vue100面试题（https://github.com/minsion）

面试题 1. 请简述Vue插件和组件的区别？

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

一、组件是什么
（1）组件的定义：
组件就是把图形、非图形的各种逻辑均抽象为一个统一的概念（组件）来实现开发的模式，在Vue中每一个.vue文件都可以视为一个组件
（2）组件的优势：
降低整个系统的耦合度，在保持接口不变的情况下，我们可以替换不同的组件快速完成需求，例如输入框，可以替换为日历、时间、范围等组件作具体的实现调试方便，由于整个系统是通过组件组合起来的，在出现问题的时候，可以用排除法直接移除组件，或者根据报错的组件快速定位问题，之所以能够快速定位，是因为每个组件之间低耦合，职责单一，所以逻辑会比分析整个系统要简单提高可维护性，由于每个组件的职责单一，并且组件在系统中是被复用的，所以对代码进行优化可获得系统的整体升级

二、插件是什么
插件通常用来为 Vue 添加全局功能。插件的功能范围没有严格的限制——一般有下面几种：

添加全局方法或者属性。如: vue-custom-element
添加全局资源：指令/过滤器/过渡等。如 vue-touch
通过全局混入来添加一些组件选项。如vue-router
添加 Vue 实例方法，通过把它们添加到 Vue.prototype 上实现。
一个库，提供自己的 API，同时提供上面提到的一个或多个功能。如vue-router

三、两者的区别
两者的区别主要表现在以下几个方面：
（1）编写形式
（2）注册形式
（3）使用场景
（4）编写形式
（5）编写组件
编写一个组件，可以有很多方式，我们最常见的就是vue单文件的这种格式，每一个.vue文件我们都可以看成是一个组件
vue文件标准格式

```
<br />
<template><br />
</template><br />
<script><br />
export default{ <br />
    ...<br />
}<br />
</script><br />
```

我们还可以通过template属性来编写一个组件，如果组件内容多，我们可以在外部定义template组件内容，如果组件内容并不多，我们可直接写在template属性上

```
<br />
<template id="testComponent">      // 组件显示的内容<br />
    <div>component!</div>   <br />
</template><br />
```

```
Vue.component('componentA',{
template: '#testComponent'
template: `
component
` // 组件内容少可以通过这种形式
})
```

四：编写插件
vue插件的实现应该暴露一个 install 方法。这个方法的第一个参数是 Vue 构造器，第二个参数是一个可选的选项对象

```
MyPlugin.install = function (Vue, options) {
// 1. 添加全局方法或 property
Vue.myGlobalMethod = function () {
// 逻辑...
}
// 2. 添加全局资源
Vue.directive('my-directive', {
bind (el, binding, vnode, oldVnode) {
// 逻辑...
}
...
})
// 3. 注入组件选项
Vue.mixin({
created: function () {
// 逻辑...
}
...
})
// 4. 添加实例方法
Vue.prototype.$myMethod = function (methodOptions) {
// 逻辑...
}
}
```

注册形式
组件注册
vue组件注册主要分为全局注册与局部注册
全局注册通过Vue.component方法，第一个参数为组件的名称，第二个参数为传入的配置项

Vue.component('my-component-name', { /* ... */ })
局部注册只需在用到的地方通过components属性注册一个组件
const component1 = {...} // 定义一个组件
export default {
components:{
component1 // 局部注册
}
}
插件注册
插件的注册通过Vue.use()的方式进行注册（安装），第一个参数为插件的名字，第二个参数是可选择的配置项
Vue.use(插件名字,{ /* ... */})
注意的是：
注册插件的时候，需要在调用 new Vue() 启动应用之前完成
Vue.use会自动阻止多次注册相同插件，只会注册一次

五：使用场景
具体的其实在插件是什么章节已经表述了，这里在总结一下
组件 (Component) 是用来构成你的 App 的业务模块，它的目标是 App.vue
插件 (Plugin) 是用来增强你的技术栈的功能模块，它的目标是 Vue 本身
简单来说，插件就是指对Vue的功能的增强或补充

面试题 2. 简述Vue的MVVM 模式?

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

MVVM 是 Model-View-ViewModel的缩写，即将数据模型与数据表现层通过数据驱动进行分离，从而只需要关系数据模型的开发，而不需要考虑页面的表现，具体说来如下：Model代表数据模型：主要用于定义数据和操作的业务逻辑。View代表页面展示组件（即dom展现形式）：负责将数据模型转化成UI 展现出来。ViewModel为model和view之间的桥梁：监听模型数据的改变和控制视图行为、处理用户交互。通过双向数据绑定把 View 层和 Model 层连接了起来，而View 和 Model 之间的同步工作完全是自动的，无需人为干涉在MVVM架构下，View 和 Model 之间并没有直接的联系，而是通过ViewModel进行交互，Model 和 ViewModel 之间的交互是双向的， 因此View 数据的变化会同步到Model中，而Model 数据的变化也会立即反应到View 上。

1：Model（模型）
模型是指代表真实状态内容的领域模型（面向对象），或指代表内容的数据访问层（以数据为中心）。
2：View（视图）
就像在MVC和MVP模式中一样，视图是用户在屏幕上看到的结构、布局和外观（UI）。
3：ViewModel（视图模型）
视图模型是暴露公共属性和命令的视图的抽象。MVVM没有MVC模式的控制器，也没有MVP模式的
presenter，有的是一个绑定器。在视图模型中，绑定器在视图和数据绑定器之间进行通信。

MVVM 优点:

低耦合 :View可以独立于Model变化和修改,一个ViewModel可以绑定到不同的View上,当View变化
的时候Model可以不变,当Model变化的时候View也可以不变。
可重用性 ：可以把一些视图逻辑放在一个ViewModel里面,让很多View重用这段视图逻辑。
独立开发 ：开发人员可以专注于业务逻辑和数据的开发,设计人员可以专注于页面的设计

面试题 3. 简述MVC与MVVM的区别 ？

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

MVC和MVVM的区别并不是VM完全取代了C，ViewModel存在目的在于抽离Controller中展示的业务逻辑，而不是替代Controller，其它视图操作业务等还是应该放在Controller中实现。也就是说MVVM实现的是业务逻辑组件的重用。

1：MVC中Controller演变成MVVM中的ViewModel
2：MVVM通过数据来显示视图层而不是节点操作
3：MVVM主要解决了MVC中大量的dom操作使页面渲染性能降低,加载速度变慢,影响用户体验

面试题 4. 简述Vue组件通讯有哪些方式 ？

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

Vue组件通讯有方式：
1、props 和 $emit 父组件向子组件传递数据是通过props传递的，子组件传递给父组件是通过$emit触发事件来做到的。
2、$parent 和 $children 获取单签组件的父组件和当前组件的子组件。
3、$attrs 和 $listeners A -> B -> C。Vue2.4开始提供了$attrs和$listeners来解决这个问题。
4、父组件中通过 provide 来提供变量，然后在子组件中通过 inject 来注入变量。（官方不推荐在实际业务中适用，但是写组件库时很常用。）
5、$refs 获取组件实例。
6、envetBus 兄弟组件数据传递，这种情况下可以使用事件总线的方式。
7、vuex 状态管理

面试题 5. 简述Vue的生命周期方法有哪些?

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

Vue的生命周期方法：

1：beforeCreate 在实例初始化之后，数据观测（data observe）和 event/watcher 事件配置之前被调用。在当前阶段 data、methods、computed 以及 watch 上的数据和方法都不能被访问。

2：created 实例已经创建完成之后被调用。在这一步，实例已经完成以下的配置：数据观测（data observe），属性和方法的运算，watch/event 事件回调。这里没有 $el，如果非要想与 DOM 进行交互，可以通过vm.$nextTick 来访问 DOM。
3：beforeMount 在挂载开始之前被调用：相关的 render 函数首次被调用。
4：mounted 在挂载完成后发生，在当前阶段，真实的 Dom 挂载完毕，数据完成双向绑定，可以访问到 Dom节点。
5：beforeUpdate 数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁 （patch）之前。可以在这个钩子中进一步地更改状态，这不会触发附加的重渲染过程。（数据修改页面未修改）
6：updated 发生在更新完成之后，当前阶段组件 Dom 已经完成更新。要注意的是避免在此期间更新数据，因为这个可能导致无限循环的更新，该钩子在服务器渲染期间不被调用。
7：beforeDestroy 实例销毁之前调用。在这一步，实例仍然完全可用。我们可以在这时进行善后收尾工作，比如清除定时器。
8：destroyed Vue实例销毁后调用。调用后，Vue实例指示的东西都会解绑定，所有的事件监听器会被移除，左右的子实例也会被销毁，该钩子在服务器端渲染不被调用。
9：activated keep-alive 专属，组件被激活时调用
10：deactivated keep-alive 专属，组件被销毁时调用

面试题 6. 简述 v-if 和 v-show 的区别？

推荐指数：★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

v-if 在编译过程中会被转化成三元表达式，条件不满足时不渲染此节点。元素销毁和重建控制显示隐藏
v-show 会被编译成指令，条件不满足时控制样式将此节点隐藏（display:none） css样式控制

使用场景
v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景。
v-show 适用于需要非常频繁切换条件的场景。
扩展补充：display:none 、 visibility:hidden 和 opacity:0 之间的区别?

三者公共点都是：隐藏

v-if 和 v-show 不同点：
是否占据空间。
display:none，隐藏之后不占位置；visibility:hidden、opacity:0，隐藏后任然占据位置。
子元素是否继承。
display:none --- 不会被子元素继承，父元素都不存在了，子元素也不会显示出来。
visibility:hidden --- 会被子元素继承，通过设置子元素 visibility:visible 来显示子元素。
opacity:0 --- 会被子元素继承，但是不能设置子元素 opacity:0 来先重新显示。

事件绑定。
display:none 的元素都已经不存在了，因此无法触发他绑定的事件。
visibility:hidden 不会触发他上面绑定的事件。
opacity:0 元素上面绑定的事件时可以触发的。

过度动画。
transition对于display是无效的。
transition对于visibility是无效的。
transition对于opacity是有效的。

面试题 7. 简述 Vue 有哪些内置指令？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

vue 内置指令：
v-once – 定义它的元素或组件只渲染一次，包括元素或组件的所有节点，首次渲染后，不再随数据的变化重新渲染，将被视为静态内容。
v-cloak – 这个指令保持在元素上直到关联实例结束编译 -- 解决初始化慢到页面闪动的最佳实践。
v-bind – 绑定属性，动态更新HTML元素上的属性。例如 v-bind:class。
v-on – 用于监听DOM事件。例如 v-on:click v-on:keyup
v-html – 赋值就是变量的innerHTML -- 注意防止xss攻击
v-text – 更新元素的textContent
v-model – 1、在普通标签。变成value和input的语法糖，并且会处理拼音输入法的问题。2、再组件上。也是处理value和input语法糖。
v-if / v-else / v-else-if。可以配合template使用；在render函数里面就是三元表达式。
v-show – 使用指令来实现 -- 最终会通过display来进行显示隐藏
v-for – 循环指令编译出来的结果是 -L 代表渲染列表。优先级比v-if高最好不要一起使用，尽量使用计算属性去解决。注意增加唯一key值，不要使用index作为key。
v-pre – 跳过这个元素以及子元素的编译过程，以此来加快整个项目的编译速度。

面试题 8. 简述怎样理解 Vue 的单项数据流？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

Vue的单向数据流是指数据在Vue应用中的流动方向是单向的，数据总是从父组件传到子组件，子组件没有权利修改父组件传过来的数据，只能请求父组件对原始数据进行修改。这样会防止从子组件意外改变父组件的状态，从而导致你的应用的数据流向难以理解。
注意：在子组件直接用 v-model 绑定父组件传过来的 props 这样是不规范的写法，开发环境会报警告。
如果实在要改变父组件的 props 值可以再data里面定义一个变量，并用 prop 的值初始化它，之后用$emit 通知父组件去修改。
多种方法实现：在子组件直接用 v-model 绑定父组件传过来的 props

这种单向数据流的设计有以下几个优点：

1. 易于追踪数据流：由于数据的流动方向是单向的，我们可以很容易地追踪数据的来源和去向，减少了数据流动的复杂性，提高了代码的可读性和可维护性。
2. 提高组件的可复用性：通过props将数据传递给子组件，使得子组件可以独立于父组件进行开发和测试。这样一来，我们可以更方便地复用子组件，提高了组件的可复用性。

3. 避免数据的意外修改：由于子组件不能直接修改父组件的数据，可以避免数据被意外修改的情况发生。这样可以提高应用的稳定性和可靠性。

单向数据流也有一些限制和不足之处。例如，当数据需要在多个组件之间进行共享时，通过props传递数据会变得繁琐，这时可以考虑使用Vuex等状态管理工具来管理共享数据。单向数据流也可能导致组件之间的通信变得复杂，需要通过事件的方式进行数据传递和更新。
Vue的单向数据流是一种有助于提高应用可维护性和可预测性的设计模式，通过明确数据的流动方向，使得代码更易于理解和维护。

面试题 9. 简述Vue computed 和 watch 的区别和运用的场景？

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

一、Computed
在Vue.js，computed 是一个非常有用的属性，它允许声明计算属性，这些属性会根据其依赖的数据进行自动更新，而无需手动触发。computed 属性常用于根据现有的响应式数据进行计算，以生成派生的数据，而这些数据的值会随着依赖数据的变化而自动更新。
computed 的使用场景和方法如下：
计算属性的缓存： computed 属性会在其依赖的数据发生变化时进行重新计算，但是它会缓存计算结果。这意味着，只有在依赖数据变化时，才会触发计算函数重新执行，如果依赖数据没有变化，会直接返回之前缓存的计算结果，这有助于性能优化。
依赖关系管理： 当有一些数据需要根据其他数据进行计算，而这些数据之间存在依赖关系时，使用 computed 可以更清晰地管理这些依赖关系，而不需要手动跟踪它们的变化。
模板中的使用： 在模板中，可以像使用普通属性一样使用计算属性。这使得能够将复杂的计算逻辑从模板中分离出来，让模板更加简洁和易读。

```
<br />
<template><br />
  <div><br />
    <p>原始价格: { {<br />
              originalPrice<br />
       }<br />
}<br />
</p><br />
    <p>折扣后价格: { {<br />
              discountedPrice<br />
       }<br />
}<br />
</p><br />
  </div><br />
</template><br />
<script><br />
export default {<br />
       data() {<br />
              return {<br />
                     price: 100,<br />
```

```
                        discount: 0.2,<br />
                }<br />
                ;<br />
        }<br />
        ,<br />
          computed: {<br />
                originalPrice() {<br />
                        return this.price;<br />
                }<br />
                ,<br />
                    discountedPrice() {<br />
                        return this.price * (1 - this.discount);<br />
                }<br />
                ,<br />
        }<br />
        ,<br />
}<br />
;<br />
</script><br />
```

在这个例子中，discountedPrice 计算属性依赖于 price 和 discount 数据，当它们发生变化时，discountedPrice 会自动更新。
总之，computed 在Vue中用于将响应式数据的计算逻辑从模板中分离出来，提高代码可读性和维护性，并自动处理依赖关系和缓存计算结果，从而帮助优化性能。

二、与watch的区别
虽然 computed 和 watch 在某些情况下可以实现相似的功能，但它们的用途和工作方式有一些重要区别。以下是它们之间的主要区别：
1. 计算属性 (computed):
用途： 用于派生出一些新的响应式属性，这些属性的值基于其他已有的响应式属性计算得出。
自动缓存： computed 属性会自动缓存计算结果，只有在依赖属性变化时才会重新计算。这对于频繁访问和计算的情况非常有用，以避免不必要的重复计算。
同步： 计算属性是同步的，它们会立即返回计算结果。
2. 观察属性 (watch):
用途： 用于监听一个特定的数据变化，并在变化发生时执行自定义的操作，如异步操作、API 请求等。
无缓存： watch 不会缓存旧值或计算结果，每当被监听的属性发生变化时，都会触发对应的回调函数。
异步支持： watch 回调函数可以执行异步操作，例如发送网络请求或执行定时器等。
适用于复杂逻辑： 当需要在属性变化时执行一些复杂逻辑、副作用或异步操作时，watch 更合适。
要根据你的具体需求来选择使用 computed 还是 watch：
如果你需要根据已有的响应式属性进行计算，而且这个计算是同步的，那么使用 computed 更合适。
如果你需要监听特定属性的变化，并在变化时执行一些异步操作或复杂逻辑，那么使用 watch 更合适。
绝大多数情况下，computed 能够满足计算和派生数据的需求，而 watch 则更适用于监听属性的变化并执行副作用操作。

三、频繁访问和计算的情况，有哪些

当涉及到频繁访问和计算的情况时，使用计算属性可以避免不必要的重复计算，提高性能。以下是一些具体的例子：

价格计算： 在电子商务应用中，商品价格可能会受到多个因素的影响，如折扣、运费等。如果你需要在页面中显示折扣后的价格、总价等，这些计算可以使用计算属性来处理。

过滤和搜索： 在一个列表中过滤和搜索数据时，你可能需要根据搜索关键字或选定的过滤条件来计算显示在页面上的数据集。这些过滤和搜索逻辑可以放在计算属性中，以保持页面的简洁性和性能。

图表数据生成： 当你在应用中显示图表或图形时，图表的数据通常是从原始数据集中进行聚合、统计等计算得出的。使用计算属性可以确保图表数据在依赖数据变化时自动更新，而无需手动处理。

动态样式： 如果你需要根据一些条件来设置元素的样式，例如根据用户的输入改变按钮的颜色，可以使用计算属性来根据条件动态计算样式。

排序和排名： 在显示排行榜、评分列表或任何需要排序的列表时，你可能需要根据某些规则对数据进行排序和排名。使用计算属性可以将排序逻辑与模板分离，并确保排序是响应式的。

数据格式化： 当你需要在页面上显示格式化的数据，比如日期、货币、百分比等，你可以使用计算属性来处理数据格式化逻辑。

四、关于同步和异步

computed 和 watch 的最大区别在于缓存和同步/异步的处理。下面进一步解释一下关于同步和异步方面的内容，特别是在 watch 中。

同步和异步：

同步（Sync）： 同步操作意味着操作会立即执行，直到操作完成后才继续执行下一个操作。在 Vue 的上下文中，这表示计算会立即发生，没有阻塞，没有延迟。

异步（Async）： 异步操作允许在操作开始后，不必等待其完成，而是可以继续执行其他操作。在 Vue 中，异步操作通常涉及到网络请求、定时器等，这些操作不会阻塞 JavaScript 的主线程，而是在后台进行。

在 watch 中的异步行为：

watch 提供了一种监视数据变化并采取响应行动的机制。当被监视的数据发生变化时，watch 的回调函数将被调用。这个回调函数可以包含异步操作，比如发送网络请求、执行定时器等。这些异步操作不会阻塞主线程，而是在后台执行，这样页面仍然可以继续响应用户操作。

为什么说发送网络请求或执行定时器是异步请求?

异步操作的概念：

异步操作是指在代码执行过程中，不需要等待某个操作完成就可以继续执行其他操作的方式。在异步操作中，你通常会指定一个回调函数，以便在操作完成时得到通知。这允许你同时执行多个操作而不会阻塞整个程序或页面。

发送网络请求的异步性质：

当你发送一个网络请求时（例如使用 XMLHttpRequest、Fetch API 或 Axios），你的程序并不会等待服务器响应返回。相反，它会继续执行后续代码。当服务器响应到达时，会触发你提供的回调函数。这样，你可以在等待响应的同时继续执行其他操作，从而实现并发性。

执行定时器的异步性质：

当你设置一个定时器（例如使用 setTimeout 或 setInterval）时，你指定了一个时间，经过该时间后，指定的回调函数会被触发。在等待这段时间内，JavaScript 不会阻塞程序的其他部分。这使得你可以同时处理其他任务，而不必等待定时器时间结束。

总之，异步操作允许你在某些操作等待完成时继续执行其他操作，而不会阻塞程序的执行。这与同步操作不同，后者会在操作完成之前阻塞代码的执行。在 Vue 的 watch 中，你可以使用异步操作，如发送网络请求或执行定时器，而不会影响整个应用的响应性。

五、回调函数

回调函数是一种在某个事件发生或者某个条件满足时被调用的函数。它是一种常见的编程概念，用于实现异步操作、事件处理、以及在特定情况下执行代码等。

在 JavaScript 中，函数可以作为值传递，因此你可以将一个函数传递给另一个函数，以便在适当的时机调用它。这就是回调函数的基本概念。

回调函数的特点和用途：

异步操作： 回调函数常用于处理异步操作，如网络请求、文件读取等。你可以在异步操作完成后执行回调函数，以响应操作的结果。

事件处理： 在事件驱动的编程中，你可以指定某个事件发生时应该执行的回调函数。比如，在点击按钮时触发的点击事件，就可以关联一个回调函数来响应按钮点击。

参数传递： 回调函数可以接受参数，这使得你可以将数据传递给回调函数，让它在执行时使用这些数据。

动态逻辑： 通过回调函数，你可以在不同的情况下执行不同的逻辑。根据不同的参数或条件，可以传递不同的回调函数来实现动态逻辑。

示例：

```
function doSomething(callback) {
console.log("Doing something...");
// 模拟操作的延迟
setTimeout(function() {
console.log("Operation completed!");
// 执行回调函数
callback();
}
, 1000);
}
function onOperationComplete() {
console.log("Callback executed: Operation complete!");
}
// 调用 doSomething 并传递回调函数
doSomething(onOperationComplete);
```

在这个例子中，doSomething 函数模拟了一个异步操作，然后在操作完成后调用传递的回调函数 onOperationComplete。这种方式可以在异步操作完成后执行额外的逻辑。

总之，回调函数是一种灵活的方式，允许你在特定事件发生或条件满足时执行一些代码，从而实现异步操作和动态逻辑。

对于上面的例子，doSomething(onOperationComplete);

首先打印了console.log("Doing something...");

然后打印了 console.log("Operation completed!");

最后打印了 console.log("Callback executed: Operation complete!");

doSomething(onOperationComplete) 的执行顺序如下：

首先，doSomething 函数被调用，并在控制台打印出 "Doing something..."。

然后，通过 setTimeout 模拟了一个异步操作，等待了1秒（1000毫秒）。

在异步操作完成后，"Operation completed!" 被打印到控制台。

接着，传递的回调函数 onOperationComplete 被执行，控制台打印 "Callback executed: Operation complete!"。

也就是说，对于doSomething，需要用到onOperationComplete的结果，但是doSomething本身也有需要执行的逻辑，使用回调函数可以在不影响doSomething的情况下异步拿到onOperationComplete

回调函数的一个重要用途是在处理异步操作时，可以在操作完成后执行额外的逻辑，同时不影响原始操作本身的流程。在上面提到的例子中，doSomething 函数本身可能有一些需要执行的逻辑，但也需要在操作完成后得到 onOperationComplete 函数的结果或执行一些额外操作。
使用回调函数，可以将 onOperationComplete 作为参数传递给 doSomething，然后 doSomething 在异步操作完成后调用这个回调函数，以确保操作完成后执行相关的逻辑。这种方式使得代码结构更清晰，并且能够处理异步操作的结果。
这种模式在处理异步操作、事件处理以及动态逻辑等方面非常有用。
vue中我们通常在哪里使用回调函数?
在Vue中，回调函数通常在以下几个场景中使用：
生命周期钩子函数： Vue组件有一系列的生命周期钩子函数，它们会在组件生命周期中的不同阶段被自动调用。你可以在这些钩子函数中传递回调函数来执行在特定生命周期阶段执行的逻辑。例如，在 created、mounted 或 updated 钩子函数中。
异步操作： 当涉及到异步操作，例如网络请求、定时器等，你可以将回调函数作为异步操作完成后的处理逻辑。这样可以确保在操作完成后执行额外的操作，比如更新组件的状态。
事件处理： Vue中的事件处理也经常使用回调函数。你可以在模板中绑定事件，并在事件触发时执行指定的回调函数。例如，监听按钮的点击事件。
Watch 监听器： 在Vue的组件中，你可以使用 watch 监听器来监听数据的变化。当被监听的数据发生变化时，你可以指定一个回调函数来执行相应的操作，如执行异步请求、更新视图等。
父子组件通信： 当你在父子组件之间进行通信时，可以通过 props 和自定义事件（$emit）来传递回调函数。子组件可以通过调用传递的回调函数来将数据传递回父组件。
路由导航守卫： 在Vue的路由中，你可以使用导航守卫来在路由导航发生时执行特定的逻辑。这些守卫可以接受回调函数，用于在不同的导航阶段执行代码。
总之，回调函数在Vue中用于处理生命周期事件、异步操作、事件处理、数据变化监听等多个场景。它们是一种实现灵活、动态逻辑的重要方式，帮助你编写更加响应式和交互性的应用程序。
生命周期使用回调函数：
几乎所有生命周期都可以使用回调函数，但并不是所有生命周期都适合使用回调函数。每个生命周期阶段都有其特定的用途和适用情况。以下是一些常见的生命周期阶段以及是否适合使用回调函数的简要说明：
beforeCreate： 在实例初始化之后，数据观测 (data observation) 和事件配置之前被调用。这个阶段适合执行一些初始化设置，但不太适合使用回调函数，因为此时组件还没有被完全创建。
created： 在实例已经创建完成之后被调用。在这个阶段，组件的数据和方法已经初始化，你可以在这里执行一些异步操作，或者执行一些需要在组件实例创建后才能执行的逻辑。回调函数可以用于执行初始化数据加载等操作。
beforeMount： 在挂载开始之前被调用。这个阶段是在模板编译成渲染函数之后，但在渲染函数首次调用之前。在这个阶段使用回调函数的情况较少，通常用于一些底层操作。
mounted： 在挂载结束后被调用，即 DOM 元素已经被插入页面中。在这个阶段，你可以执行与 DOM 相关的操作，如初始化第三方库、操作 DOM 元素等。这个阶段较为适合使用回调函数。
beforeUpdate： 在数据更新之前被调用，发生在虚拟 DOM 重新渲染和打补丁之前。这个阶段通常不适合使用回调函数，而是用于一些底层操作。
updated： 在数据更改导致虚拟 DOM 重新渲染和打补丁后被调用。这个阶段适合执行与数据更改有关的操作，但通常情况下不是最适合使用回调函数的地方。
beforeDestroy： 在实例销毁之前调用。在这个阶段，实例还完全可用，你可以执行一些清理工作。回调函数可以用于执行销毁前的一些操作。
destroyed： 在实例销毁之后被调用。这个阶段适合执行一些最终清理工作，如解绑事件监听器、清理定时器等。回调函数可以用于执行销毁后的操作。
综上所述，虽然几乎所有生命周期都可以使用回调函数，但回调函数的使用会根据生命周期的特

性和目的而有所不同。要根据具体的需求来决定是否在特定的生命周期中使用回调函数。

六、再讲一下this.$nextTick
当在 Vue 中更新数据时，Vue 实际上并不会立即更新 DOM。相反，它会将更新放入一个队列中，然后在适当的时机进行更新。这种方式可以优化性能，避免不必要的 DOM 操作。但有时候你可能需要在 DOM 更新后执行一些操作，这时就可以使用 this.$nextTick。
this.$nextTick 是 Vue 提供的一个实例方法，它接受一个回调函数作为参数，并会在下次 DOM 更新循环结束之后调用这个回调函数。这意味着你可以确保在 DOM 更新完成后执行一些操作，例如读取更新后的 DOM 元素属性、执行某些操作等。
使用场景：
DOM 更新后的操作： 如果你想在更新后访问更新后的 DOM 元素，比如获取元素的高度或宽度，可以将这些操作放在 this.$nextTick 的回调函数中。
保证更新完成后执行： 有时候你可能需要在数据更新后执行一些操作，但又不想在每次数据更新时都执行，而是确保在整个更新周期完成后执行。

面试题 10. 简述 v-if 和 v-for 为什么不建议一起使用？

推荐指数： ★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

v-for和v-if不要在同一标签中使用，因为解析时先解析v-for在解析v-if。如果遇到需要同时使用时可以考虑写成计算属性的方式。
永远不要把 v-if 和 v-for 同时用在同一个元素上，带来性能方面的浪费（每次渲染都会先循环再进行条件判断）
如果避免出现这种情况，则在外层嵌套template（页面渲染不生成dom节点），在这一层进行v-if判断，然后在内部进行v-for循环

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

```
computed: {
items: function() {
return this.list.filter(function (item) {
return item.isShow
})
}
}
```

面试题 11. 简述 Vue 2.0 响应式数据的原理（重点）？

推荐指数： ★★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

整体思路是 数据劫持 + 观察者模式
Vue 在初始化数据时，会使用 Object.defineProperty 重新定义 data 中的所有属性，当页面使用对应属性时，首先会进行依赖收集（收集当前组件的 watcher），如果属性发生变化会通知相关依赖进行更新操作（发布订阅）
Vue2.x 采用数据劫持结合发布订阅模式（PubSub 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。
当把一个普通 Javascript 对象传给 Vue 实例来作为它的 data 选项时，Vue 将遍历它的属性，用
Object.defineProperty 将它们转为 getter/setter。用户看不到 getter/setter，但是在内部它们让
Vue 追踪依赖，在属性被访问和修改时 通知变化 。
Vue 的数据 双向绑定 整合了 Observer，Compile 和 Watcher 三者，通过 Observer 来监听 自己的
model 的数据变化，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（例如 input 操作）->数据
model 变更的双向绑定效果。
Vue3.x 放弃了 Object.defineProperty ，使用 ES6 原生的 Proxy，来解决以前使用 Object.defineProperty 所存在的一些问题。
1、Object.defineProperty 数据劫持
2、使用 getter 收集依赖 ，setter 通知 watcher派发更新。
3、watcher 发布订阅模式。

面试题 12. 简述Vue 如何检测数组变化 ？

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

Vue2.x 中实现检测数组变化的方法，是将数组的常用方法进行了 重写 。Vue 将 data 中的数组进行了 原型链重写 ，指向了自己定义的数组原型方法。这样当调用数组 api 时，可以通知依赖更新 。如果数组中包含着引用类型，会对数组中的引用类型 再次递归遍历进行监控 。这样就实现了 监测数组变化 。
流程:
1. 初始化传入 data 数据执行 initData
2. 将数据进行观测 new Observer
3. 将数组原型方法指向重写的原型
4. 深度观察数组中的引用类型
有两种情况无法检测到数组的变化 。
1. 当利用索引直接设置一个数组项时，例如 vm.items[indexOfItem] = newValue
2. 当修改数组的长度时，例如 vm.items.length = newLength
不过这两种场景都有对应的解决方案。
利用索引设置数组项的替代方案
//使用该方法进行更新视图

// vm.$set，Vue.set的一个别名
vm.$set(vm.items, indexOfItem, newValue)

面试题 13. 请解释Vue的父子组件生命周期钩子函数执行顺序 ？

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

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

面试题 14. 简述 v-model 双向绑定的原理是什么?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

v-model 本质 就是 ：value + input 方法的语法糖 。可以通过 model 属性的 prop 和 event 属性来进行自定义。原生的 v-model，会根据标签的不同生成不同的事件和属性。
例如：
1. text 和 textarea 元素使用 value 属性和 input 事件
2. checkbox 和 radio 使用 checked 属性和 change 事件
3. select 字段将 value 作为 prop 并将 change 作为事件
以输入框为例，当用户在输入框输入内容时，会触发 input 事件，从而更新 value。而 value 的改变同样会更新视图，这就是 vue 中的双向绑定。双向绑定的原理，其实现思路 如下：
首先要对 数据进行劫持监听 ，所以我们需要设置 一个监听器 Observe r，用来 监听 所有属性。如果属性发上变化了，就需要告 诉订阅者 Watcher 看是否需要更新 。
因为订阅者是有很多个，所以我们需要有一个 消息订阅器 Dep 来专门收集这些订阅者 ，然后在监听器 Observer 和订阅者 Watcher 之间 进行统一管理的。
接着，我们还需要有一个 指令解析器 Compile ，对每个节点元素进 行扫描和解析 ，将相关指令对应初始化成一个订阅者 Watcher，并替换模板数据或者绑定相应的函数，此时当订阅者 Watcher 接收到相应属性的变化，就会执行对应的更新函数，从而更新视图。
因此接下去我们执行以 下 3 个步骤，实现数据的双向绑定 ：
1. 实现一个 监听器 Observer ，用来 劫持并监听所有属 性，如果有变动的，就通知订阅者。
2. 实现一个 订阅者 Watcher ，可以 收到属性的变化通知并执 行相应的函数，从而更新视图。
3. 实现一个 解析器 Compile ，可以 扫描和解析每个 节点的相关指令，并根据 初始化模板数据以及初始化相应的订阅器

面试题 15. 请简述Vue3.x 响应式数据原理是什么？（重点）

推荐指数：★★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

Vuejs 作为在众多 MVVM(Model-View-ViewModel) 框架中脱颖而出的佼佼者，无疑是值得任何一个前端开发者去深度学习的。
不可置否尤大佬的 VueJs 中有许多值得我们深入研究的内容，但是作为最核心的数据响应式 Reactivity 模块正是我们日常工作中高端相关的内容同时也是 VueJs 中最核心的内容之一。
至于 Vuejs 中的响应式原理究竟有多重要，这里我就不必累赘了。相信大家都能理解它的重要性。
不过这里我想强调的是，所谓响应式原理本质上也是基于 Js 代码的升华实现而已。你也许会觉得它很难，但是这一切只是源于你对他的未知。毕竟只要是你熟悉的 JavaScript ，那么问题就不会很大对吧。
今天我们就让我们基于最新版 Vuejs 3.2 来稍微聊聊 VueJs 中核心模块 Reactive 是如何实现数据响应式的。
前置知识
ES6 Proxy & Reflect
Proxy 是 ES6 提供给我们对于原始对象进行劫持的 Api ，同样 Reflect 内置 Api 为我们提供了对于原始对象的拦截操作。
这里我们主要是用到他们的 get 、 set 陷阱。
Typescript
TypeScript 的作用不言而喻了，文中代码我会使用 TypeScript 来书写。
Esbuild
EsBuild 是一款新型 bundle build tools ，它内部使用 Go 对于我们的代码进行打包整合。
Pnpm
pnpm 是一款优秀的包管理工具，这里我们主要用它来实现 monorepo 。
如果你还没在你的电脑上安装过 pnpm ，那么请你跟随官网安装它很简单，只需要一行 npm install -g pnpm即可。
搭建环境
工欲善其事，必先利其器。在开始之前我们首先会构建一个简陋的开发环境，便于将我们的 TypeScript 构建成为 life 形式，提供给浏览器中直接使用。
因为文章主要针对于响应式部分内容进行梳理，构建环境并不是我们的重点。所以我并不会深入构建环境的搭建中为大家讲解这些细节。
如果你有兴趣，可以跟着我一起来搭建这个简单的组织结构。如果你并不想动手，没关系。我们的重点会放在在之后的代码。
初始化项目目录
首先我们创建一个简单的文件夹，命名为 vue 执行 pnpm init -y 初始化 package.json 。
接下来我们依次创建：
pnpm-workspace.yaml文件
这是一个有关 pnpm 实现 monorepo 的 yaml 配置文件，我们会在稍微填充它。
.npmrc文件
这是有关 npm 的配置信息存放文件。
packages/reactivity目录
我们会在这个目录下实现核心的响应式原理代码，上边我们提过 vue3 目录架构基于 monorepo 的结构，所以这是一个独立用于维护响应式相关的模块目录。
当然，每个 packages 下的内容可以看作一个独立的项目，所以它们我们在 reactivity 目录中执行 pnpm init -y 初始化自己的 package.json。
同样新建 packages/reactivity/src 作为 reactivity 模块下的文件源代码。
packages/share目录
同样，正如它的文件夹名称，这个目录下存放所有 vuejs 下的工具方法，分享给别的模块进行引入使用。
它需要和 reactivity 维护相同的目录结构。

scripts/build.js文件
我们需要额外新建一个 scripts 文件夹，同时新建 scripts/build.js 用于存放构建时候的脚本文件。
image.png
此时目录如图所示。
安装依赖
接下来我们来依次安装需要使用到的依赖环境，在开始安装依赖之前。我们先来填充对应的 .npmrc 文件：
shamefully-hoist = true
默认情况下 pnpm 安装的依赖是会解决幽灵依赖的问题，所谓什么是幽灵依赖你可以查看这篇文章。
这里我们配置 shamefully-hoist = true 意为我们需要第三方包中的依赖提升，也就是需要所谓的幽灵依赖。
这是因为我们会在之后引入源生 Vue 对比实现效果与它是否一致。
你可以在这里详细看到它的含义。
同时，接下里让我们在 pnpm-workspace.yaml 来填入以下代码：
packages:
所有在 packages/ 和 components/ 子目录下的 package
- 'packages/**'
- 'components/**'
不包括在 test 文件夹下的 package
- '!**/test/**'
因为基于 monorepo 的方式来组织包代码，所以我们需要告诉 pnpm 我们的 repo 工作目录。
这里我们指定了 packages/ 为 monorepo 工作目录，此时我们的 packages 下的每一个文件夹都会被 pnpm 认为是一个独立的项目。
接下来我们去安装所需要的依赖：
pnpm install -D typescript vue esbuild minimist -w
注意，这里 -w 意为 --workspace-root ，表示我们将依赖安装在顶层目录，所以包可以共享到这些依赖。
同时 minimist 是 node-optimist 的核心解析模块，它的主要作为即为解析执行 Node 脚本时的环境变量。
填充构建
接下来我们就来填充构建部分逻辑。
更改 package.json
首先，让我们切换到项目跟目录下对于整个 repo 的 pacakge.json 进行改造。 {
"name": "@vue",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
"dev": "node ./scripts/dev.js reactivity -f global"
}
,
"keywords": [],
"author": "",
"license": "ISC",
"devDependencies": {

"esbuild": "^0.14.27",
"typescript": "^4.6.2",
"vue": "^3.2.31"
}
}
首先我们将包名称修改为作用域，@vue 表示该包是一个组织包。
其次，我们修改 scripts 脚本。表示当运行 pnpm run dev 时会执行 ./scripts/dev.js 同时传入一个 reactivity 参数以及 -f 的 global 环境变量。
更改项目内 package.json
接下来我们需要更改每个 repop 内的 package.json（以下简称 pck） 。这里我们以 reactivity 模块为例，share 我就不重复讲解了。 {
"name": "@vue/reactive",
"version": "1.0.0",
"description": "",
"main": "index.js",
"buildOptions": {
"name": "VueReactivity",
"formats": [
"esm-bundler",
"esm-browser",
"cjs",
"global"
]
}
,
"keywords": [],
"author": "",
"license": "ISC"
}
首先，我们将 reactivity 包中的名称改为作用域名 @vue/reactive 。
其次我们为 pck 中添加了一些自定义配置，分别为：
buildOptions.name 该选项表示打包生成 IIFE 时，该模块挂载在全局下的变量名。
buildOptions.formats 该选项表示该模块打包时候需要输出的模块规范。
填充scripts/dev.js
之后，让我们切换到 scripts/dev.js 来实现打包逻辑：
// scripts/dev.js
const {
build
}
= require('esbuild');
const {
resolve
}
= require('path');
const argv = require('minimist')(process.argv.slice(2));
// 获取参数 minimist
const target = argv['_'];
const format = argv['f'];

```
const pkg = require(resolve(__dirname, '../packages/reactivity/package.json'));
const outputFormat = format.startsWith('global')
? 'iife'
: format.startsWith('cjs')
? 'cjs'
: 'esm';
// 打包输出文件
const outfile = resolve(
__dirname,
`../packages/$ {
target
}
/dist/$ {
target
}
.$ {
outputFormat
}
.js`
);
// 调用ESbuild的NodeApi执行打包
build( {
entryPoints: [resolve(__dirname, `../packages/$ {
target
}
/src/index.ts`)],
outfile,
bundle: true, // 将所有依赖打包进入
sourcemap: true, // 是否需要sourceMap
format: outputFormat, // 输出文件格式 IIFE、CJS、ESM
globalName: pkg.buildOptions?.name, // 打包后全局注册的变量名 IIFE下生效
platform: outputFormat === 'cjs' ? 'node' : 'browser', // 平台
watch: true, // 表示检测文件变动重新打包
}
);
```

脚本中的已经进行了详细的注释，这里我稍微在啰嗦一些。

其次整个流程看来像是这样，首先当我们运行 npm run dev 时，相当于执行了 node ./scripts/dev.js reactivity -f global。

所以在执行对应 dev.js 时，我们通过 minimist 获得对应的环境变量 target 和 format 表示我们本次打包分别需要打包的 package 和模式，当然你也可以通过 process.argv 自己截取。

之后我们通过判断如果传入的 -f 为 global 时将它变成 iife 模式，执行 esbuild 的 Node Api 进行打包对应的模块。

需要注意的是，ESbuild 默认支持 typescript 所以不需要任何额外处理。

当然，我们此时并没有在每个包中创建对应的入口文件。让我们分别创建两个packages/reactivity/src/index.ts以及packages/share/src/index.ts作为入口文件。

此时，当你运行 npm run dev 时，会发现会生成打包后的js文件：

image.png

写在环境结尾的话

至此，针对于一个简易版 Vuejs 的项目构建流程我们已经初步实现了。如果有兴趣深入了解这个完整流程的同学可以自行查看对应 源码。

当然这种根据环境变量进行动态打包的思想，我在之前的React-Webpack5-TypeScript打造工程化多页面应用中详细讲解过这一思路，有兴趣的同学可以自行查阅。

其实关于构建思路我大可不必在这里展开，直接讲述响应式部分代码即可。但是这一流程在我的日常工作中的确帮助过我在多页面应用业务上进行了项目构建优化。

所以我觉得还是有必要拿出来和大家稍微聊一聊这一过程，希望大家在以后业务中遇到该类场景下可以结合 Vuejs 的构建思路来设计你的项目构建流程。

响应式原理

上边我们对于构建稍稍花费了一些篇幅，接下来终于我们要步入正题进行响应式原理部分了。

首先，在开始之前我会稍微强调一些。文章中的代码并不是一比一对照源码来实现响应式原理，但是实现思想以及实现过程是和源码没有出入的。

这是因为源码中拥有非常多的条件分支判断和错误处理，同时源码中也考虑了数组、Set、Map之类的数据结构。

这里，我们仅仅先考虑基础的对象，至于其他数据类型我会在之后的文章中详细和大家一一道来。

同时我也会在每个步骤的结尾贴出对应的源代码地址，提供给大家参照源码进行对比阅读。

开始之前

在我们开始响应式原理之前，我想和大家稍微阐述下对应背景。因为可能有部分同学对应 Vue3 中的源码并不是很了解。

在 VueJs 中的存在一个核心的 Api Effect ，这个 Api 在 Vue 3.2 版本之后暴露给了开发者去调用，在3.2之前都是 Vuejs 内部方法并不提供给开发者使用。

简单来说我们所有模版（组件）最终都会被 effect 包裹 ，当数据发生变化时 Effect 会重新执行，所以 vuejs 中的响应式原理可以说是基于 effect 来实现的 。

当然这里你仅仅需要了解，最终组件是会编译成为一个个 effect ，当响应式数据改变时会触发 effect 函数重新执行从而更新渲染页面即可。

之后我们也会详细介绍 effect 和 响应式是如何关联到一起的。

基础目录结构

首先我们来创建一些基础的目录结构：

reactivity/src/index.ts 用于统一引入导出各个模块

reactivity/src/reactivity.ts 用于维护 reactive 相关 Api。

reactivity/src/effect.ts 用户维护 effect 相关 Api。

这一步我们首先在 reactivity 中新建对应的文件：

image.png

reactive 基础逻辑处理

接下来我们首先进入相关的 reactive.ts 中去。

思路梳理

关于 Vuejs 是如何实现数据响应式，简单来说它内部利用了 Proxy Api 进行了访问/设置数据时进行了劫持。

对于数据访问时，需要进行依赖收集。记录当前数据中依赖了哪些 Effect ，当进行数据修改时候同样会进行触发更新，重新执行当前数据依赖的 Effect。简单来说，这就是所谓的响应式原理。

关于 Effect 你可以暂时的将它理解成为一个函数，当数据改变函数（Effect）重新执行从而函数执行导致页面重新渲染。

Target 实现目标

在开始书写代码之前，我们先来看看它的用法。我们先来看看 reactive 方法究竟是如何搭配 effect 进行页面的更新：

不太了解 Effect 和响应式数据的同学可以将这段代码放在浏览器下执行试试看。
首先我们使用 reactive Api 创建了一个响应式数据 reactiveData 。
之后，我们创建了一个 effect，它会接受一个 fn 作为参数 。这个 effect 内部的逻辑非常简单：它将 id 为 app 元素的内容置为 reactiveData.name 的值。
注意，这个 effect 传入的 fn 中依赖了响应式数据 reactiveData 的 name 属性，这一步通常成为依赖收集。
当 effect 被创建时，fn 会被立即执行所以 app 元素会渲染对应的 19Qingfeng 。
当 0.5s 后 timer 达到时间，我们修改了 reactiveData 响应式数据的 name 属性，此时会触发改属性依赖的 effct 重新执行，这一步同样通常被称为触发更新。
所以页面上看起来的结果就是首先渲染出 19Qingfeng 在 0.5s 后由于响应式数据的改变导致 effect 重新执行所以修改了 app 的 innerHTML 导致页面重新渲染。
这就是一个非常简单且典型的响应式数据 Demo ，之后我们会一步一步基于结果来逆推实现这个逻辑。
基础 Reactive 方法实现
接下来我们先来实现一个基础版的 Reactive 方法，具体使用 API 你可以参照 这里。
上边我们提到过 VueJs 中针对于响应式数据本质上就是基于 Proxy & Reflect 对于数据的劫持，那么自然我们会想到这样的代码：

```
// reactivity/src/reactivity.ts
export function isPlainObj(value: any): value is object {
return typeof value === 'object' && value !== null;
}
const reactive = (obj) => {
// 传入非对象
if (!isPlainObj(obj)) {
return obj;
}
// 声明响应式数据
const proxy = new Proxy(obj, {
get() {
// dosomething
}
,
set() {
// dosomething
```

```
}
}
);
return proxy;
}
;
```

上边的代码非常简单，我们创建了一个 reactive 对象，它接受传入一个 Object 类型的对象。我们会对于函数传入的 obj 进行校验，如果传入的是 object 类型那么会直接返回。

接下来，我们会根据传入的对象 obj 创建一个 proxy 代理对象。并且会在该代理对象上针对于 get 陷阱（访问对象属性时）以及 set （修改代理对象的值时）进行劫持从而实现一系列逻辑。

依赖收集

之前我们提到过针对于 reactive 的响应式数据会在触发 get 陷阱时会进行依赖收集。

这里你可以简单将依赖收集理解为记录当前数据被哪些Effect使用到，之后我们会一步一步来实现它。

```
// reactivity/src/reactivity.ts
export function isPlainObj(value: any): value is object {
return typeof value === 'object' && value !== null;
}
const reactive = (obj) => {
// 传入非对象
if (!isPlainObj(obj)) {
return obj;
}
// 声明响应式数据
const proxy = new Proxy(obj, {
get(target, key, receiver) {
// 依赖收集方法 track
track(target, 'get', key);
// 调用 Reflect Api 获得原始的数据 你可以将它简单理解成为 target[key]
let result = Reflect.get(target, key, receiver);
// 依赖为对象 递归进行reactive处理
if (isPlainObj(result)) {
return reactive(result);
}
// 配合Reflect解决当访问get属性递归依赖this的问题
return result;
}
,
set() {
// dosomething
}
}
);
return proxy;
}
;
```

上边我们填充了在 Proxy 中的 get 陷阱的逻辑：

当访问响应式对象 proxy 中的属性时，首先会针对于对应的属性进行依赖收集。主要依靠的是 track 方法。
之后如果访问该响应式对象 key 对应的 value 仍为对象时，会再次递归调用 reactive 方法进行处理。
需要注意的是递归进行 reactive 时是一层懒处理，换句话说只有访问时才会递归处理并不是在初始化时就会针对于传入的 obj 进行递归处理。
当然这里的依赖收集主要依靠的就是 track 方法，我们会在稍后详解实现这个方法。
依赖收集
接下来我们来看看 set 陷阱中的逻辑，当触发对于 proxy 对象的属性修改时会触发 set 陷阱从而进行触发对应 Effect 的执行。
我们来一起看看对应的 set 陷阱中的逻辑：

```
// reactivity/src/reactivity.ts
export function isPlainObj(value: any): value is object {
return typeof value === 'object' && value !== null;
}
const reactive = (obj) => {
// 传入非对象
if (!isPlainObj(obj)) {
return obj;
}
// 声明响应式数据
const proxy = new Proxy(obj, {
get(target, key, receiver) {
// 依赖收集方法 track
track(target, 'get', key);
// 调用 Reflect Api 获得原始的数据 你可以将它简单理解成为 target[key]
let result = Reflect.get(target, key, receiver);
// 依赖为对象 递归进行reactive处理
if (isPlainObj(result)) {
return reactive(result);
}
// 配合Reflect解决当访问get属性递归依赖this的问题
return result;
}
,
// 当进行设置时进行触发更新
set(target, key, value, receiver) {
const oldValue = target[key];
// 配合Reflect解决当访问get属性递归依赖this的问题
const result = Reflect.set(target, key, value, receiver);
// 如果两次变化的值相同 那么不会触发更新
if (value !== oldValue) {
// 触发更新
trigger(target, 'set', key, value, oldValue);
}
return result;
}
,
```

```
}
);
return proxy;
}
;
```

同样，我们在上边填充了对应 set 陷阱之中的逻辑，当设置响应式对象时会触发对应的 set 陷阱。我们会在 set 陷阱中触发对应的 trigger 逻辑进行触发更新：将依赖的 effect 重新执行。

关于为什么我们在使用 Proxy 时需要配合 Refelct ，我在这篇文章有详细讲解。感兴趣的朋友可以查看这里 [为什么Proxy一定要配合Reflect使用？]。

上边我们完成了 reactive.ts 文件的基础逻辑，遗留了两个核心方法 track & trigger 方法。在实现着两个方法之前，我们先来一起看看 effect 是如何被实现的。

effect 文件

effect 基础使用

让我们把视野切到 effcet.ts 中，我们稍微来回忆一下 effect Api 的用法：

```
const {
reactive,
effect
}
= Vue
const obj = {
name: '19Qingfeng'
}
// 创建响应式数据
const reactiveData = reactive(obj)
// 创建effect依赖响应式数据
effect(() => {
app.innerHTML = reactiveData.name
}
)
```

effect 基础原理

上边我们看到，effect Api 有以下特点：

effect 接受一个函数作为入参。

当调用effect(fn) 时，内部的函数会直接被调用一次。

其次，当 effect 中的依赖的响应式数据发生改变时。我们期望 effect 会重新执行，比如这里的 effect 依赖了 reactiveData.name 上的值。

接下来我们先来一起实现一个简单的 Effect Api：

```
function effect(fn) {
// 调用Effect创建一个的Effect实例
const _effect = new ReactiveEffect(fn);
// 调用Effect时Effect内部的函数会默认先执行一次
_effect.run();
// 创建effect函数的返回值：_effect.run() 方法（同时绑定方法中的this为_effect实例对象）
const runner = _effect.run.bind(_effect);
// 返回的runner函数挂载对应的_effect对象
runner.effect = _effect;
return runner;
```

}
这里我们创建了一个基础的 effect Api，可以看到它接受一个函数 fn 作为参数。
当我们运行 effect 时，会创建一个 const _effect = new ReactiveEffect(fn);
对象。
同时我们会调用 _effect.run() 这个实例方法立即执行传入的 fn ，之所以需要立即执行传入的 fn 我们在上边提到过：当代码执行到 effect(fn) 时，实际上会立即执行 fn 函数。
我们调用的 _effect.run() 实际内部也会执行 fn ，我们稍微回忆下上边的 Demo 当代码执行 effect(fn) 时候相当于执行了：

```
// ...
effect(() => {
app.innerHTML = reactiveData.name
}
)
```

会立即执行传入的 fn 也就是 () => {

```
app.innerHTML = reactiveData.name
}
```

会修改 app 节点中的内容。
同时，我们之前提到过因为 reactiveData 是一个 proxy 代理对象，当我们访问它的属性时实际上会触发它的 get 陷阱。

```
// effect.ts
export let activeEffect;
export function effect(fn) {
// 调用Effect创建一个的Effect实例
const _effect = new ReactiveEffect(fn);
// 调用Effect时Effect内部的函数会默认先执行一次
_effect.run();
// 创建effect函数的返回值：_effect.run() 方法（同时绑定方法中的this为_effect实例对象）
const runner = _effect.run.bind(_effect);
// 返回的runner函数挂载对应的_effect对象
runner.effect = _effect;
return runner;
}
/**
* Reactive Effect
*/
export class ReactiveEffect {
private fn: Function;
constructor(fn) {
this.fn = fn;
}
run() {
try {
activeEffect = this;
// run 方法很简单 就是执行传入的fn
return this.fn();
}
finally {
```

```
activeEffect = undefined
}
}
}
```

这是一个非常简单的 ReactiveEffect 实现，它的内部非常简单就是简单的记录了传入的 fn，同时拥有一个 run 实例方法当调用 run 方法时会执行记录的 fn 函数。

同时，我们在模块内部声明了一个 activeEffect 的变量。当 我们调用运行 effect(fn) 时，实际上它会经历以下步骤：

首先用户代码中调用 effect(fn)

VueJs 内部会执行 effect 函数，同时创建一个 _effect 实例对象。立即调用 _effect.run() 实例方法。

重点就在所谓的 _effect.run() 方法中。

首先，当调用 _effect.run() 方法时，我们会执行 activeEffect = this 将声明的 activeEffect 变成当前对应的 _effect 实例对象。

同时，run() 方法接下来会调用传入的 fn() 函数。

当 fn() 执行时，如果传入的 fn() 函数存在 reactive() 包裹的响应式数据，那么实际上是会进入对应的 get 陷阱中。

当进入响应式数据的 get 陷阱中时，不要忘记我们声明全局的 activeEffect 变量，我们可以在对应响应式数据的 get 陷阱中拿到对应 activeEffect (也就是创建的 _effect) 变量。

接下来我们需要做的很简单:

在响应式数据的 get 陷阱中记录该数据依赖到的全局 activeEffect 对象(_effect)（依赖收集）也就是我们之前遗留的 track 方法。

同时:

当改变响应式数据时，我们仅仅需要找出当前对应的数据依赖的 _effect ，修改数据同时重新调用 _effect.run() 相当于重新执行了 effect（fn）中的 fn。那么此时不就是相当于修改数据页面自动更新吗？这一步就被称为依赖收集，也就是我们之前遗留的 trigger 方法。

track & trigger 方法

让我们会回到之前遗留的 track 和 trigger 逻辑中，接下来我们就尝试去实现它。

这里我们将在 effect.ts 中来实现这两个方法，将它导出提供给 reactive.ts 中使用。

思路梳理

上边我们提到过，核心思路是当代码执行到 effect(fn) 时内部会调用对应的 fn 函数执行。当 fn 执行时会触发 fn 中依赖的响应式数据的 get ，当 get 触发时我们记录到对应 声明的 (activeEffect) _effect 对象和对应的响应式数据的关联即可。

当响应式数据改变时，我们取出关联的 _effect 对象，重新调用 _effect.run() 重新执行 effect(fn) 传入的 fn 函数即可。

看到这里，一些同学已经反应过来了。我们有一份记录对应 activeEffect(_effect) 和 对应的响应式数据的表，于是我们自然而然的想到使用一个 WeakMap 来存储这份关系。

之所以使用 WeakMap 来存储，第一个原因自然是我们需要存储的 key 值是非字符串类型这显然只有 map 可以。其次就是 WeakMap 的 key 并不会影响垃圾回收机制。

创建映射表

上边我们分析过，我们需要一份全局的映射表来维护 _effect 实例和依赖的响应式数据的关联:

于是我们自然想到通过一个 WeakMap 对象来维护映射关系，那么如何设计这个 WeakMap 对象呢？这里我就不卖关子了。

我们再来回忆下上述的 Demo ：

```
// ...
const {
reactive,
```

```
effect
}
= Vue
const obj = {
name: '19Qingfeng'
}
// 创建响应式数据
const reactiveData = reactive(obj)
// 创建effect依赖响应式数据
effect(() => {
app.innerHTML = reactiveData.name
}
)
// 上述Demo的基础上增加了一个effect依赖逻辑
effect(() => {
app2.innerHTML = reactiveData.name
}
)
```

首先针对于响应式数据 reactiveData 它是一个对象，上述代码中的 effect 中依赖了 reactiveData 对象上的 name 属性。

所以，我们仅仅需要关联当前响应式对象中的 name 属性和对应 effect 即可。

同时，针对于同一个响应式对象的属性比如这里的 name 属性被多个 effect 依赖。自然我们可以想到一份响应式数据的属性可以和多个 effect 依赖。

根据上述的分析最终 Vuejs 中针对于这份映射表设计出来了这样的结构：

当一个 effect 中依赖对应的响应式数据时，比如上述 Demo ：

我们创建的全局的 WeakMap 首先会将响应式对象的原始对象（未代理前的对象）作为 key ，value 为一个 Map 对象。

同时 effect 内部使用了上述对象的某个属性，那么此时 WeakMap 对象的该对象的值（我们刚才创建的 Map ）。我们会在这个 Map 对象中设置 key 为使用到的属性，value 为一个 Set 对象。

为什么对应属性的值为一个 Set ，这非常简单。因为该属性可能会被多个 effect 依赖到。所以它的值为一个 Set 对象，当该属性被某个 effect 依赖到时，会将对应 _effect 实例对象添加进入 Set 中。

也许有部分同学乍一看对于这份映射表仍然比较模糊，没关系接下来我会用代码来描述这一过程。你可以结合代码和这段文字进行一起理解。

track 实现

接下来我们来看看 track 方法的实现：

```
// *用于存储响应式数据和Effect的关系Hash表
const targetMap = new WeakMap();
/**
* 依赖收集函数 当触发响应式数据的Getter时会进入track函数
* @param target 访问的原对象
* @param type 表示本次track从哪里来
* @param key 访问响应式对象的key
*/
export function track(target, type, key) {
// 当前没有激活的全局Effect 响应式数据没有关联的effect 不用收集依赖
if (!activeEffect) {
```

```
return;
}
// 查找是否存在对象
let depsMap = targetMap.get(target);
if (!depsMap) {
targetMap.set(target, (depsMap = new Map()));
}
// 查找是否存在对应key对应的 Set effect
let deps = depsMap.get(key);
if (!deps) {
depsMap.set(key, (deps = new Set()));
}
// 其实Set本身可以去重 这里判断下会性能优化点
const shouldTrack = !deps.has(activeEffect) && activeEffect;
if (shouldTrack) {
// *收集依赖，将 effect 进入对应WeakMap中对应的target中对应的keys
deps.add(activeEffect);
}
}
```

我们一行一行分析上边的 track 方法，这个方法我们之前提到过。它是在 reactive.ts中对于某个响应式属性进行依赖收集（触发proxy的 get 陷阱）时触发的，忘记了的小伙伴可以翻回去重新看下。

首先，它会判断当前 activeEffect 是否存在，所谓 actvieEffect 也就是当前是否存在 effect 。换句话说，比如这样：

```
// ...
app.innerHTML = reactiveData.name
```

那么我们有必要进行依赖收集吗，虽然 reactiveData 是一个响应式数据这不假，但是我们并没有在模板上使用它。它并不存在任何关联的 effect ，所以完全没有必要进行依赖收集。

而在这种情况下：

```
effect(() => {
app.innerHTML = reactiveData.name
}
)
```

只有我们在 effect(fn) 中，当 fn 中使用到了对应的响应式数据。简单来说也就是 activeEffect 存在值得时候，对于响应式数据的依赖收集才有意义。

其次，接下来会去全局的 targetMap 中寻找是否已经存在对应响应式数据的原始对象 -> depsMap 。如果该对象首次被收集，那么我们需要在 targetMap 中设置 key 为 target ，value 为一个新的Map。

```
// 查找是否存在对象
let depsMap = targetMap.get(target);
if (!depsMap) {
// 不存在则创建一个Map作为value，将target作为key放入depsMap中
targetMap.set(target, (depsMap = new Map()));
}
```

同时我们会继续去上一步返回的 deps ，此时的 deps 是一个 Map 。它的内部会记录改对象中被进行依赖收集的属性。

我们回去寻找 name 属性是否存在，显然它是第一次进行依赖收集。所以会进行：

```
// 查找是否存在对应key对应的 Set effect
```

```
let deps = depsMap.get(key);
if (!deps) {
// 同样，不存在则创建set放入
depsMap.set(key, (deps = new Set()));
}
```

此时，比如上方的 Demo 中，当代码执行到 effect 中的 fn 碰到响应式数据的 get 陷阱时，触发 track 函数。

我们会为全局的 targetMap 对象中首先设置 key 为 obj （reactiveData的原始对象），value 为一个 Map 。

其次，我们会为该创建的 Map 中再次进行设置 key 为该响应式对象需要被收集的属性，也就是我们在 effect 中访问到该对象的 name ，value 为一个 Set 集合。

接下里 Set 中存放什么其实很简单，我们仅需在对应 Set 中记录当前正在运行的 effct 实例对象，也就是 activeEffct 就可以达到对应的依赖收集效果。

此时，targetMap 中就会存放对应的对象和关联的 effect 了。

trigger 实现

当然，上述我们已经通过对应的 track 方法收集了相关响应式数据和对应它依赖的 effect 。

那么接下来如果当改变响应式数据时（触发 set 陷阱时）自然我们仅仅需要找到对应记录的 effect 对象，调用它的 effect.run() 重新执行不就可以让页面跟随数据改变而改变了吗。

我们来一起看看 trigger 方法：

```
// ... effect.ts
/**
* 触发更新函数
* @param target 触发更新的源对象
* @param type 类型
* @param key 触发更新的源对象key
* @param value 触发更新的源对象key改变的value
* @param oldValue 触发更新的源对象原始的value
*/
export function trigger(target, type, key, value, oldValue) {
// 简单来说 每次触发的时 我拿出对应的Effect去执行 就会触发页面更新
const depsMap = targetMap.get(target);
if (!depsMap) {
return;
}
let effects = depsMap.get(key);
if (!effects) {
return;
}
effects = new Set(effects);
effects.forEach((effect) => {
// 当前zheng
if (activeEffect !== effect) {
// 默认数据变化重新调用effect.run()重新执行清空当前Effect依赖重新执行Effect中fn进行重新收集依赖以及视图更新
effect.run();
}
}
);
```

```
}
```

接下来我们在 effect.ts 中来补充对应的 trigger 逻辑，其实 trigger 的逻辑非常简单。每当响应式数据触发 set 陷阱进行修改时，会触发对应的 trigger 函数。

他会接受对应的 5 个 参数，我们在函数的注释中已经标明了对应的参数。

当触发响应式数据的修改时，首先我们回去 targetMap 中寻找 key 为对应原对象的值，自然因为在 track 中我们已经保存了对应的值，所以当然可以拿到一个 Map 对象。

因为该 Map 对象中存在对应 key 为 name 值为该属性依赖的 effect 的 Set 集合，所以我们仅需要依次拿出对应修改的属性，比如我们调用：

```
// ...
const {
reactive,
effect
}
= Vue
const obj = {
name: '19Qingfeng'
}
// 创建响应式数据
const reactiveData = reactive(obj)
// 创建effect依赖响应式数据
effect(() => {
app.innerHTML = reactiveData.name
}
)
// 修改响应式数据 触发set陷阱
reactiveData.name = 'wang.haoyu'
```

当我们调用 reactiveData.name = 'wang.haoyu' 时，我们会一层一层取到 targetMap 中 key 为 obj 的 depsMap（Map） 对象。

再从 depsMap 中拿到 key 为 name 属性的 Set 集合（Set 中保存该响应式对象属性依赖的 effect）。

迭代当前 Set 中的所有 effect 进行 effect.run() 重新执行 effect 对象中记录的 fn 函数。

因为我们在 reactive.ts 中的 set 陷阱中对于数据已经修改之后调用了 trigger 方法，trigger 导致重新执行 effect(fn) 中的 fn，所以自然而然 fn() 重新执行 app.innerHTML 就会变成最新的 wang.haoyu 。

整个响应式核心原理其实一点都不难对吧，核心思想还是文章开头的那句话：对于数据访问时，需要进行依赖收集。记录当前数据中依赖了哪些 Effect ，当进行数据修改时候同样会进行触发更新，重新执行当前数据依赖的 Effect。

阶段总结

其实写到这里已经 8K 多字了，原本打算是和大家过一遍整个 Vue 3.2 中关于 reactivity 的逻辑，包括各种边界情况。

比如文章中的代码其实仅仅只能说是实现了一个乞丐版的响应式原理，其他一些边界情况，比如：

多个 effect 嵌套时的处理。

多次 reactive 调用同一对象，或者对于已经 reactive 包裹的响应式对象。

每次触发更新时，对于前一次依赖收集的清理。

shallow、readonly 情况等等...

这些边界情况其实文章中的代码我都没有考虑，如果后续有小伙伴对这方面感兴趣我会再次开一篇文章去继续这次的代码去实现一个完整的 reactive 方法。

不过，透过现象看本质。VueJs 中所谓主打的数据响应式核心原理即是文章中代码所表现的思想。
我在这个代码地址，也自己实现了一版比较完整的精简版 reactivity 模块，有兴趣的同学可以自行查阅。
当然，你也可以直接参照源代码进行阅读。毕竟 Vue 3 的代码相较于 2 来说其实已经很人性化了。

面试题 16. 简述vue-router 路由钩子函数是什么？执行顺序是什么？

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

路由钩子的执行流程，钩子函数种类有：全局守卫、路由守卫、组件守卫。
完整的导航解析流程：
1 、导航被触发。
2 、在失活的组件里调用 beforeRouterLeave 守卫。
3 、调用全局的 beforeEach 守卫。
4 、在重用的组件调用 beforeRouterUpdate 守卫 （ 2.2+ ） 。
5 、在路由配置里面 beforeEnter 。
6 、解析异步路由组件。
7 、在被激活的组件里调用 beforeRouterEnter 。
8 、调用全局的 beforeResolve 守卫 （ 2.5+ ） 。
9 、导航被确认。
10 、调用全局的 afterEach 钩子。
11 、触发 DOM 更新。
12 、调用 beforeRouterEnter 守卫中传给 next 的回调函数，创建好的组件实例会作为回调函数的参数传入。

面试题 17. 请简述vue-router 动态路由是什么？

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

我们经常需要把某种模式匹配到的所有路由，全都映射到同个组件。例如，我们有一个 User 组件，对于所有 ID 各不相同的用户，都要使用这个组件来渲染。那么，我们可以在 vue-router 的路由路径中使用“ 动态路径参数 ”（ dynamic segment ）来达到这个效果：

```
const User = {
template: "
User", };
const router = new VueRouter({
routes: [
// 动态路径参数 以冒号开头
{ path: "/user/:id", component: User },
],
});
```

面试题 18. 简述vue-router 组件复用导致路由参数失效怎么办?

推荐指数: ★★★ 试题难度: 初级 试题类型: 原理题

试题回答参考思路:

解决方案 :
通 过 watch 监听 路由参数再发请求
watch: {
"router":function(){
this.getData(this.$router.params.xxx)
}
}

面试题 19. Vue生命周期钩子是如何实现的?

推荐指数: ★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

生命周期钩子在内部会被vue维护成一个数组(vue 内部有一个方法mergeOption)和全局的生命周期合并最终转换成数组，当执行到具体流程时会执行钩子(发布订阅模式)，callHook来实现调用。

理解vue中模板编译原理?

1 会将模板变成ast语法树(模板编译原理的核心就是 ast-> 生成代码)
2 对 ast 语 法 树 进 行 优 化 ， 标 记 静 态 节 点 .(vue3 中 模 板 编 译 做 了 哪 些 优 化 patchFlag,blockTree,事件缓存，节点缓存...)
3 代码生成 拼接render函数字符串 + new Function + with;

面试题 20. 如何理解Vue中的模板编译原理?

推荐指数: ★★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

关于Vue编译原理这块的整体逻辑主要分为三步:

第一步将模版字符串转换成element ASTs(解析器)
第二步是对AST进行静态节点标记，主要用来做虚拟DOM的渲染优化(优化器)
第三步是使用element ASTs生成render函数代码字符串(代码生成器)
解析器

{{name}}

上面一个简单 的模版转换成element AST树形结构后是这样的:
{
tag: "div"

type: 1,
staticRoot: false,
static: false,
plain: true,
parent: undefined,
attrsList: [],
attrsMap: {},
children: [
{
tag: "p"
type: 1,
staticRoot: false,
static: false,
plain: true,
parent: {tag: "div", ...},
attrsList: [],
attrsMap: {},
children: [{
type: 2,
text: "{{name}}",
static: false,
expression: "_s(name)"
}]
}
]
}
我们可以看到上面的dom被解析成了解析器，它的原理主要是两部分内容，一部分是截取字符串，一部分是对截取的字符串做解析。

优化器
优化器的目的就是找出那些静态节点并打上标记，而静态节点指的是DOM不需要发生变化的节点，也就是里面都是静态标签和静态文本。

标记静态节点有两个好处：
一、每次重新渲染的时候不需要为静态节点创建新节点，也就是静态节点的解析器不需要重新创建
二、在Virtual DOM中patching的过程可以被跳过
优化器的实现原理主要分两步：
一、用递归的方式将静态节点添加static属性，用来标识是不是静态节点
二、标记所有静态根节点(子节点全是静态节点就是静态根节点)
代码生成器
代码生成器的作用是使用element ASTs生成render函数代码字符串。
使用本文开头举的例子中的模版生成后的AST来生成render后是这样的：

{
render: with(this){return _c('div',[_c('p',[_v(_s(name))])])}
}
格式化后是这样的：

```
with(this){
return _c(
'div',
[
_c(
'p',
[
_v(_s(name))
]
)
]
)
}
```

生成后的代码字符串中看到了有几个函数调用_c、_v、_s。
_c对应的是createElement，它的作用是创建一个元素。
1.第一个参数是一个HTML标签名
2.第二个参数是元素上使用的属性所对应的数据对象，可选项
3.第三个参数是children
_v的意思是创建一个文本节点。
_s是返回参数中的字符串。
代码生成器的总体逻辑其实就是使用element ASTs去递归，然后拼出这样的_c('div',[_c('p',[_v(_s(name))])]) 字符串。

总结

本篇文章我们说了 vue 对模板编译的整体流程分为三个部分：解析器（parser），优化器（optimizer）和代码生成器（code generator）。
解析器（parser）的作用是将 模板字符串 转换成 element ASTs。
优化器（optimizer）的作用是找出那些静态节点和静态根节点并打上标记。
代码生成器（code generator）的作用是使用 element ASTs 生成 render函数代码（generate render function code from element ASTs）。

解析器（parser）的原理是一小段一小段的去截取字符串，然后维护一个 stack 用来保存DOM深度，每截取到一段标签的开始就 push 到 stack 中，当所有字符串都截取完之后也就解析出了一个完整的 AST。

优化器（optimizer）的原理是用递归的方式将所有节点打标记，表示是否是一个 静态节点，然后再次递归一遍把 静态根节点 也标记出来。

代码生成器（code generator）的原理也是通过递归去拼一个函数执行代码的字符串，递归的过程根据不同的节点类型调用不同的生成方法，如果发现是一颗元素节点就拼一个 _c(tagName, data, children) 的函数调用字符串，然后 data 和 children 也是使用 AST 中的属性去拼字符串。

如果 children 中还有 children 则递归去拼。

最后拼出一个完整的 render 函数代码。

面试题 21. 简述vue.mixin的使用场景和原理?

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

Mixin是面向对象程序设计语言中的类，提供了方法的实现。
其他类可以访问mixin类的方法而不必成为其子类
当一段代码非常相似的时候就可以抽离成一个mixin
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，如data、components、methods 、created、computed等等。只要将共用的功能以对象的方式传入mixins选项中，当组件使用 mixins对象时所有mixins对象的选项都将被混入该组件本身的选项中来，这样就可以提高代码的重用性，使你的代码保持干净和易于维护。
使用场景
当存在多个组件中的数据或者功能很相近时，就可以利用mixins将公共部分提取出来，通过mixins封装的函数，组件调用他们是不会改变函数作用域外部的。
mixins和vuex的区别
vuex公共状态管理，在一个组件被引入后，如果该组件改变了vuex里面的数据状态，其他引入vuex数据的组件也会对应修改，所有的vue组件应用的都z是同一份vuex数据。（在js中，有点类似于浅拷贝）
vue引入mixins数据，mixins数据或方法，在每一个组件中都是独立的，互不干扰的，都属于vue组件自身。（在js中，有点类似于深度拷贝）
mixins和组件的区别
组件：在父组件中引入组件，相当于在父组件中给出一片独立的空间供子组件使用，然后根据props来传值，但本质上两者是相对独立的。
Mixins：则是在引入组件之后与组件中的对象和方法进行合并，相当于扩展了父组件的对象与方法，可以理解为形成了一个新的组件。
（装饰器模式）
mixin的使用
定义一个mixin名字为myMixins

```
export default {
data () {
return {
num:1
}
}
,
methods: {
mymixin() {
console.log(this.num);
}
,
}
}
```

在组件中使用

```
import {
myMixins
}
from './myMixins';
```

```
export default {
mixins: [myMixins],
data() {
return {
}
}
,
created() {
//使用mixin可以直接用，但是组件就得传值
this.num++
}
,
}
```

面试题 22. 简述v-for中的key的理解?

推荐指数：★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

需要使用key来给每个节点做一个唯一标识，Diff算法就可以正确的识别此节点。主要是为了高效的更新虚拟DOM

面试题 23. 解释Vue中transition的理解?

推荐指数：★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

1）定义transition时需要设置对应的name，具体语法为：需要动画的内容或者组件或者页面

2）过渡动画主要包含6个class，分别为：

v-enter：定义元素进入过渡的初始状态，在元素插入前生效，插入后一帧删除，
v-enter-active：在元素插入前生效，在动画完成后删除，
v-enter-to：在元素插入后一帧生效，在动画完成后删除，
v-leave：离开过渡的初始状态，在元素离开时生效，下一帧删除
v-leave-active：在离开过渡时生效，在动画完成后删除
v-leave-to：离开过渡结束状态，在离开过渡下一帧生效，在动画完成后删除
⚠️：v会转化为对应的transition的name值

3）当然我们也可以自定义这六个class 可以直接在transition中设置对应的属性为对应的class名称，属性有：enter-class，enter-active-class，enter-to-class，leave-class，leave-active-class，leave-to-class

4）在同时使用过渡和css动画的时候 可以设置type属性来制定vue内部机制监听transitioned或者animationed事件来完成过渡还是动画的监听

5）如果需要设置对应的过渡时间，可以直接设置属性duration，可以直接接收一个数字（单位

为毫秒），也可以接收一个对象{enter:1000,leave:300}

6）也可以设置过渡的钩子函数，具体有：before-enter，enter，after-enter，enter-cancelled，before-leave，leave，after-leave，leave-cancelled

面试题 24. 请简述什么是Vue的自定义指令?

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

自定义指令分为全局指令和组件指令，其中全局指令需要使用directive来进行定义，组件指令需要使用directives来进行定义，具体定义方法同过滤器filter或者其他生命周期，具体使用方法如下：

全局自定义指令 directive(name,{})，其中name表示定义的指令名称（定义指令的时候不需要带v-，但是在调用的时候需要哦带v-），第二个参数是一个对象，对象中包括五个自定义组件的钩子函数，具体包括：

bind函数：只调用一次，指令第一次绑定在元素上调用，即初始化调用一次，
inserted函数：并绑定元素插入父级元素（即new vue中el绑定的元素）时调用（此时父级元素不一定转化为了dom）
update函数：在元素发生更新时就会调用，可以通过比较新旧的值来进行逻辑处理
componentUpdated函数：元素更新完成后触发一次
unbind函数：在元素所在的模板删除的时候就触发一次
钩子函数对应的参数el,binding,vnode,oldnode,具体参数讲解如下：

a、el指令所绑定的元素 可以直接操组dom元素

b、binding一个对象，具体包括以下属性：

1）name：定义的指令名称 不包括v-
2）value：指令的绑定值，如果绑定的是一个计算式，value为对应计算结果
3）oldvalue：指令绑定元素的前一个值，只对update和componentUpdated钩子函数有值
4）expression：指令绑定的原始值 不对值进行任何加工
5）arg：传递给指令的参数
6）modifiers：指令修饰符，如：v-focus.show.async 则接收的modifiers为 {show：true，async：true}
c、vnode：vue编译生成的虚拟dom

d、oldVnode：上一个vnode，只在update和componentUpdated钩子函数中有效

⚠️：如果不需要其他钩子函数，可以直接简写为：directive(“focus”,function(el,binding){})

面试题 25. 请描述Vue的实现原理 ?

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

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。

具体步骤：

第一步：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

第二步：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

第三步：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。
第四步：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。

面试题 26. 简述对于Vue的diff算法理解？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

1）diff算法的作用：用来修改dom的一小段，不会引起dom树的重绘

2）diff算法的实现原理：diff算法将virtual dom的某个节点数据改变后生成的新的vnode与旧节点进行比较，并替换为新的节点，具体过程就是调用patch方法，比较新旧节点，一边比较一边给真实的dom打补丁进行替换

3）具体过程详解：

a、在采用diff算法进行新旧节点进行比较的时候，比较是按照在同级进行比较的，不会进行跨级比较：

b、当数据发生改变的时候，set方法会调用dep.notify通知所有的订阅者watcher，订阅者会调用patch函数给响应的dom进行打补丁，从而更新真实的视图

c、patch函数接受两个参数，第一个是旧节点，第二个是新节点，首先判断两个节点是否值得

比较，值得比较则执行patchVnode函数，不值得比较则直接将旧节点替换为新节点。如果两个节点一样就直接检查对应的子节点，如果子节点不一样就说明整个子节点全部改变不再往下对比直接进行新旧节点的整体替换

d、patchVnode函数：找到真实的dom元素；判断新旧节点是否指向同一个对象，如果是就直接返回；如果新旧节点都有文本节点，那么直接将新的文本节点赋值给dom元素并且更新旧的节点为新的节点；如果旧节点有子节点而新节点没有，则直接删除dom元素中的子节点；如果旧节点没有子节点，新节点有子节点，那么直接将新节点中的子节点更新到dom中；如果两者都有子节点，那么继续调用函数updateChildren

e、updateChildren函数：抽离出新旧节点的所有子节点，并且设置新旧节点的开始指针和结束指针，然后进行两辆比较，从而更新dom（调整顺序或者插入新的内容 结束后删掉多余的内容）

面试题 27. 请简述 Vue组件的通信（父子组件和非父子组件）？

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

父子组件通信
传递参数可以使用props，传递函数可以直接在调用子组件的时候传递自定义事件，并使用$emit来调用，例如：

//父组件

```
export default {
data(){
return {
usermessage:'我是父亲'
}
},
methods:{
myname(name){
console.log('我的名字叫'+name)
}
}
}
```

//子组件

来源：{{userinfo}}
显示我的名字

```
export default {
props:['userinfo'],
```

```
methods:{
getname(){
this.$emit('getinfo','bilibili')
}
}
}
```

面试题 28. 请简述 Vue组件的通信（ 兄弟组件通信 ）?

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

首先建立一个vue实例空白页（js文件）

```
import Vue from 'vue'
export default new Vue()
```

组件a（数据发送方）通过使用 $emit 自定义事件把数据带过去

组件b（数据接收方）使用而通过 $on监听自定义事件的callback接收数据

面试题 29. 请叙述Vue与React、Angular的比较?

推荐指数： ★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

Vue
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:相比较与react而言，同样是操作虚拟dom，就性能而言，vue存在很大的优势。

React
相同点：
React采用特殊的JSX语法，Vue.js在组件开发中也推崇编写.vue特殊文件格式，对文件内容都有一些约定，两者都需要编译后使用；中心思想相同：一切都是组件，组件实例之间可以嵌套；都提供合理的钩子函数，可以让开发者定制化地去处理需求；都不内置列数AJAX，Route

等功能到核心包，而是以插件的方式加载；在组件开发中都支持mixins的特性。

不同点：
React采用的Virtual DOM会对渲染出来的结果做脏检查；Vue.js在模板中提供了指令，过滤器等，可以非常方便，快捷地操作Virtual DOM。

Angular
相同点：
都支持指令：内置指令和自定义指令；都支持过滤器：内置过滤器和自定义过滤器；都支持双向数据绑定；都不支持低端浏览器。

不同点：
AngularJS的学习成本高，比如增加了Dependency Injection特性，而Vue.js本身提供的API都比较简单、直观；在性能上，AngularJS依赖对数据做脏检查，所以Watcher越多越慢；Vue.js使用基于依赖追踪的观察并且使用异步队列更新，所有的数据都是独立触发的

面试题 30. 请简述Vuex的使用？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

vuex 是什么 ？
vuex 是一个专为 Vue 应用程序开发 的状态管理器， 采用集中式 存储管理 应用的所有组件的状态。每 一个 vuex 应用的核心就是 store（仓库）。“store”基本上就是一个容器，它包含着应用中大部分 的状态 (state)。
为什么需要 vuex
由于组件只维护自身的状态(data)，组件创建时或者路由切换时，组件会被初始化，从而导致 data 也 随之销毁。
使用方法
在 main.js 引入 store，注入。只用来读取的状态集中放在 store 中， 改变状态的方式是提交
mutations，这是个同步的事物，异步逻辑应该封装在 action 中。
什么场景下会使用到 vuex 如果是 vue 的小型应用，那么没有必要使用 vuex，这个时候使用 vuex 反而会带来负担。组件之间的 状态传递使用 props、自定义事件来传递即可。 但是如果涉及到 vue 的大型应用 ，那么就需要类似于 vuex 这样的集中管 理状态的状态机来管理所有组件的状态。例如登录状态、加入购物车、音乐播放等，总之只要是开发 vue 的大型应用，都推荐使 用 vuex 来管理所有组件状态
主要包括以下几个模块：
State:定义了应用状态的数据结构，可以在这里设置默认的初始化状态。
Getter:允许组件从Store中获取数据，mapGetters 辅助函数仅仅是将 store 中的
getter 映射到局部计算属性。
Mutation:是唯一更改 store 中状态的方法，且必须是同步函数。
Action:用于提交 mutation，而不是直接变更状态，可以包含任意异步请求。
Module:允许将单一的 Store 拆分更多个 store 且同时保存在单一的状态树中

面试题 31. Vuex 页面刷新数据丢失怎么解决?

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

需要做 vuex 数据持久化 ，一般使 用本地储存的方案 来保存数据，可以自己设计存储方案，也可以使用第三方插件。
推荐使用 vuex-persist（ 脯肉赛斯特 ）插件，它是为 Vuex 持久化储存而生的一个插件。
不需要你手动存取 storage ， 而是直接将状态保存至 cookie 或者 localStorage 中

面试题 32. 请叙述Vue 中使用了哪些设计模式?

推荐指数： ★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

1 工厂模式 – 传入参数即可创建实例 虚拟 DOM 根据参数的不同返回基础标签的 Vnode 和组件 Vnode。
2 单例模式 – 整个程序有且仅有一个实例 vuex 和 vue-router 的插件3注册方法 install 判断如果系统存在实例就直接返回掉。
3 发布-订阅模式。（vue 事件机制）
4 观察者模式。（响应式数据原理）
5 装饰器模式（@装饰器的用法）
6 策略模式，策略模式指对象有某个行为，但是在不同的场景中，该行为有不同的实现方案 – 比如选项的合并策略。

面试题 33. 请简述Vue 的性能优化可以从哪几个方面去思考设计 ？

推荐指数： ★★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

这里只列举针对 Vue 的性能优化，整个项目的性能优化是一个大工程。
对象层级不要过深，否则性能就会差。
不需要响应式的数据不要放在 data 中（可以使用 Object.freeze() 冻结数据）
v-if 和 v-show 区分使用场景
computed 和 watch 区分场景使用
v-for 遍历必须加 key，key最好是id值，且避免同时使用 v-if
大数据列表和表格性能优化 – 虚拟列表 / 虚拟表格
防止内部泄露，组件销毁后把全局变量和时间销毁
图片懒加载
路由懒加载
异步路由
第三方插件的按需加载
适当采用 keep-alive 缓存组件
防抖、节流的运用
服务端渲染 SSR or 预渲染

面试题 34. 简述nextTick 的作用是什么？他的实现原理是什么 ？

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

作用 ： vue 更新 DOM 是异步更新的，数据变化，DOM 的更新不会马上完成， nextTick的回调是在下次 DOM 更新循环结束之后执行的延迟回调 。
实现原理 ： nextTick 主要使用了 宏任务和微任务 。根据执行环境分别尝试采用
Promise：可以将函数延迟到当前函数调用栈最末端
MutationObserver ： 是 H5 新加的一个功能，其功能是监听 DOM 节点的变动，在所有 DOM 变动完成后，执行回调函数setImmediate：用于中断长时间运行的操作，并在浏览器完成其他操作（如事件和显 示更新）后立即运行回调函数
如果以上都不行则采用 setTimeout 把函数延迟到 DOM 更新之后再使用，原因是宏任务消耗大于微任务，优先使用微任务，最后使用消耗最大的宏任务。

面试题 35. keep-alive 使用场景和原理 ？

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

keep-alive 组件是 vue 的内置组件 ，用于 缓存内部组件 实例。这样做的目的在于，keep alive 内部的组件切回时， 不用重新创建 组件实例，而直接使用缓存中的实例，一方面能够避免创建组件带来的开销，另一方面可以保留组件的状态 。
keep-alive 具有 include 和 exclude 属性，通过它们可以控制哪些组件进入缓存。另外它还提供了 max 属性，通过它可以设置最大缓存数，当缓存的实例超过该数时，vue 会移除最久没有使用的组件缓存。
受keep-alive的影响，其内部所有嵌套的组件都具有两个生命周期钩子函数，分别是 activated 和 deactivated，它们分别在组件激活和失活时触发。第一次 activated 触发是在 mounted 之后
在具体的实现上，keep-alive 在内部维护了一个 key 数组和一个缓存对象
// keep-alive 内部的声明周期函数
created () {
this.cache = Object.create(null)
this.keys = []
}
key 数组记录目前缓存的组件 key 值，如果组件没有指定 key 值，则会为其自动生成一个唯一的 key 值 cache 对象以 key 值为键，vnode 为值，用于缓存组件对应的虚拟 DOM
在 keep-alive 的渲染函数中，其基本逻辑是判断当前渲染的 vnode 是否有对应的缓存，如果有，从缓存中读取到对应的组件实例；如果没有则将其缓存。、
当缓存数量超过 max 数值时，keep-alive 会移除掉 key 数组的第一个元素

面试题 36. 简述Vue.set 方法原理 ？

推荐指数： ★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

Vue 响应式原理的同学都知道在两种情况下修改 Vue 是不会触发视图更新的。
在实例创建之后添加新的属性到实例上（给响应式对象新增属性）
直接更改数组下标来修改数组的值。
Vue.set 或者说是 $set 原理如下
因为响应式数据 我们给对象和数组本身新增了 __ob__ 属性，代表的是 Observer 实例。
当给对象新增不存在的属性，首先会把新的属性进行响应式跟踪 然后会触发对象 __ob__
的 dep 收集到的 watcher 去更新，当修改数组索引时我们调用数组本身的 splice 方法去
更新数组。

面试题 37. Vue的组件data为什么必须是一个函数?

推荐指数: ★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

new Vue是一个单例模式，不会有任何的合并操作，所以根实例不必校验data一定是一个函数。 组件的data必须是一个函数，是为了防止两个组件的数据产生污染。 如果都是对象的话，会在合并的时候，指向同一个地址。 而如果是函数的时候，合并的时候调用，会产生两个空间。

面试题 38. 请解释Vue为什么要用虚拟Dom ，详细解释原理 ?

推荐指数: ★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

什么是虚拟DOM?

在Vue.js 2.0版本中引入了 Virtual DOM 的概念，Virtual DOM 其实就是一个以JavaScript对象（VNode节点）作为基础来模拟DOM结构的树形结构，这个树形结构包含了整个DOM结构的信息。简单来说，可以把Virtual DOM理解为一个简单的JS对象，并且最少包含标签名（tag）、属性（attrs）和子元素对象（children）三个属性。不同的框架对这三个属性的命名会有所差别。

虚拟DOM的作用?

虚拟DOM的最终目标是将虚拟节点渲染到视图上。但是如果直接使用虚拟节点覆盖旧节点的话，会有很多不必要的DOM操作。例如，一个ul标签下有很多个li标签，其中只有一个li标签有变化，这种情况下如果使用新的ul去替代旧的ul，会因为这些不必要的DOM操作而造成性能上的浪费。

为了避免不必要的DOM操作，虚拟DOM在虚拟节点映射到视图的过程中，将虚拟节点与上一次渲染视图所使用的旧虚拟节点做对比，找出真正需要更新的节点来进行DOM操作，从而避免操作其他不需要改动的DOM元素。

其实，虚拟DOM在Vue.js中主要做了两件事情:

1、提供与真实DOM节点所对应的虚拟节点VNode

2、将虚拟节点VNode和旧虚拟节点oldVNode进行对比，然后更新视图

具备跨平台优势，由于Virtual DOM 是以JavaScript对象为基础而不依赖真实平台环境，所以使它具有了跨平台的能力，比如说浏览器平台、Weex、Node等。

操作DOM慢，JS运行效率高，可以将DOM对比操作放在JS层，提高效率。因为DOM操作的执行速度远不如JavaScript运算速度快，因此，把大量的DOM操作搬运到JavaScript中，运用patching算法来计算出真正需要更新的节点，最大限度地减少DOM操作，从而显著提高性能。Vritual DOM本质上就是在JS和DOM之间做了一个缓存，JS只操作Virtual DOM，最后把变更写入到真实DOM。

提高渲染性能，Virtual DOM的优势不在于单次的操作，而是在大量、频繁的数据更新下，能够对视图进行合理、高效的更新。

面试题 39. vue通过数据劫持可以精准的探测数据变化，为什么还要进行diff检测差异?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

现代前端框架有两种方式侦测变化,一种是pull一种是push

pull: 其代表为React,我们可以回忆一下React是如何侦测到变化的,我们通常会用setStateAPI显式更新,然后React会进行一层层的Virtual Dom Diff操作找出差异,然后Patch到DOM上,React从一开始就不知道到底是哪发生了变化,只是知道「有变化了」,然后再进行比较暴力的Diff操作查找「哪发生变化了」，另外一个代表就是Angular的脏检查操作。

push: Vue的响应式系统则是push的代表,当Vue程序初始化的时候就会对数据data进行依赖的收集,一但数据发生变化,响应式系统就会立刻得知,因此Vue是一开始就知道是「在哪发生变化了」,但是这又会产生一个问题,如果你熟悉Vue的响应式系统就知道,通常一个绑定一个数据就需要一个Watcher,一但我们的绑定细粒度过高就会产生大量的Watcher,这会带来内存以及依赖追踪的开销,而细粒度过低会无法精准侦测变化,因此Vue的设计是选择中等细粒度的方案,在组件级别进行push侦测的方式,也就是那套响应式系统,通常我们会第一时间侦测到发生变化的组件,然后在组件内部进行Virtual Dom Diff获取更加具体的差异,而Virtual Dom Diff则是pull操作,Vue是push+pull结合的方式进行变化侦测的.

面试题 40. 请说明Vue key的作用及原理 ?

推荐指数：★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

key是虚拟DOM对象的标识，当数据发生变化时，Vue会根据[新数据]生成[新的虚拟DOM]，随后Vue进行[新虚拟DOM]与[旧虚拟DOM]的差异比较
原理（比较规则）：
1.旧虚拟DOM中找到了与新虚拟DOM相同的key:
a.若虚拟DOM中内容没变，直接使用之前的直DOM!
b.若虚拟DOM中内容变了，则生成新的真实DOM，随后替换掉页面中之前的真实DOM。
2.旧虚拟DOM中未找到与新虚拟DOM相同的key:

a.创建新的真实DOM，随后渲染到到页面。

index作为key可能会引发的问题:
1.若对数据进行:逆序添加、逆序删除等破坏顺序操作:
a.会产生没有必要的真实DOM更新==> 界面效果没问题，但效率低。
2.如果结构中还包含输入类的DOM:
a.会产生错误DOM更新==>界面有问题

面试题 41. 简单描述Vue的组件渲染流程 ？

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

1、给组件创建个构造函数，基于Vue。

```
export default function globalApiMixin(Vue){
Vue.options = {}
Vue.mixin = function (options){
this.options = mergeOptions(this.options,options);//合并options
}
Vue.options.components = {};
Vue.options._base = Vue
Vue.component = function (id,definition){
//保证组件的隔离，每个组件都会产生一个新的类，去继承父类
definition = this.options._base.extend(definition);
this.options.components[id] = definition;
}
//给个对象返回类
Vue.extend = function (definition){//extend方法就是返回一个继承于Vue的类
//并且身上应该有父类的所有功能
let Super = this;
let Sub = function VueComponent(options){
this._init(options);
}
//原型继承
Sub.prototype = Object.create(Super.prototype);
Sub.prototype.constructor = Sub;
Sub.options = mergeOptions(Super.options, definition);//只和vue.options合并
return Sub
}
}
```

2、开始生成虚拟节点，对组件进行特殊处理 data.hook = {init(){}}

```
export function createElement(vm, tag, data = {}, ...children) {
if(isReservedTag(tag)){
return vnode(vm, tag, data, data.key, children, undefined);
}else{
const Ctor = vm.$options.components[tag];
return createComponent(vm, tag, data, data.key, undefined, undefined,Ctor);
}
```

```
}
function createComponent(vm, tag, data, key, children, text, Ctor) {

if(isObject(Ctor)){
Ctor = vm.$options._base.extend(Ctor)
}
data.hook = {
init(vnode){
let vm = vnode.componentInstance = new Ctor({_isComponent:true})//new sub
debugger
vm.$mount();
}
}
return                vnode(vm,`vue-component-${tag}`,data,key,undefined,undefined,
{Ctor,children});
}
export function createTextElement(vm, text) {
return vnode(vm, undefined, undefined, undefined, undefined, text);
}
function vnode(vm, tag, data, key, children, text,componentOptions) {
return { vm, tag, data, key, children, text, componentOptions };
}

function isReservedTag(str){ //判断是否是组件
let strList = 'a,div,span,p,ul,li';
return strList.includes(str);
}
3、生成dom元素，如果当前虚拟节点上有hook.init属性，说明是组件
function createComponent(vnode){
let i = vnode.data;
if((i = i.hook) && (i = i.init)){
i(vnode);//调用init方法
}
if (vnode.componentInstance) {
//有属性说明子组件new完毕了，并且组件的真实dom挂载到了vnode。componentInstance
return true;
}
}
function createElm(vnode){
debugger
let {vm,tag,data,children,text} = vnode;
if(typeof tag === 'string'){
//判断是否是组件
if( createComponent(vnode)){
//返回组件对应的真实节点
console.log(vnode.componentInstance.$el);
return vnode.componentInstance.$el
}
```

```
vnode.el = document.createElement(tag);
if(children.length){
children.forEach(child=>{
vnode.el.appendChild(createElm(child));
})
}
}else{
vnode.el = document.createTextNode(text);
}
return vnode.el;
}
```

4、对组件进行new 组件().$mount()=>vm.$el; 将组件的$el插入到父容器中 （父组件）

```
Vue.prototype.$mount = function (el) {
debugger
const vm = this;
const options = vm.$options;
el = document.querySelector(el);
vm.$el = el;
//将模板转化成对应的渲染函数=》虚拟函数概念 vnode =》diff算法更新虚拟 dom =》产生真实节点，更新
if (!options.render) {
//没有render 用template，目前没有render
let template = options.template;
if (!template && el) {
//用户也没有传入template，就取页面上的el
template = el.outerHTML;
}
let render = compileToFunction(template);
//options.render 就是渲染函数
options.render = render;
}
debugger
mountComponent(vm, el); //组件的挂载流程
};
```

面试题 42. Vie3.0 Proxy 相比 defineProperty 的优势在哪里?

推荐指数： ★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

Vue3.x 改用 Proxy 替代 Object.defineProperty
原因在于 Object.defineProperty 本身存在的一 些问题 ：
Object.defineProperty 只能劫持对象属性的 getter 和 setter 方法。
Object.definedProperty 不支持数组(可以监听数组,不过数组方法无法监听自己重写)，更准确的说是不支持数组的各种 API(所以 Vue 重写了数组方法。
而相比 Object.defineProperty，Proxy 的优点在于：
Proxy 是直接代理劫持整个对象。

Proxy 可以直接监听对象和数组的变化，并且有多达 13 种拦截方法。
目前，Object.definedProperty 唯一比 Proxy 好的一点就是兼容性，不过 Proxy 新标准也受到浏览器厂商重点持续的性能优化当中

面试题 43. 请说明Vue的filter的理解与用法?

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

1）全局过滤器必须写在vue实例创建之前

```
Vue.filter('testfilter', function (value,text) { // 返回处理后的值
return value+text
})
```

2）局部写法：在组件实例对象里挂载。

```
filters: {
changemsg:(val,text)\=>{ return val + text
}
}
```

3）使用方式：只能使用在{{}}和：v-bind中，定义时第一个参数固定为预处理的数，后面的数为调用时传入的参数，调用时参数第一个对应定义时第二个参数，依次往后类推

```
{{test|changemsg(4567)}}
```

```
//多个过滤器也可以串行使用
{{name|filter1|filter2|filter3}}
```

4）vue-cli项目中注册多个全局过滤器写法：

```
//1.创建一个单独的文件定义并暴露函数对象
const filter1 = function (val) {
return val + '--1'
}
const filter2 = function (val) {
return val + '--2'
}
const filter3 = function (val) {
return val + '--3'
}

export default {
filter1,
filter2,
filter3
```

```
}

//2.导入main.js(在vue实例之前)
import filters from './filter/filter.js'

//3.循环注册过滤器
Object.keys(filters).forEach(key=>{
Vue.filter(key,filters[key])
})
```

面试题 44. Vue首屏白屏如何解决?

推荐指数: ★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

1）路由懒加载
2）vue-cli开启打包压缩 和后台配合 gzip访问
3）进行cdn加速
4）开启vue服务渲染模式
5）用webpack的externals属性把不需要打包的库文件分离出去，减少打包后文件的大小
6）在生产环境中删除掉不必要的console.log

```
plugins: [
new webpack.optimize.UglifyJsPlugin({ //添加-删除console.log
compress: {
warnings: false,
drop_debugger: true,
drop_console: true
},
sourceMap: true
})
```

7）开启nginx的gzip ,在nginx.conf配置文件中配置

```
http { //在 http中配置如下代码,
gzip on;
gzip_disable "msie6";
gzip_vary on;
gzip_proxied any;
gzip_comp_level 8; #压缩级别
gzip_buffers 16 8k;
#gzip_http_version 1.1;
gzip_min_length 100; #不压缩临界值
gzip_types text/plain application/javascript application/x-javascript text/css
application/xml text/javascript application/x-httpd-php image/jpeg image/gif
image/png;
```

}
8）添加loading效果，给用户一种进度感受

面试题 45. 请简述Vue中的v-cloak的理解？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

使用 v-cloak 指令设置样式，这些样式会在 Vue 实例编译结束时，从绑定的 HTML 元素上被移除。

一般用于解决网页闪屏的问题，在对一个的标签中使用v-cloak，然后在样式中设置[v-cloak]样式,[v-cloak]需写在 link 引入的css中，或者写一个内联css样式，写在import引入的css中不起作用

面试题 46. 简述Vue单页面和传统的多页面区别?

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

单页面应用（SPA）
通俗一点说就是指只有一个主页面的应用，浏览器一开始要加载所有必须的 html, js, css。所有的页面内容都包含在这个所谓的主页面中。但在写的时候，还是会分开写（页面片段），然后在交互的时候由路由程序动态载入，单页面的页面跳转，仅刷新局部资源。多应用于pc端。

多页面（MPA）
指一个应用中有多个页面，页面跳转时是整页刷新

单页面的优点：
用户体验好，快，内容的改变不需要重新加载整个页面，基于这一点spa对服务器压力较小；前后端分离；页面效果会比较炫酷（比如切换页面内容时的专场动画）。

单页面缺点：
不利于seo；导航不可用，如果一定要导航需要自行实现前进、后退。（由于是单页面不能用浏览器的前进后退功能，所以需要自己建立堆栈管理）；初次加载时耗时多；页面复杂度提高很多

面试题 47. 请描述Vue常用的修饰符？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

.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

面试题 48. 请简述Vue更新数组时触发视图更新的方法?

推荐指数：★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

push()；pop()；shift()；unshift()；splice()；sort()；reverse()

面试题 49. 解释 Vue route和router的区别?

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

$router
router是VueRouter的一个对象，通过Vue.use(VueRouter)和VueRouter构造函数得到一个router的实例对象，这个对象中是一个全局的对象，他包含了所有的路由包含了许多关键的对象和属性,常见的有：
1）push：向 history 栈添加一个新的记录，当我们点击浏览器的返回按钮时可以看到之前的页面

```
// 字符串
this.$router.push('home')
// 对象
this.$router.push({ path: 'home' })
// 命名的路由
this.$router.push({ name: 'user', params: { userId: 123 }})
// 带查询参数，变成 /register?plan=123
this.$router.push({ path: 'register', query: { plan: '123' }})
```

2）go：页面路由跳转 前进或者后退

```
// 页面路由跳转 前进或者后退
this.$router.go(-1) // 后退
```

3）replace：push方法会向 history 栈添加一个新的记录，而replace方法是替换当前的页面，不会向 history 栈添加一个新的记录

$route
$route对象表示当前的路由信息，包含了当前URL解析得到的信息。包含当前的路径、参数、query对象等。
1）$route.path：字符串，对应当前路由的路径，总是解析为绝对路径，如 "/foo/bar"。
2）$route.params：一个 key/value 对象，包含了 动态片段 和 全匹配片段，如果没有路由参数，就是一个空对象。
3）$route.query：一个 key/value 对象，表示 URL 查询参数。例如，对于路径 /foo?user=1，则有 $route.query.user == 1，如果没有查询参数，则是个空对象。
4）$route.hash：当前路由的 hash 值（不带#），如果没有 hash 值，则为空字符串。
5.$route.fullPath：完成解析后的 URL，包含查询参数和 hash 的完整路径。

6$route.matched：数组，包含当前匹配的路径中所包含的所有片段所对应的配置参数对象。
7.$route.name：当前路径名字
8.$route.meta：路由元信息

面试题 50. 简述如何使用Vue-router实现懒加载的方式?

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

vue异步组件
vue异步组件技术 ==== 异步加载
vue-router配置路由 ，使用vue的异步组件技术 ，可以实现按需加载 。但是,这种情况下一个组件生成一个js文件

```
/* vue异步组件技术 */
{
path: '/home',
name: 'home',
component: resolve => require(['@/components/home'],resolve)
},{
path: '/index',
name: 'Index',
component: resolve => require(['@/components/index'],resolve)
},{
path: '/about',
name: 'about',
component: resolve => require(['@/components/about'],resolve)
}
```

es提案的import()
路由懒加载(使用import)

```
// 下面2行代码，没有指定webpackChunkName，每个组件打包成一个js文件。
/* const Home = () => import('@/components/home')
const Index = () => import('@/components/index')
const About = () => import('@/components/about') */
// 下面2行代码，指定了相同的webpackChunkName，会合并打包成一个js文件。把组件按组分块
const Home = () => import(/* webpackChunkName: 'ImportFuncDemo' */ '@/components/home')
const Index = () => import(/* webpackChunkName: 'ImportFuncDemo' */ '@/components/index')
const About = () => import(/* webpackChunkName: 'ImportFuncDemo' */ '@/components/about')


{
path: '/about',
component: About
```

```
}, {
path: '/index',
component: Index
}, {
path: '/home',
component: Home
}
```

webpack的require,ensure()

vue-router配置路由，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。

```
/* 组件懒加载方案三: webpack提供的require.ensure() */
{
path: '/home',
name: 'home',
component: r => require.ensure([], () => r(require('@/components/home')), 'demo')
}, {
path: '/index',
name: 'Index',
component: r => require.ensure([], () => r(require('@/components/index')), 'demo')
}, {
path: '/about',
name: 'about',
component: r => require.ensure([], () => r(require('@/components/about')), 'demo-01')
}
```

面试题 51. Vue中delete和Vue.delete删除数组的区别？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

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

面试题 52. 简述Vue路由跳转和location.href的区别?

推荐指数：★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

使用location.href='/url'来跳转，简单方便，但是刷新了页面；
使用路由方式跳转，无刷新页面，静态跳转

面试题 53. 请说明Vue的solt的用法?

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

在子组件内使用特殊的＜slot>元素就可以为这个子组件开启一个slot（插槽），在父组件模板里，插入在子组件标签内的所有内容将替代子组件的＜slot> 标签及它的内容。

简单说来就是：在子组件内部用 标签占位，当在父组件中使用子组件的时候，我们可以在子组件中插入内容，而这些插入的内容则会替换 标签的位置。

当然：单个solt的时候可以不对solt进行命名，如果存在多个 则一个可以不命名，其他必须命名，在调用的时候指定名称的对应替换slot，没有指定的则直接默认无名称的solt

面试题 54. 说明对于Vue $emit 、$on 、$once 、$off理解?

推荐指数：★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

$emit
触发当前实例上的自定义事件（并将附加参数都传给监听器回调）

$on
监听实例上自定义事件并调用回调函数，监听emit触发的事件

$once
监听一个自定义事件，但是只触发一次，在第一次触发之后移除监听器。

$off
用来移除自定义事件监听器。如果没有提供参数，则移除所有的事件监听器；如果只提供了事件，则移除该事件所有的监听器；如果同时提供了事件与回调，则只移除这个回调的监听器。

这四个方法的实现原理是：通过对vue实例挂载，然后分别使用对象存储数组对应的函数事件，其中emit通过循环查找存储的数组中对应的函数进行调用，once只匹配一次就就结束，on是将对应的函数存储到数组中，off是删除数组中指定的元素或者所有的元素事件。具体可以参考文章：VUEemit实现

面试题 55. 请说明Vue中$root、$refs、$parent的使用？

推荐指数：★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

$root
可以用来获取vue的根实例，比如在简单的项目中将公共数据放再vue根实例上(可以理解为一个全局 store),因此可以代替vuex实现状态管理；

$refs
在子组件上使用ref特性后，this.属性可以直接访问该子组件。可以代替事件emit 和$on 的作用。

使用方式是通过ref特性为这个子组件赋予一个ID引用，再通过this.$refs.testId获取指定元素。

注意：$refs只会在组件渲染完成之后生效，并且它们不是响应式的。这仅作为一个用于直接操作子组件的“逃生舱”——你应该避免在模板或计算属性中访问$refs。

$parent
$parent属性可以用来从一个子组件访问父组件的实例，可以替代将数据以 prop 的方式传入子组件的方式；当变更父级组件的数据的时候，容易造成调试和理解难度增加；

面试题 56. Vue路由的使用以及如何解决在router-link上添加点击事件无效的情况 ？

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

使用@click.native。原因:router-link会阻止click事件,.native指直接监听一个原生事件

面试题 57. 简述v-el 作用是什么以及Vue的el属性和$mount优先级?

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

v-el 提供一个在页面上已存在的 DOM 元素作为 Vue 实例的挂载目标。可以是 CSS 选择器，也可以是一个 HTMLElement 实例。

```
new Vue({
router,
store,
el: '#app',
render: h => h(App)
}).$mount('#div')
/*当出现上面的情况就需要对el和$mount优先级进行判断，
el的优先级是高于$mount的，因此以el挂载节点为准*/
```

面试题 58. 简述vue-loader是什么?使用它的用途有哪些?

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

解析.vue文件的一个加载器，跟template/js/style转换成js模块。
用途：js可以写es6、style样式可以scss或less、template可以加jade等

面试题 59. 请说出vue.cli项目中src目录每个文件夹和文件的用法?

推荐指数： ★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

1. assets 文件夹是放静态资源
2. components 文件夹是放全局组件的
3. router 文件夹是定义路由相关的配置
4. store 文件夹是管理 vuex管理数据的位置 模块化开发 全局getters
5. views 视图 所有页面 路由级别的组件
6. App.vue 入口页面 根组件
7. main.js 入口文件 加载组件 初始化等

面试题 60. 简述Vue的普通Slot以及作用域Slot的区别？

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

普通插槽
普通插槽是渲染后做替换的工作。父组件渲染完毕后，替换子组件的内容。

作用域插槽
作用域插槽可以拿到子组件里面的属性。在子组件中传入属性然后渲染。

// 有name的属于具名插槽，没有name属于匿名插槽

xxxx

xxxx

普通插槽渲染的位置是在它的父组件里面，而不是在子组件里面
作用域插槽渲染是在子组件里面
1.插槽slot

在渲染父组件的时候，会将插槽中的先渲染。

创建组件虚拟节点时，会将组件的儿子的虚拟节点保存起来。当初始化组件时,通过插槽属性将儿 子进行分类 {a:[vnode],b[vnode]}渲染组件时会拿对应的slot属性的节点进行替换操作。（插槽的作用域为父组件，插槽中HTML模板显示不显示、以及怎样显示由父组件来决定）

有name的父组件通过html模板上的slot属性关联具名插槽。没有slot属性的html模板默认关联匿名插槽。

2.作用域插槽slot-scope

作用域插槽在解析的时候，不会作为组件的孩子节点。会解析成函数，当子组件渲染时，会调用此函数进行渲染。

或者可以说成作用域插槽是子组件可以在slot标签上绑定属性值，在父组件可以拿到子组件的数据，通过子组件绑定数据传递给父组件。（插槽的作用域为子组件）

子组件：

父组件：

面试题 61. 请简述vue2和vue3的区别？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

1、双向数据绑定原理不同
vue2：vue2的双向数据绑定是利用ES5的一个API：Object.definePropert() 对数据进行劫持，结合发布订阅模式的方式来实现的。
vue3：vue3中使用了ES6的Proxy API对数据代理。相比vue2.x，使用proxy的优势如下：
defineProperty只能监听某个属性，不能对全对象监听
可以省去for in，闭包等内容来提升效率(直接绑定整个对象即可)
可以监听数组，不用再去单独的对数组做特异性操作vue3.x可以检测到数组内部数据的变化。
2、是否支持碎片
vue2：vue2不支持碎片。
vue3：vue3支持碎片（Fragments），就是说可以拥有多个根节点。
3、API类型不同
vue2：vue2使用选项类型api，选项型api在代码里分割了不同的属性：data,computed,methods等。
vue3：vue3使用组合式api，新的合成型api能让我们使用方法来分割，相比于旧的api使用属性来分组，这样代码会更加简便和整洁。
4、定义数据变量和方法不同
vue2：vue2是把数据放入data中，在vue2中定义数据变量是data() {
}
，创建的方法要在methods: {
}
中。
vue3：，vue3就需要使用一个新的setup()方法，此方法在组件初始化构造的时候触发。使用以下三个步骤来建立反应性数据：
从vue引入reactive；
使用reactive() 方法来声明数据为响应性数据；
使用setup()方法来返回我们的响应性数据，从而template可以获取这些响应性数据。
5、生命周期钩子函数不同

vue2：vue2中的生命周期：
beforeCreate 组件创建之前
created 组件创建之后
beforeMount 组价挂载到页面之前执行
mounted 组件挂载到页面之后执行
beforeUpdate 组件更新之前
updated 组件更新之后
vue3：vue3中的生命周期：
setup 开始创建组件前
onBeforeMount 组价挂载到页面之前执行
onMounted 组件挂载到页面之后执行
onBeforeUpdate 组件更新之前
onUpdated 组件更新之后
而且vue3.x 生命周期在调用前需要先进行引入。除了这些钩子函数外，vue3.x还增加了onRenderTracked 和onRenderTriggered函数。
6、父子传参不同
vue2：父传子，用props,子传父用事件 Emitting Events。在vue2中，会调用this$emit然后传入事件名和对象。
vue3：父传子，用props,子传父用事件 Emitting Events。在vue3中的setup()中的第二个参数content对象中就有emit，那么我们只要在setup()接收第二个参数中使用分解对象法取出emit就可以在setup方法中随意使用了。
7、指令与插槽不同
vue2：vue2中使用slot可以直接使用slot；v-for与v-if在vue2中优先级高的是v-for指令，而且不建议一起使用。
vue3：vue3中必须使用v-slot的形式；vue3中v-for与v-if,只会把当前v-if当做v-for中的一个判断语句，不会相互冲突；vue3中移除keyCode作为v-on的修饰符，当然也不支持config.keyCodes；vue3中移除v-on.native修饰符；vue3中移除过滤器filter。
8、main.js文件不同
vue2：vue2中我们可以使用pototype(原型)的形式去进行操作，引入的是构造函数。
vue3：vue3中需要使用结构的形式进行操作，引入的是工厂函数；vue3中app组件中可以没有根标签。
关键词：
组合式api；proxy；支持碎片；组合式api；composition；生命周期；

面试题 62. SPA首屏加载速度慢的怎么解决?

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

请求优化：CDN 将第三方的类库放到 CDN 上，能够大幅度减少生产环境中的项目体积，另外 CDN 能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。

缓存：将长时间不会改变的第三方类库或者静态资源设置为强缓存，将 max-age 设置为一个非常长的时间，再将访问路径加上哈希达到哈希值变了以后保证获取到最新资源，好的缓存策略有助于减轻服务器的压力，并且显著的提升用户的体验

gzip：开启 gzip 压缩，通常开启 gzip 压缩能够有效的缩小传输资源的大小。

http2：如果系统首屏同一时间需要加载的静态资源非常多，但是浏览器对同域名的 tcp 连接数量是有限制的(chrome 为 6 个)超过规定数量的 tcp 连接，则必须要等到之前的请求收到响应后才能继续发送，而 http2 则可以在多个 tcp 连接中并发多个请求没有限制，在一些网络较差的环境开启 http2 性能提升尤为明显。

懒加载：当 url 匹配到相应的路径时，通过 import 动态加载页面组件，这样首屏的代码量会大幅减少，webpack 会把动态加载的页面组件分离成单独的一个 chunk.js 文件

预渲染：由于浏览器在渲染出页面之前，需要先加载和解析相应的 html、css 和 js 文件，为此会有一段白屏的时间，可以添加loading，或者骨架屏幕尽可能的减少白屏对用户的影响体积优化
合理使用第三方库：对于一些第三方 ui 框架、类库，尽量使用按需加载，减少打包体积

使用可视化工具分析打包后的模块体积：webpack-bundle- analyzer 这个插件在每次打包后能够更加直观的分析打包后模块的体积，再对其中比较大的模块进行优化

提高代码使用率：利用代码分割，将脚本中无需立即调用的代码在代码构建时转变为异步加载的过程

封装：构建良好的项目架构，按照项目需求就行全局组件，插件，过滤器，指令，utils 等做一些公共封装，可以有效减少我们的代码量，而且更容易维护资源优化

图片懒加载：使用图片懒加载可以优化同一时间减少 http 请求开销，避免显示图片导致的画面抖动，提高用户体验使用 svg 图标：相对于用一张图片来表示图标，svg 拥有更好的图片质量，体积更小，并且不需要开启额外的 http 请求

压缩图片：可以使用 image-webpack-loader，在用户肉眼分辨不清的情况下一定程度上压缩图片

面试题 63. 简述Vue初始化过程中（new Vue(options)）都做了什么?

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

处理组件配置项；初始化根组件时进行了选项合并操作，将全局配置合并到根组件的局部配置上；初始化每个子组件时做了一些性能优化，将组件配置对象上的一些深层次属性放到 vm.$options 选项中，以提高代码的执行效率；
初始化组件实例的关系属性，比如 parent、parent、parent、children、root、root、root、refs 等
处理自定义事件
调用 beforeCreate 钩子函数
初始化组件的 inject 配置项，得到 ret[key] = val 形式的配置对象，然后对该配置对象进行响应式处理，并代理每个 key 到 vm 实例上
数据响应式，处理 props、methods、data、computed、watch 等选项
解析组件配置项上的 provide 对象，将其挂载到 vm._provided 属性上
调用 created 钩子函数
如果发现配置项上有 el 选项，则自动调用 $mount 方法，也就是说有了 el 选项，就不需要再

手动调用 $mount 方法，反之，没提供 el 选项则必须调用 $mount
接下来则进入挂载阶段

```
// core/instance/init.js
export function initMixin (Vue: Class) {
Vue.prototype._init = function (options?: Object) {
const vm: Component = this
vm._uid = uid++

// 如果是Vue的实例，则不需要被observe
vm._isVue = true

if (options && options._isComponent) {
// optimize internal component instantiation
// since dynamic options merging is pretty slow, and none of the
// internal component options needs special treatment.
initInternalComponent(vm, options)
} else {
vm.$options = mergeOptions(
resolveConstructorOptions(vm.constructor),
options || {},
vm
)
}

if (process.env.NODE_ENV !== 'production') {
initProxy(vm)
} else {
vm._renderProxy = vm
}

vm._self = vm

initLifecycle(vm)
initEvents(vm)
callHook(vm, 'beforeCreate')
initInjections(vm) // resolve injections before data/props

initState(vm)
initProvide(vm) // resolve provide after data/props
callHook(vm, 'created')

if (vm.$options.el) {
vm.$mount(vm.$options.el)
}
}
}
```

面试题 64. Vue3.0 里为什么要用 Proxy API替代 defineProperty API?

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

1.defineProperty API 的局限性最大原因是它只能针对单例属性做监听。
Vue2.x中的响应式实现正是基于defineProperty中的descriptor，对 data 中的属性做了遍历 + 递归，为每个属性设置了 getter、setter。这也就是为什么 Vue 只能对 data 中预定义过的属性做出响应的原因。
2.Proxy API的监听是针对一个对象的，那么对这个对象的所有操作会进入监听操作， 这就完全可以代理所有属性，将会带来很大的性能提升和更优的代码。
Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。
3.响应式是惰性的。
在 Vue.js 2.x 中，对于一个深层属性嵌套的对象，要劫持它内部深层次的变化，就需要递归遍历这个对象，执行 Object.defineProperty 把每一层对象数据都变成响应式的，这无疑会有很大的性能消耗。
在 Vue.js 3.0 中，使用 Proxy API 并不能监听到对象内部深层次的属性变化，因此它的处理方式是在 getter 中去递归响应式，这样的好处是真正访问到的内部属性才会变成响应式，简单的可以说是按需实现响应式，减少性能消耗

面试题 65. 简述Proxy 与 Object.defineProperty 优劣对比?

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

1.Proxy 可以直接监听对象而非属性；
2.Proxy 可以直接监听数组的变化；
3.Proxy 有多达 13 种拦截方法,不限于 apply、ownKeys、deleteProperty、has 等等是 Object.defineProperty 不具备的；
4.Proxy 返回的是一个新对象,我们可以只操作新的对象达到目的,而 Object.defineProperty 只能遍历对象属性直接修改；
5.Proxy 作为新标准将受到浏览器厂商重点持续的性能优化，也就是传说中的新标准的性能红利；
6.Object.defineProperty 的优势如下:
兼容性好，支持 IE9，而 Proxy 的存在浏览器兼容性问题,而且无法用 polyfill 磨平，因此 Vue 的作者才声明需要等到下个大版本(3.0)才能用 Proxy 重写。

面试题 66. Vue中data的属性可以和methods中方法同名吗，为什么?

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

可以同名，methods的方法名会被data的属性覆盖；调试台也会出现报错信息，但是不影响执行；
原因：源码定义的initState函数内部执行的顺序：props>methods>data>computed>watch

```
//initState部分源码
export function initState (vm: Component) {
vm._watchers = []
const opts = vm.$options
if (opts.props) initProps(vm, opts.props)
if (opts.methods) initMethods(vm, opts.methods)
if (opts.data) {
initData(vm)
} else {
observe(vm._data = {}, true /* asRootData */)
}
if (opts.computed) initComputed(vm, opts.computed)
if (opts.watch && opts.watch !== nativeWatch) {
initWatch(vm, opts.watch)
}
}
```

面试题 67. Vue中created与mounted区别？

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

在created阶段，实例已经被初始化，但是还没有挂载至el上，所以我们无法获取到对应的节点，但是此时我们是可以获取到vue中data与methods中的数据的；
在mounted阶段，vue的template成功挂载在$el中，此时一个完整的页面已经能够显示在浏览器中，所以在这个阶段，可以调用节点了；

```
//以下为测试vue部分生命函数，便于理解
beforeCreate(){ //创建前
console.log('beforecreate:',document.getElementById('first'))//null
console.log('data:',this.text);//undefined
this.sayHello();//error:not a function
},
created(){ //创建后
console.log('create:',document.getElementById('first'))//null
console.log('data:',this.text);//this.text
this.sayHello();//this.sayHello()
},
beforeMount(){ //挂载前
console.log('beforeMount:',document.getElementById('first'))//null
console.log('data:',this.text);//this.text
this.sayHello();//this.sayHello()
},
mounted(){ //挂载后
console.log('mounted:',document.getElementById('first'))//
```

```
console.log('data:',this.text);//this.text
this.sayHello();//this.sayHello()
}
```

面试题 68. 简述Vue中watch用法详解？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

在vue中，使用watch来监听数据的变化；
1.监听的数据后面可以写成对象形式，包含handler方法，immediate和deep。
2.immediate表示在watch中首次绑定的时候，是否执行handler，值为true则表示在watch中声明的时候，就立即执行handler方法，值为false，则和一般使用watch一样，在数据发生变化的时候才执行handler。
3.当需要监听一个对象的改变时，普通的watch方法无法监听到对象内部属性的改变，只有data中的数据才能够监听到变化，此时就需要deep属性对对象进行深度监听。

```
watch: {
name: {
handler(newName, oldName) {

},
deep: true,
immediate: true
}
}
```

面试题 69. 简述为什么Vue采用异步渲染？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

vue是组件级更新，当前组件里的数据变了，它就会去更新这个组件。当数据更改一次组件就要重新渲染一次，性能不高，为了防止数据一更新就更新组件，所以做了个异步更新渲染。（核心的方法就是nextTick）

源码实现原理：
当数据变化后会调用notify方法，将watcher遍历，调用update方法通知watcher进行更新，这时候watcher并不会立即去执行，在update中会调用queueWatcher方法将watcher放到了一个队列里，在queueWatcher会根据watcher的进行去重，多个属性依赖一个watcher，如果队列中没有该watcher就会将该watcher添加到队列中，然后通过nextTick异步执行flushSchedulerQueue方法刷新watcher队列。flushSchedulerQueue中开始会触发一个before的方法，其实就是beforeUpdate，然后watcher.run() 才开始真正执行watcher，执行完页面就渲染完成啦，更新完成后会调用updated钩子。

面试题 70. 解释Vue.set 改变数组和对象中的属性？

推荐指数：★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

在一个组件实例中，只有在data里初始化的数据才是响应的，Vue不能检测到对象属性的添加或删除，没有在data里声明的属性不是响应的,所以数据改变了但是不会在页面渲染；
解决办法：
使用 Vue.set(object, key, value) 方法将响应属性添加到嵌套的对象上

面试题 71. 请解释 vm.$set(obj, key, val)？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

由于 Vue 无法探测对象新增属性或者通过索引为数组新增一个元素，所以这才有了 vm.set，它是 Vue.set 的别名。 vm.set，它是 Vue.set 的别名。 vm.set，它是Vue.set的别名。vm.set 用于向响应式对象添加一个新的 property，并确保这个新的 property 同样是响应式的，并触发视图更新。

为对象添加一个新的响应式数据：调用 defineReactive 方法为对象增加响应式数据，然后执行 dep.notify 进行依赖通知，更新视图
为数组添加一个新的响应式数据：通过 splice 方法实现

面试题 72. 简述什么情况下使用 Vuex?

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

如果应用够简单，最好不要使用 Vuex，一个简单的 store 模式即可；
需要构建一个中大型单页应用时，使用Vuex能更好地在组件外部管理状态；

面试题 73. 简述Vuex和单纯的全局对象有什么区别?

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

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

面试题 74. 为什么 Vuex 的 mutation 中不能做异步操作?

推荐指数： ★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

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

Mutation 必须是同步函数一条重要的原则就是要记住 mutation 必须是同步函数。为什么？请参考下面的例子：
mutations: { someMutation (state) { api.callAsyncMethod(() =>
{ state.count++ }) }}
现在想象，我们正在 debug 一个 app 并且观察 devtool 中的 mutation 日志。
每一条 mutation 被记录，devtools 都需要捕捉到前一状态和后一状态的快照。
然而，在上面的例子中 mutation 中的异步函数中的回调让这不可能完成：因为当 mutation 触发的时候，回调函数还没有被调用，devtools 不知道什么时候回调函数实际上被调用——实质上任何在回调函数中进行的状态的改变都是不可追踪的。
在组件中提交 Mutation 你可以在组件中使用 this.$store.commit('xxx') 提交mutation，或者使用 mapMutations 辅助函数将组件中的 methods 映射为store.commit 调用（需要在根节点注入 store）。

```
import { mapMutations } from 'vuex'export default {
// ... methods: {
...mapMutations([
'increment', // 将 `this.increment()` 映射为
`this.$store.commit('increment')`
// `mapMutations` 也支持载荷：
'incrementBy' // 将 `this.incrementBy(amount)` 映射为
`this.$store.commit('incrementBy',
amount)`
]),
...mapMutations({
add: 'increment' // 将 `this.add()` 映射为`this.$store.commit('increment')`
})
}
}
```

面试题 75. 请解释axios 是什么，其特点和常用语法？

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

axios 是什么?

Axios 是一个基于 promise 的 HTTP 库，可以用在浏览器和 node.js 中。前端最流行的 ajax 请求库，
react/vue 官方都推荐使用 axios 发 ajax 请求
特点:

基于 promise 的异步 ajax 请求库，支持promise所有的API
浏览器端/node 端都可以使用，浏览器中创建XMLHttpRequests
支持请求 / 响应拦截器
支持请求取消
可以转换请求数据和响应数据，并对响应回来的内容自动转换成 JSON类型的数据
批量发送多个请求
安全性更高，客户端支持防御 XSRF，就是让你的每个请求都带一个从cookie中拿到的key，根据浏览器同源策略，假冒的网站是拿不到你cookie中得key的，这样，后台就可以轻松辨别出这个请求是否是用户在假冒网站上的误导输入，从而采取正确的策略。

常用语法：
axios(config): 通用/最本质的发任意类型请求的方式
axios(url[, config]): 可以只指定 url 发 get 请求
axios.request(config): 等同于 axios(config)
axios.get(url[, config]): 发 get 请求
axios.delete(url[, config]): 发 delete 请求
axios.post(url[, data, config]): 发 post 请求
axios.put(url[, data, config]): 发 put 请求
axios.defaults.xxx: 请求的默认全局配置
axios.interceptors.request.use(): 添加请求拦截器
axios.interceptors.response.use(): 添加响应拦截器
axios.create([config]): 创建一个新的 axios(它没有下面的功能)
axios.Cancel(): 用于创建取消请求的错误对象
axios.CancelToken(): 用于创建取消请求的 token 对象
axios.isCancel(): 是否是一个取消请求的错误
axios.all(promises): 用于批量执行多个异步请求
axios.spread(): 用来指定接收所有成功数据的回调函数的方法

面试题 76. Vue 3.0 所采用的 Composition Api 与 Vue 2.x使用的Options Api 有什么区别?

推荐指数： ★★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

Options Api

包含一个描述组件选项（data、methods、props等）的对象 options；
API开发复杂组件，同一个功能逻辑的代码被拆分到不同选项 ；
使用mixin重用公用代码，也有问题：命名冲突，数据来源不清晰；

Composition Api

vue3 新增的一组 api，它是基于函数的 api，可以更灵活的组织组件的逻辑。
解决options api在大型项目中，options api不好拆分和重用的问题

面试题 77. 请问什么是SSR ，它主要解决什么问题?

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

Server-Side Rendering 我们称其为SSR，意为服务端渲染指由服务侧完成页面的 HTML 结构拼接的页面处理技术，发送到浏览器，然后为其绑定状态与事件，成为完全可交互页面的过程；

解决了以下两个问题：

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

复杂度：整个项目的复杂度
性能会受到影响
服务器负载变大，相对于前后端分离务器只需要提供静态资源来说，服务器负载更大，所以要慎重使用

面试题 78. Vue.js中的v-bind指令有何作用？如何使用？

推荐指数： ★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

Vue.js中的v-bind指令用于绑定HTML元素的属性或特性。使用方式是在HTML元素上添加v-bind属性，并指定需要绑定的属性或特性。
解析：这是一个基础问题，主要考察面试者Vue.js中v-bind指令的熟悉程度，并且是否能够简单、清晰地描述出来

面试题 79. Vue.js中的事件修饰符有哪些？如何使用？

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

Vue.js中的事件修饰符包括stop、prevent、capture、self、once、passive等。使用方式是在v-on指令后添加相应的修饰符。
解析：这是一个基础问题，主要考察面试者Vue.js中事件修饰符的熟悉程度，并且是否能够简单、清晰地描述出来

面试题 80. Vue.js中的路由导航钩子有哪些？如何使用？

推荐指数： ★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

三种，一是全局导航钩子：router.beforeEach(to,from,next)，作用：跳转前进行判断拦截。
第二种：组件内的钩子
第三种：单独路由独享组件
beforeRouteEnter、afterEnter、beforeRouterUpdate、beforeRouteLeave
参数：有to（去的那个路由）、from（离开的路由）、next（一定要用这个函数才能去到下一个路由，如果不用就拦截）最常用就这几种
Vue的路由实现: hash模式和history模式(Vue的两种状态)

hash——即地址栏URL的#符号，特点： 通过window.onhashchange的监听，匹配不同的url路径，进行解析，加载不同的组件，然后动态的渲染出区域内的html内容，不会被包含在HTTP请求中,对后端完全没有影响
HashHistory有两个方法：
HashHistory.push（）是将路由添加到浏览器访问历史的栈顶
hashHistory.replace( ) 是替换掉当前栈顶的路由
因为hash发生变化的url都会被浏览器历史访问栈记录下来，这样一来，尽管浏览器没有请求服务器，但是页面状态和url一一关联起来的，浏览器还是可以进行前进后退的

history —— 利用了HTML5 History Interface中新增的pushState（）和replaceState（）方法。这两个方式应用于浏览器的历史记录栈，提供了对历史记录的修改功能。history模式不怕页面的前进和后腿，就怕刷新，当刷新时，如果服务器没有相应的响应或者资源，就会刷出404，而hash模式不会。

面试题 81. 简述vue如何监听键盘事件中的按键?

推荐指数： ★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

在我们的项目经常需要监听一些键盘事件来触发程序的执行，而Vue中允许在监听的时候添加关键修饰符：
对于一些常用键，还提供了按键别名：
全部的按键别名：
.enter
.tab
.delete (捕获“删除”和“退格”键)
.esc
.space
.up
.down
.left
.right
修饰键：
.ctrl
.alt
.shift
.meta
与按键别名不同的是，修饰键和 keyup 事件一起用时，事件引发时必须按下正常的按键。换一种说法：如果要引发 keyup.ctrl，必须按下 ctrl 时释放其他的按键；单单释放 ctrl 不会引发

事件

对于elementUI的input，我们需要在后面加上.native，因为elementUI对input进行了封装，原生的事件不起作用。

“昵称”

面试题 82. 简述如何在vue-cli生产环境使用全局常量？

推荐指数：★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

第一步，在 static 下新建 config.js：
第二步，在 config.js 里面设置全局变量：
第三步，在 index.html 里面引入：
第四步，在其他 .js 文件中即可使用：
第五步，打包后修改：通过 `npm run build` 命令打包后，此 config.js 文件会被打包到`dist/static`文件夹下，此时如果需要修改 `PUBLIC_IP`，打开`config.js`即可修改，无需重新打包

面试题 83. 简述父组件给子组件props传参，子组件接收的6种方法？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

1. data中 变量 = this.props里面的数据
2. watch监听 赋值
3. mounted 渲染完成后调用一个函数 进行赋值
4. vuex
5. computed 计算属性，用法和watch类似，computed是同步，watch可以异步
6. 父组件v-if 触发渲染和销毁，子组件触发传参

面试题 84. 如何解决Vuex页面刷新数据丢失？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

F5页面刷新，页面销毁之前的资源，重新请求，因此写在生命周期里的vuex数据是重新初始化，无法获取的，这也就是为什么会打印出空的原因。
解决思路1：
使用Localstorage sessionStorage 或cookie
实际使用时当vuex值变化时，F5刷新页面，vuex数据重置为初始状态，所以还是要用到localStorage，
解决方法2：

插件vuex-persistedstate
vuex-persistedstate默认持久化所有state，可以指定需要持久化的state

面试题 85. 简述MINI UI是什么？怎么使用？说出至少三个组件的使用方法？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

//基于vue框架的ui组件库很多，这里主要简单阐述一下组件的使用方法。
基于vue前端的组件库。使用npm安装，然后import样式和js；
vue.use(miniUi)全局引入。在单个组件局部引入；
import {Toast} form ‘mini-ui’;
组件一: Toast（‘登录成功’）
组件二： mint-header;
组件三：mint-swiper

面试题 86. 自定义指定(v-check/v-focus)的方法有哪些？有哪些钩子函数？还有哪些钩子函数参数？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

全局定义指令：在vue对象的directive方法里面有两个参数，一个是指令名称，另外一个是函数。组件内定义指令：directives钩子函数：bind（绑定事件触发）、inserted（节点插入的时候触发）、updata（组件内相关更新）
钩子函数的参数：el、binding

面试题 87. 请说明Vue Watch和Dep的关系？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

watcher中实例化了dep并向dep.subs中添加了订阅者，dep通过notify遍历了dep.subs通知每个watcher更新。
依赖收集

initState时，对computed属性初始化时，触发computed watcher依赖收集
initState时，对侦听属性初始化是，触发user watcher依赖收集
render（）的过程， 触发render watcher依赖收集
re-render时，vm.render( )再次执行，会溢出所有subs中的watcher的订阅，重新赋值
派发更新
组件对响应的数据进行了修改，触发setter的逻辑
调用dep.notify( )
遍历所有的subs(Watcher实例),调用每一个watcher的update方法

面试题 88. 简述Vue中同时发送多个请求怎么操作？

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

创建两个Promise，在Promise中使用axios
调用Promise.all([p1,p2]).then(res =>{ }).catch(err => { }) 方法
举例说明：

```
getInfo(){
//创建promise,在promise中调用axios，then里使用resolve回调，catch里使用reject回调
var p1 = new Promie((resolve,reject) => {
this.$axios.get(httpUrl.getUser).then(res => {
resolve(res);
}).catch(err =>{
reject (err);
})
})

var p2 = new Promie((resolve,reject) => {
this.$axios.get(httpUrl.getCompany).then(res => {
resolve(res);
}).catch(err =>{
reject (err);
})
})
//调用Promise.add().then(res => {})
Promise.all([p1,p2]).then(res => {
console.log(res);
})
}
resolve(res);
```

面试题 89. Vue不使用v-model的时候怎么监听数据变化?

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

可以使用watch监听数据的变化

面试题 90. 怎么定义vue-router的动态路由以及如何获取传过来的动态参数?

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

在router目录下的index.js文件中，对path属性加上/：id
使用router对象的params id
Vue中,如何用watch去监听router变化

当路由发生变化的时候，在watch中写具体的业务逻辑

```
let vm = new Vue({
el:"#app",
data:{},
router,
watch:{
$router(to,from){
console.log(to.path);
}
}
})
```

面试题 91. 请简述Vue 开发框架的的优点？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

Vue 是一个构建数据驱动的 Web 界面的渐进式框架。

Vue 的目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件。核心是一个响应的数据绑定系统。

关于 Vue 的优点，主要有响应式编程、组件化开发、虚拟 DOM

响应式编程

这里的响应式不是 @media 媒体查询中的响应式布局，而是指 Vue 会自动对页面中某些数据的变化做出响应。这也就是 Vue 最大的优点，通过 MVVM 思想实现数据的双向绑定，让开发者不用再操作 DOM 对象，有更多的时间去思考业务逻辑。

组件化开发

Vue 通过组件，把一个单页应用中的各种模块拆分到一个一个单独的组件（component）中，我们只要先在父级应用中写好各种组件标签（占坑），并且在组件标签中写好要传入组件的参数（就像给函数传入参数一样，这个参数叫做组件的属性），然后再分别写好各种组件的实现（填坑），然后整个应用就算做完了。

组件化开发的优点：提高开发效率、方便重复使用、简化调试步骤、提升整个项目的可维护性、便于协同开发。

虚拟 DOM

在传统开发中，用 JQuery 或者原生的 JavaScript DOM 操作函数对 DOM 进行频繁操作的时候，浏览器要不停的渲染新的 DOM 树，导致在性能上面的开销特别的高。

而 Virtual DOM 则是虚拟 DOM 的英文，简单来说，他就是一种可以预先通过 JavaScript 进行各种计算，把最终的 DOM 操作计算出来并优化，由于这个 DOM 操作属于预处理操作，

并没有真实的操作 DOM，所以叫做虚拟 DOM。最后在计算完毕才真正将 DOM 操作提交，将 DOM 操作变化反映到 DOM 树上

面试题 92. 简述vue2.x 和 vuex3.x 渲染器的 diff 算法？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

简单来说，diff 算法有以下过程

同级比较，再比较子节点
先判断一方有子节点一方没有子节点的情况(如果新的 children 没有子节点，将旧的子节点移除)
比较都有子节点的情况(核心 diff)
递归比较子节点
正常 Diff 两个树的时间复杂度是 O(n^3)，但实际情况下我们很少会进行跨层级的移动 DOM，所以 Vue 将 Diff 进行了优化，从O(n^3) -> O(n)，只有当新旧 children 都为多个子节点时才需要用核心的 Diff 算法进行同层级比较。

Vue2 的核心 Diff 算法采用了双端比较的算法，同时从新旧 children 的两端开始进行比较，借助 key 值找到可复用的节点，再进行相关操作。相比 React 的 Diff 算法，同样情况下可以减少移动节点次数，减少不必要的性能损耗，更加的优雅。

Vue3.x 借鉴了 ivi 算法和 inferno 算法

在创建 VNode 时就确定其类型，以及在 mount/patch 的过程中采用位运算来判断一个 VNode 的类型，在这个基础之上再配合核心的 Diff 算法，使得性能上较 Vue2.x 有了提升。该算法中还运用了动态规划的思想求解最长递归子序列。

面试题 93. 如何让 CSS 值在当前的组件中起作用?

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

在 vue 文件中的 style 标签上，有一个特殊的属性：scoped。当一个 style 标签拥有 scoped 属性时，它的 CSS 样式就只能作用于当前的组件，也就是说，该样式只能适用于当前组件元素。通过该属性，可以使得组件之间的样式不互相污染。如果一个项目中的所有 style 标签全部加上了 scoped，相当于实现了样式的模块化。

scoped 的实现原理

vue 中的 scoped 属性的效果主要通过 PostCSS 转译实现的。PostCSS 给一个组件中的所有 DOM 添加了一个独一无二的动态属性，然后，给 CSS 选择器额外添加一个对应的属性选择器来选择该组件中 DOM，这种做法使得样式只作用于含有该属性的 DOM，即组件内部 DOM。

例如：

转译前

转译后：

面试题 94. Vue 中如何进行组件的使用？Vue 如何实现全局组件的注册？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

要使用组件，首先需要使用 import 来引入组件，然后在 components 属性中注册组件，之后就可以在模板中使用组件了。
可以使用 Vue.component 方法来实现全局组件的注册。

面试题 95. 请简述构建 vue-cli 工程都用到了哪些技术？他们的作用分别是什么？

推荐指数：★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

vue.js：vue-cli 工程的核心，主要特点是双向数据绑定和组件系统。
vue-router：vue 官方推荐使用的路由框架。
vuex：专为 Vue.js 应用项目开发的状态管理器，主要用于维护 vue 组件间共用的一些 变量和 方法。
axios（或者 fetch、ajax）：用于发起 GET 、或 POST 等 http请求，基于 Promise 设计。
vux等：一个专为vue设计的移动端UI组件库。
webpack：模块加载和vue-cli工程打包器。
eslint：代码规范工具
vue-cli 工程常用的 npm 命令有哪些?
下载 node_modules 资源包的命令：npm install
启动 vue-cli 开发环境的 npm命令：npm run dev
vue-cli 生成 生产环境部署资源 的 npm命令：npm run build
用于查看 vue-cli 生产环境部署资源文件大小的 npm命令：npm run build --report

面试题 96. 简述Vue SSR 的实现原理 ？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

app.js 作为客户端与服务端的公用入口，导出 Vue 根实例，供客户端 entry 与服务端 entry 使用。客户端 entry 主要作用挂载到 DOM 上，服务端 entry 除了创建和返回实例，还需要进行路由匹配与数据预获取。
webpack 为客服端打包一个 ClientBundle，为服务端打包一个 ServerBundle。
服务器接收请求时，会根据 url，加载相应组件，获取和解析异步数据，创建一个读取 Server Bundle 的 BundleRenderer，然后生成 html 发送给客户端。
客户端混合，客户端收到从服务端传来的 DOM 与自己的生成的 DOM 进行对比，把不相同的 DOM 激活，使其可以能够响应后续变化，这个过程称为客户端激活（也就是转换为单页应用）。为确保混合成功，客户 端与服务器端需要共享同一套数据。在服务端，可以在渲染之前获取数据，填充到 store 里，这样，在客户端挂载到 DOM 之前，可以直接从 store 里取数据。首屏的动态数据通过 window._INITIAL_STATE_ 发送到客户端
VueSSR 的原理，主要就是通过 vue-server-renderer 把 Vue 的组件输出成一个完整 HTML，输出到客户端，到达客户端后重新展开为一个单页应用。

面试题 97. 简述Vue 的 computed 的实现原理 ？

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

1、每个computed属性都会生成对应的Watcher实例，watcher拥有value属性和get方法，computed的getter函数会在get方法中调用，并返回赋值给value。初始设置dirty和lazy为true，当lazy为true时不会立即执行get方法，而是会在读取computed值时执行；

2、将computed属性添加到组件实例上，通过get、set进行属性值的获取或设置，并且重新定义getter方法；

3、页面初始化时，会读取computed属性值，触发重新定义的getter，由于观察者的dirty值为true，将会调用原始的getter函数，当getter方法读取data数据时会触发原始的get方法(数据劫持中的get方法)，将computed对应的watcher添加到data依赖收集器(dep)中。观察者的get方法执行完后，更新观察者的value，并将dirty置为false，表示value值已更新，之后执行观察者的depend方法，将上层观察者也添加到getter函数中data的依赖收集器(dep)中，最后返回computed的value值；

4、当更改了computed属性getter函数依赖的data值时，将会触发之前dep收集的watcher，依次调用watcher的update方法，先调用computed的观察者的update方法，由于lazy为true，会将dirty先设置为true，表示computed属性getter函数依赖data发生变化，但不调用观察者的get方法更新value值。这时调用包含更新页面方法的观察者的update方法，在更新页面时会读取computed属性值，触发重新定义的getter函数，由于dirty为true，调用该观察者的get方法，更新value并返回，完成页面渲染；

面试题 98. 简述Vue complier 的实现原理是什么样的?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

在使用 vue 的时候，我们有两种方式来创建我们的 HTML 页面，第一种情况，也是大多情况下，我们会使用模板 template 的方式，因为这更易读易懂也是官方推荐的方法；第二种情况

是使用 render 函数来生成 HTML，它比 template 更接近最终结果。

complier 的主要作用是解析模板，生成渲染模板的 render， 而 render 的作用主要是为了生成 VNode

complier 主要分为 3 大块：

parse：接受 template 原始模板，按着模板的节点和数据生成对应的 ast
optimize：遍历 ast 的每一个节点，标记静态节点，这样就知道哪部分不会变化，于是在页面需要更新时，通过 diff 减少去对比这部分DOM，提升性能
generate 把前两步生成完善的 ast，组成 render 字符串，然后将 render 字符串通过 new Function 的方式转换成渲染函数

面试题 99. Vue 如何快速定位那个组件出现性能问题的？

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

用 timeline 工具。 通过 timeline 来查看每个函数的调用时常，定位出哪个函数的问题，从而能判断哪个组件出了问题。

面试题 100. 请说明scoped 是如何实现样式穿透的?

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

首先说一下什么场景下需要 scoped 样式穿透。
在很多项目中，会出现这么一种情况，即：引用了第三方组件，需要在组件中局部修改第三方组件的样式，而又不想去除 scoped 属性造成组件之间的样式污染。此时只能通过特殊的方式，穿透 scoped。
有三种常用的方法来实现样式穿透。
方法一
使用 ::v-deep 操作符(>>> 的别名)
如果希望 scoped 样式中的一个选择器能够作用得“更深”，例如影响子组件，可以使用 >>> 操作符：

上述代码将会编译成：
.a[data-v-f3f3eg9] .b {
/* ... */
}
后面的类名没有 data 属性，所以能选到子组件里面的类名。
有些像 Sass 之类的预处理器无法正确解析 >>>，所以需要使用 ::v-deep 操作符来代替。
方法二
定义一个含有 scoped 属性的 style 标签之外，再定义一个不含有 scoped 属性的 style 标签，即在一个 vue 组件中定义一个全局的 style 标签，一个含有作用域的 style 标签：

此时，我们只需要将修改第三方样式的 css 写在第一个 style 中即可。

方法三

上面的方法一需要单独书写一个不含有 scoped 属性的 style 标签，可能会造成全局样式的污染。

更推荐的方式是在组件的外层 DOM 上添加唯一的 class 来区分不同组件，在书写样式时就可以正常针对针对这部分 DOM 书写样式。