需要使用预编译的渲染函数或者通过单文件组件配合构建工具来使用。这个版本的 Vue 体积较小，适用于生产环境，因为模板的编译和优化一般在构建过程中完成。

当需要在客户端编译模板 (比如传入一个字符串给 template 选项，或挂载到一个元素上并以其 DOM 内部的 HTML 作为模板)，就将需要加上编译器，即完整版：

```
// 需要编译器
new Vue({
  template: '<div>{{ hi }}</div>'
})

// 不需要编译器
new Vue({
  render (h) {
    return h('div', this.hi)
  }
})
```

当使用 vue-loader 或 vueify 的时候，*.vue 文件内部的模板会在构建时预编译成 JavaScript。在最终打好的包里实际上是不需要编译器的，所以只用运行时版本即可。 因为运行时版本(Runtime only 版本)相比完整版体积要小大约 30%，所以应该尽可能使用这个版本。

模板变异过程解析

在 Vue 的整个编译过程中，会做三件事：

- 解析模板 parse，生成 AST
- 优化 AST optimize
- 生成代码 generate

在 Vue 的编译过程中，其核心步骤可以分为以下三个阶段： 1、解析模板 parse，生成 AST：在这一阶段，Vue 会将模板解析为抽象语法树（AST），以便后续进行优化和代码生成。Vue 的解析器采用了 HTML Parser 和 Text Parser 两个解析器，分别用于解析 HTML 标签和模板文本。解析器会将模板解析为一组元素描述对象和指令描述对象，然后使用这些对象构建出整个模板的 AST。 2、优化 AST optimize：在生成 AST 后，Vue 会对其进行优化处理，以便更高效地生成代码。这一阶段包括以下三个优化步骤：

- 静态节点标记。Vue 会对那些静态节点进行标记，避免其在之后的更新中重新渲染；
- 静态节点提升。Vue 会将那些只包含静态内容的节点，在编译时提升为常量，从而减少渲染开销。
- 插槽优化。Vue 会对所有的插槽节点进行标记和优化，从而更高效地处理 slot 元素的渲染。

3、生成代码 generate：在 AST 优化后，Vue 会进一步将其转换为可执行的代码，以便生成组件的渲染函数。具体来说，Vue 会根据每个节点生成相应的渲染函数，并将这些函数组合成一个完整的组件渲染函数。同时，Vue 还会为每个组件生成相应的静态 Render 函数，以便在第一次渲染时可以直接使用，提高性能。综上所述，Vue 的编译过程包括解析模板、优化 AST 和生成代码三个核心步骤，每个阶段都有其特定的处理方式，以便最终生成高效的组件渲染函数。

一个典型的模板编译例子 假设我们有如下 Vue template 代码：

```
<template>
  <div>
    <h1>{{ message }}</h1>
    <button @click="handleClick">Click me</button>
  </div>
</template>
```

经过 Vue 的编译优化后，生成的代码可能如下所示：

```
function render(_ctx, _cache) {
  return (_openBlock(), _createBlock("div", null, [
    _createVNode("h1", null, _toDisplayString(_ctx.message), 1 /* TEXT */),
    _createVNode("button", { onClick: _cache[1] || (_cache[1] = $event => _ctx.handleClick($event)) },
"Click me")
  ]))
}
```

可以看到，在编译后生成的渲染函数中： 1、使用 _createBlock 创建根节点，并通过 _createVNode 来创建子节点。 2、模板中的动态内容 {{ message }} 被转换为 _toDisplayString(_ctx.message) 来实现动态渲染 3、事件处理函数(@click="handleClick") 被转换为 { onClick: _cache[1] || (_cache[1] = $event => _ctx.handleClick($event)) } 当然上述代码只是一个简化的示例，实际的编译结果会更加复杂，包含更多的优化和处理逻辑。

了解 Vue template 与 React JSX 的差异及优劣势对比 虽然 Vue 也能使用 JSX，但是大部分时候更多的是使用 template。 而在 React 框架中，对于结构的表现更多是使用 JSX。 首先，它们之间存在一些差异：

1、语法风格：Vue 使用基于 HTML 的模板语法，使用双大括号 {{ }} 来进行插值，使用指令如 v-if、v-for 等来处理逻辑。而 React 使用 JavaScript 的语法扩展 JSX，在 JavaScript 代码中直接编写组件的结构和逻辑。 2、组件定义方式：在 Vue 中，可以通过定义一个带有 template 字段的 Vue 组件对象来编写组件。这个 template 字段就是 Vue 模板。而在 React 中，通过定义一个继承自 React.Component 的类组件，或使用函数式组件来编写组件，通过 render 方法来返回要渲染的 JSX 结构。 3、动态属性和样式绑定：Vue 的模板语法支持直接在元素上使用动态属性绑定，例如 :class、:style 等。而在 React 中，使用 JSX 属性来进行动态属性和样式的绑定，需要使用大括号 {} 进行包裹，例如 className、style。 4、条件渲染和循环：Vue 中使用指令 v-if 和 v-for 来进行条件渲染和循环。而在 React 中，使用 JavaScript 表达式来进行条件渲染，例如使用三元表达式 {condition ? trueBlock : falseBlock} 和 map() 函数来进行循环渲染。 5、事件处理：Vue 的模板语法通过指令 @ 或 v-on 来绑定事件处理函数。而在 React 中，通过在 JSX 的属性中直接使用事件处理函数来进行绑定，例如 onClick、onChange 等。 总的来说，Vue 的模板语法更加类似于传统的 HTML + 模板指令的写法，更容易理解和上手；而 React 的 JSX 则将组件的结构和逻辑紧密地融合在一起，更加灵活和可控。

优劣势分析

- Vue Template 的优势：

    - 语法友好：Vue 的模板采用类似 HTML 的标记语言，容易理解和上手。
    - 可视化编辑工具支持：由于模板具有标记语言的特点，可以在一些编辑器中获得较好的支持。
    - 兼容性强：Vue 的模板相对容易迁移，可适应旧项目，且模板和逻辑分离，避免了代码混杂问题。

- Vue Template 的劣势：
    - 扩展性差：Vue 的模板只能使用内置的 JavaScript 功能，扩展性相对较差，无法灵活地处理复杂逻辑。
    - 难以调试：模板语法虽然类似 HTML，但实际上是 JavaScript 表达式，需要特定工具来进行调试。
    - 可能产生重复代码：模板中相似的操作可能需要在多个组件中重复编写，导致代码冗余和维护困难。

- React JSX 的优势：
    - 强大的扩展性：JSX 可以使用 JavaScript 的完整功能，允许在模板中编写复杂逻辑和自定义组件。
    - 更好的一致性：JSX 与 JavaScript 具有更高的语法一致性，对于熟悉 JavaScript 开发者来说更加自然。
    - 更灵活的调试：由于 JSX 是 JavaScript 代码，可以直接在浏览器中进行调试。

- React JSX 的劣势：
    - 上手难度较高：相较于 Vue 的模板，JSX 的语法相对复杂，对于新手来说上手难度较大。
    - 学习成本高：需要了解和熟悉 JSX 语法和 React 的生命周期等概念。
    - 编辑器支持较差：JSX 的语法对于一些编辑器可能支持不够好，可能需要额外的插件或配置。

82.详解虚拟 DOM 与 Vue DIFF 算法原理

1、什么是虚拟 DOM

- Vue 的虚拟 DOM（Virtual DOM）是 Vue 框架中的一种技术，用于优化 DOM 操作的效率和性能。虚拟 DOM 是一个 JavaScript 对象，它是对实际 DOM 结构的一种抽象表示。
- 每个虚拟DOM节点（VNode）代表一个真实DOM节点或一段文本。虚拟DOM对象包含与之相关的信息，如标签名、属性、子节点等。

2、什么是 DIFF 算法

- Vue.js 的 Virtual DOM 实现了一种高效的 diff 算法，能够快速比较两个虚拟 DOM 树的差异。

3、它们解决了什么问题?

- 当Vue组件的状态发生改变时，Vue会通过 DOM DIFF 算法比较新旧虚拟DOM树找出需要更新的部分，并将这些部分批量地更新到实际 DOM 中。这种批量更新可以减少对实际 DOM 的直接操作，从而提高性能。

4、为什么要用虚拟 DOM?

- 避免频繁操作 DOM，更好的实现跨平台、组件状态追踪等特性

5、DOM DIFF 算法中运用了哪些优化思路?

- 给列表项添加 Key 值，带有唯一 key 值的节点可以减少虚拟DOM的比对操作
- 将树的完全 DIFF 改为同层 DIFF，时间复杂度从 O(n3) 下降到 O(n)
- 同级比较中为了减少元素移动次数，采用了双指针配合最长连续子串算法
- Vue3 的静态标记算法，在编译阶段，通过编译器静态地分析模板，并识别其中的静态节点，在 DIFF 中可以跳过静态节点

拓展回答 DOM diff 作为工程问题，需要具有一定算法思维，因此经常出现在面试场景中，毕竟这是难得出现在工程领域的算法问题。 需要注意的是，为了提高 diff 算法的性能，现代前端框架往往采用一些优化技巧，比如只比较同层级的节点、使用 key 值进行优化等，从而进一步提高渲染性能。

为什么要使用虚拟 DOM？虚拟 DOM 的性能一定比原生 DOM 好吗?

- 当然是前端优化方面，避免频繁操作 DOM，频繁操作 DOM 会可能让浏览器回流和重绘，性能也会非常低，还有就是手动操作 DOM 还是比较麻烦的，要考虑浏览器兼容性问题，当前 jQuery 等库简化了 DOM 操作，但是项目复杂了，DOM 操作还是会变得复杂，数据操作也变得复杂
- 并不是所有情况使用虚拟 DOM 都提高性能，是针对在复杂的的项目使用。如果简单的操作，使用虚拟 DOM 要创建虚拟 DOM 对象等等一系列操作，还不如普通 的DOM 操作
- 跨平台：虚拟 DOM 可以实现跨平台渲染，服务器渲染、小程序、原生应用都使用了虚拟 DOM
- 降低真实 DOM 操作：使用虚拟 DOM，可以实现改变了当前的状态不需要立即的去更新 DOM。并且实现通过前后两次差异进行比较，只更新需要更新的内容，对于没有改变的内容不做任何操作
- 状态追踪：虚拟 DOM 可以比较好的维护组件的状态，更好的跟踪的状态

Vue 2 和 Vue 3 的 DOM DIFF 完整过程

Vue 2 使用了经典的 Diff 算法，也称为双指针算法。其原理可以概括为以下几个步骤： 1、生成新旧虚拟DOM树：在重新渲染组件时，Vue会生成新的虚拟DOM树，并将其与旧的虚拟DOM树进行比较。 2、深度优先遍历：Vue 2 使用深度优先遍历算法来遍历新旧虚拟DOM树的节点。同时，Vue 2 会为每个节点添加唯一的标识符(VNode Key)以提高性能。 3、Diff过程：在遍历的过程中，Vue 2 会对比新旧虚拟DOM节点的类型和属性，判断是否需要更新实际DOM。具体的对比逻辑如下：

- 如果新旧虚拟DOM节点相同（类型相同且Key相同），则只需要更新节点的属性；
- 如果新旧虚拟DOM节点不同，则直接替换整个节点及其子节点；
- 如果节点存在子节点，则递归地对子节点进行Diff；
- 如果节点在旧虚拟DOM树中存在但在新虚拟DOM树中不存在，则直接删除该节点及其子节点。

4、更新实际DOM：根据Diff的结果，Vue 2 会将需要更新的部分进行批量更新，使实际DOM与新的虚拟DOM树保持一致。

Vue 3 采用了基于观察者的 Diff 算法，也称为静态分析算法。其主要原理如下： 1、生成新旧虚拟DOM树：与Vue 2 相同，Vue 3 在重新渲染组件时会生成新的虚拟DOM树，并将其与旧的虚拟DOM树进行比较。 2、标记静态节点：Vue 3 通过静态标记（Static Marking）技术，将那些不会发生变化的节点标记为静态节点，以减少对它们的Diff计算。 3、Patch 过程：在 Diff 过程中，Vue 3 采用了 Patch 策略来处理不同类型的节点：

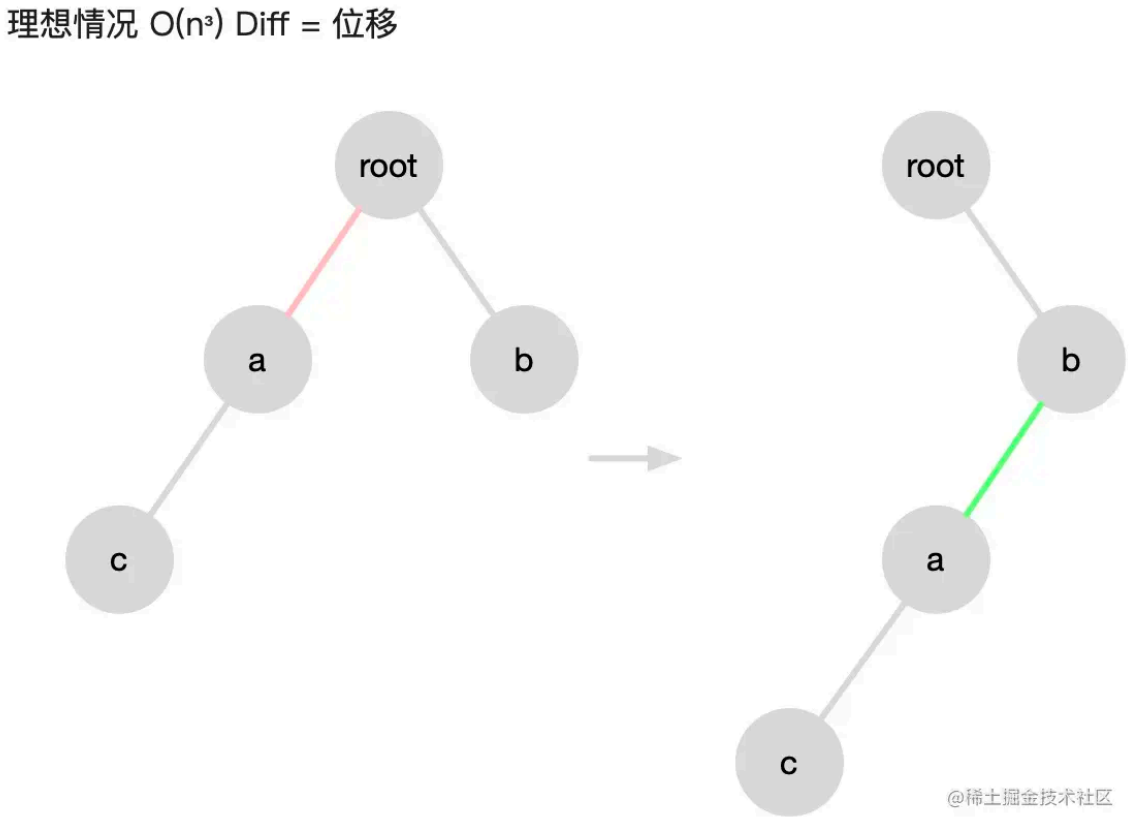
- 对静态节点，Vue 3 将跳过其子节点的Diff过程，省略一些无谓的计算；
- 对动态节点，即有可能发生变化的节点，Vue 3 采用了优化的Heuristic算法，通过比较新旧虚拟DOM节点的选择器（Selector）信息，判断是否需要进行详细的Diff比较。

4、更新实际DOM：根据Patch的结果，Vue 3 将需要更新的部分进行批量更新，使实际DOM与新的虚拟DOM树保持一致。

下面，我们继续了解 Vue 是如何将两棵树的完全 DIFF 的时间复杂度从 O(n3) 降到 O(n) 的。

常而言两个树的完全对比 DIFF 比较算法：O(n3)

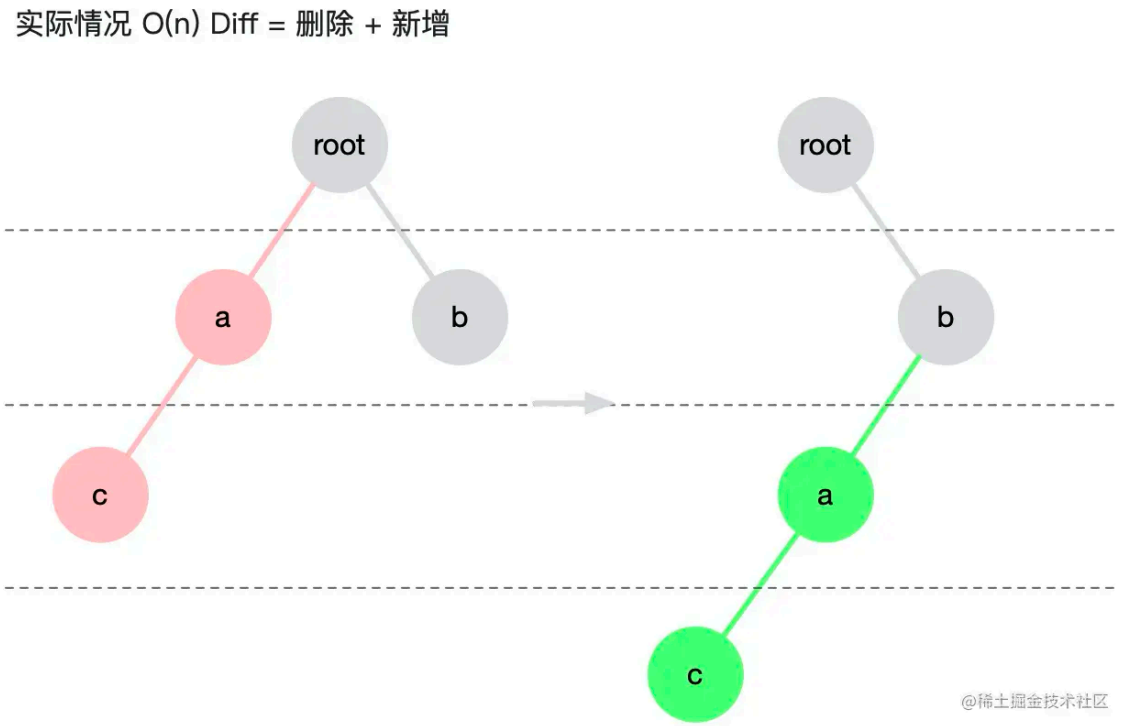
首先，我们需要理解，正常而言，想比较两棵树有什么不一样，其时间复杂度是 O(n3)：



如图所示，理想的 Dom diff 自然是滴水不漏的复用所有能复用的，实在遇到新增或删除时，才执行插入或删除。 可惜程序无法猜到你的想法，想要精确复用就必须付出高昂的代价：时间复杂度 O(n³) 的 diff 算法，这显然是无法接受的，因此理想的 Dom diff 算法无法被使用。 关于 O(n³) 的由来。由于左树中任意节点都可能出现在右树，所以必须在对左树深度遍历的同时，对右树进行深度遍历，找到每个节点的对应关系，这里的时间复杂度是 O(n²)，之后需要对树的各节点进行增删移的操作，这个过程简单可以理解为加了一层遍历循环，因此再乘一个 n。

这里的核心点在于，用传统算法对两棵树进行 DOM DIFF，时间复杂度太高，势必会导致更新效率低，因此，不管是 Vue 还是 React，都在寻求时间复杂度更高的 DOM DIFF 算法。

Vue 实现 O(n) 时间复杂度同层 DOM DIFF 算法



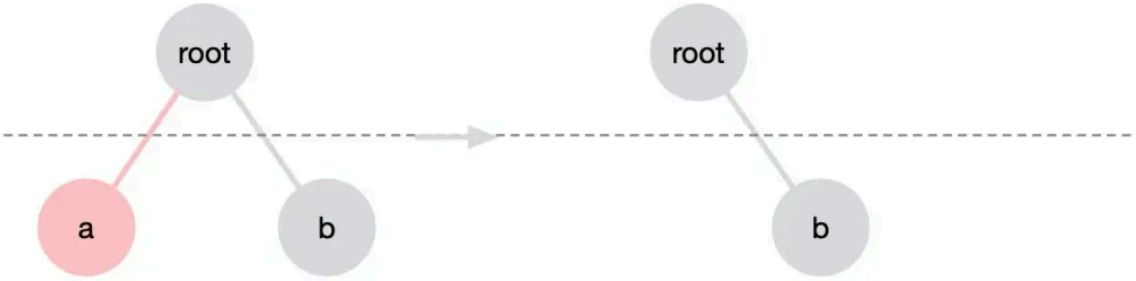
如图所示，只按层比较，就可以将时间复杂度降低为 O(n)。按层比较也不是广度遍历，其实就是判断某个节点的子元素间 diff，跨父节点的兄弟节点也不必比较。

这样做确实非常高效，但代价就是，判断的有点傻，比如 ac 明明是一个移动操作，却被误识别为删除 + 新增。

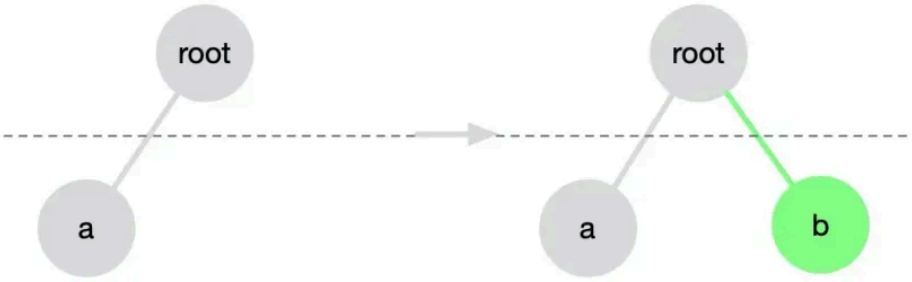
只同层比较的可行性原因：跨 DOM 复用在实际业务场景中是很少出现的，，这时候我们就不要太追求学术思维上的严谨了，毕竟框架是给实际项目用的，实际项目中很少出现的场景，算法是可以不考虑的。

面是同层 diff 可能出现的三种情况，非常简单，看图即可：

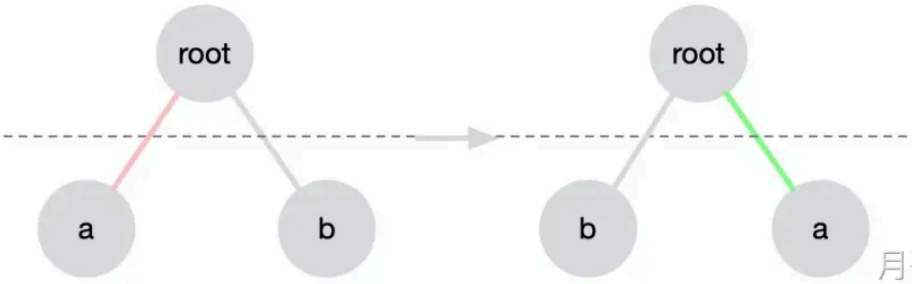
同层 Diff – 删



同层 Diff – 增

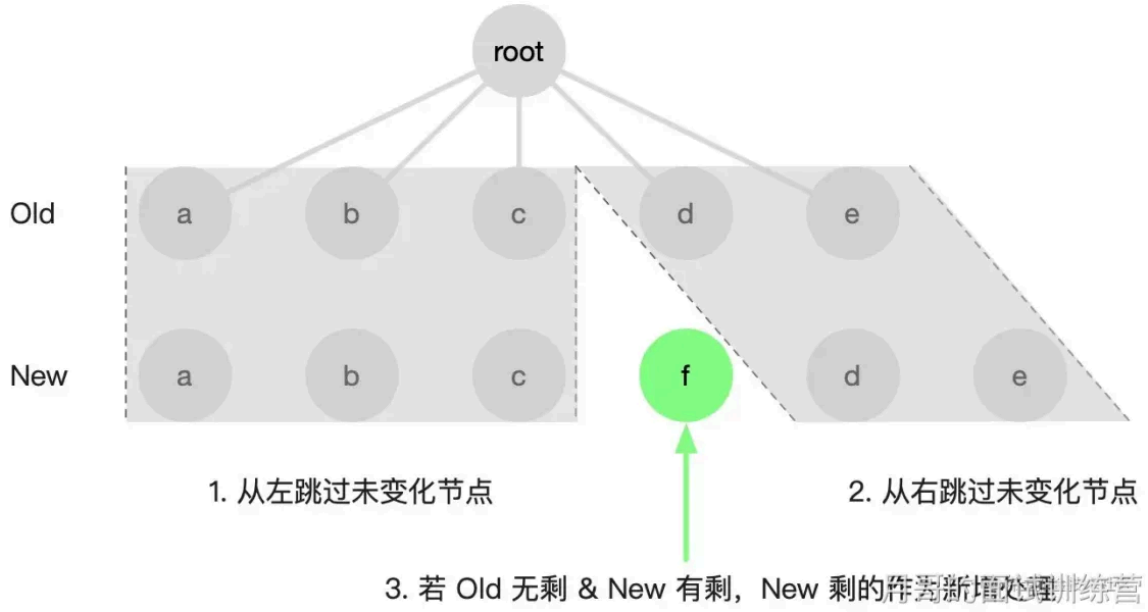


同层 Diff – 移



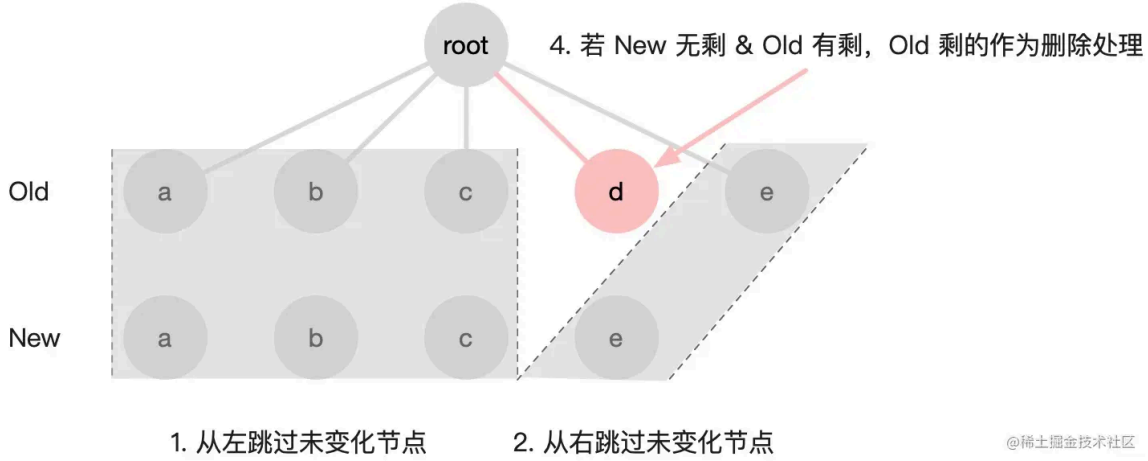
那么同层比较是怎么达到 O(n) 时间复杂度的呢？我们来看具体框架的思路。

Vue 的 Dom diff 一共 5 步，我们结合下图先看前三步：



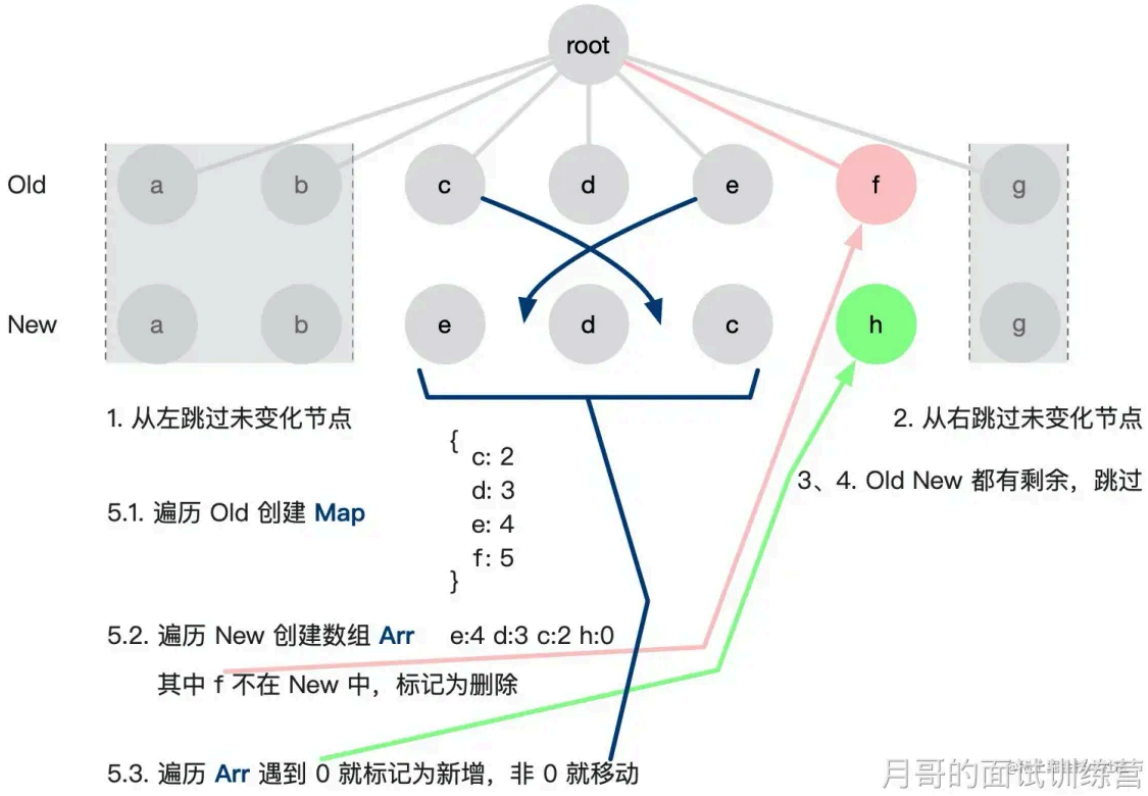
如图所示，第一和第二步分别从首尾两头向中间逼近，尽可能跳过首位相同的元素，因为我们的目的是 尽量保证不要发生 dom 位移。

这种算法一般采用双指针。如果前两步做完后，发现旧树指针重合了，新树还未重合，说明什么？说明新树剩下来的都是要新增的节点，批量插入即可。很简单吧？那如果反过来呢？如下图所示：



第一和第二步完成后，发现新树指针重合了，但旧树还未重合，说明什么？说明旧树剩下来的在新树都不存在了，批量删除即可。

当然，如果 1、2、3、4 步走完之后，指针还未处理完，那么就进入一个小小算法时间了，我们需要在 O(n) 时间复杂度内把剩下节点处理完。熟悉算法的同学应该很快能反映出，一个数组做一些检测操作，还得把时间复杂度控制在 O(n)，得用一个 Map 空间换一下时间，实际上也是如此，我们看下图具体做法：



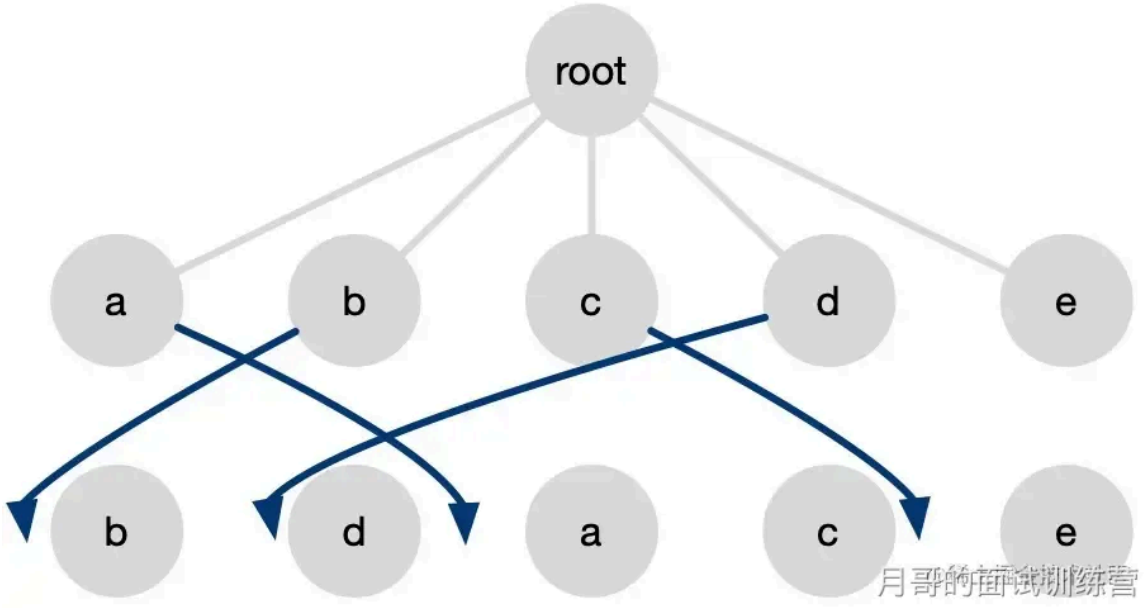
如图所示，1、2、3、4 步走完后，Old 和 New 都有剩余，因此走到第五步，第五步分为三小步：

1、遍历 Old 创建一个 Map，这个就是那个换时间的空间消耗，它记录了每个旧节点的 index 下标，一会好在 New 里查出来。

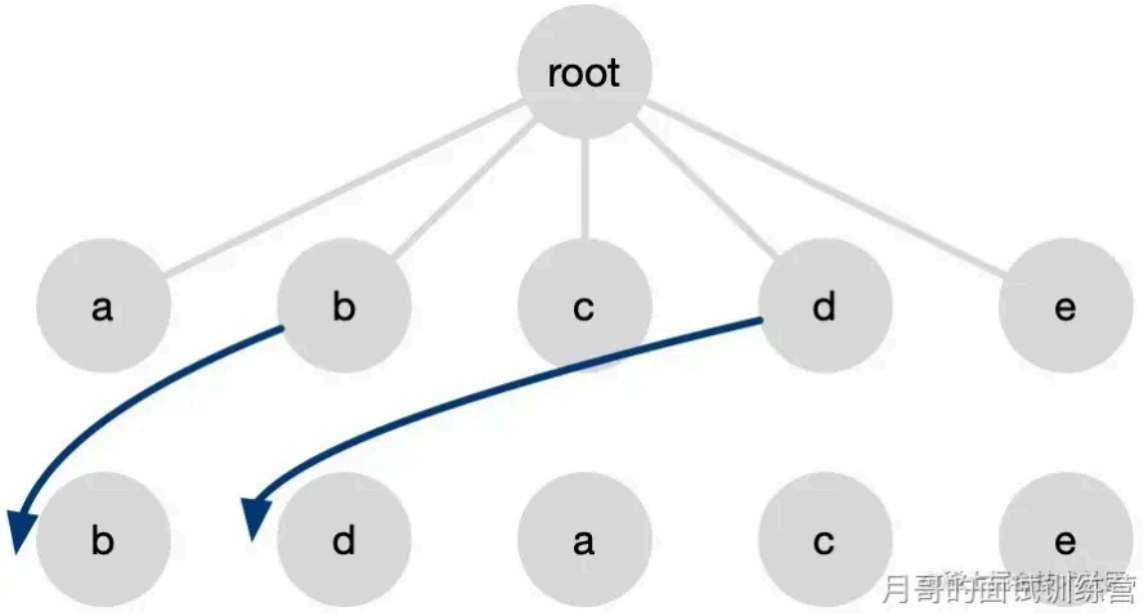
2、遍历 New，顺便利用上面的 Map 记录下下标，同时 Old 在 New 中不存在的说明被删除了，直接删除。

3、不存在的位置补 0，我们拿到 e:4 d:3 c:2 h:0 这样一个数组，下标 0 是新增，非 0 就是移过来的，批量转化为插入操作即可。

最后一步的优化也很关键，我们不要看见不同就随便移动，为了性能最优，要保证移动次数尽可能的少，那么怎么才能尽可能的少移动呢？假设我们随意移动，如下图所示：



但其实最优的移动方式是下面这样：



为什么呢？因为移动的时候，其他元素的位置也在相对变化，可能做了 A 效果同时，也把 B 效果给满足了，也就是说，找到那些相对位置有序的元素保持不变，让那些位置明显错误的元素挪动即是最优的。

什么是相对有序？a c e 这三个字母在 Old 原始顺序 a b c d e 中是相对有序的，我们只要把 b d 移走，这三个字母的位置自然就正确了。因此我们只需要找到 New 数组中的 最长子序列。具体的找法可以当作一个小算法题了，由于知道每个元素的实际下标，比如这个例子中，下标是这样的：

[b:1, d:3, a:0, c:2, e:4]

肉眼看上去，连续自增的子串有 b d 和 a c e，由于 a c e 更长，所以选择后者。

换成程序去做，可以采用贪心 + 二分法进行查找，详细可以看这道题 最长递增子序列，时间复杂度 O(nlogn)。

由于该算法得出的结果顺序是乱的，Vue 采用提前复制数组的方式辅助找到了正确序列。

注意：上面是 Vue DIFF 算法的一个原理展示。

Diff 总结有这么几点考虑：

1、完全对比 $O(n^3)$ 无法接受，故降级为同层对比的 O(n) 方案。

2、为什么降级可行？因为跨层级很少发生，可以忽略。

3、同层级也不简单，难点是如何高效位移，即最小步数完成位移。

4、Vue 为了尽量不移动，先左右夹击跳过不变的，再找到最长连续子串保持不动，移动其他元素。

Vue 列表渲染中 Key 的作用

在 Vue 的列表渲染中，key 是用于标识不同元素的唯一属性。它主要有以下作用： 1、提供稳定的节点身份：在使用 v-for 进行列表渲染时，Vue 会根据被迭代数据生成对应的虚拟DOM节点。如果列表项的顺序发生变化，没有设置 key 的情况下，Vue 将无法准确追踪每个虚拟DOM节点的状态，导致出现错误的更新。而设置了 key 后，Vue 可以通过 key 值来准确地检测每个节点的变化，提供更准确和高效的更新策略。 2、优化列表渲染性能：带有唯一 key 值的节点可以减少虚拟DOM的比对操作。Vue 在进行列表渲染时会使用 Diff 算法来找出新旧虚拟DOM节点之间的差异并进行局部更新。如果列表项没有设置 key，则会使用默认的比对规则，导致需要对全部节点进行比对。而设置了唯一的 key 值后，Vue 可以通过 key 进行快速查找，只对变化的节点进行更新，从而提高渲染性能。 3、保持节点状态和避免重排：当列表项重新排序或动态增删时，没有设置 key 的情况下，DOM 元素会频繁地在父组件中移动，导致浏览器重新计算元素的样式和布局，影响性能。而通过设置合适的 key 值，可以让 Vue 识别出每个元素的稳定身份，减少 DOM 元素的移动，从而保持节点的状态和避免不必要的重排。 需要注意的是，key 必须是唯一且稳定的字符串或数字，每个列表项都应该有一个对应的唯一 key 值。在使用 v-for 进行列表渲染时，建议优先选择具有稳定唯一值的属性作为 key，避免使用索引等动态变化的值。此外，不同层级的节点之间的 key 应该是不同的，并且不同列表之间的 key 也应该是不同的，以确保唯一性。

Vue3 的静态标记算法

Vue 3 的静态标记算法是一种在虚拟DOM Diff 过程中用来标记静态节点的技术。它的主要目的是减少对那些不会发生变化的节点进行无谓的Diff计算，从而提升渲染性能。 静态节点是指在组件渲染过程中不会改变的节点，包括元素节点、文本节点等。这些节点的内容在组件的多次更新中保持不变。 Vue 3 的静态标记算法将标记过程从Diff过程中分离出来，可以在编译阶段对模板进行静态分析并标记。具体的过程如下：

1、编译阶段：在Vue 3 中，模板会在编译阶段被转换为渲染函数。在此过程中，编译器会静态地分析模板，并识别其中的静态节点。 2、静态标记：通过静态标记算法，编译器会为静态节点添加额外的标记信息。这些标记信息可以帮助运行时的Diff算法在比较新旧虚拟DOM树时忽略那些已经被标记为静态的节点。 3、Diff 过程：在组件重新渲染时，Vue 3 的Diff算法会利用静态标记信息，在遍历虚拟DOM树时跳过那些被标记为静态的节点，从而减少了对它们的Diff计算。 通过静态标记算法，Vue 3 可以更加智能地处理那些不会发生变化的节点，避免了不必要的计算和更新操作，提高了渲染性能。同时，静态标记算法也为 Vue 3 引入其他优化策略（如Patch算法）提供了基础，进一步提升了组件的渲染效率。

Vue3 的 Patch 算法

其实我们上面提到的 DOM DIFF 算法，在 Vue 中的表现就是 Patch 算法。 Vue 3 中的 Patch 算法是用于比较新旧虚拟 DOM，并将变化应用到实际 DOM 的一种算法。它是 Vue 3 在渲染过程中使用的核心算法之一，用于实现高效的 DOM 更新。 Patch 算法的主要目标是尽量复用现有的 DOM 节点，最小化对真实 DOM 的操作，从而提高性能。具体的过程如下： 1、创建新节点：首先，根据新的虚拟 DOM，创建一个新的节点树。 2、比较新旧节点：将新的虚拟 DOM 树与旧的虚拟 DOM 树进行逐层比较。在比较过程中，Patch 算法会根据节点的类型、属性、子节点等信息来判断节点的变化情况。 3、更新差异：当发现节点存在差异时，Patch 算法会有针对性地进行更新操作，以尽量减少对真实 DOM 的操作。可能的更新操作包括替换节点、更新属性、移动节点位置等。 4、递归处理子节点：如果存在子节点，Patch 算法会递归调用自身，继续比较和更新子节点。 通过 Patch 算法，Vue 3 可以有效地识别出虚拟 DOM 中哪些节点发生了变化，并且只对变化的部分进行实际 DOM 的更新操作，从而提高了渲染的效率。 在 Patch 算法的实现中，Vue 3 还使用了一些优化策略，例如使用静态标记算法来跳过静态节点的比较和更新，以及使用 key 属性来进行更精确的更新操作。这些优化措施进一步提升了 Patch 算法的性能和效率。 总之，Patch 算法是 Vue 3 实现高效渲染的关键算法之一，它通过差异比对和最小化 DOM 操作的方式，使得 Vue 3 在数据变化时能够快速、准确地更新视图。

参考资料： ●精读《DOM diff 原理详解》