学员优秀答案

1.20 v-if和v-for哪个优先级高？如果两个同时出现，应该怎么优化得到更好的性能?

5组：

当 v-if 与 v-for 一起使用时，v-for 具有比 v-if 更高的优先级。

原因分析

在v-for和v-if同时存在与同一个标签或组件上时：

例：有如下template 模板

```
<ul>
<li v-for="item in lists" :key="item.id" v-if="item.bol">{{item.text}}</li>
</ul>
```

vue在生成render函数时会先循环`lists`生成虚拟节点,之后根据每一个item上的bol来判断是否渲染当前li元素

优化方案

分两种情况：

1. 以上面示例来看 在v-if中的变量用的是`item`中的属性时
    - 可以在对`lists`渲染之前先过滤掉`item`中`bol`属性值为false的元素，可以借用computed计算属性来完成
2. 如果v-if使用的变量为其他属性（不和item有关） -
    - 可以在li的外层加一层把v-if加在其上面即可

7组：

优先级

当 v-if 与 v-for 一起使用时，v-for 具有比 v-if 更高的优先级。 这意味着 v-if 将分别重复运行于每个 v-for 循环中。当你只想为部分项渲染节点时，这种优先级的机制会十分有用，如下：

```
<li v-for="todo in todos" v-if="!todo.isComplete">
  {{ todo }}
</li>
```

上面的代码将只渲染未完成的 todo。 而如果你的目的是有条件地跳过循环的执行，那么可以将 v-if 置于外层元素 上。如：

```
<ul v-if="todos.length">
  <li v-for="todo in todos">
    {{ todo }}
  </li>
</ul>
<p v-else>No todos left!</p>
```

时使用时的分析

不推荐同时使用 v-if 和 v-for。

一般我们在两种常见的情况下会倾向于这样做：

- 为了过滤一个列表中的项目 (比如 v-for="user in users" v-if="user.isActive")。在这种情形下，请将 users 替换为一个计算属性 (比如 activeUsers)，让其返回过滤后的列表。
- 为了避免渲染本应该被隐藏的列表 (比如 v-for="user in users" v-if="shouldShowUsers")。这种情形下，请将 v-if 移动至容器元素上 (比如 ul, ol)。

当 Vue 处理指令时，v-for 比 v-if 具有更高的优先级，所以这个模板：

```
<ul>
  <li
    v-for="user in users"
    v-if="user.isActive"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

将会经过如下运算：

```
this.users.map(function (user) {
  if (user.isActive) {
    return user.name
  }
})
```

因此哪怕我们只渲染出一小部分用户的元素，也得在每次重渲染的时候遍历整个列表，不论活跃用户是否发生了变化。

通过将其更换为在如下的一个计算属性上遍历：

```
computed: {
  activeUsers: function () {
    return this.users.filter(function (user) {
      return user.isActive
    })
  }
}
```

```
<ul>
  <li
    v-for="user in activeUsers"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

我们将会获得如下好处：

- 过滤后的列表只会在 users 数组发生相关变化时才被重新运算，过滤更高效。
- 使用 v-for="user in activeUsers" 之后，我们在渲染的时候只遍历活跃用户，渲染更高效。
- 解耦渲染层的逻辑，可维护性 (对逻辑的更改和扩展) 更强。

为了获得同样的好处，我们也可以把：

```
<ul>
  <li
    v-for="user in users"
    v-if="shouldShowUsers"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

更新为：

```
<ul v-if="shouldShowUsers">
  <li
    v-for="user in users"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

通过将 v-if 移动到容器元素，我们不会再对列表中的每个用户检查 shouldShowUsers。取而代之的是，我们只检查它一次，且不会在 shouldShowUsers 为否的时候运算 v-for。

18组：

当 Vue 处理指令时，v-for 比 v-if 具有更高的优先级。

若v-for 和 v-if同时出现，为了得到更好的性能，我们需要先判断，v-if是为了过滤列表中的单条item，还是为了避免渲染该被隐藏的整个列表。

【目的】：过滤列表中的单条item，有条件地跳过循环的执行

(1)可先把列表进行过滤，再将过滤完的列表进行条件循环语句。

```
computed: {
  activeUsers () {
    return this.users.filter(user => user.isActive)
  }
}
```

(2)可将 v-if 置于上(此方法次之)

```
<ul>
  <template  v-for="user in users" >
    <li
      v-if="user.isActive"
      :key="user.id"
    >
      {{ user.name }}
    </li>
  </template>
</ul>
```

【目的】：避免渲染本应该被隐藏的列表

将 v-if 移动至需要循环的元素的父级元素上，如ul。

```
<ul v-if="shouldShowUsers">
  <li
    v-for="user in users"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

所带来的性能优化：

1.过滤后的列表只会在 users 数组发生相关变化时才被重新运算，过滤更高效。

2.使用 v-for="user in activeUsers" 之后，我们在渲染的时候只遍历活跃用户，按需渲染，使渲染更高效。

3.解耦渲染层的逻辑，可维护性 (对逻辑的更改和扩展) 更强。详见vue风格指南。

记住：永远不要把 v-if 和 v-for 同时用在同一个元素上。

1.21 Vue组件data选项为什么必须是个函数而Vue的根实例则没有此限制?

7组：

- 在Vue根实例中data可以为函数，可以为对象
  - 源码层方面分析:

```
// 在initData时候会先判断data的类型，所以在根实例中无论是对象或者是函数类型均会解析
    let data = vm.$options.data
    data = vm._data = typeof data === 'function'
        ? getData(data, vm)
        : data || {}
```

  - 原因: 因为无论是什么类型, 只要挂载在根实例上, 均属于全局类型, 可以全局调用, 不存在数据污染
- 在Vue的组件中data必须为函数
  - 源码层面分析:

```
// 在创建或注册模板的时候，会判断data的类型，如果不是函数类型，则会抛出一个错误
    strats.data = function (
        parentVal: any,
        childVal: any,
        vm?: Component
    ): ?Function {
        if (!vm) {
            if (childVal && typeof childVal !== 'function') {
            process.env.NODE_ENV !== 'production' && warn(
                'The "data" option should be a function ' +
                'that returns a per-instance value in component ' +
                'definitions.',
                vm
            )

            return parentVal
            }
            return mergeDataOrFn(parentVal, childVal)
        }

        return mergeDataOrFn(parentVal, childVal, vm)
    }
```

- 原因: 其实每一个组件均可当作一个构造器, 注册组件的本质其实就是构造器的引用. 如果直接使用对象, 他们的内存地址是一样的, 一个数据改变了其他也改变了, 这就造成了数据污染, 如果使用函数的话, 会形成一个全新的作用域, 这样data中的数据不会相互影响, 从而避免数据污染, 下面从原型链出发举个例子:

```
const MyComponents = function() {}
   MyComponents.prototype.data = {
       number: 1
   }

   let component1 = new MyComponents()
   let component2 = new MyComponents()
   component1.data.number = 2

   console.log(component1.data.number, 'component1-data') // 2
   console.log(component2.data.number, 'component2-data') // 2
```

12组：

```
// vue源码里 对data进行数据响应式的部分代码
function initData (vm: Component) {
  let data = vm.$options.data
  // 如果data是一个函数，执行该函数，获取其值
  // 如果是一个对象，直接使用，如果该值没有意义，将其置为空对象
  data = vm._data = typeof data === 'function'
    ? getData(data, vm)
    : data || {}
    ...
}

export function getData (data: Function, vm: Component): any {
  ...
  try {
    return data.call(vm, vm)
  } catch (e) {
    handleError(e, vm, `data()`)
    return {}
  } finally {
    ...
  }}
```

1. 可以看出Vue组件的data选项可以是一个函数，也可以是一个对象，如果data选项是一个函数则执行getData()取其值。
2. 组件是可以复用的，如果Vue组件的data选项是一个对象，组件复用时会将组件内data指向同一块内存地址，造成数据污染。如果data选项是一个函数，通过getData()函数里的return data.call(vm, vm)可看出返回的data只针对当前Vue组件实例。

3. Vue的根实例独此一份，不会有其他组件与其共享数据，所以Vue的根实例的data选项可以为一个函数，也可以是一个对象。

15组：

官方文档描述：

> 根据官方文档的说法，一个组件的data必须是一个函数，因此每个实例可以维护一份被返回对象的独立的拷贝。如果没有这条规则，操作一个组件实例时可能同时影响到其他组件实例。

从源码上来看：

首先从init方法开始，该方法中有mergeOptions方法，合并组件属性到vm。

```
Vue.prototype._init = function (options?: Object) {
    const vm: Component = this
    ...

    // merge options
    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
      )
    }

    /* istanbul ignore else */
    ...
  }
```

从mergeOptions方法中我们可以找到合并各个属性的方法，其中合并data属性的方法源码如下：

```
export function mergeDataOrFn (
  parentVal: any,
  childVal: any,
  vm?: Component
): ?Function {
  if (!vm) {
    if (!childVal) {
      return parentVal
```

```
    }
    if (!parentVal) {
      return childVal
    }
    return function mergedDataFn () {
      return mergeData(
        typeof childVal === 'function' ? childVal.call(this, this) : childVal,
        typeof parentVal === 'function' ? parentVal.call(this, this) :
parentVal
      )
    }
  } else {
    return function mergedInstanceDataFn () {
      // instance merge
      const instanceData = typeof childVal === 'function'
        ? childVal.call(vm, vm)
        : childVal
      const defaultData = typeof parentVal === 'function'
        ? parentVal.call(vm, vm)
        : parentVal
      if (instanceData) {
        return mergeData(instanceData, defaultData)
      } else {
        return defaultData
      }
    }
  }
}
```

上述源码体现了data选项为什么在组件中需要是函数，而在根目录中不需要。

对于子data，判断是否是函数，如果是函数则执行，返回一个对象，挂载到vm上。

此时若data是对象形式，将该data直接挂载到了vm上，则另一个组件实例将和此组件实例共用一个data

对于根目录来说，一个Vue实例只有一个根目录，它使用函数或是对象来定义data，都只有根目录使用，不会有第二个根目录而造成影响。

1.22 你知道vue中key的作用和工作原理吗？说说你对它的理解。

7组：

作用

- 主要用在 Vue 的虚拟 DOM 算法，在新旧 nodes 对比时辨识 VNodes，相当于唯一标识ID。
- Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染， 因此使用key值可以提

高渲染效率，同理，改变某一元素的key值会使该元素重新被渲染。

工作原理

> 因为key值主要在虚拟DOM算法，即diff算法。所以我们在src\core\vdom\patch.js文件中，从源码级别进行探讨

- 用一个源码中的方法进行举例：

```
// 主要在patch方法中调用
function sameVnode (a, b) {
  return (
    a.key === b.key && (
      (
        a.tag === b.tag &&
        a.isComment === b.isComment &&
        isDef(a.data) === isDef(b.data) &&
        sameInputType(a, b)
      ) || (
        isTrue(a.isAsyncPlaceholder) &&
        a.asyncFactory === b.asyncFactory &&
        isUndef(b.asyncFactory.error)
      )
    )
  )
}

// 以下代码在patchVnode方法中
// reuse element for static trees.
// note we only do this if the vnode is cloned -
// if the new node is not cloned it means the render functions have been
// reset by the hot-reload-api and we need to do a proper re-render.
if (isTrue(vnode.isStatic) &&
    isTrue(oldVnode.isStatic) &&
    vnode.key === oldVnode.key &&
    (isTrue(vnode.isCloned) || isTrue(vnode.isOnce))
) {
    vnode.componentInstance = oldVnode.componentInstance
    return
}
```

- 分析：在例子中可以看出，对Vnode进行patch的时候会调用sameVnode方法，里面会使用key值是否相等来判断Vnode是否为同一个。并且在对比过程中作为Vnode复用的一个判断条件。
- 结论：key值是在DOM树进行diff算法时候发挥作用。一个是用来判断新旧Vnode是否为同一个，从而进行下一步的比较以及渲染。另外一个作用就是判断Vnode是否可以复用，是否需要重新渲染。

15组：

作用

1.Vue为了高效渲染，将会复用已有元素而不会进行重新渲染，这样速度是快了，但是有时候我们并不想要复用组件，而需要独立，此时可以通过添加唯一的key值，使两个元素各自渲染。
2.主要用在 Vue 的虚拟 DOM 算法，在新旧 nodes 对比时辨识 VNodes。如果不使用 key，Vue 会使用一种最大限度减少动态元素并且尽可能的尝试就地修改/复用相同类型元素的算法。而使用 key 时，它会基于 key 的变化重新排列元素顺序，并且会移除 key 不存在的元素。
有相同父元素的子元素必须有独特的 key。重复的 key 会造成渲染错误。
3.它也可以用于强制替换元素/组件而不是重复使用它。当你遇到如下场景时它可能会很有用：
    1.完整地触发组件的生命周期钩子
    2.触发过渡

工作原理

在源码目录/src/core/vdom/patch.js，我们找到了diff算法执行的位置，在这个文件中有一个sameVnode方法，用来比较节点是否一样。

```
function sameVnode (a, b) {
  return (
    a.key === b.key && (
      (
        a.tag === b.tag &&
        a.isComment === b.isComment &&
        isDef(a.data) === isDef(b.data) &&
        sameInputType(a, b)
      ) || (
        isTrue(a.isAsyncPlaceholder) &&
        a.asyncFactory === b.asyncFactory &&
        isUndef(b.asyncFactory.error)
      )
    )
  )
}
```

从这里看出，该方法首先会比较两个key是否相同，如果不相同，则直接返回false，而不会继续下面的逻辑，大大提高了性能。

再然后我们来看下在updateChildren方法中,有这么一段：

```
 if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx)
        idxInOld = isDef(newStartVnode.key)
          ? oldKeyToIdx[newStartVnode.key]
          : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)
    if (isUndef(idxInOld)) { // New element
        createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, false, newCh, newStartIdx)
    } else {
```

```
        vnodeToMove = oldCh[idxInOld]
        if (sameVnode(vnodeToMove, newStartVnode)) {
        patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue, newCh,
newStartIdx)
        oldCh[idxInOld] = undefined
        canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm,
oldStartVnode.elm)
        } else {
        // same key but different element. treat as new element
        createElm(newStartVnode, insertedVnodeQueue, parentElm,
oldStartVnode.elm, false, newCh, newStartIdx)
        }
    }
```

当未定义oldKeyToIdx时，调用createKeyToOldIdx方法，如果新元素节点有key，直接从oldKeyToIdx获取，如果没有key，调用findIdxInOld方法。 如果没有找到idxInOld，则创建新元素。

下面来看看这两个方法做了什么：

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

```
function findIdxInOld (node, oldCh, start, end) {
    for (let i = start; i < end; i++) {
      const c = oldCh[i]
      if (isDef(c) && sameVnode(node, c)) return i
    }
  }
```

createKeyToOldIdx创建一个map，将元素上的key值做属性名，索引做属性值，然后返回map。findIdxInOld遍历老元素上的节点，和新元素开始节点比较，如果找到了，则返回该节点在老元素中的索引。

总结一下：

首先在对比节点是否相同时，会先对比key值，key值不相同，不再对比之后的逻辑，提高了性能。

其次在更新时会生成一个以key值为属性，index为值的map对象。当新开始节点有key值，更新时可通过key值直接找到相应的节点的索引，如果没有则会通过遍历找到对应的节点的索引，返回给idxInOld。**通过key值直接找到元素索引比通过遍历找更快，也提高了性能。**

1.23 你怎么理解vue中的diff算法?

7组：

1. 什么是diff算法 diff算法直观来说是在virtual DOM发生变化的时候和真实DOM进行比较，找出不一样的地方进行替换和更新的一种算法
2. diff算法的具体作用是什么 我们先根据真实DOM生成一颗virtual DOM，当virtual DOM某个节点的数据改变后会生成一个新的Vnode，然后Vnode和oldVnode作对比，发现有不一样的地方就直接修改在真实的DOM上，然后使oldVnode的值为Vnode。 diff的过程就是调用名为patch的函数，比较新旧节点，一边比较一边给真实的DOM打补丁。
3. diff算法是如何进行比较的 上面提到diff算法主要是调用patch函数，所以我们从patch函数入手进行比较，直接上源码

```
// src\core\vdom\patch.js patch函数 开始打补丁

// 如果不是真实DOM并且是相同的Vnode
if (!isRealElement && sameVnode(oldVnode, vnode)) {
    // patch existing root node
    patchVnode(oldVnode, vnode, insertedVnodeQueue, null, null,
removeOnly)
}

// 进入patchVnode函数 开始对比新老Vnode
// 新旧虚拟dom子节点
const oldCh = oldVnode.children
const ch = vnode.children
// 是否是元素节点
if (isUndef(vnode.text)) {
  // 是元素节点
  if (isDef(oldCh) && isDef(ch)) { // 都有子节点 diff对比子节点
    // 旧的子节点和新的子节点不相同 开始进行子节点比较
    if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue,
removeOnly)
  } else if (isDef(ch)) { // 只有新vnode有子节点 往旧vnode插入子节点
    if (process.env.NODE_ENV !== 'production') {
      checkDuplicateKeys(ch)
    }
    if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
    addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
  } else if (isDef(oldCh)) { // 旧vnode有子节点 干掉上面的子节点
    removeVnodes(oldCh, 0, oldCh.length - 1)
  } else if (isDef(oldVnode.text)) { // 旧vnode有文本节点 直接赋值
    nodeOps.setTextContent(elm, '')
  }
}

// 对比子节点Vnode，进入updateChildren函数
// 这段就是diff算法的核心代码，主要用于循环比较Vnode节点进行打补丁
```

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  if (isUndef(oldStartVnode)) {
    oldStartVnode = oldCh[++oldStartIdx] // Vnode has been moved left
  } else if (isUndef(oldEndVnode)) {
    oldEndVnode = oldCh[--oldEndIdx]
  } else if (sameVnode(oldStartVnode, newStartVnode)) {
    patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue, newCh,
newStartIdx)
    oldStartVnode = oldCh[++oldStartIdx]
    newStartVnode = newCh[++newStartIdx]
  } else if (sameVnode(oldEndVnode, newEndVnode)) {
    patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue, newCh,
newEndIdx)
    oldEndVnode = oldCh[--oldEndIdx]
    newEndVnode = newCh[--newEndIdx]
  } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right
    patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue, newCh,
newEndIdx)
    canMove && nodeOps.insertBefore(parentElm, oldStartVnode.elm,
nodeOps.nextSibling(oldEndVnode.elm))
    oldStartVnode = oldCh[++oldStartIdx]
    newEndVnode = newCh[--newEndIdx]
  } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left
    patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue, newCh,
newStartIdx)
    canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm,
oldStartVnode.elm)
    oldEndVnode = oldCh[--oldEndIdx]
    newStartVnode = newCh[++newStartIdx]
  } else {
    if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh,
oldStartIdx, oldEndIdx)
    idxInOld = isDef(newStartVnode.key)
      ? oldKeyToIdx[newStartVnode.key]
      : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)
    if (isUndef(idxInOld)) { // New element
      createElm(newStartVnode, insertedVnodeQueue, parentElm,
oldStartVnode.elm, false, newCh, newStartIdx)
    } else {
      vnodeToMove = oldCh[idxInOld]
      if (sameVnode(vnodeToMove, newStartVnode)) {
        patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue, newCh,
newStartIdx)
        oldCh[idxInOld] = undefined
        canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm,
oldStartVnode.elm)
      } else {
        // same key but different element. treat as new element
```

```
        createElm(newStartVnode, insertedVnodeQueue, parentElm,
oldStartVnode.elm, false, newCh, newStartIdx)
      }
    }
    newStartVnode = newCh[++newStartIdx]
  }
}
```

通过源码，我们可以得出以下diff的工作原理：

1. 首先判断有没有oldVnode mount挂载时 界面中是没有vnode 直接调用createElm不走diff算法 这里可理解为初始化生成真实dom树 之后替换掉界面中存在的dom
2. 再次渲染时首先判断oldVnode是否是真实dom 是的话将转化为虚拟dom 即vnode
3. 如果oldVnode是虚拟dom 并且和newVnode key值等相同 即同一个vnode 则进入vnode patch比较 调用patchVnode
4. patchVnode执行时正式进入新旧vnode对比模式 即diff 并且在diff前会先判断文本节点还是元素节点 文本节点直接diff 元素节点会找有没有子节点 有的话对比子节点
    1. 如果新旧vnode均有子节点 则调用updateChildren() 进行对比计算
    2. 如果只有新vnode有子节点 则在旧vnode中追加
    3. 如果只有旧vnode有子节点 则直接干掉这些节点
5. 进入updateChildren后新旧vnode头尾各创建两个指针 newStartIndex newEndIndex oldStartIndex oldEndIndex
6. 开始while循环 当newStartIndex > newEndIndex || oldStartIndex > oldEndIndex 跳出循环 以下以ov代替oldVnode nv代替newNode nsi nei osi oei 代替指针
    1. ov[osi] === nv[nsi] 直接替换的vnode osi nsi 指针后移
    2. ov[oei] === nv[nsi] ov[oei] 位置移到最前 替换 osi nsi 指针后移
    3. ov[osi] === nv[nei] ov[osi] 位置一到最后 替换 nei oei 指针前移
    4. ov[oei] === nv[nei] 直接替换的vnode nei oei 指针前移
    5. 如果以上四种基础情况均未找到 只能开始遍历nv节点循环查找ov节点 ovi nvi 代表找到的节点
        1. ovi === nvi 将nvi移到对前面 替换 osi nsi 指针后移
        2. ovi 没有找到 插入到nvi第一个 osi nsi 指针后移
7. 跳出while之后 对比nsi nei osi oei
    1. nsi < nei nv比较长 将最后的vnode 追加到 ov
    2. 反之 osi < oei ov比较长 将最后的vnode干掉

1.27 谈谈你对MVC、MVP和MVVM的理解?

5组：

什么是MVC?

MVC全称**Model View Controller** 是由数据模型，视图，控制器组成以数据驱动视图的一个架构模式。

通信关系

- model 可以与view和controller通信
- view与controller通信
- controller与model和view之间都可通信

工作模式

在view中把事件传递给controller 而在controller中去通知model更新数据 model更新数据完成后再来告诉view更新视图

什么是MVP?

MVP全称**Model View Presenter** 是基于MVC进化的一个架构模式。

通信关系

- model与Presenter通信
- view与Presenter通信
- Presenter与model和view之间通信

工作模式

在MVP中model和view之间互不通信都由presenter来通知视图和数据的更新，如果视图发生变化例如输入框有输入 view会通知presente 之后有presenter来告诉model更新数据再更新视图

什么是MVVM?

MVVM全称Model View ViewModel vue就是一个典型的MVVM模式框架。

通信关系

- model与ViewModel通信
- view与ViewModel通信
- ViewModel与model和view之间通信

工作模式

MVVM中实现了双向数据绑定思想，MVVM把View和Model的同步逻辑自动化了。以前Presenter负责的View和Model同步不再手动地进行操作，而是交给框架所提供的数据绑定功能进行负责，只需要告诉它View显示的数据对应的是Model哪一部分即可。

1.28 谈谈你对vue组件之间通信的理解?

5组：

vue组件通信的几种方式

父子组件之间通信

- props / `$emit`

- 父传子：父组件在子组件上挂载属性并携带数据 子组件中使用props接收子传父：父组件在子组件上注册监听一个自定义事件 子组件中通过`$emit`触发
- `$parent` / `$children`

  子组件中通过`$parent`可直接访问父组件实例

  父组件中通过`$children`访问子组件实例

子代通信

- provide / inject

  依赖注入：实现子代组件通信 由父组件使用provide方法提供依赖 子代组件使用inject注入
- `$attrs` / `$listeners`

  `$attrs`: 除了class和style之外未经prop声明 即传递的属性都会放在 `$attrs`里面

  `$listeners`: 继承父组件的事件 可以通过v-on="$listeners" 传入内部组件

任何组件之间通信

- `$on` / `$emit`可以用这种方式实现 `$bus`事件总线模式达到随意两个组件之间的通信
- vuexvue中用于管理全局状态的一个插件

应用场景

组件封装

在组件的封装中由于不能依赖除vue官方提供的组件通信方式之外的插件 如vuex等所以通常可以使用

provide / inject ，$parent$/children，props / $emit的方式来通信

大型项目

一般在组件过多结构庞大时使用vuex来进行全局的数据状态管理

12组：

组件是Vue里最强大的功能之一。而组件实例的作用域是相互独立的，这就意味着不同组件之间的数据是无法相互引用的。

一般来说组件之间的关系有父子关系、兄弟关系、隔代关系。

1. 父组件向子组件传值

   父组件通过props向子组件传值
2. 子组件向父组件传值

   通过事件形式子组件通过events给父组件发消息，其实就是子组件将自己的数据发送给父组件。

   `$emit` / `$on`

   这种方法通过一个空的Vue实例作为中央事件总线（事件中心），用它来触发和监听事件，巧妙而轻量的实现了任何组件间的通信，包括父子、兄弟、跨级。
3. vuex

vuex实现了一个单项数据流，在全局拥有一个State存放数据，当组件中想要修改state时必须通过Mutation进行，Mutation同时提供了订阅者模式供外部调用获取state数据的更新。而当所有异步操作（常见于调用后端接口异步获取更新数据）或批量的同步操作需要走Action，但action也是无法直接修改State的，还是需要通过Mutation来修改State的数据。

4. `$attrs` / `$listeners`

   多级组件嵌套需要传导数据时，通常使用的方法是vuex，但是如果仅仅是传递数据而不做中间处理，使用vuex处理，未免有一些大材小用，为此Vue2.4版本提供了以一种方法：

   `$attrs`: 包含了父作用域中不被props所识别且获取的特定绑定（除了style和class）。当一个组件中没有生命任何props时，`$attrs`会包含所有父作用域的绑定（除了style和class），并可以通过v-bind="`$attrs`"传入内部组件, 通常配合inheritAttrs一起使用。

   `$listerners`: 包含了父作用域中的（不包含.native）事件v-on监听器。可以通过v-on="`$listeners`"传入组件内部。

   简单来说：`$attrs`与`$listeners`是两个对象，`$attrs`是父组件里绑定的非props的属性，$listeners是父组件里绑定的非原生事件。

5. provide / inject

   Vue2.2.0里新增的API，这对选项需要一起使用，以允许一个祖先组件向所有子孙组件注入一个依赖，不论组件层次有多深，并在组件上下游的关系里始终成立。祖先组件通过provider来提供变量，然后在子孙组件中使用inject来注入变量。provide / injectAPI主要解决的了跨级通信问题不过它的使用场景，主要是子组件获取上级组件的状态，跨级组件间建立了一种主动提供与依赖注入的关系。

6. parent、`$children`和ref

   ref: 如果在普通的dom元素上使用，引用指向的就是dom元素；如果用在子组件上，引用指向的就是组件实例。

   `$parent`: 指向父组件实例。

   `$childred`: 指向子组件实例

1.29 你了解哪些vue性能优化方法

4组：

一、代码层面优化

1.v-if和v-show的区分使用：v-if是条件渲染，只有在渲染条件为真时才会渲染条件块，如初始渲染条件为假时，则什么也不做，而v-show无论初始条件是什么都会渲染，且是给予css的display属性进行切换；因此v-if适用于很少改变条件，不需要频繁切换条件的场景；v-show则适用于频繁切换条件的场景。

2.computed和watch区分使用场景：computed是计算属性且computed的值有缓存，只有它依赖的属性值发生改变，下一次获取computed的值时才会重新计算；watch类似于数据监听，每当监听的数据变化时都会执行回调后的操作；当需要进行数值计算，并依赖其他数据时应该使用computed， 这样可以利用computed的缓存属性避免重新计算，当需要在数据变化时执行异步或开销较大的操作时，应该使用watch。

3.v-for遍历必须为item添加key，且避免同时使用v-if：在列表循环时为每项item设置唯一的key可以方便vue精确找到列表数据，当state更新时，新的状态和旧的状态对比，快速定位到diff；由于v-for的 优先级高于v-if，因此避免同时使用，以免每次循环需要去遍历整个数组，可使用computed属性或把v-if放在循环的父元素上。

4.长列表性能优化：vue会通过Object.defineProperty对数据进行劫持来实现视图响应数据的变化，当仅作为展示，无需数据变动时可以禁止vue劫持这些数据，可以通过Object.freeze的方法来冻结一个对象。

5.图片资源懒加载：对于图片过多的页面，为了加速页面加载速度，所以很多时候我们需要将页面内未出现在可视区域内的图片先不做加载， 等到滚动到可视区域后再去加载。

二、Webpack层优化

1.Webpack对图片进行压缩：在vue项目中除了可以在webpack.base.conf.js中url-loader中设置limit大小来对图片处理对小于limit的图片转化为base64格式，其余不做操作。 所有有些较大的图片资源，在请求资源的时候，加载会很慢，我们可以用 image-webpack-loader来压缩图片。

2.减少ES6转化为ES5的冗余代码：Bable插件会将ES6代码转换成ES5代码时注入一些辅助函数，，如果多个源代码文件都依赖这些辅助函数，那么这些辅助函数的代码将会出现很多次，造成代码冗余

1.30 你知道vue3有哪些新特性吗？它们会带来什么影响

12组：

Vue3新特性大致可以分为：更快、更小、更容易维护、更多的原生支持、更易于开发使用。

Vue3的出现将大规模引爆TypeScript的使用，会改变现有的Vue应用开发方式，开发更加灵活，也会吸引越来越多的开发者加入Vue的阵容。

更快

1.虚拟dom重写，mounting和patching的速度提高100%。

2.更多的编译时的提示来减少运行时的开销。

3.组件快速路径 + 单个调用 + 子类型检查

跳过不必要的条件分支

JS引擎更容易优化

4.优化插槽的生成

确保实例正确的跟踪依赖关系

避免不必要的父子组件重新渲染

5.静态树提升

跳过修补整棵树，从而降低渲染成本

即使多次出现也能正常工作

6.静态属性提升

跳过不会修改节点的修补过程，但是它的子组件会保持修补过程

7.内联的事件提升

避免因为不同的内联函数标识导致的不必要的重新渲染

8.基于Proxy的观察者机制，全语言覆盖 + 更好的性能

目前Vue使用的是Object.defineProperty的getter和setter

组件实例化的速度提升100%

使用Proxy节省了以前一半的内存开销，加快速度，但是存在低版本的浏览器不兼容问题

为了继续支持IE11，Vue3将发布一个支持就观察者机制和新Proxy版本的构建

更小

1.更友好的tree-shaking

2.新的core-runtime压缩后大概10KB更易于维护1.Flow -> TypeScript2.包的解耦

3.编译器重写：可插拔的架构；提供更强大的IDE支持来作为基础设施。

更多的原生支持

运行时内核也将与平台无关，使Vue可以更容易的与任何平台（Web、Android、IOS）一起使用。

更易于开发使用

1.暴露响应式API import { observable, effect } from 'vue'

2.轻松识别组件重新渲染的原因

3.提供对TypeScript的支持

4.更友好的warning traces

现在包括功能组件o可检查的props

在更多的警告中提供可用的traces

实验性的Hooks API

类似react hook的API

实验性的 Time Slicing支持

当许多组件同时尝试重新渲染时，浏览器会变的很慢，利用Time Slicing将JS执行分为几部分，此时，用户的交互不会被阻塞

1.31 vue如果想扩展某个现有的组件时应该怎么做?

14组：

1. 使用extends直接扩展官方文档
2. 使用Vue.mixin全局混入缺点：1.我们必须要知道改造组件的内部结构，需要知道原先组件内部方

法名，及组件内部属性。2.两个组件有很强的依赖性，如果是嵌套加嵌套，代码就很难去追寻本源，多人开发可能产生混乱。

3. HOC封装所谓高阶组件其实就是高阶函数，React 和 Vue 都证明了一件事儿：一个函数就是一个组件。所以组件是函数这个命题成立了，那高阶组件很自然的就是高阶函数，即一个返回函数的函数参考1：为何在React中推荐使用HOC，而不是mixins来实现组件复用。但在Vue中，很少有HOC的尝试？ 参考2：奇技淫巧 - Vue Mixins 高级组件 与 Vue HOC 高阶组件 实践
4. 加slot扩展

2.1 watch和computed的区别以及怎么选用?

5组：

1. computed 创建新的属性， watch 监听 data 已有的属性；
2. computed 会产生依赖缓存；
3. 当 watch 监听 computed 时，watch 在这种情况下无效，仅会触发 computed.setter。
    - watch和computed各自擅长处理的数据关系场景不同：
    1. watch：监听某个数据的变化，一个数据影响多个数据。
    2. computed：需要复杂逻辑和运算才能得到的值，或某些属性依赖其他数据的结果，那么建议将这个属性放在computed中，一个数据受多个数据影响。

2.2 谈谈你对vue生命周期的理解?

12组：

Vue生命周期就是Vue组件实例从创建到销毁的过程。

首先要创建一个Vue实例，new Vue()，会执行Vue的构造函数，在构造函数里执行了*init()方法，在*init方法里会先执行：

initLifecycle() // 声明`$parent`、`$root`、`$children`、`$refs`

initEvents() // 对父组件传入的事件和回调添加监听

initRender() // 声明`$slots`、`$createElement()`

调用beforeCreate()钩子

然后再执行：

initInjections() // 注入数据

initstate() // 数据响应式

initProvide()

调用created()钩子，这时数据已经初始化。

当created()完成之后，会判断instance（实例）里面是否包含el选项，如果存在el选项，准备挂载。首先会判断是否存在render选项（优先级：render > template > el），如果不存在render选项，则将template选项解析为渲染函数，如果template也不存在，则template = getOuterHTML(el)，然后再将其解析为渲染函数，主要是为了得到渲染函数。

调用beforeMount()钩子

执行render函数得到VNode，再执行patch得到真正的`$el`，`$el`真正初始化。

调用mounted()钩子

beforeMount之时，`$el`还只是我们在HTML里写的节点，然后到mounted的时候，它就把渲染出来的内容挂载到了DOM节点上。

当有数据变化时，会调用组件对应watcher的update方法将该watcher加入nextTick将要刷新的watcher队列里。在执行watcher的run方法（执行patch，刷新组件）之前会先调用beforeUpdate()钩子。

清空本次微任务里的watcher队列后，会调用updated()钩子。

在组建销毁之前，会先调用beforeDestroy()钩子，然后找到其父节点，将自身从父节点之上移除，将和其相关的watcher全部卸载、移除ob，更新dom，调用destroyed()钩子，卸载监听器，释放父级。

13组：从源码我们可以知道，在new Vue（）创建实例的过程中，在src\core\instance\init.js的文件中会依次执行下方代码

```
initLifecycle(vm) //声明
$parent,$root,$children,$refsinitEvents(vm) //对父组件传入的事件和回调添加监听
initRender(vm) //声明
$slots,$createElement()callHook(vm, 'beforeCreate') //调用beforeCreate钩子
initInjections(vm) // resolve injections before data/props 注入数据
initState(vm) //数据的初始化，响应式
initProvide(vm) // resolve provide after data/props 提供数据
callHook(vm, 'created')
```

在initState(vm)的时候，进行数据初始化，实现数据响应式，以及对computed（计算属性）、watch侦听器初始化，在此之前data里的数据和methods里的方法无法操作，最早只能在created中操作

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

判断是否有el属性，如果有就调用`$mount`，而`$mount`中主要是调用mountComponent，源码如下

```
//实现$mount
Vue.prototype.$mount = function (el?: string | Element,hydrating?: boolean):
Component {el = el && inBrowser ? query(el) : undefined

                    //初始化，将首次渲染的结果替换elreturn

                    mountComponent(this, el, hydrating)}
```

mountComponent中，相关的render 函数首次被调用，有了render函数后再调用beforeMount，最后调用mounted

```
export function mountComponent (vm: Component,el: ?Element,hydrating?:
boolean): Component {
  vm.$el = el
  if (!vm.$options.render) {
    vm.$options.render = createEmptyVNode}callHook(vm, 'beforeMount')
  let updateComponentupdateComponent = ()=>{
    vm._update(vm._render(), hydrating)
  }
  // we set this to vm._watcher inside the watcher's constructor
  // since the watcher's initial patch may call $forceUpdate (e.g. inside
child// component's mounted hook), which relies on vm._watcher being already
definednew
  Watcher(vm, updateComponent, noop, {
    before () {
      if (vm._isMounted && !vm._isDestroyed) {
        ? callHook(vm, 'beforeUpdate')
      }}}, true /* isRenderWatcher */)
  hydrating = false
  // manually mounted instance, call mounted on self// mounted is called for
render-created child components in its inserted hook
  if (vm.$vnode == null) {
    vm._isMounted = truecallHook(vm, 'mounted')
  }
 return vm}
```

以上四个钩子函数在初始化时被调用，其他钩子的调用需要外部的触发才能执行。

当有数据的变化，会调用beforeUpdate，然后经过Virtual DOM，最后updated更新完毕。

当组件被销毁的时候，它会调用beforeDestory，以及destoryed。activated、deactivated是和vue中一个原生的组件keep-alive有关系,当keep-alive组件激活时调用activated钩子。keep-alive 组件停用时调用deactivated钩子钩子errorCaptured是当捕获一个来自子孙组件的错误时被调用。此钩子会收到三个参数：错误对象、发生错误的组件实例以及一个包含错误来源信息的字符串。此钩子可以返回 false 以阻止该错误继续向上传播。

2.3 谈谈你对vuex使用及其理解?

13组：

首先数据从初始化data开始，使用observe监控数据；

给每个数据属性添加dep，并且在它的getter过程添加收集依赖操作，在setter过程添加通知更新的操作；

在解析指令或者给vue实例设置watch选项或者调用$watch时，生成对应的watcher并收集依赖。

之后，如果数据触发更新，会通知watcher，wacther在重新收集依赖之后，触发模板视图更新。 里面的data；存放数据方式为响应式，vue组件从store中读取数据，如数据发生变化，组件也会对应的更新。2、getter：可以认为是 store 的计算属性，它的返回值会根据它的依赖被缓存起来，且只有当它的依赖值发生了改变才会被重新计算。3、mutation：更改 Vuex 的 store 中的状态的唯一方法是提交

mutation。4、action：包含任意异步操作，通过提交 mutation 间接更变状态。5、module：将 store 分割成模块，每个模块都具有state、mutation、action、getter、甚至是嵌套子模块。当组件进行数据修改的时候我们需要调用dispatch来触发actions里面的方法。actions里面的每个方法中都会有一个commit方法，当方法执行的时候会通过commit来触发mutations里面的方法进行数据的修改。mutations里面的每个函数都会有一个state参数，这样就可以在mutations里面进行state的数据修改，当数据修改完毕后，会传导给页面。页面的数据也会发生改变。

2.4 你知道nextTick的原理吗?

4组：

异步执行 （宏任务，微任务）

Vue 在更新 DOM 时是异步执行的。 只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。 如果同一个 watcher 被多次触发，只会被推入到队列中一次。 这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。 然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。 Vue 在内部对异步队列尝试使用原生的 Promise.then、MutationObserver 和 setImmediate， 如果执行环境不支持，则会采用 setTimeout(fn, 0) 代替。

next-tick.js 对外暴露了nextTick这一个参数，所以每次调用Vue.nextTick时会执行：

把传入的回调函数cb压入callbacks数组

执行timerFunc函数，延迟调用 flushCallbacks 函数

遍历执行 callbacks 数组中的所有函数

这里的 callbacks 没有直接在 nextTick 中执行回调函数的原因是保证在同一个 tick 内多次执行 nextTick，不会开启多个异步任务，而是把这些异步任务都压成一个同步任务，在下一个 tick 执行完毕。

5组：

nextTick 实现主要分为三个部分：

1.首先 nextTick 把传入的 cb 回调函数用 try-catch 包裹后放在一个匿名函数中推入callbacks数组中，这么做是因为防止单个 cb 如果执行错误不至于让整个JS线程挂掉，每个 cb 都包裹是防止这些回调函数如果执行错误不会相互影响，比如前一个抛错了后一个仍然可以执行。

2.然后检查 pending 状态，这个跟之前介绍的 queueWatcher 中的 waiting 是一个意思，它是一个标记位，一开始是 false 在进入 macroTimerFunc、microTimerFunc 方法前被置为 true，因此下次调用 nextTick 就不会进入 macroTimerFunc、microTimerFunc 方法，这两个方法中会在下一个 macro/micro tick 时候 flushCallbacks 异步的去执行callbacks队列中收集的任务，而 flushCallbacks 方法在执行一开始会把 pending 置 false，因此下一次调用 nextTick 时候又能开启新一轮的 macroTimerFunc、microTimerFunc，这样就形成了vue中的 event loop。

3.最后检查是否传入了 cb，因为 nextTick 还支持Promise化的调用：nextTick().then(() => {})，所以如果没有传入 cb 就直接return了一个Promise实例，并且把resolve传递给_resolve，这样后者执行的时候就跳到我们调用的时候传递进 then 的方法中。

12组：提到DOM的更新是异步执行的，只要数据发生变化，将会开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。如果同一个 watcher 被多次触发，只会被推入到队列中一次。

简单来说，就是当数据发生变化时，视图不会立即更新，而是等到同一事件循环中所有数据变化完成之后，再统一更新视图。

nextTick定义如下：

1.首先判断当前运行环境是否支持Promise，如果支持则使用之。

2.如果当前运行环境不支持Promise，再判断当前环境是否支持MutationObserver，如果支持则使用之。

3.如果当前环境不支持MutationObserver，再判断当前运行环境是否支持setImmediate，如果支持则使用之。

4.如果当前环境不支持setImmediate，则使用setTimeout。

2.5你知道vue的双向数据绑定的原理吗?

4组：

实现一个数据监听器Observer，能够对数据对象的所有属性进行监听，如有变动可拿到最新值并通知订阅者

实现一个指令解析器Compile，对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相应的更新函数

实现一个Watcher，作为连接Observer和Compile的桥梁，能够订阅并收到每个属性变动的通知，执行指令绑定的相应回调函数，从而更新视图

实现Observer

利用Obeject.defineProperty()来监听属性变动。那么将需要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。

那么监听到变化之后就是怎么通知订阅者了，所以接下来我们需要实现一个消息订阅器。维护一个数组，用来收集订阅者，数据变动触发notify，再调用订阅者的update方法。

实现Compile

compile主要做的事情是解析模板指令，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。

实现Watcher

Watcher订阅者作为Observer和Compile之间通信的桥梁，主要做的事情是:

在自身实例化时往属性订阅器(dep)里面添加自己

自身必须有一个update()方法

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。

2.6 vue-router导航钩子有哪些?

12组：

全局前置守卫

const router = new VueRouter({ ... })

router.beforeEach((to, from, next) => {

// ...

})

可以使用router.beforeEach(to, from, next)注册一个全局前置守卫，当一个导航被触发时，全局前置守卫按照创建顺序调用，守卫是异步解析执行，此时导航在所有守卫resolve之前一直处于等待中。

全局解析守卫

可以使用router.beforeResolve()注册一个全局解析守卫，这和router.beforeEach()类似，区别在于导航被确认之前，同时在所有组件内守卫和异步路由组件被解析之后，守卫就被调用。

全局后置钩子

可以注册全局后置钩子，然而和守卫不同的是，这些钩子不会接受 next 函数也不会改变导航本身：

router.afterEach((to, from) => {

// ...

})

路由独享的守卫

可以在路由配置上直接定义beforeEnter守卫

const router = new VueRouter({

routes: [{

path: '/foo',

component: Foo,

beforeEnter: (to, from, next) => {

// ...

}

}

]})

组件内的守卫

可以再路由组件内直接定义一下路由导航守卫：

beforeRouteEnter

beforeRouteUpdate

beforeRouteLeave

template: `...`,

beforeRouteEnter (to, from, next) {

// 在渲染该组件的对应路由被 confirm 前调用

// 不！能！获取组件实例 `this`

// 因为当守卫执行前，组件实例还没被创建

},

beforeRouteUpdate (to, from, next) {

// 在当前路由改变，但是该组件被复用时调用

// 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，

// 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。

// 可以访问组件实例 `this`

},

beforeRouteLeave (to, from, next) {

// 导航离开该组件的对应路由时调用

// 可以访问组件实例 `this`

}

}

beforeRouteEnter守卫不能访问this，因为守卫在导航确认前被调用，因此组件还没有被创建
beforeRouteLeave 守卫通常用来禁止用户在为保存修改前突然离开，该导航可以通过next(false)来取消

beforeRouteLeave (to, from, next) {

const answer = window.confirm('Do you really want to leave? you have unsaved changes!')

if (answer) {

next()

}else {

next(false)

}

}

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离开守卫beforeRouteLeave

3.调用全局的beforeEach守卫

4.在重用的组件里调用beforeRouteUpdate守卫

5.在路由配置里调用beforeEnter守卫

6.解析异步路由组件

7.在被激活的组件里调用beforeRouteEnter守卫

8.调用全局的beforeResolve守卫

9.导航被确认

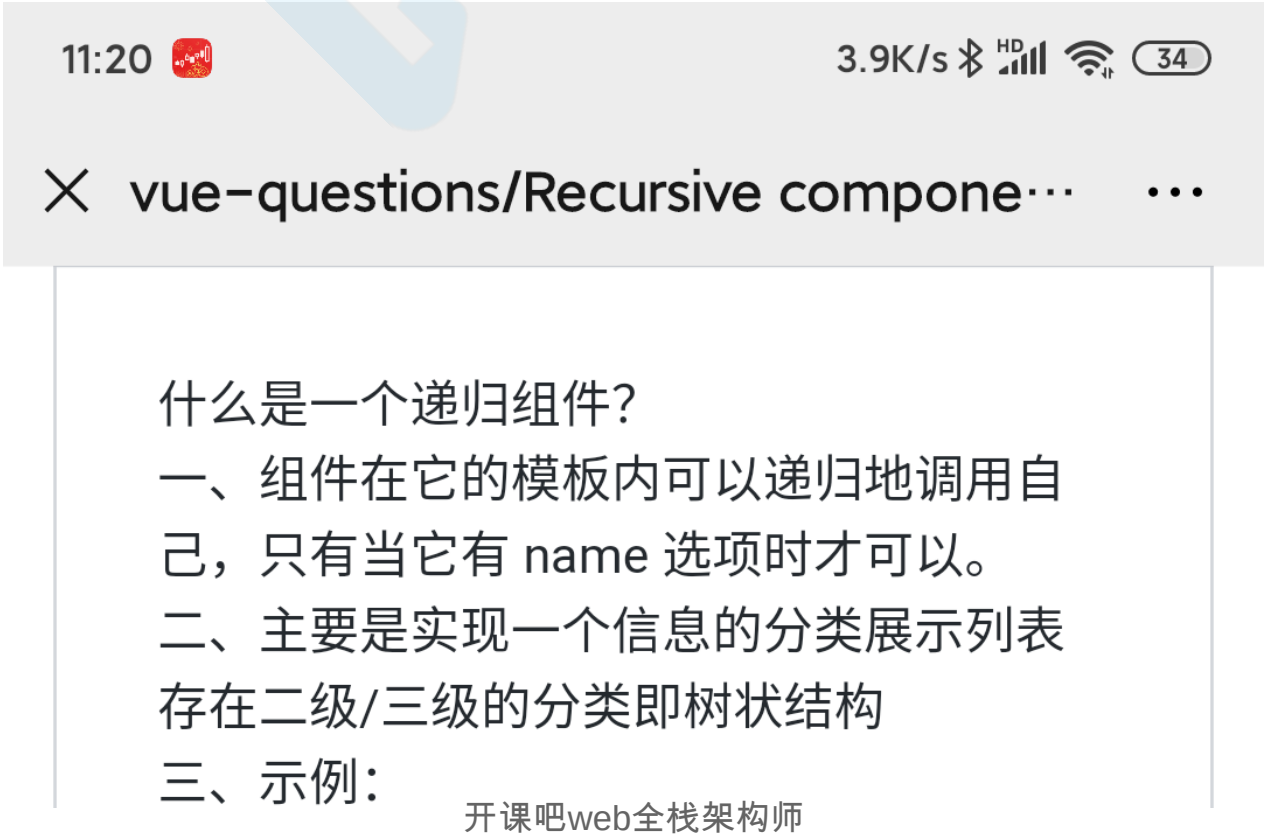
10.调用全局的afterEach钩子

11.触发DOM更新

12.用创建好的实例调用beforeRouteEnter守卫里传给next的回调函数

2.7 什么是一个递归组件?

4组:



首先创建一个List的递归组件

```
<template>
  <div>
    <div class="list-item v-for="(i
      <div class="item-name>
        <span>{{item.name}}</span>
      </div>
      <div v-if="item.chilren" clas
        <list :list="item.children>
      </div>
    </div>
  </div>
</template>
<script>
  export default {
    name: "List",
    props:{list:Arrar}
  }
</script>
```

注意上面的代码我们用了List组件本身，完成后我们在外部父组件中使用List组件时，不管数据有多少嵌套关系，都可以实现自适应加载而不需要去层层嵌套了。

```
<template>
  <div class="list-detail">
    <list :list="list"></list>
  </div>
```

```
</template>
<script>
import List from "./components/list
export default {
  name: "parent",
  components: {List},
  data () {
    return {
      list:[{
        name:"经济",
        children:[{
          name:"如家",
          children:[{
            name:"上江路-如家"
          },
          {
            name:"望江路-如家"
          }]
        },
        {
          name: "7天",
          children: [{
            name: "长江路-7天"
          },
          {
            name: "望江路-7天"
          }]
      }]
    }
  }
}
</script>
```

类似与信息分类的展示我们都可以用递归组件解决。

2.8 谈一谈你对vue响应式原理的理解?

12组：

vue.js采用数据劫持结合发布者-订阅者的方式，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。

具体的来讲，Vue.js通过Directives指令去对DOM做封装，当数据发生变化，会通知指令去修改对应的DOM，数据驱动DOM的变化。vue.js还会对操作做一些监听（DOM Listener），当我们修改视图的时候，vue.js监听到这些变化，从而改变数据。这样就形成了数据的双向绑定。

具体步骤如下：

1.首先，需要对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 getter。这样的话，给这个对象的某个属性赋值，就会触发setter，那么就能监听到数据变化。（其实是通过Object.defineProperty()实现监听数据变化的）

2.然后，需要compile解析模板指令，将模板中的变量替换成数据，接着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，订阅者收到通知，就会更新视图

3.接着，Watcher订阅者是Observer和Compile之间通信的桥梁，主要负责：

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，就调用自身的update()方法，并触发Compile中绑定的回调

4.最后，viewmodel(vue实例对象)作为数据绑定的入口，整合Observer、Compile、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 (ViewModel)-》视图更新(view)；视图变化(view)-》数据(ViewModel)变更的双向绑定效果。