01-组件通信

组建通信有八种

1. props
2. provide+indect
3. `$emit/$on`
4. `$children/$parent`
5. `$attrs/$listoners`
6. ref
7. $root
8. eventbus
    - vue3中不好使用，因为eventbus需要 `$on`
9. vuex/pinia
    - 在祖先组件定义 `provide` 属性，返回传递的值
    - 在后代组件通过 `inject` 接收组件传递过来的值

在vue3中，由于$on的移除所以事件总线已经不再使用了，实现起来也很复杂，不如直接用vuex或pinia

初次之外， `$parent` 和 `$listoner` 也被移除了所以也不会用这些方法

一般在项目中的话，主要分成以下几种方式传

- 父子之间： `props/$emit/$parent/ref/$attrs`
    - 使用$parent访问父亲组件
    - 使用ref访问子组件
- 兄弟之间： `$parent/$root/eventbus/vuex`
    - 可以桥接，使用兄弟之间共同的父组件或者就是用vuex
- 跨层级之间： `vuex/provide+indect`

02-v-for和v-if优先级

实践中，不可以把v-for和v-if放在一起，这个在文档中是重点标注的

- 在Vue2中，v-for的优先级高于v-if
    - 在vue2的语法中，生成的渲染函数里面，v-for里面包含这v-if，这个v-if会被渲染成一个三元表达式，所以v-for的优先级高于v-if
- 在Vue3中，v-if的优先级高于v-for

- Vue3中刚好相反，而且Vue3中如果说逻辑出现了冲突会报错，这样的解决方式更好

所以我们在写v-for和v-if的时候最好拆成两层来使用

03-简述Vue生命周期

1. 给出概念
   - Vue组件创建的时候会进行一系列的初始化步骤，比如说，数据观测，模板编译，挂在实例到DOM上，数据变化更新DOM等，这个过程中，会运行叫生命周期的钩子函数，以便用户在特定阶段有机会添加自己的代码
2. 列出生命周期的各个阶段
   - Vue的声明周期主要分成八个阶段:创建前后，beforeCreate，Created，载入前后，beforeMount，Mounted，更新前后，beforUpdate，Updated，销毁前后beforeDestroy，destroyed
   - 在Vue3中新增了三个用于调试和服务端渲染场景

生命周期V2	生命周期V3	描述
beforeCreate	beforeCreate/setup	组件实例创建之初
Created	Created/setup	组件实例已经被完全创建
beforeMount	beforeMount	组件挂在之前
Mounted	Mounted	组件挂载到实例上去之后
beforeUpdate	beforeUpdate	数据发生变化之后，更新之前
updated	updated	数据更新之后
beforeDestroy	beforeUnmount	组件实例销毁之前
Destroyed	Unmounted	组件实例销毁之后
activated	activated	keep-alive缓存的组件激活时
deactivated	deactivated	keep-alive缓存的组件停用的时候调用
	renderTracked	调试钩子，响应式依赖被收集时调用
	renderTriggered	调试钩子，响应式依赖被触发时调用
	serverPrefetch	ssr only，组件实例在服务器上被渲染前调用
errorCaptured	errorCaptured	捕获一个来自子孙组件错误时被调用

3. 阐述整体流程
    1. 首先是创建阶段
    2. 最早发生的是setup（CompositionAPI先处理）
    3. 之后进入beforeCreate和created（OptionsAPI）此时组件已经创建完毕
    4. 当组件创建完成后，判断是否存在已经预编译的模板，如果则继续，否则在运行时实时编译模板得到render函数
    5. 之后进入挂载阶段，其实到mounted之后，一个组件才算是彻底的初始化结束，因此在mounted阶段我们才能访问dom节点
    6. 之后进入更新阶段，其实就是不断循环实时监控数据是否发生改变，如果数据发生改变救护rerender和patch来进行更新
    7. 当组件不被需要的时候进入卸载流程，在beforeUnmount阶段进行定时器的卸载，取消订阅的任务的等
4. 结合实践
    1. beforeCreate：常用于插件开发中执行一些任务
    2. created：组件初始化完毕，可以访问各种数据，获取接口数据等，这个是最常见的，因为在这个阶段数据数据已经初始化完成了，说明响应式数据已经准备好了
    3. mounted：dom已经创建，可用于获取访问数据dom元素，访问子组件等，因为生命周期父子之间的调用关系是创建是从外到内，渲染是从内到外，所以在mounted阶段子组件已经挂载完成了
    4. beforeunmount：实例被销毁前调用，一般用于一些定时器或订阅的取消
    5. unmouted：销毁一个实例
5. Vue3中的变化
    1. setup和created谁先执行：setup
    2. setup中为啥没有beforeCreate和created?
        - 因为setup的时候其实组件已经创建完毕了，此时beforeCreate和created其实已经没有意义了

04-双向绑定的使用和原理

1. 双向绑定的定义
    - Vue中双向绑定是一个指令v-model，可以绑定一个响应式数据到视图，同时视图中变化能改变该值
2. 双向绑定的好处
    - v-model其实就是一个语法糖，默认情况下相当于：value和@input的实现
    - 使用v-model可以减少大量的事件处理代码，提高开发效率
3. 在哪里使用双向绑定
    1. 通常在表单项上面使用v-model，还可以在自定义组件上使用，表示某个值的输入和输出控制

4. 使用方式、细节、Vue3变化

   - 在input上使用v-model，其实就是将值绑定到表单元素value上，对于checkbox，可以使用true-value或则false-value指定特殊的值，对于radio可以使用value指定特殊的值，对于select可以通过options元素的value色值特殊的值，还可以结合lazy，number，trim作进一步的限定
   - v-model作用在自定义组件上时有很大的不同，vue3中类似于sync修饰符，最终展开的结果是modelValue属性和update:modelValue事件（这里的事件名可以自定义，比如说update:foo,update:bar等），在Vue3中可以使用参数形式指定多个不同的绑定比如：v-model:foo和v-model:bar，非常强大

5. 原理实现描述

   - v-model是一个指令，其实是在vue的编译器上完成，包含v-model的模板转换成渲染函数，这个渲染函数会把v-model编译成value属性的绑定和input事件的监听

```
// <input type="text" v-model="foo">
_c('input', {
  directives: [{ name: "model", rawName: "v-model", value: (foo), expression: "foo" }],
  attrs: { "type": "text" },
  domProps: { "value": (foo) },
  on: {
    "input": function ($event) {
      if ($event.target.composing) return;
      foo = $event.target.value
    }
  }
})
```

```
// <input type="checkbox" v-model="bar">
_c('input', {
  directives: [{ name: "model", rawName: "v-model", value: (bar), expression: "bar" }],
  attrs: { "type": "checkbox" },
  domProps: {
    "checked": Array.isArray(bar) ? _i(bar, null) > -1 : (bar)
  },
  on: {
    "change": function ($event) {
      var $$a = bar, $$el = $event.target, $$c = $$el.checked ? (true) : (false);
      if (Array.isArray($$a)) {
        var $$v = null, $$i = _i($$a, $$v);
        if ($$el.checked) { $$i < 0 && (bar = $$a.concat([$$v])) }
        else {
          $$i > -1 && (bar = $$a.slice(0, $$i).concat($$a.slice($$i + 1))) }
      } else {
        bar = $$c
      }
    }
  }
})
```

```
// <select v-model="baz">
//     <option value="vue">vue</option>
//     <option value="react">react</option>
// </select>
_c('select', {
  directives: [{ name: "model", rawName: "v-model", value: (baz), expression: "baz" }],
  on: {
    "change": function ($event) {
      var $$selectedVal = Array.prototype.filter.call(
        $event.target.options,
        function (o) { return o.selected }
      ).map(
        function (o) {
          var val = "_value" in o ? o._value : o.value;
          return val
        }
      );
      baz = $event.target.multiple ? $$selectedVal : $$selectedVal[0]
    }
  }
}, [
  _c('option', { attrs: { "value": "vue" } }, [_v("vue")]), _v(" "),
  _c('option', { attrs: { "value": "react" } }, [_v("react")])
])
```

05-Vue中如何扩展一个组件

1. 按照逻辑扩展和内容扩展来举例
    - 逻辑扩展：mixins，compositionAPI
    - 内容扩展：slots
2. 分别说出使用方法、场景和差异
    - 混入mixin在混入对象的时候会有一些问题，当mixin比较多的时候，可能会发生冲突，比如说变量名一样，变量方法名一样等
    - 后面就有了CompositionAPI
    - 插槽主要用于Vue组件中的内容开发，也可以用与组件扩展
        - 子组件Child

```
<div>
    <slot>这个内容会被父组件传递的内容替换</slot>
</div>
```

        - 父组件Parent

```
<div>
    <Child>来自老爹的内容</Child>
</div>
```

3. 作为扩展，说一下Vue3中的CompositionAPI

- useXX,useYY，然后用结构赋值的方式获取，十分方便

06-子组件可以修改父组件数据吗?

1. 讲讲单向数据流原则，表示为什么不能这么做
   - 父组件的prop更新会向下流动到子组件中，但是反过来不行，这样会防止子组件意外变更父组件的状态，导致数据流向难以理解
   - 每次父组件发生变更的时候，子组件中所有的prop都会刷新为最新的值，所以不应该在一个子组件内部修改prop，如果做了Vue会发出警告
2. 举几个常见的例子说说解决方案

   实际开发中会有两个场景想要修改一个属性

   1. prop用来传递一个初始值，子组件接下来希望将其作为一个本地prop数据来使用，在这个时候，最好定义一个本地的data，将prop作为其初始值

      ```
      const prop = defineProps(['initialCounter'])
      const counter = ref(props.initialCounter)
      ```

   2. prop以一种原始的值传入，且需要进行转换，这个时候使用prop的值来定义一个计算属性

      ```
      const prop = defineProps(['size'])
      // prop变化，计算属性自动更新
      const normalizedSize = computed(() ⇒
      props.size.trim().toLowerCase())
      ```

3. 结合实践讲讲如果要修改父组件状态应该如何做
   - 事件中如果想要改变父组件属性其实应该emit一个事件，让父组件内部自己去变更

07-Vue项目权限管理如何做?

1. 权限管理一般需求是两个:
   1. 页面权限
   2. 按钮权限
2. 具体实现的时候分成前端和后端两种方式
   - 前端方案：前端方案会把所有的路由信息在前端进行配置，通过路由守卫要求用户登录，用户登录后根据用户的角色过滤出路由表，比如说可以在认证页面的路由上的meta中配置一个roles字段，获取角色后来取两者的交集，若结果不为空则可以访问，如果可以访问的话，使用roter.addRoutes方法动态添加路由即可

    - 后端方案：后端方案会把所有也么路由信息存储在数据库中，用户登录的时候根据其角色查询得到能访问的所有页面的路由信息返回给前端，前端在通过addRoutes动态添加路由信息即可
    - 按钮控制权限的话通常会实现一个指令，将按钮要求角色通过值传递给指令，在指令的mounted钩子中可以判断当前用户角色和按钮是否存在交集，有就保留，无就移除
3. 纯前端方案的优点是实现简单，不需要额外权限管理页面，但是维护起来问题比较大，有新的页面和角色需求就要修改前端代码重新打包部署

   服务端就不会存在这个问题，通过专门的角色和权限管理页面，配置页面信息和按钮信息到数据库，每次登录的时候获取最新的路由信息即可

学生管理系统中用的就是后端实现的方案

08-Vue数据响应式的理解

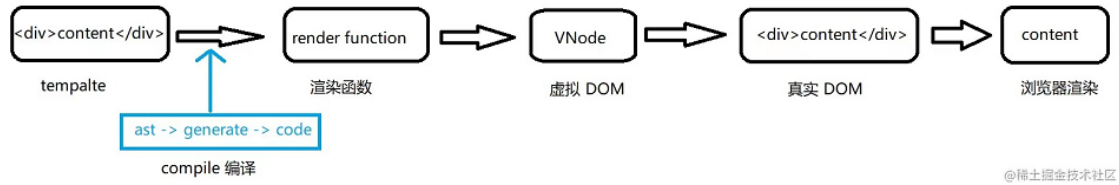
1. 什么是响应式
    - 数据响应式就是能够使数据发生变化可以被检测并对这种变化作为响应的机制
2. 为什么Vue需要响应式
    - MVVM框架中要解决的一个核心问题是链接数据层和视图层，通过数据驱动应用，数据变化，视图更新，要做到这一点就需要对数据做响应式处理，这样一旦发生数据变化就可以立即做出更新处理
3. 响应式能带给我们什么好处
    - 以Vue为例，通过数据响应式加上虚拟DOM和patch算法，开发人员只需要操作数据，关心业务，完全不用接触频繁的DOM操作，从而大大提升开发效率，降低开发难度
4. Vue的响应式是怎么实现的？有哪些优点
    - Vue2中的数据响应式是会根据数据类型来做不同的处理，如果是对象，则采用Object.defineProperty()方法来定义数据拦截，当数据被访问或者发生变化的时候，感知做出响应
    - 如果是数组则通过覆盖数组原型对象的7个变更方法，让这些方法可以做额外的更细通知，从而做出响应
    - 这种机制很好的解决了数据响应化的问题，但是在实际使用的过程中也存在一些缺点：比如初始化递归比那里会造成很大的性能损失，而且新增或则删除属性的时候需要用Vue.set和Vue。delete这种特殊的api才可以生效，而且在ES6新增的Map，Set这些数据结构也都不支持
5. Vue3中响应式的变化
    - 为了解决刚才的问题Vue3中，重新编写了这个实现：用Es6的Proxy代理要响应化的数据，不需要特殊的api，而且他还对响应式的实现代码抽离出来了一个reactivite的包，独立于Vue，让一些第三方模块可以不依赖Vue，只需要对reactivite产生依赖即可，但是就是可能会有兼容性的问题

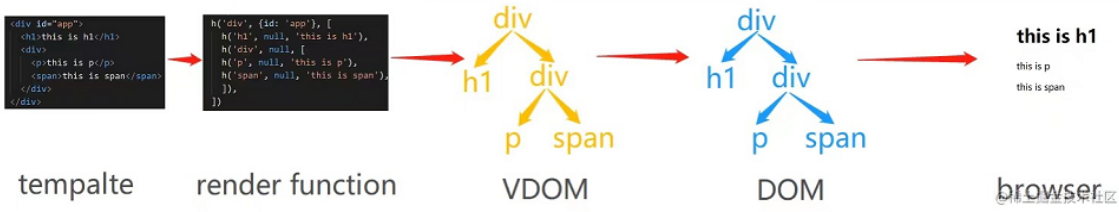
09-虚拟DOM

1. vdom是什么
   - 虚拟dom就是一个虚拟的dom对象，本身就是一个js对象，只是他是通过不同属性去描绘一个视图性能
2. 引入vdom的好处
   - 将真实元素节点抽象成vnode，有效减少操作dom此时，从而提高程序性能
     - 直接操作dom是有限制的，比如diff，clone等操作，一个真实元素上有许多的内容，如果直接对其进行diff操作，会额外diff一些没必要的内容，同样的，如果需要进行clone，那么需要将其全部内容进行复制这也是没必要的，如果把这写操作放到js上就会很简单
     - 而且操作dom是很昂贵的，因为频繁的操作dom会隐藏重绘和回流，但是通过抽象vnode进行中间处理的话，可以有效减少直接操作dom的次数，减少页面的重绘和回流
3. vdom如何生成，又如何成为dom

   如下图，Vue的执行过程大概为这个方式

   首先会进行模板编译，把模板利用complie编译成render渲染函数，之后由render渲染函数生成虚拟DOM，然后虚拟DOM在渲染成真实DOM，最后进行浏览器渲染





4. 后续的diff中的作用

10-diff算法

1. diff算法是干什么的
   - Vue中的diff算法称为patching算法，它有Snabbdom修改而来，虚拟DOM转换为真实DOM就需要通过patch方法来转换
2. 它的必要性
   - 为什么需要patch方法呢？在Vue1的时候每个依赖都有更新函数对应，可以做到进准更新，因此不需要虚拟DOM和patching算法支持，但是这样粒度过细导致Vue1无法承载较大的英语
   - 在Vue2中为了降低粒度，每个组件只有一个watcher与之对应，此时需要引入patching算法才能精确找到发生变化的地方并高效更新

3. 它何时执行
    - vue中diff执行的时刻是组件内响应式数据变更触发实例执行更新函数时，更新函数会再次找到render函数获取最新的虚拟DOM，然后执行patch函数，并传入新旧两次虚拟DOM，进行对比找到变化的地方，最后转为对应的DOM操作
4. 具体执行方式：对比的过程是一个递归过程，遵循深度优先，同层比较的策略
    - 首先判断两个节点是否是相同同类节点（判断key，类型等），不同删除重建创建
    - 如果双方都是文本更新文本内容
    - 如果双方都是元素节点则递归更新子元素，同时更更新元素属性
    - 更新子节点分成几种情况：
        1. 新的子节点是文本，旧子节点是数组（子元素）则清空，设置文本
        2. 新的子节点是文本，旧子节点是文本则直接更新文本
        3. 新的子节点是数组，旧子节点是文本则情况文本，并创建新的子节点数组中的子元素
        4. 新的子节点是数组，旧子节点是数组（子元素），则比较两组子节点继续递归
            - 双端比较，新旧各设置头尾节点，从头到尾进行比较
            - 一个数组利用key，做一个map，在另一个节点中去看有没有这个key
5. Vue3中的优化
    - patchFlags：可以告诉我们节点中什么是动态的，那么更新的是就只更新动态的就可以了
    - block

11-Vue3新特性

1. api层面上主要是：CompositionAPI、SFCCompositionAPI语法糖、Teleport传送门、Fragment片段、Emits选项、自定义渲染器、SFCCSS变量、Suspense
2. 另外VUe3更快更小

   更快：
    - 虚拟DOM重写
    - 编译器优化：静态提升、patchFlags、block等
    - Proxy的响应式系统

   更小：更好的treeshcking优化

   更容易维护：TS+模块化

   更容易扩展：独立的响应式模块，自定义渲染器

12-vue-router动态路由有什么用

1. 什么是动态路由

   很多时候，访问一个页面需要将给定匹配模式的路由映射到同一个组件，这种情况需要定义动态路由

2. 什么时候使用动态路由，怎样定义动态路由

   比如说有一个User组件，它根据所有用户进行渲染，但是用户ID不同，在VueRouter中，可以在路径中设置一个动态字段来实践

   ```
   {
       path:'user/:id',
       component: User
   }
   ```

   其中 `:id` 就是路径参数

3. 参数如何获取

   - 路径参数用冒号获取，当一个路由被匹配时，他的params的值将在每个组件中以 `this.$route.params` 的形式暴露出来

4. 细节、注意事项

   - 参数可以有多个，而且除了params之外，还有query、hash等方法

13-如何实现一个vue-router

首先分析：SPA应用路由需要解决的问题是页面跳转不刷新，同时路由要以插件的方式存在

1. 首先我回顶一个createRender函数，返回路由器的实例
   1. 保存用户传入的配置项
   2. 箭筒hah或者popstate事件
   3. 会调离根据path匹配对应路由
2. 将路由定义成一个Vue插件，实现install方法，内部做两件事情：
   1. 实现两个全局组件：router-link和router-view，实现跳转和内容显示
   2. 实现两个全局变量： `$route` 和 `$router` ，组件内部可以访问当前路由和路由器实例

14-key的作用

1. 给出结论，key的作用是用于优化patch性能
    - key的左右就是更搞笑的更新虚拟DOM
2. key的必要性
    - 在diff过程中，判断连个节点是不是同一个节点的话可以key是必要条件，渲染一组列表时，key往往是一个唯一表示，如果不定义key的话，vue只能比较两个结点是同一个，哪怕不是同一个，这样会导致频繁更新元素，让diff过程低效影响性能
3. 实际使用方式
    - 实际使用中在渲染一组列表时key必须设置，而且必须是唯一标识，应该避免使用数组索引作为key，这可能导致一些隐蔽的bug；vue中在使用相同标签元素过渡切换时，也会使用key属性，其目的也是为了让vue可以区分它们，否则vue只会替换其内部属性而不会触发过渡效果
4. 总结：可从源码层面描述一下vue如何判断两个节点是否相同

15-nextTick的使用和原理

1. nextTick是做什么的?
    - nextTick是等待下一次DOM更新刷新的工具方法
2. 为什么需要它呢?
    - Vue有个异步更新策略，意思是如果数据变化，Vue不会立刻更新DOM，而是开启一个队列，把组件更新函数保存在队列中，在同一事件循环中发生的所有数据变更会异步的批量更新。这一策略导致我们对数据的修改不会立刻体现在DOM上，此时如果想要获取更新后的DOM状态，就需要使用nextTick
3. 开发时何时使用它？抓抓头，想想你在平时开发中使用它的地方
    - created中想要获取DOM时；
    - 响应式数据变化后获取DOM更新后的状态，比如希望获取列表更新后的高度
4. 原理解读，结合异步更新和nextTick生效方式，会显得你格外优秀

    在Vue内部，nextTick之所以能够让我们看到DOM更新后的结果，是因为我们传入的callback会被添加到队列刷新函数(flushSchedulerQueue)的后面，这样等队列内部的更新函数都执行完毕，所有DOM操作也就结束了，callback自然能够获取到最新的DOM值

16-computed和watch的区别

1. 计算属性可以**从组件数据派生出新数据**，最常见的使用方式是设置一个函数，返回计算之后的结果，computed和methods的差异是它具备缓存性，如果依赖项不变时不会重新计算。侦听器**可以侦测某个响应式数据的变化并执行副作用**，常见用法是传递一个函数，执行副作用，watch没有返回值，但可以执行异步操作等复杂逻辑。

2. 计算属性常用场景是简化行内模板中的复杂表达式，模板中出现太多逻辑会是模板变得臃肿不易维护。侦听器常用场景是状态变化之后做一些额外的DOM操作或者异步操作。选择采用何用方案时首先看是否需要派生出新值，基本能用计算属性实现的方式首选计算属性。
3. 使用过程中有一些细节，比如计算属性也是可以传递对象，成为既可读又可写的计算属性。watch可以传递对象，设置deep、immediate等选项。
4. vue3中watch选项发生了一些变化，例如不再能侦测一个点操作符之外的字符串形式的表达式； reactivity API中新出现了watch、watchEffect可以完全替代目前的watch选项，且功能更加强大

17-父子组件创建、挂载的顺序是怎么样的

创建的时候：现有父组件，后有子组件

挂载的时候：先挂载子组件，后挂载父组件

1. 创建的过程是自上而下，挂载过程是自下而上的，因为是利用递归的关系，创建的时候就是递归进去，挂载的时候就是在回溯
    1. parent created
    2. child created
    3. child mounted
    4. parent mouted
2. 之所以只有是因为Vue创建的过程是一个递归的过程，先创建父组件，有子组件就会创建子组件，因此创建时，先有父组件，再有子组件

    子组件首次创建时会添加mouted钩子到队列里面，等patch结束后再执行它们，因此可见子组件的mounted钩子是先进入队列的，因此等patch结束执行的时候这些钩子也会先执行

18-如何缓存和更新组件

1. 缓存用keep-alive，它的作用与用法

    开发中缓存组件使用keep-alive组件，keep-alive是vue内置组件，keep-alive包裹动态组件component时，会缓存不活动的组件实例，而不是销毁它们，这样在组件切换过程中将状态保留在内存中，防止重复渲染DOM。

    ```
    <keep-alive>
      <component :is="view"></component>
    </keep-alive>
    ```

2. 使用细节，例如缓存指定/排除、结合router和transition

结合属性include和exclude可以明确指定缓存哪些组件或排除缓存指定组件。vue3中结合vue-router时变化较大，之前是 `keep-alive` 包裹 `router-view` ，现在需要反过来用 `router-view` 包裹 `keep-alive` ：

```
<router-view v-slot="{ Component }">
  <keep-alive>
    <component :is="Component"></component>
  </keep-alive>
</router-view>
```

3. 组件缓存后更新可以利用activated或者beforeRouteEnter

   缓存后如果要获取数据，解决方案可以有以下两种：

   - beforeRouteEnter：在有vue-router的项目，每次进入路由的时候，都会执行 `beforeRouteEnter`

     ```
     beforeRouteEnter(to, from, next){
       next(vm=>{
         console.log(vm)
         // 每次进入路由执行
         vm.getData()  // 获取数据
       })
     },
     复制代码
     ```

   - actived：在 `keep-alive` 缓存的组件被激活的时候，都会执行 `actived` 钩子

     ```
     activated(){
           this.getData() // 获取数据
     },
     ```

4. 原理阐述

   keep-alive是一个通用组件，它内部定义了一个map，缓存创建过的组件实例，它返回的渲染函数内部会查找内嵌的component组件对应组件的vnode，如果该组件在map中存在就直接返回它。由于component的is属性是个响应式数据，因此只要它变化，keep-alive的render函数就会重新执行

19-如何从0~1架构一个项目

20-你知道哪些Vue最佳实践

21-Vuex的理解

22-从template到render发生了什么

1. 引入vue编译器概念

   Vue中有个独特的编译器模块，称为“compiler”，它的主要作用是将用户编写的template编译为js中可执行的render函数。

2. 说明编译器的必要性

   之所以需要这个编译过程是为了便于前端程序员能高效的编写视图模板。相比而言，我们还是更愿意用HTML来编写视图，直观且高效。手写render函数不仅效率底下，而且失去了编译期的优化能力

3. 阐述编译器工作流程

   在Vue中编译器会先对template进行解析，这一步称为parse，结束之后会得到一个JS对象，我们成为抽象语法树AST，然后是对AST进行深加工的转换过程，这一步成为transform，最后将前面得到的AST生成为JS代码，也就是render函数

23-Vue实例挂载过程发生了什么

24-Vue3设计目标和优化点有哪些

25-Vue优化方法有哪些

26-Vue组件为什么只能有一个根结点

27-什么情况需要使用Vuex模块?

28-为什么要有路由懒加载

29-ref和reactive有什么区别

30-watch和watchEffect有什么不同

31-SPA和SSR有何不同

32-vue-loader是什么

33-写过自定义指令吗

1. 定义

   Vue有默认住了比如v-model，v-for，同时vue也允许用户自己自定义指令来扩展VUe

2. 何时使用

   自定义指令主要完成一些课复用的低层级DOM操作

3. 如何用

   - 定义自定义指令有两种方式：对象和函数形式，前者类似组件定义，有各种生命周期，后者只会在mouted和updated时执行
   - 注册自定义之类类似组件，可以使用app.rdirective()全局注册，也可以使用{directives:{xxx}}局部注册

4. 常用指令

   - 复制粘贴 v-copy
   - 长按 v-longpress
   - 防抖 v-debounce
   - 图片懒加载 v-lazy
   - 按钮权限 v-premission

5. Vue3中的变化

   vue3中发生了较大的变化，主要是钩子的名称和组件保持一致，这样更容易记忆，不容易犯错，3.2中可以在setup中用一个小写的v开头方便的定义

34- `$attrs` 和 `$listener` 是做什么的

35-v-once的使用场景

36-什么是递归组件

37-什么是异步组件

1. 异步组件作用

   在大型应用中，我们需要分割应用为更小的块，并且在需要组件时再加载它们。

2. 何时使用异步组件

   我们不仅可以在路由切换时懒加载组件，还可以在页面组件中继续使用异步组件，从而实现更细的分割粒度。

3. 使用细节

   使用异步组件最简单的方式是直接给defineAsyncComponent指定一个loader函数，结合ES模块动态导入函数import可以快速实现。我们甚至可以指定loadingComponent和errorComponent选项从而给用户一个很好的加载反馈。另外Vue3中还可以结合Suspense组件使用异步组件

4. 和路由懒加载的不同

38-Vue如何处理错误?

39-如何从0实现Vuex

官方说 `vuex` 是一个状态管理模式和库，并确保这些状态以可预期的方式变更。可见要实现一个 `vuex` ：

- 要实现一个 `Store` 存储全局状态
- 要提供修改状态所需API： `commit(type, payload)` ， `dispatch(type, payload)`

实现 `Store` 时，可以定义Store类，构造函数接收选项options，设置属性state对外暴露状态，提供commit和dispatch修改属性state。这里需要设置state为响应式对象，同时将Store定义为一个Vue插件。

`commit(type, payload)` 方法中可以获取用户传入 `mutations` 并执行它，这样可以按用户提供的方法修改状态。 `dispatch(type, payload)` 类似，但需要注意它可能是异步的，需要返回一个Promise给用户以处理异步结果。

40-mutation和action有什么区别

修改状态只能是 `mutations` ， `actions` 只能通过提交 `mutation` 修改状态即可

1. 给出两者概念说明区别

   官方文档说：更改 Vuex 的 store 中的状态的唯一方法是提交 `mutation` ， `mutation` 非常类似于事件：每个 `mutation` 都有一个字符串的**类型(type)和一个回调函数（handler）**。 `Action` 类似于 `mutation` ，不同在于： `Action` 可以包含任意异步操作，但它不能修改状态， 需要提交 `mutation` 才能变更状态。

2. 举例说明应用场景

   因此，开发时，包含异步操作或者复杂业务组合时使用action；需要直接修改状态则提交mutation。但由于dispatch和commit是两个API，容易引起混淆，实践中也会采用统一使用dispatch action的方式

3. 使用细节不同

   调用dispatch和commit两个API时几乎完全一样，但是定义两者时却不甚相同，mutation的回调函数接收参数是state对象。action则是与Store实例具有相同方法和属性的上下文context对象，因此一般会解构它为 `{commit, dispatch, state}` ，从而方便编码。另外dispatch会返回Promise实例便于处理内部异步结果。

4. 简单阐述实现上差异

   实现上commit(type)方法相当于调用 `options.mutations[type](state)` ； `dispatch(type)` 方法相当于调用 `options.actions[type](store)` ，这样就很容易理解两者使用上的不同了

41-Vue长列表优化思路

1. 描述大数据量带来的问题

   大型项目中经常需要渲染大量数据，此时容易出现卡顿，比如大数据表格、树等

2. 分不同情况的不同处理

   - 避免出现大数据量：采取分页的方式获取
   - 避免渲染大连舒：vue-virtual-scroller等虚拟滚动的方案，只渲染视口范围内的数据
   - 避免更新：使用v-once只渲染一次
   - 优化更新：通过v-memo缓存子树，有条件更细，提高复用，避免不必要更新
   - 按需加载数据，采用懒加载方式，在用户需要的时候再加载数据，比如tree组件的子树懒加载

3. 总结

要看具体需求，首先从设计上避免大数据获取和渲染，实在需要这样做的话，可以采用虚拟列表的方式优化

42-如何监听vuex状态变化

43-router-link和router-view是如何起作用的

44 - vue的reactivity是独立的

第三题:

第四题：C 6

第五题：D 归并排序