Vue3开发与实战

> **千锋精品教程，好学得不像实力派！**

一.vue3介绍

1. 官网初识

> Vue (发音为 /vjuː/，类似 **view**) 是一款用于构建用户界面的 JavaScript 框架。它基于标准 HTML、CSS 和 JavaScript 构建，并提供了一套声明式的、组件化的编程模型，帮助你高效地开发用户界面。无论是简单还是复杂的界面，Vue 都可以胜任。
>
> https://cn.vuejs.org/

2.环境搭建

2-1 线上尝试

- 想要快速体验 Vue，你可以直接试试我们的 演练场。

2-2 CDN使用

```
<script
src="https://unpkg.com/vue
@3/dist/vue.global.js">
</script>
```

通过 CDN 使用 Vue 时，不涉及“构建步骤”。这使得设置更加简单，并且可以用于增强静态的 HTML 或与后端框架集成。但是，你将无法使用单文件组件 (SFC) 语法。

2-3 Vue CLI

Vue CLI 是官方提供的基于 Webpack 的 Vue 工具链，它现在处于维护模式。我们建议使用 Vite 开始新的项目，除非你依赖特定的 Webpack 的特性。在大多数情况下，Vite 将提供更优秀的开发体验。

2-4 Vite

Vite 是一个轻量级的、速度极快的构建工具，对 Vue SFC 提供第一优先级支持。作者是尤雨溪，同时也是 Vue 的作者！

要使用 Vite 来创建一个 Vue 项目，非常简单：

```
$ npm init vue@latest
```

这个命令会安装和执行 create-vue，它是 Vue 提供的官方脚手架工具。跟随命令行的提示继续操作即可。

二.vue3基础

1.模板语法

1-1 我的第一个vue应用

```
<div id="box">
  {{10+20}}
  {{name}}
</div>
<script>
  var app = Vue.createApp({
    data(){
      return {
        name:"kerwin"
      }
    }
  }).mount("#box")
</script>
```

- 推荐使用的 IDE 是 VSCode，配合 Vue 语言特性 (Volar) 插件。该插件提供了语法高亮、TypeScript 支持，以及模板内表达式与组件 props 的智能提示。
- Volar 取代了我们之前为 Vue 2 提供的官方 VSCode 扩展 Vetur。如果你之前已经安装了 Vetur，请确保在 Vue 3 的项目中禁用它。

1-2 应用背后的真相

（1）Object.defineProperty

```
var obj = {}
var obox = document.getElementById("box")
Object.defineProperty(obj,"myname",{
  get(){
    console.log("有人访问了")
    return obox.innerHTML
  },
  set(value){
    console.log("有人改变我了",value)
    obox.innerHTML = value
  }
})

/*
    缺陷：无法监听数组的改变， 无法监听class改变， 无法监听Map Set结构。
```

```
    */
```

（2）

```
var obj = {

}
var obox = document.getElementById("box")

var vm = new Proxy(obj, {
    get(target, key) {
        console.log("get")
        return target[key]
    },
    set(target, key, value) {
        console.log("set")

        target[key] = value
        obox.innerHTML = value
    }
})


/*
        vue3 基于Proxy ,ES6 Proxy ,
         if(支持proxy){
            // proxy进行拦截处理， 实现功能
         }else{
            // object.defineProtery
         }

    */
```

1-3 模板语法-新的皮肤来了

（1）最基本的数据绑定形式是文本插值，它使用的是“Mustache”语法 (即双大括号):

```
<span>Message: {{ msg }}</span>
```

双大括号标签会被替换为相应组件实例中 `msg` 属性的值。同时每次 `msg` 属性更改时它也会同步更新。

（2）双大括号不能在 HTML attributes 中使用。想要响应式地绑定一个 attribute，应该使用 `v-bind` 指令:

template

```
<div v-bind:id="dynamicId"></div>
```

`v-bind` 指令指示 Vue 将元素的 `id` attribute 与组件的 `dynamicId` 属性保持一致。如果绑定的值是 `null` 或者 `undefined`，那么该 attribute 将会从渲染的元素上移除。

（3）表达式的支持

```
{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div :id="`list-${id}`"></div>
```

（4）指令

```
<a v-on:click="doSomething"> ... </a>

<!-- 简写 -->
<a @click="doSomething"> ... </a>
```

1-4 Todolist-来个案例

```
<!--
 * @作者: kerwin
-->
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src="vue.js"></script>
</head>

<body>
    <div id="box">

        <input type="text" v-model="mytext">
        <button @click="handleAdd">add</button>
        <ul>
            <li v-for="data,index in datalist">
                {{data}}
                <button @click="handleDel(index)">del</button>
            </li>
        </ul>
    </div>
    <script>
        var obj = {

            data() {
```

```
            return {
                mytext: "",
                datalist:["11","22","33"],
            }
        },
        methods: {
            handleAdd(ev){
                this.datalist.push(this.mytext)
            },
            handleDel(index){
                this.datalist.splice(index,1)
            }
        }
    }
    var app = Vue.createApp(obj).mount("#box")
  </script>
</body>
</html>
```

1-5 点击变心案例 - 是变色

```
<!--
 * @作者: kerwin
-->
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="vue.js"></script>
  <style>
    .active{
      background-color: red;
    }
  </style>
</head>

<body>
  <div id="box">
    <ul>
      <li v-for="data,index in datalist" @click="handleclick(index)" :class="current===index?'active':''">
        {{data}}
      </li>
    </ul>
  </div>
  <script>
    var obj = {
      data() {
```

```
            return {
                datalist:["11","22","33"],
                current:0
            }
        },
        methods: {
            handleclick(index){
                this.current = index
            }
        }
    }
    var app = Vue.createApp(obj).mount("#box")
  </script>
</body>
</html>
```

1-6 v-html- 模板的陷阱

双大括号会将数据解释为纯文本，而不是 HTML。若想插入 HTML，你需要使用 v-html 指令:

```
<p>Using text interpolation: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

> ⚠ 安全警告
>
> 在网站上动态渲染任意 HTML 是非常危险的，因为这非常容易造成 **XSS 漏洞**。请仅在内容安全可信时再使用 `v-html`，并且**永远不要**使用用户提供的 HTML 内容。

2.class与style

2-1 class的绑定-方法太多了

对象写法

```
data() {
 return {
  classObject: {
   active: true,
   'text-danger': false
  }
 }
}
```

```
<div :class="classObject"></div>
```

数组写法

```
data() {
  return {
    activeClass: 'active',
    errorClass: 'text-danger'
  }
}
```

```
<div :class="[activeClass, errorClass]"></div>
```

2-2 style的绑定-同上

对象写法

```
data() {
  return {
    styleObject: {
      color: 'red',
      fontSize: '13px'
    }
  }
}
```

```
<div :style="styleObject"></div>
```

数组写法

```
data() {
    return {
        arr:[{
            width:"200px",
            height:"200px",
            backgroundSize:"cover"
        }],
    }
}

 this.arr.push({
backgroundImage:"url(https://pic.maizuo.com/usr/movie/862ab6736237acd11599e5eecbbc83d7.jpg?x-oss-process=image/quality,Q_70)"
 })
```

3.条件渲染

3-1 条件渲染-生或死的选择

`v-if` 是“真实的”按条件渲染，因为它确保了在切换时，条件区块内的事件监听器和子组件都会被销毁与重建。

`v-if` 也是**惰性**的：如果在初次渲染时条件值为 false，则不会做任何事。条件区块只有当条件首次变为 true 时才被渲染。

相比之下，`v-show` 简单许多，元素无论初始条件如何，始终会被渲染，只有 CSS `display` 属性会被切换。

总的来说，`v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。因此，如果需要频繁切换，则使用 `v-show` 较好；如果在运行时绑定条件很少改变，则 `v-if` 会更合适。

```
<ul >
   <li v-for="item,index in datalist">
      {{item.title}}
      <div v-if="item.state===0">
        未付款
      </div>
      <div v-else-if="item.state===1">
        未发货
      </div>
      <div v-else-if="item.state===2">
        已发货
      </div>
      <div v-else>
        已完成
      </div>
   </li>
</ul>


datalist:[
   {
      state:0,
      title:"111"
   },
   {
      state:1,
      title:"222"
   },
   {
      state:2,
      title:"333"
   }
]
```

4.列表渲染

4-1 v-for列表渲染 - 影分身术

v-for与对象

```
data() {
  return {
    myObject: {
      title: 'How to do lists in Vue',
      author: 'Jane Doe',
      publishedAt: '2016-04-10'
    }
  }
}
```

```
<ul>
  <li v-for="value in myObject">
    {{ value }}
  </li>
</ul>
```

v-for与v-if

> ⚠ 注意

```
<ul >
    <template v-for="{title,state},index in datalist" >
        <li v-if="state===1">{{title}}</li>
    </template>
</ul>
```

4-2 key设置 - 性能的保障

Vue 默认按照“就地更新”的策略来更新通过 `v-for` 渲染的元素列表。当数据项的顺序改变时，Vue 不会随之移动 DOM 元素的顺序，而是就地更新每个元素，确保它们在原本指定的索引位置上渲染。

为了给 Vue 一个提示，以便它可以跟踪每个节点的标识，从而重用和重新排序现有的元素，你需要为每个元素对应的块提供一个唯一的 `key` attribute：

```
<div v-for="item in items" :key="item.id">
  <!-- 内容 -->
</div>
```

虚拟dom

```
{
  type: 'div',
  props: {

    id: 'container'
```

```
  },
  children: [
    {
      type: 'span',
      props: {
        class: 'text1'
      },
      children: 'hello '
    },
    {
      type: 'span',
      props: {
        class: 'text2'
      },
      children: 'kerwin'
    },
  ]
}
```

真实dom

```
<div id="container">
  <span class="text1">hello </span>
  <span class="text2">kerwin</span>
</div>
```

4-3 数组变动侦测

Vue 能够侦听响应式数组的变更方法，并在它们被调用时触发相关的更新。这些变更方法包括：

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

对于一些不可变 (immutable) 方法，例如 `filter()`，`concat()` 和 `slice()`，这些都不会更改原数组，而总是**返回一个新数组。**当遇到的是非变更方法时，我们需要将旧的数组替换为新的：

```
this.items = this.items.filter((item) => item.message.match(/Foo/))
```

4-4 模糊搜索案例

方案1

```
<input type="text" v-model="mytext"/>
<ul>
  <template v-for="data in datalist" :key="data">
    <li v-if="data.includes(mytext)">{{data}}</li>
  </template>
</ul>
```

方案2

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="vue.js"></script>
</head>
<body>
  <div id="box">
    <input type="text" v-model="mytext"/>
    <ul>
      <li v-for="data in filterList()" :key="data">
        {{data}}
      </li>
    </ul>
  </div>

  <script>
    var obj  ={
      data(){
        return {
          mytext:"",
          datalist:["aaa","abb","aab","bcc","abc","bcd","add","acd"]
        }
      },
      methods:{
        filterList(evt){
          return this.datalist.filter(item=>item.includes(this.mytext))
        }
      }
    }
    Vue.createApp(obj).mount("#box")
  </script>
</body>
</html>
```

5.事件处理器

5-1 事件处理器 - 告别原生事件

内联事件处理器

```
<button @click="count++">Add 1</button>
<button @click="test('hello')">test hello</button>
```

方法事件处理器

```
<button @click="test">test</button>
```

5-2 事件修饰符 - 事件偷懒符?

Vue 为 `v-on` 提供了**事件修饰符。**修饰符是用 `.` 表示的指令后缀，包含以下这些：

- `.stop`
- `.prevent`
- `.self`
- `.capture`
- `.once`
- `.passive`

`.passive` 修饰符一般用于触摸事件的监听器，可以用来改善移动端设备的滚屏性能。

> TIP
>
> 请勿同时使用 `.passive` 和 `.prevent`，因为 `.passive` 已经向浏览器表明了你不想阻止事件的默认行为。如果你这么做了，则 `.prevent` 会被忽略，并且浏览器会抛出警告。

按键修饰符

Vue 为一些常用的按键提供了别名：

- `.enter`
- `.tab`
- `.delete` (捕获“Delete”和“Backspace”两个按键)
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`

6.表单控件绑定

6-1 表单输入绑定-一根绳上的蚂蚱

普通文本

```
<input v-model="message" placeholder="edit me" />
```

复选框

```
<input type="checkbox" id="checkbox" v-model="checked" />
```

单选框

```
<div>Picked: {{ picked }}</div>
<input type="radio" id="one" value="One" v-model="picked" />
```

选择器

```
<div>Selected: {{ selected }}</div>

<select v-model="selected">
  <option disabled value="">Please select one</option>
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
```

6-2 购物车案例

```
<!--
 * @作者: kerwin
-->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src="vue.js"></script>
    <style>
        li{
            display: flex;
            justify-content: space-between;
            padding:10px;
            border:1px solid lightgray;
            align-items: center;
        }
        li img{
            width: 100px;
        }
    </style>
</head>
```

```
<body>
  <div id="box">
    <ul >
      <li>
        <div>
          <input type="checkbox" v-model="isAllChecked" @change="handleChange"/>
          <span>全选/全不选</span>
        </div>
      </li>
      <template v-if="datalist.length">
        <li v-for="data,index in datalist" :key="data.id">
          <div>
            <input type="checkbox" v-model="checkList" :value="data" @change="handleItemChange"/>
          </div>
          <div>
            <img :src="data.poster" alt="">
          </div>

          <div>
            <div>{{data.title}}</div>
            <div style="color:red;">{{data.price}}</div>
          </div>

          <div>
            <button @click="data.number--" :disabled="data.number===1">-</button>
            {{data.number}}
            <button  @click="data.number++" :disabled="data.number===data.limit">+</button>
          </div>
          <div>
            <button @click="handleDel(index,data.id)">delete</button>
          </div>
        </li>
      </template>
      <li v-else>购物车空空如也</li>
      <li><div>总金额:{{sum()}}</div></li>
    </ul>
  </div>
  <script>

    var obj = {
      data() {
        return {
          isAllChecked:false,
          checkList:[],
          datalist: [{
              id: 1,
              title: "商品1",
              price: 10,
              number: 1,
              poster:
"https://p0.meituan.net/movie/dc2fed6001e809e4553f90cc6fad9a59245170.jpg@1l_1e_1c_128w_180h",

              limit: 5
```

```
                },
                {
                    id: 2,
                    title: "商品2",
                    price: 20,
                    number: 2,
                    poster:
"https://p0.meituan.net/moviemachine/3084e88f63eef2c6a0df576153a3fad0327782.jpg@1l_1e_1c_128w_180h",
                    limit: 5
                },
                {
                    id: 3,
                    title: "商品3",
                    price: 30,
                    number: 3,
                    poster:
"https://p0.meituan.net/movie/897b836475594922699514 4bfc2261ee4493381.jpg@1l_1e_1c_128w_180h",
                    limit: 5
                }
            ]
        }
    },
    methods:{
        sum(){
            return this.checkList.reduce((total,item)=>total+item.price*item.number,0)
        },
        handleDel(index,id){
            this.datalist.splice(index,1)

            this.checkList = this.checkList.filter(item=>item.id!==id)

            this.handleItemChange()
        },
        handleChange(){
            this.checkList = this.isAllChecked?this.datalist:[]
        },
        handleItemChange(){
            if(this.datalist.length===0){
                this.isAllChecked = false
                return
            }
            this.isAllChecked = this.datalist.length===this.checkList.length
        }
    }
}
var app = Vue.createApp(obj).mount("#box")
</script>
</body>

</html>
```

6-3 表单修饰符

.lazy

```
<!-- 在 "change" 事件后同步更新而不是 "input" -->
<input v-model.lazy="msg" />
```

.number

用户输入自动转换为数字，你可以在 `v-model` 后添加 `.number` 修饰符来管理输入：

```
<input v-model.number="age" />
```

`number` 修饰符会在输入框有 `type="number"` 时自动启用。

.trim

默认自动去除用户输入内容中两端的空格，你可以在 `v-model` 后添加 `.trim` 修饰符：

```
<input v-model.trim="msg" />
```

7.计算属性

7-1 计算属性 - 很聪明，会缓存

模板中的表达式虽然方便，但也只能用来做简单的操作。如果在模板中写太多逻辑，会让模板变得臃肿，难以维护。因此我们推荐使用**计算属性**来描述依赖响应式状态的复杂逻辑。

```
{
 data() {
  return {
   author: {
    name: 'John Doe',
    books: [
     'Vue 2 - Advanced Guide',
     'Vue 3 - Basic Guide',
     'Vue 4 - The Mystery'
    ]
   }
  }
 },
 computed: {
  // 一个计算属性的 getter
  publishedBooksMessage() {
   // `this` 指向当前组件实例
   return this.author.books.length > 0 ? 'Yes' : 'No'
  }
 }

 {{publishedBooksMessage}}
```

若我们将同样的函数定义为一个方法而不是计算属性，两种方式在结果上确实是完全相同的，然而，不同之处在于**计算属性值会基于其响应式依赖被缓存。**一个计算属性仅会在其响应式依赖更新时才重新计算。

7-2 之前案例的小改造

```
computed:{
   isAllChecked:{
      get(){
         return this.datalist.length===this.checkList.length
      },
         set(checked){
            this.checkList = checked?this.datalist:[]
         }
   }
}
```

注意：

（1）Getter 不应有副作用

计算属性的 getter 应只做计算而没有任何其他的副作用，这一点非常重要，请务必牢记。举例来说，**不要在 getter 中做异步请求或者更改 DOM**！

（2）避免直接修改计算属性值

从计算属性返回的值是派生状态。可以把它看作是一个“临时快照”，每当源状态发生变化时，就会创建一个新的快照。更改快照是没有意义的，因此计算属性的返回值应该被视为只读的，并且永远不应该被更改——应该更新它所依赖的源状态以触发新的计算。

7-3 侦听器watch的对比

`watch` 选项期望接受一个对象，其中键是需要侦听的响应式组件实例属性 (例如，通过 `data` 或 `computed` 声明的属性)——值是相应的回调函数。该回调函数接受被侦听源的新值和旧值。

```
watch: {
   // 侦听根级属性
   a(val, oldVal) {
     console.log(`new: ${val}, old: ${oldVal}`)
   },
   // 字符串方法名称
   b: 'someMethod',
   // 该回调将会在被侦听的对象的属性改变时调动，无论其被嵌套多深
   c: {
     handler(val, oldVal) {
       console.log('c changed')
     },
     deep: true
   },
   // 侦听单个嵌套属性：
   'c.d': function (val, oldVal) {
     // do something
   },

   // 该回调将会在侦听开始之后立即调用
```

```
  e: {
    handler(val, oldVal) {
      console.log('e changed')
    },
    immediate: true
  }
 }
```

8.数据请求

8-1 Fetch

XMLHttpRequest 是一个设计粗糙的 API，配置和调用方式非常混乱， 而且基于事件的异步模型写起来不友好。

兼容性不好 polyfill: https://github.com/camsong/fetch-ie8

```
fetch("http://localhost:3000/users")
        .then(res=>res.json())
        .then(res=>{
            console.log(res)
        })


fetch("http://localhost:3000/users",{
        method:"POST",
        headers:{
            "content-type":"application/json"
        },
        body:JSON.stringify({
            username:"kerwin",
            password:"123"
        })
    })
        .then(res=>res.json())
        .then(res=>{
            console.log(res)
        })

fetch("http://localhost:3000/users/5",{
        method:"PUT",
        headers:{
            "content-type":"application/json"
        },
        body:JSON.stringify({
            username:"kerwin",
            password:"456"
        })
    })
        .then(res=>res.json())

        .then(res=>{
```

```
        console.log(res)
    })

fetch("http://localhost:3000/users/5",{
        method:"DELETE"
    })
        .then(res=>res.json())
        .then(res=>{
            console.log(res)
        })
```

8-2 axios

> Axios是一个基于promise 的 HTTP 库，可以用在浏览器和 node.js中。

> https://www.npmjs.com/package/axios

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

1. get请求

```
axios.get("http://localhost:3000/users",{
    params:{
        name:"kerwin"
    }
}).then(res=>{
    console.log(res.data)
})
```

2.post请求

```
axios.post("http://localhost:3000/users",{
    name:"kerwin",
    age:100
}).then(res=>{
    console.log(res.data)
})
```

3. put请求

```
axios.put("http://localhost:3000/users/12",{
    name:"kerwin111",
    age:200
}).then(res=>{
    console.log(res.data)
})
```

4. delete请求

```
axios.delete("http://localhost:3000/users/11").then(res=>{
   console.log(res.data)
})
```

5. axios(config)配置

```
axios({
   method: 'post',
   url: 'http://localhost:3000/users',
   data: {
      name: 'kerwin',
      age: 100
   }
})
   .then(res => {
   console.log(res.data)
}).catch(err=>{
   console.log(err)
})
```

9.过滤器

9-1 vue3过滤器不支持了 - 怎么办?

在 2.x 中，开发者可以使用过滤器来处理通用文本格式。

```
<p>{{ accountBalance | currencyUSD }}</p>

filters: {
    currencyUSD(value) {
      return '$' + value
    }
}
```

虽然这看起来很方便，但它需要一个自定义语法，打破了大括号内的表达式“只是 JavaScript”的假设，这不仅有学习成本，而且有实现成本。

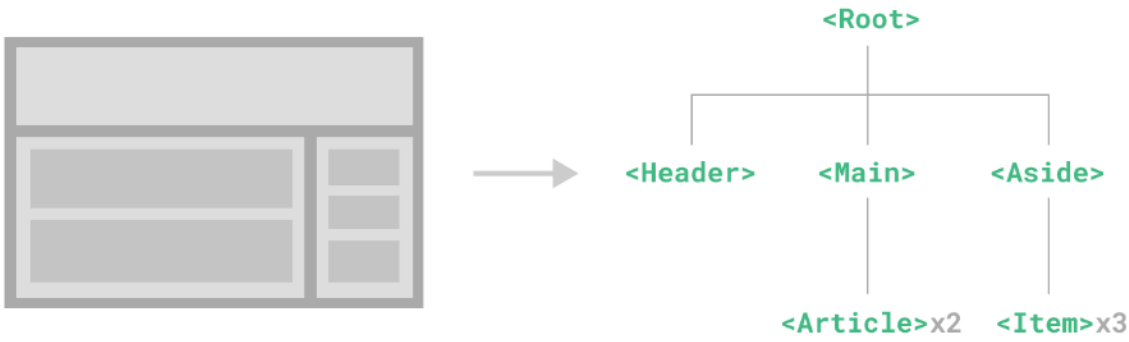
在 3.x 中，过滤器已移除，且不再支持。取而代之的是，我们建议用方法调用或计算属性来替换它们。

三.vue3进阶

1.单文件组件

1-1 组件定义 - 重塑经脉，断了?

组件允许我们将 UI 划分为独立的、可重用的部分，并且可以对每个部分进行单独的思考。在实际应用中，组件常常被组织成层层嵌套的树状结构：




```
//全局定义方法
app.component("child",{
   template:`<div>
child
</div>`
})
```

```
app.component("tabbar",{
        template:`
          <div style="color:blue">
            <ul>
              <li>首页</li>
              <li>列表</li>
              </ul>
            <child></child>
            <tabbarchild></tabbarchild>
          </div>
        `,
        //局部定义
        components:{
         tabbarchild:{
              template:`<div>
                tabbarchild
              </div>`
          }
        }
    })
```

组件带来的好处

（1） 结构清晰

（2） 复用性增加

（3） 封装性

当前写法的吐槽：

（1）dom高亮和代码提示没有

（2）css只能行内

1-2 单文件组件(SFC) - 独立日

Vue 的单文件组件 (即 `*.vue` 文件，英文 Single-File Component，简称 **SFC**) 是一种特殊的文件格式，使我们能够将一个 Vue 组件的模板、逻辑与样式封装在单个文件中。下面是一个单文件组件的示例：

```
<script>
export default {
  data() {
    return {
      greeting: 'Hello World!'
    }
  }
}
</script>

<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

如你所见，Vue 的单文件组件是网页开发中 HTML、CSS 和 JavaScript 三种语言经典组合的自然延伸。

- 使用熟悉的 HTML、CSS 和 JavaScript 语法编写模块化的组件
- [让本来就强相关的关注点自然内聚]
- 预编译模板，避免运行时的编译开销
- [组件作用域的 CSS]
- [在使用组合式 API 时语法更简单]
- 通过交叉分析模板和逻辑代码能进行更多编译时优化
- [更好的 IDE 支持]，提供自动补全和对模板中表达式的类型检查
- 开箱即用的模块热更新 (HMR) 支持

1-3 Vue-CLI创建项目 - 锅灶升级

Vue CLI 是一个基于 Vue.js 进行快速开发的完整系统，提供：

- 通过 `@vue/cli` 实现的交互式的项目脚手架。
- 通过 `@vue/cli` + `@vue/cli-service-global` 实现的零配置原型开发。
- 一个运行时依赖 (

  ```
  @vue/cli-service
  ```

)，该依赖：

    - 可升级；
    - 基于 webpack 构建，并带有合理的默认配置；
    - 可以通过项目内的配置文件进行配置；
    - 可以通过插件进行扩展。
- 一个丰富的官方插件集合，集成了前端生态中最好的工具。
- 一套完全图形化的创建和管理 Vue.js 项目的用户界面。

Vue CLI 致力于将 Vue 生态中的工具基础标准化。它确保了各种构建工具能够基于智能的默认配置即可平稳衔接，这样你可以专注在撰写应用上，而不必花好几天去纠结配置的问题。与此同时，它也为每个工具提供了调整配置的灵活性，无需 eject。

安装：

```
npm install -g @vue/cli
# OR
yarn global add @vue/cli
```

创建一个项目：

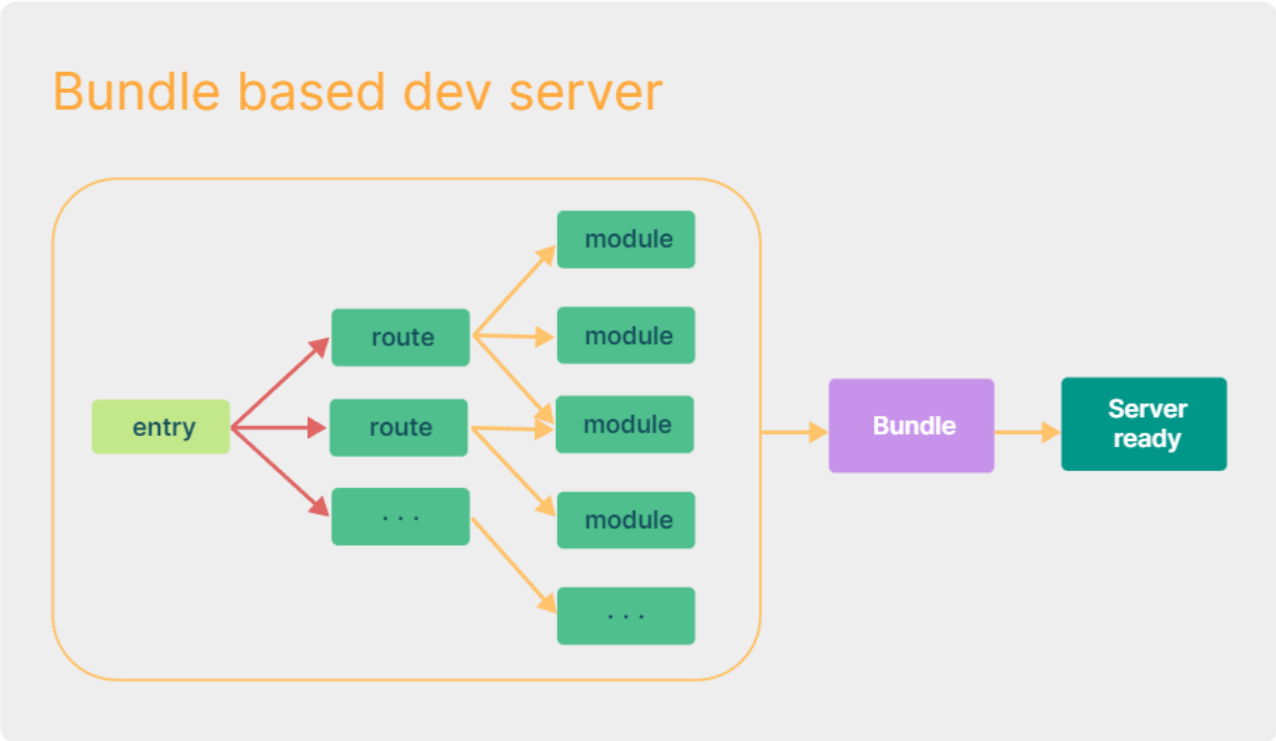
```
vue create my-project
# OR
vue ui
```

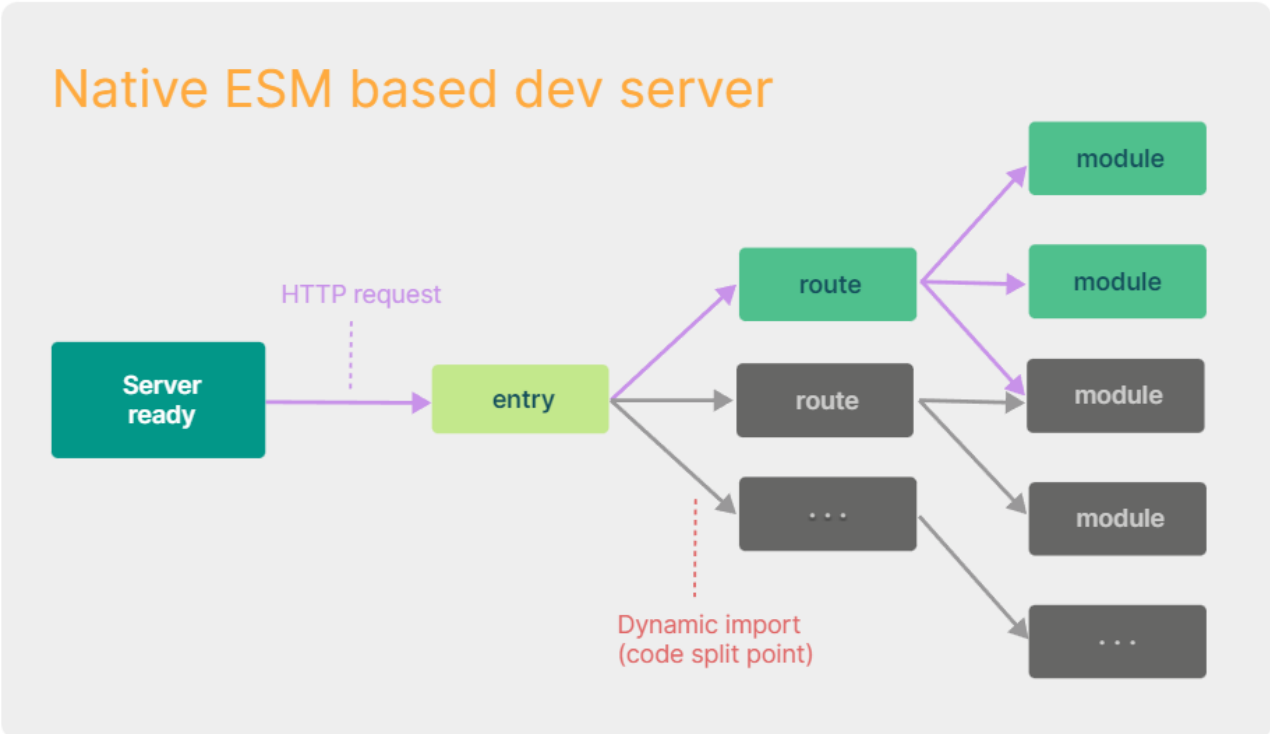
1-4 Vite 创建项目 - 官方推荐

Vite（法语意为 "快速的"，发音 `/vit/`，发音同 "veet"）是一种新型前端构建工具，能够显著提升前端开发体验。它主要由两部分组成：

- 一个开发服务器，它基于 原生 ES 模块 提供了 丰富的内建功能，如速度快到惊人的 模块热更新（HMR）。
- 一套构建指令，它使用 Rollup 打包你的代码，并且它是预配置的，可输出用于生产环境的高度优化过的静态资源。

Vite 意在提供开箱即用的配置，同时它的 插件 API 和 JavaScript API 带来了高度的可扩展性，并有完整的类型支持。





创建项目

Vite 需要 Node.js 版本 14.18+，16+。然而，有些模板需要依赖更高的 Node 版本才能正常运行，当你的包管理器发出警告时，请注意升级你的 Node 版本。

使用 NPM:

```
$ npm create vite@latest
```

使用 Yarn:

```
$ yarn create vite
```

使用 PNPM:

```
$ pnpm create vite
```

1-5 启动流程&入口文件

2.组件基础

2-1 父传子prop - 沟通的重要

```
<Navbar  mytitle='我的电影' left='返回' right='首页'></Navbar>

<Navbar v-bind="{
        mytitle:'我的电影',
        left:'返回',
        right:'首页'
    }" ></Navbar>
```

注意:

所有的 props 都遵循着**单向绑定**原则，props 因父组件的更新而变化，自然地将新的状态向下流往子组件，而不会逆向传递。这避免了子组件意外修改父组件的状态的情况，不然应用的数据流将很容易变得混乱而难以理解。

另外，每次父组件更新后，所有的子组件中的 props 都会被更新到最新值，这意味着你**不应该**在子组件中去更改一个 prop。若你这么做了，Vue 会在控制台上向你抛出警告:

```
export default {
  props: ['foo'],
  created() {
    // ❌ 警告！prop 是只读的！
    this.foo = 'bar'
  }
}
```

2-2 属性验证&默认属性

```
export default {
  props: {
    // 基础类型检查
    // （给出 `null` 和 `undefined` 值则会跳过任何类型检查）
    propA: Number,
    // 多种可能的类型
    propB: [String, Number],
    // 必传，且为 String 类型
    propC: {
      type: String,
      required: true
    },
    // Number 类型的默认值
    propD: {
      type: Number,
      default: 100
    },
    // 对象类型的默认值
    propE: {
      type: Object,
      // 对象或者数组应当用工厂函数返回。
      // 工厂函数会收到组件所接收的原始 props
      // 作为参数
      default(rawProps) {
        return { message: 'hello' }
      }
    },
    // 自定义类型校验函数
    propF: {
      validator(value) {
        // The value must match one of these strings
        return ['success', 'warning', 'danger'].includes(value)
      }
    }
  }
}
```

校验选项中的 `type` 可以是下列这些原生构造函数：

- `String`

- `Number`
- `Boolean`
- `Array`
- `Object`
- `Date`
- `Function`
- `Symbol`

2-3 子传父 - 自定义事件

在组件的模板表达式中，可以直接使用 `$emit` 方法触发自定义事件 (例如：在 `v-on` 的处理函数中)：

```
<!-- MyComponent -->
<button @click="$emit('someEvent')">click me</button>
```

`$emit()` 方法在组件实例上也同样以 `this.$emit()` 的形式可用：

```
export default {
  methods: {
    submit() {
      this.$emit('someEvent')
    }
  }
}
```

父组件可以通过 `v-on` (缩写为 `@`) 来监听事件：

```
<MyComponent @some-event="callback" />
```

同样，组件的事件监听器也支持 `.once` 修饰符：

```
<MyComponent @some-event.once="callback" />
```

2-4 $refs - 父组件的强权

- ref如果绑定在dom节点上，拿到的就是 原生dom节点
- ref如果绑定在组件上，拿到的就是 组件对象，可以实现通信功能

```
<van-field label="用户名" ref="myusername"></van-field>
<van-field label="密码" type="password" ref="mypassword"></van-field>
<van-field label="年龄" type="number"></van-field>
```

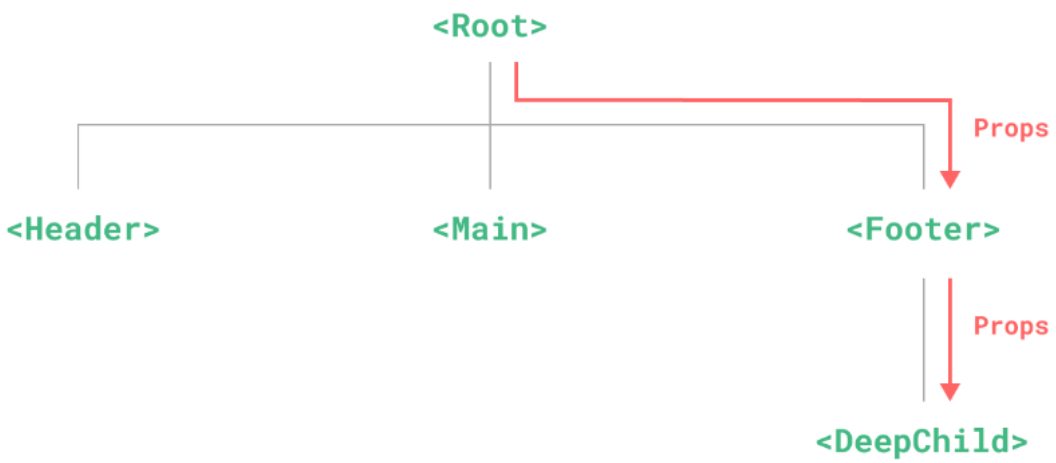
```
methods:{
      handleRegister(){
        console.log(this.$refs.myusername.value,this.$refs.mypassword.value)
      },
      handelReset(){
        this.$refs.myusername.value=""
        this.$refs.mypassword.value=""
      }
}
```

2-5 Misplaced & root - 子组件的无法无天

在子组件中通过`$parent`访问父组件，通过`$root`访问根组件

```
{
  methods:{
      handleClick(){
         this.$parent.$refs.layout.isShow=this.$parent.$refs.layout.isShow
      }
   }
}
```

2-6 跨级通信- provide&inject




```
//child.vue //grandchild.vue.....
export default {
   inject:["message"]
}

//app.vue  -非响应式
.....
 data(){
      return {
```

```
        message:"hello kerwin"
      }
  },
  provide() {
    return {
      message: this.message
    }
  }
....
```

2-7 动态组件 - 墙头草

在切换时创建新的组件实例通常是有意义的，但在这个例子中，我们的确想要组件能在被“切走”的时候保留它们的状态。要解决这个问题，我们可以用 `` 内置组件将这些动态组件包装起来：

```
<!-- 非活跃的组件将会被缓存！ -->
<KeepAlive>
  <component :is="activeComponent" />
</KeepAlive>
```

```
<!-- 以英文逗号分隔的字符串 -->
<KeepAlive include="a,b">
  <component :is="view" />
</KeepAlive>

<!-- 正则表达式 (需使用 `v-bind`) -->
<KeepAlive :include="/a|b/">
  <component :is="view" />
</KeepAlive>

<!-- 数组 (需使用 `v-bind`) -->
<KeepAlive :include="['a', 'b']">
  <component :is="view" />
</KeepAlive>
```

2-8 组件中的v-model

```
```

2-9 异步组件

在大型项目中，我们可能需要拆分应用为更小的块，并仅在需要时再从服务器加载相关组件。Vue 提供了 defineAsyncComponent 方法来实现此功能：

```
<script>
import { defineAsyncComponent } from 'vue'

export default {
  components: {
    AdminPage: defineAsyncComponent(() =>
      import('./components/AdminPageComponent.vue')
    )
  }
}
</script>

<template>
  <AdminPage />
</template>
```

加载与错误提示

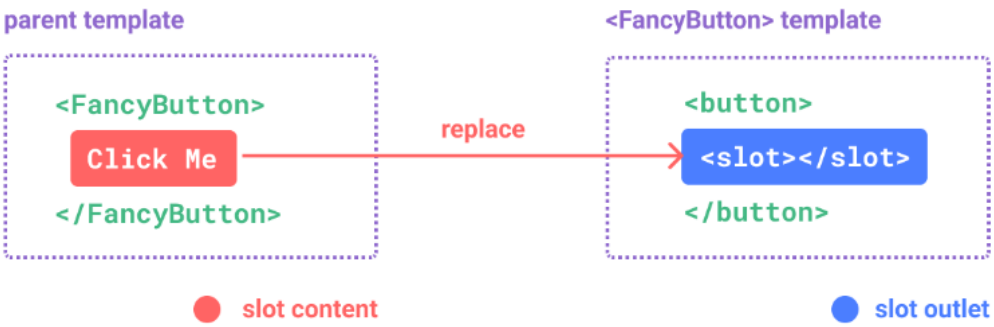
```
const AsyncComp = defineAsyncComponent({
  // 加载函数
```

```
  loader: () => import('./Foo.vue'),

  // 加载异步组件时使用的组件
  loadingComponent: LoadingComponent,
  // 展示加载组件前的延迟时间，默认为 200ms
  delay: 200,

  // 加载失败后展示的组件
  errorComponent: ErrorComponent,
  // 如果提供了一个 timeout 时间限制，并超时了
  // 也会显示这里配置的报错组件，默认值是：Infinity
  timeout: 3000
})
```

3.组件插槽

3-1 插槽的基本应用

`<slot>` 元素是一个**插槽出口** (slot outlet)，标示了父元素提供的**插槽内容** (slot content) 将在哪里被渲染。



注意：插槽内容可以访问到父组件的数据作用域，因为插槽内容本身是在父组件模板中定义的

插槽内容**无法访问**子组件的数据。Vue 模板中的表达式只能访问其定义时所处的作用域，这和 JavaScript 的词法作用域规则是一致的。换言之：

> 父组件模板中的表达式只能访问父组件的作用域；子组件模板中的表达式只能访问子组件的作用域。

3-2 具名插槽

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

```
<BaseLayout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>

  <template #default>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </template>

  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

3-3 作用域插槽

然而在某些场景下插槽的内容可能想要同时使用父组件域内和子组件域内的数据。要做到这一点，我们需要一种方法来让子组件在渲染时将一部分数据提供给插槽。

```
<Nowplaying v-slot="props">
        <ul>
          <li v-for="data in props.list"
:key="data.id">
            <div v-
if="data.nm.includes(text) && text!=''"
style="font-size: 30px;">
              {{data.nm}}
            </div>
            <div v-else>
              {{data.nm}}
            </div>
          </li>
        </ul>
    </Nowplaying>
```

{{ slotProps.count }}

```
<slot :list="datalist">
        <ul>
          <li v-for="data in datalist"
:key="data.id">
            {{ data.nm }}
          </li>
        </ul>
 </slot>
```

4. 生命周期

每个 Vue 组件实例在创建时都需要经历一系列的初始化步骤，比如设置好数据侦听，编译模板，挂载实例到 DOM，以及在数据改变时更新 DOM。在此过程中，它也会运行被称为生命周期钩子的函数，让开发者有机会在特定阶段运行自己的代码。

(1) beforeCreate()会在实例初始化完成、props 解析之后、`data()` 和 `computed` 等选项处理之前立即调用。

(2)created() 当这个钩子被调用时，以下内容已经设置完成：响应式数据、计算属性、方法和侦听器。然而，此时挂载阶段还未开始，因此 `$el` 属性仍不可用。

(3)beforeMount()当这个钩子被调用时，组件已经完成了其响应式状态的设置，但还没有创建 DOM 节点。它即将首次执行 DOM 渲染过程。

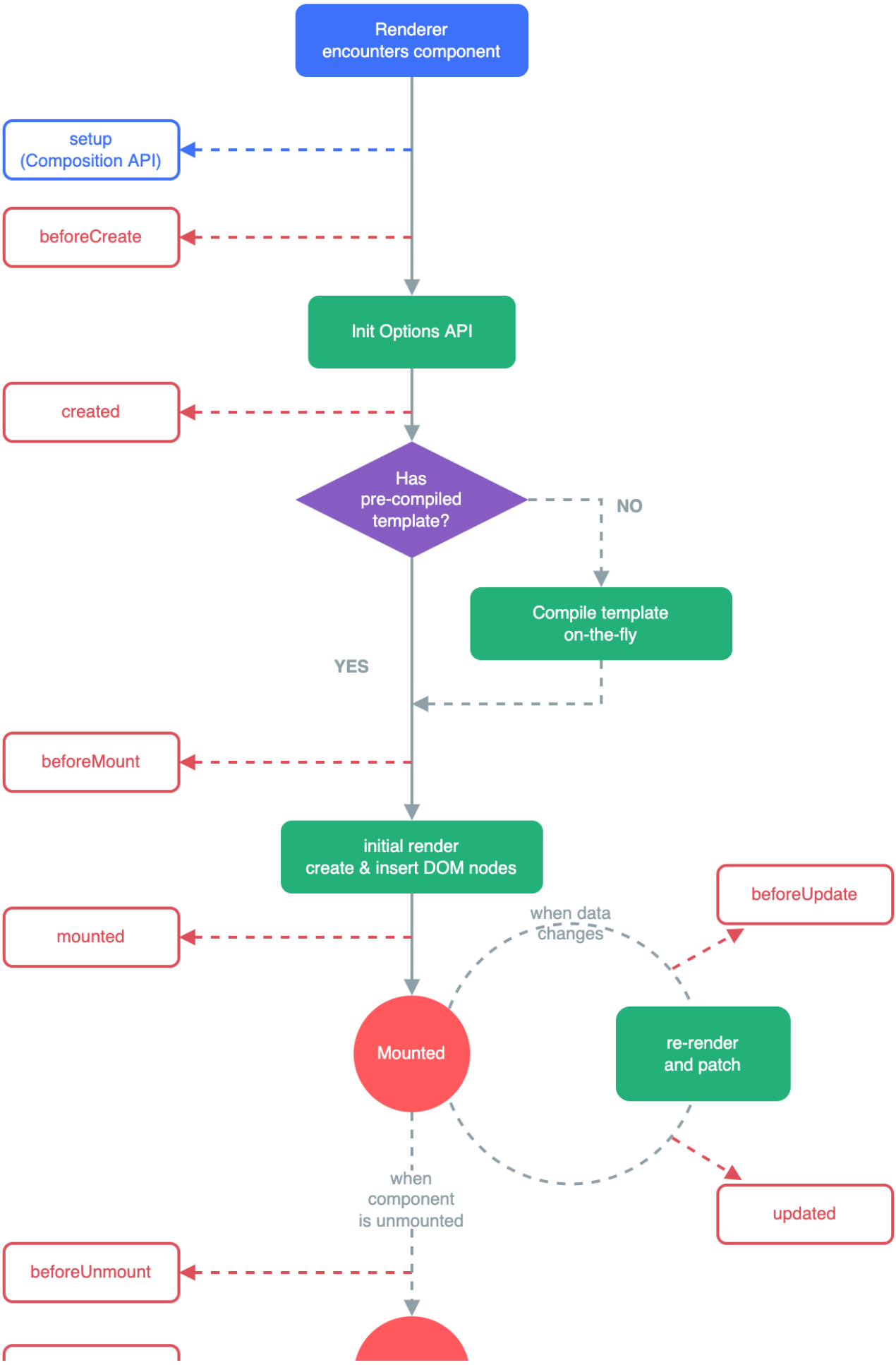
(4)mounted()所有同步子组件都已经被挂载。这个钩子通常用于执行需要访问组件所渲染的 DOM 树相关的副作用

(5)beforeUpdate()这个钩子可以用来在 Vue 更新 DOM 之前访问 DOM 状态。在这个钩子中更改状态也是安全的。

(6)updated()这个钩子会在组件的任意 DOM 更新后被调用，这些更新可能是由不同的状态变更导致的。如果你需要在某个特定的状态更改后访问更新后的 DOM，请使用 nextTick() 作为替代。

(7)beforeUnmount()当这个钩子被调用时，组件实例依然还保有全部的功能。

(8)unmounted()在一个组件实例被卸载之后调用。


Renderer
encounters component
setup
(Composition API)
beforeCreate
Init Options API
created
Has
pre-compiled
template?
NO
Compile template
on-the-fly
YES
beforeMount
initial render
create & insert DOM nodes
mounted
when data
changes
beforeUpdate
Mounted
re-render
and patch
when
component
is unmounted
updated
beforeUnmount

5.组件的封装

轮播组件

```
<!--
 * @作者: kerwin
-->
<template>
  <div>
    <MySwiper v-if="datalist.length" :loop="false" @slideChange="onSlideChange">
      <MySwiperItem v-for="(data, index) in datalist" :key="index">
        {{ data }}
      </MySwiperItem>
    </MySwiper>
  </div>
</template>
<script>
import MySwiper from './MySwiper.vue'
import MySwiperItem from './MySwiperItem.vue'
export default {
  components: {
    MySwiper,
    MySwiperItem
  },
  data() {
    return {
      datalist: []
    }
  },
  mounted() {
    setTimeout(() => {
      this.datalist = [11, 22, 33, 44]
    }, 2000)
  },
  methods: {
    onSlideChange(index) {
      console.log(index)
    }
  }
}
</script>
<style >
.swiper {
  width: 600px;
  height: 400px;
}
</style>
```

```
<!--
 * @作者: kerwin
-->
<template>
  <div class="swiper">
    <div class="swiper-wrapper">
      <slot></slot>
    </div>
    <!-- 如果需要分页器 -->
    <div class="swiper-pagination"></div>
  </div>
</template>
<script>
import Swiper from 'swiper/bundle';
import 'swiper/css/bundle';

export default {
  props: ["loop"],
  mounted() {
    this.mySwiper = new Swiper('.swiper', {
      loop: this.loop,
      // 如果需要分页器
      pagination: {
        el: '.swiper-pagination',
      },
      on: {
        slideChange: () => {
          this.$emit("slideChange", this.mySwiper.activeIndex)
        },
      },
    });
  }
}
</script>
```

```
<!--
 * @作者: kerwin
-->
<template>
  <div class="swiper-slide">
    <slot></slot>
  </div>
</template>
```

6.自定义指令

6-1 指令写法&钩子

除了 Vue 内置的一系列指令 (比如 `v-model` 或 `v-show`) 之外，Vue 还允许你注册自定义的指令 (Custom Directives)。自定义指令主要是为了重用涉及普通元素的底层 DOM 访问的逻辑。

全局

```
const app = createApp({})

// 使 v-focus 在所有组件中都可用
app.directive('focus', {
  /* ... */
})
```

局部

```
const focus = {
  mounted: (el) => el.focus()
}

export default {
  directives: {
    // 在模板中启用 v-focus
    focus
  }
}
```

指令钩子

```
const myDirective = {
  // 在绑定元素的 attribute 前
  // 或事件监听器应用前调用
  created(el, binding, vnode, prevVnode) {
    // 下面会介绍各个参数的细节
  },
  // 在元素被插入到 DOM 前调用
  beforeMount(el, binding, vnode, prevVnode) {},
  // 在绑定元素的父组件
  // 及他自己的所有子节点都挂载完成后调用
  mounted(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件更新前调用
  beforeUpdate(el, binding, vnode, prevVnode) {},
  // 在绑定元素的父组件
  // 及他自己的所有子节点都更新后调用
  updated(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件卸载前调用
  beforeUnmount(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件卸载后调用
  unmounted(el, binding, vnode, prevVnode) {}
}
```

简写形式

对于自定义指令来说，一个很常见的情况是仅仅需要在 `mounted` 和 `updated` 上实现相同的行为，除此之外并不需要其他钩子。这种情况下我们可以直接用一个函数来定义指令，如下所示：

```
<div v-color="color"></div>
```

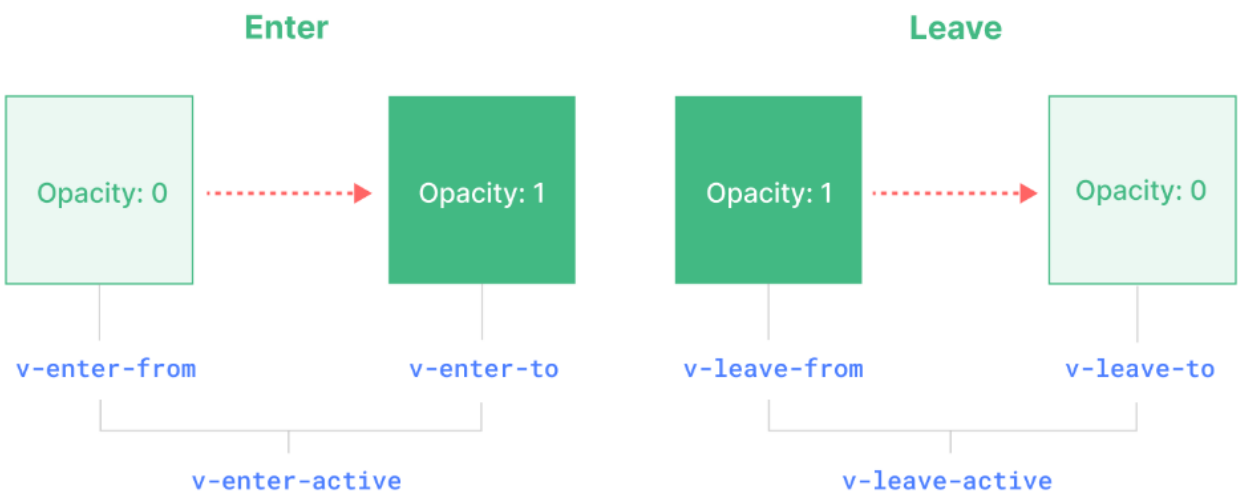
```
app.directive('color', (el, binding) => {
  // 这会在 `mounted` 和 `updated` 时都调用
  el.style.color = binding.value
})
```

7.过渡效果

Vue 提供了两个内置组件，可以帮助你制作基于状态变化的过渡和动画：

- `Transition` 会在一个元素或组件进入和离开 DOM 时应用动画。
- `TransitionGroup` 会在一个 `v-for` 列表中的元素或组件被插入，移动，或移除时应用动画。

7-1 过渡效果



JS钩子

```
<Transition
  @before-enter="onBeforeEnter"
  @enter="onEnter"
  @after-enter="onAfterEnter"
  @enter-cancelled="onEnterCancelled"
  @before-leave="onBeforeLeave"
  @leave="onLeave"
  @after-leave="onAfterLeave"
  @leave-cancelled="onLeaveCancelled"
>
  <!-- ... -->
</Transition>
```

过渡模式

```
<Transition mode="out-in">
  ...
</Transition>
```

组件间过渡

```
<Transition name="fade" mode="out-in">
  <component :is="activeComponent"></component>
</Transition>
```

7-2 列表过渡

`TransitionGroup` 是一个内置组件，用于对 `v-for` 列表中的元素或组件的插入、移除和顺序改变添加动画效果。

区别：

- 默认情况下，它不会渲染一个容器元素。但你可以通过传入 `tag` prop 来指定一个元素作为容器元素来渲染。
- 过渡模式在这里不可用，因为我们不再是在互斥的元素之间进行切换。
- 列表中的每个元素都**必须**有一个独一无二的 `key` attribute。
- CSS 过渡 class 会被应用在列表内的元素上，**而不是**容器元素上。

```
<TransitionGroup name="list" tag="ul">
  <li v-for="item in items" :key="item">
    {{ item }}
  </li>
</TransitionGroup>
```

移动动画

当某一项被插入或移除时，它周围的元素会立即发生“跳跃”而不是平稳地移动。我们可以通过添加一些额外的 CSS 规则来解决这个问题：

```
.list-move /* 对移动中的元素应用的过渡 */
{
  transition: all 0.5s ease;
}

/* 确保将离开的元素从布局流中删除
  以便能够正确地计算移动的动画。 */
.list-leave-active {
  position: absolute;
}
```

7-3 可复用过渡

得益于 Vue 的组件系统，过渡效果是可以被封装复用的。要创建一个可被复用的过渡，我们需要为 `Transition` 组件创建一个包装组件，并向内传入插槽内容：

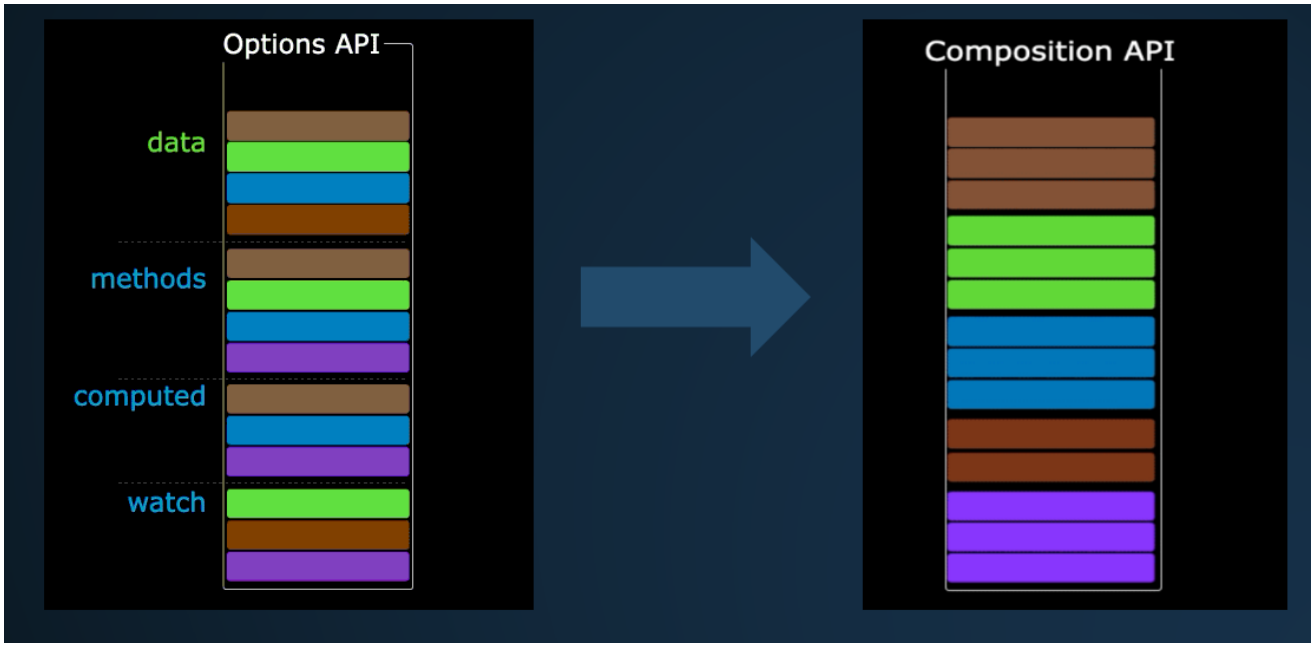
```
<!-- MyTransition.vue -->
<script>
// JavaScript 钩子逻辑...
</script>

<template>
  <!-- 包装内置的 Transition 组件 -->
  <Transition
    name="my-transition"
    @enter="onEnter"
    @leave="onLeave">
    <slot></slot> <!-- 向内传递插槽内容 -->
  </Transition>
</template>

<style>
/*
  必要的 CSS...
  注意：避免在这里使用 <style scoped>
  因为那不会应用到插槽内容上
*/
</style>
```

现在 `MyTransition` 可以在导入后像内置组件那样使用了：

```
<MyTransition>
  <div v-if="show">Hello</div>
</MyTransition>
```

四.vue3组合式API

起初定义的是Vue-Function-API，后经过社区意见收集，更名为Vue-Composition-API.



1. reactive

作用：创建响应式对象，非包装对象，可以认为是模板中的状态。

- template 可以放兄弟节点
- reactive 类似useState, **如果参数是字符串，数字,会报警告，value cannot be made reactive**, 所以应该设置对象，这样可以数据驱动页面

```
<div>
  {{countobj.count}}-<button @click="add">add</button>
 </div>

 setup () {

  const countobj = reactive({
   count: 0
  })
  const add = () => {
   countobj.count++
  }
  return {
   countobj,
   add
  }
 }
```

```
setup () {
  const A = ref('')
  const methodsA = () => {
      ......
  }
  const B = ref('')
  const methodsB = () => {
      ......
  }
  const C = ref('')
  const methodsC = () => {
      ......
  }
  return {
    A,
    methodsA,
    B,
    methodsB,
    C,
    methodsC
  }
}
```

```
import useA from './moduleA'
import useB from './moduleB'
import useC from './moduleC'
export default {
  setup () {
    const { A, methodsA } = useA()
    const { B, methodsB } = useB()
    const { C, methodsC } = useC()
    return {
      A,
      methodsA,
      B,
      methodsB,
      C,
      methodsC
    }
  }
}
```

2. ref

作用：创建一个包装式对象，含有一个响应式属性value。 它和reactive的差别，就是前者没有包装属性value
const count = ref(0) ， 可以接收普通数据类型,count.value++

```
<div>
  {{count}}-<button @click="add">add</button>
</div>

setup () {
  const add = () => {
    count.value++
  }
  const count = ref(0)

  return {
    count,
    add
  }
}
```

2-1 ref嵌套在reactive中

```
<template>
  <div class="home">
    home-{{count}}--{{state.count}}
    <button @click="add">click</button>
  </div>
</template>

<script>
import { reactive, ref } from 'vue'
export default {
  name: 'Home',

  setup () {
```

```
    const count = ref(0)
    const state = reactive({
      count
    })
    const add = () => {
      state.count++
      //state.count 跟ref count 都会更新
    }
    return {
      state,
      add,
      count
    }
  }
}
</script>
```

2-2 toRefs

默认直接展开state，那么此时reactive数据变成普通数据，通过toRefs，可以把reactive里的每个属性，转化为ref对象，这样展开后，就会变成多个ref对象， 依然具有响应式特性

```
<template>
  <div class="home">
    home-{{count}}
    <button @click="add">click</button>
  </div>
</template>

<script>
import { reactive, toRefs } from 'vue'
export default {
  name: 'Home',
  setup () {
    const state = reactive({
      count: 1
    })

    const add = () => {
      state.count++
    }
    return {
      ...toRefs(state),
      add
    }
  }
}
</script>
```

2-3 ref访问dom或者组件

```
<input type="text" ref="myinput"/>

//js
const myinput = ref(null)

console.log(myinput.value.value)
```

3. 计算属性

```
computed(回调函数)
 setup () {
  const mytext = ref('')

  const computedSum = computed(() => mytext.value.substring(0, 1).toUpperCase() + mytext.value.substring(1))
  // 注意mytext.value
  return {
   mytext,
   computedSum
  }
 }
```

4. watch

计算属性允许我们声明性地计算衍生值。然而在有些情况下，我们需要在状态变化时执行一些“副作用”：例如更改 DOM，或是根据异步操作的结果去修改另一处的状态。

在组合式 API 中，我们可以使用 `watch` 函数在每次响应式状态发生变化时触发回调函数：

```
监听器 watch 是一个方法，它包含 2 个参数
const reactivedata = reactive({count:1})
const text= ref("")
watch(() => reactivedata.count,
 val => {
  console.log(`count is ${val}`)
 })

watch(text,
 val => {
  console.log(`count is ${val}`)
 })

watch(source, (newValue, oldValue) => {
 // 立即执行，且当 `source` 改变时再次执行
 }, { immediate: true })
```

第一个参数是监听的值，count.value 表示当 count.value 发生变化就会触发监听器的回调函数，即第二个参数，第二个参数可以执行监听时候的回调

5. VCA中的watchEffect函数

注意：

（1）watch：

- 具有一定的惰性lazy 第一次页面展示的时候不会执行，只有数据变化的时候才会执行
- 参数可以拿到当前值和原始值
- 可以侦听多个数据的变化，用一个侦听器承载

```
const todoId = ref(1)
const data = ref(null)

watch(todoId, async () => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  data.value = await response.json()
}, { immediate: true })
```

（2）watchEffect：

- 立即执行，没有惰性，页面的首次加载就会执行。
- 自动检测内部代码，代码中有依赖 便会执行
- 不需要传递要侦听的内容 会自动感知代码依赖，不需要传递很多参数，只要传递一个回调函数
- 不能获取之前数据的值 只能获取当前值
- 一些异步的操作放在这里会更加合适

```
watchEffect(async () => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  data.value = await response.json()
})
```

6. prop & emit

```
props:["mytite"], //正常接收
setup (props, { emit }) {
   console.log(props.mytitle)
   const handleClick = () => {
    emit('kerwinevent')
   }

   return {
    handleClick
   }
 }
```

7. VCA中provide&inject

> provide、inject 是 vue-composition-api 的一个功能：依赖注入功能

```
import { provide, inject } from 'vue'

// 根组件 共享自己的状态
   const kerwinshow = ref(true)
   provide('kerwinshow', kerwinshow)

// detail组件
   onMounted(() => {

     const kerwinshow = inject('kerwinshow')
     kerwinshow.value = false
   })
```

8. VCA中的生命周期

```
import { onUnmounted, onMounted } from 'vue'

setup () {
   ...

 onMounted(() => {

 console.log('onMounted')
   })
   ...
 }
```

9. 在单文件组件使用VCA的语法糖

parent，$refs

$root，和$parent都能访问父组件的属性和方法，区别在于如果存在多级子组件，通过parent 访问得到的是它最近一级的父组件，通过root 访问得到的是根父组件。通过在子组件标签定义 ref 属性，在父组件中可以使用$refs 访问子组件实例。

10. Vue 中怎么自定义指令

通过directive来自定义指令，自定义指令分为全局指令和局部指令，自定义指令也有几个的钩子函数，常用的有bind和update，当 bind 和 update 时触发相同行为，而不关心其它的钩子时可以简写。

```
Vue.directive('focus', {
  // 当被绑定的元素插入到 DOM 中时……
  inserted: function (el) {
    // 聚焦元素
    el.focus()
  }
})


Vue.directive('color-swatch', function (el, binding) {
  el.style.backgroundColor = binding.value
})
```

11. Vue 中怎么自定义过滤器 (vue3不支持)

通过filter来定义过滤器，过滤器分为全局和局部过滤器，过滤器的主体为一个普通的函数，来对数据进行处理，可以传递参数。当有局部和全局两个名称相同的过滤器时候，会以就近原则进行调用，即：局部过滤器优先于全局过滤器被调用。

```
<!-- 在双花括号中 -->
{{ message | capitalize }}

<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId | formatId"></div>


filters: {
  capitalize: function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
  }
}
```

```
Vue.filter('capitalize', function (value) {
  if (!value) return ''
  value = value.toString()
  return value.charAt(0).toUpperCase() + value.slice(1)
})
```

12. Vue 等单页面应用的优缺点

```
// 优点
1单页应用的内容的改变不需要重新加载整个页面，web应用更具响应性和更令人着迷。

2、单页应用没有页面之间的切换，就不会出现“白屏现象”,也不会出现假死并有“闪烁”现象

3、单页应用相对服务器压力小，服务器只用出数据就可以，不用管展示逻辑和页面合成，吞吐能力会提高几倍。

4、良好的前后端分离。后端不再负责模板渲染、输出页面工作，后端API通用化，即同一套后端程序代码，不用修改就可以用于Web界面、手机、平板等多种客户端。

// 缺点
1、首次加载耗时比较多。

2、SEO问题，不利于百度，360等搜索引擎收录。

3、容易造成Css命名冲突。

4、前进、后退、地址栏、书签等，都需要程序进行管理，页面的复杂度很高，需要一定的技能水平和开发成本高。
```

13. Vue-router 使用params与query传参有什么区别

```
// 用法上
1：query要用path来引入，params要用name来引入，接收参数都是类似的，分别是this.$route.query和this.$route.params。

// 展示上

2：params是路由的一部分,必须要有。query是拼接在url后面的参数

// 命名的路由，并加上参数，让路由建立 url /users/eduardo
router.push({ name: 'user', params: { username: 'eduardo' } })

// 带查询参数，结果是 /register?plan=private
router.push({ path: '/register', query: { plan: 'private' } })

// 带 hash，结果是 /about#team
router.push({ path: '/about', hash: '#team' })
```

14. Vue中 keep-alive 的作用

keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染。一旦使用keepalive包裹组件，此时mouted，created等钩子函数只会在第一次进入组件时调用，当再次切换回来时将不会调用。此时如果我们还想在每次切换时做一些事情，就需要用到另外的周期函数，actived和deactived，这两个钩子函数只有被keepalive包裹后才会调用。

15. Vue如何实现单页面应用

通常的url 地址由以下内容构成：协议名 域名 端口号 路径 参数 哈希值，当哈希值改变，页面不会发生跳转，单页面应用就是利用了这一点，给window注册onhashchange事件，当哈希值改变时通过location.hash就能获得相应的哈希值，然后就能跳到相应的页面。

1.hash通过监听浏览器的onhashchange()事件变化，查找对应的路由规则

2.history原理： 利用H5的 history中新增的两个API pushState() 和 replaceState() 和一个事件onpopstate监听URL变化

16. 说出至少4种Vue当中的指令和它的用法?

v-if(判断是否隐藏，用来判断元素是否创建)
v-show(元素的显示隐藏，类似css中的display的block和hidden)
v-for(把数据遍历出来)
v-bind(绑定属性)
v-model(实现双向绑定)

17. 如何实现一个路径渲染多个组件?

可以通过命名视图(router-view)，它容许同一界面中拥有多个单独命名的视图，而不是只有一个单独的出口。如果 router-view 没有设置名字，那么默认为 default。通过设置components即可同时渲染多个组件。

```
<router-view class="view left-sidebar" name="LeftSidebar"></router-view>
<router-view class="view main-content"></router-view>
<router-view class="view right-sidebar" name="RightSidebar"></router-view>


const router = createRouter({
  history: createWebHashHistory(),
  routes: [
    {
      path: '/',
      components: {
        default: Home,
        // LeftSidebar: LeftSidebar 的缩写
        LeftSidebar,
        // 它们与 `<router-view>` 上的 `name` 属性匹配
        RightSidebar,
      },
    },
  ],
})
```

18. 如何实现多个路径共享一个组件?

只需将多个路径的component字段的值设置为同一个组件即可。

```
const routes = [
  { path: '/', component: Home },
  { path: '/home', component: Home },
]
```

19. 如何监测动态路由的变化

可以通过watch方法来对$route进行监听，或者通过导航守卫的钩子函数beforeRouteUpdate来监听它的变化。

20. vue-router 中的 router-link上 v-slot属性怎么用?

router-link 通过一个作用域插槽暴露底层的定制能力。这是一个更高阶的 API，主要面向库作者，但也可以为开发者提供便利，多数情况用在一个类似 NavLink 这样的自定义组件里。

有时我们可能想把激活的 class 应用到一个外部元素而不是 <a> 标签本身，这时你可以在一个 router-link 中包裹该元素并使用 v-slot 属性来创建链接：

```
<router-link
  to="/foo"
  custom
  v-slot="{ href, route, navigate, isActive, isExactActive }"
>
  <li
    :class="[isActive && 'router-link-active', isExactActive && 'router-link-exact-active']"
  >
    <a :href="href" @click="navigate">{{ route.fullPath }}</a>
  </li>
</router-link>
```

21. Vue 如何去除url中的

将路由模式改为history

由于我们的应用是一个单页的客户端应用，如果没有适当的服务器配置，用户在浏览器中直接访问 https://example.com/user/id，就会得到一个 404 错误。这就尴尬了。

不用担心：要解决这个问题，你需要做的就是在你的服务器上添加一个简单的回退路由。如果 URL 不匹配任何静态资源，它应提供与你的应用程序中的 index.html 相同的页面。

```
var history = require('connect-history-api-fallback');
app.use(history({
    index: '/index.html'
})); //注意放在所有的接口后面
```

22. $route$和router 的区别

$route用来获取路由的信息的，它是路由信息的一个对象，里面包含路由的一些基本信息，包括name、meta、path、hash、query、params、fullPath、matched、redirectedFrom等。而$router主要是用来操作路由的，它是VueRouter的实例，包含了一些路由的跳转方法push，go, replace，钩子函数等

23. Vue 路由守卫

vue-router 提供的导航守卫主要用来对路由的跳转进行监控，控制它的跳转或取消，路由守卫有全局的, 单个路由独享的, 或者组件级的。导航钩子有3个参数：

1、to:即将要进入的目标路由对象；
2、from:当前导航即将要离开的路由对象；
3、next：调用该方法后，才能进入下一个钩子函数（afterEach）。

```
router.beforeEach(async (to, from) => {
  if (
    // 检查用户是否已登录
    !isAuthenticated &&
    //  ❗ 避免无限重定向
    to.name !== 'Login'
  ) {
    // 将用户重定向到登录页面
    return { name: 'Login' }
  }
 })
```

24. Vue路由实现的底层原理

在Vue中利用数据劫持defineProperty在原型prototype上初始化了一些getter,分别是router代表当前Router的实例 、route 代表当前Router的信息。在install中也全局注册了router-view,router-link,其中的Vue.util.defineReactive, 这是Vue里面观察者劫持数据的方法，劫持_route，当_route触发setter方法的时候，则会通知到依赖的组件。

接下来在init中，会挂载判断是路由的模式，是history或者是hash,点击行为按钮，调用hashchange或者popstate的同时更_route,_route的更新会触发route-view的重新渲染。

25. 路由懒加载

Vue Router 支持开箱即用的动态导入，这意味着你可以用动态导入代替静态导入：

```
// 将
// import UserDetails from './views/UserDetails.vue'
// 替换成
const UserDetails = () => import('./views/UserDetails.vue')

const router = createRouter({
  // ...
  routes: [{ path: '/users/:id', component: UserDetails }],
})
```

26. 用过插槽吗？用的是具名插槽还是匿名插槽

用过，都使用过。插槽相当于预留了一个位置，可以将我们书写在组件内的内容放入，写一个插槽就会将组件内的内容替换一次，两次则替换两次。为了自定义插槽的位置我们可以给插槽取名，它会根据插槽名来插入内容，一一对应。

举例来说，这里有一个 <FancyButton> 组件，可以像这样使用：
template
<FancyButton>
  Click me! <!-- 插槽内容 -->
</FancyButton>

而 <FancyButton> 的模板是这样的：

template
<button class="fancy-btn">
  <slot></slot> <!-- 插槽出口 -->
</button>

27. Vue-loader解释一下

解析和转换 .vue 文件，提取出其中的逻辑代码 script、样式代码 style、以及 HTML 模版 template，再分别把它们交给对应的 Loader 去处理。

28. Vue和React中diff算法区别

vue和react的diff算法，都是忽略跨级比较，只做同级比较。vue diff时调动patch函数，参数是vnode和oldVnode，分别代表新旧节点。

1.vue对比节点。当节点元素相同，但是classname不同，认为是不同类型的元素，删除重建，而react认为是同类型节点，只是修改节点属性。

2.vue的列表对比，采用的是两端到中间比对的方式，而react采用的是从左到右依次对比的方式。当一个集合只是把最后一个节点移到了第一个，react会把前面的节点依次移动，而vue只会把最后一个节点移到第一个。总体上，vue的方式比较高效。

29. 请你说一下 Vue 中 create 和 mount 的区别

create为组件初始化阶段，在此阶段主要完成数据观测(data observer)，属性和方法的运算， watch/event 事件回调。然而，挂载阶段还没开始，此时还未生成真实的DOM，也就无法获取和操作DOM元素。而mount主要完成从虚拟DOM到真实DOM的转换挂载，此时html已经渲染出来了，所以可以直接操作dom节点。

30. axios是什么？怎么使用？描述使用它实现登录功能的流程?

axios 是请求后台资源的模块。 通过npm install axios -S来安装，在大多数情况下我们需要封装拦截器，在实现登录的过程

中我们一般在请求拦截器中来加入token，在响应请求器中通过判断后端返回的状态码来对返回的数据进行不同的处理。如果发送的是跨域请求，需在配置文件中 config/index.js 进行代理配置。

```
// Add a request interceptor
axios.interceptors.request.use(function (config) {
    // Do something before request is sent
    return config;
  }, function (error) {
    // Do something with request error
    return Promise.reject(error);
  });

// Add a response interceptor
axios.interceptors.response.use(function (response) {
    // Any status code that lie within the range of 2xx cause this function to trigger
    // Do something with response data
    return response;
  }, function (error) {
    // Any status codes that falls outside the range of 2xx cause this function to trigger
    // Do something with response error
    return Promise.reject(error);
  });
```

31. computed和watch的区别？watch实现原理？watch有几种写法?

计算属性computed :

1. 支持缓存，只有依赖数据发生改变，才会重新进行计算
2. 不支持异步，当computed内有异步操作时无效，无法监听数据的变化
3.computed 属性值会默认走缓存，计算属性是基于它们的响应式依赖进行缓存的，也就是基于data中声明过或者父组件传递的props中的数据通过计算得到的值
4. 如果一个属性是由其他属性计算而来的，这个属性依赖其他属性，是一个多对一或者一对一，一般用computed
5.如果computed属性属性值是函数，那么默认会走get方法；函数的返回值就是属性的属性值；在computed中的，属性都有一个get和一个set方法，当数据变化时，调用set方法。

```
computed: {
    // 一个计算属性的 getter
    publishedBooksMessage() {
      // `this` 指向当前组件实例
      return this.author.books.length > 0 ? 'Yes' : 'No'
    }
  }
```

侦听属性watch：

1. 不支持缓存，数据变，直接会触发相应的操作；
2.watch支持异步；

3.监听的函数接收两个参数，第一个参数是最新的值；第二个参数是输入之前的值；

4. 当一个属性发生变化时，需要执行对应的操作；一对多；
5. 监听数据必须是data中声明过或者父组件传递过来的props中的数据，当数据变化时，触发其他操作，函数有两个参数，
immediate：组件加载立即触发回调函数执行，
deep: 深度监听，为了发现对象内部值的变化，复杂类型的数据时使用，例如数组中的对象内容的改变，注意监听数组的变动不需要这么做。

```
watch: {
   // 每当 question 改变时，这个函数就会执行
   question(newQuestion, oldQuestion) {
     if (newQuestion.includes('?')) {
       this.getAnswer()
     }
   }
 },
watch: {
   someObject: {
     handler(newValue, oldValue) {
       // 注意：在嵌套的变更中，
       // 只要没有替换对象本身，
       // 那么这里的 `newValue` 和 `oldValue` 相同
     },
     deep: true
   }
 }
```

32. Vue $forceUpdate的原理

1、作用：

迫使 `Vue` 实例重新渲染。注意它仅仅影响实例本身和插入插槽内容的子组件，而不是所有子组件。

2、内部原理：

```
Vue.prototype.$forceUpdate = function () {
   const vm: Component = this
   if (vm._watcher) {
     vm._watcher.update()
   }
}
```

实例需要重新渲染是在依赖发生变化的时候会通知watcher，然后通知watcher来调用update方法，就是这么简单。

33. v-for key

- key是为Vue中的vnode标记的唯一id,通过这个key,我们的diff操作可以更准确、更快速
- diff算法的过程中,先会进行新旧节点的首尾交叉对比,当无法匹配的时候会用新节点的key与旧节点进行比对,然后超出差异.

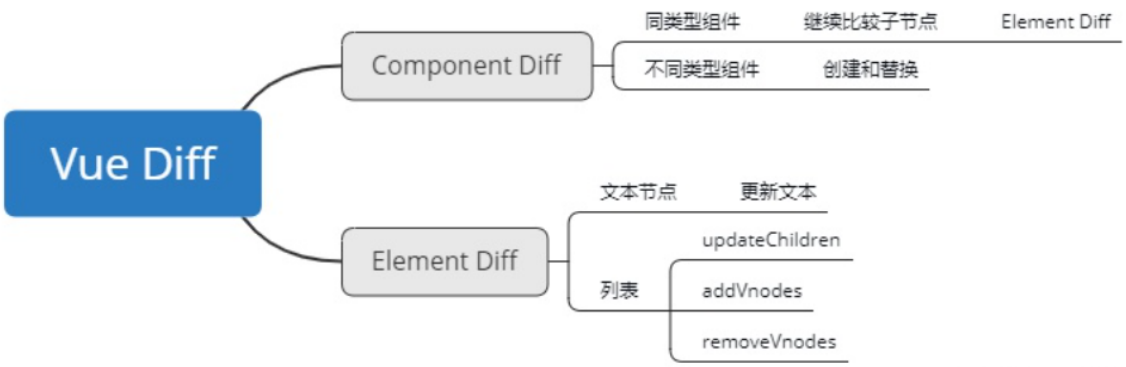
> diff程可以概括为：oldCh和newCh各有两个头尾的变量StartIdx和EndIdx，它们的2个变量相互比较，一共有4种比较方式。如果4种比较都没匹配，如果设置了key，就会用key进行比较，在比较的过程中，变量会往中间靠，一旦StartIdx>EndIdx表明oldCh和newCh至少有一个已经遍历完了，就会结束比较,这四种比较方式就是首、尾、旧尾新头、旧头新尾.
>
> 准确: 如果不加key,那么vue会选择复用节点(Vue的就地更新策略),导致之前节点的状态被保留下来,会产生一系列的bug. 快速: key的唯一性可以被Map数据结构充分利用,相比于遍历查找的时间复杂度`O(n)`,`Map`的时间复杂度仅仅为`O(1)`

34. 为什么要设置key值，可以用index吗？为什么不能?

vue中列表循环需加:key="唯一标识" 唯一标识可以是item里面id index等，因为vue组件高度复用增加Key可以标识组件的唯一性，为了更好地区别各个组件 key的作用主要是为了高效的更新虚拟DOM

35. diff复杂度原理及具体过程画图

diff算法是一种通过同层的树节点进行比较的高效算法，避免了对树进行逐层搜索遍历，所以时间复杂度只有O(n)。



diff算法有两个比较显著的特点：

1、比较只会在同层级进行, 不会跨层级比较。

2、在diff比较的过程中，循环从两边向中间收拢。

diff流程： 首先定义 oldStartIdx、newStartIdx、oldEndIdx 以及 newEndIdx 分别是新老两个 VNode 的两边的索引。

接下来是一个 while 循环，在这过程中，oldStartIdx、newStartIdx、oldEndIdx 以及 newEndIdx 会逐渐向中间靠拢。while 循环的退出条件是直到老节点或者新节点的开始位置大于结束位置。

while 循环中会遇到四种情况：

情形一：当新老 VNode 节点的 start 是同一节点时，直接 patchVnode 即可，同时新老 VNode 节点的开始索引都加 1。

情形二：当新老 VNode 节点的 end 是同一节点时，直接 patchVnode 即可，同时新老 VNode 节点的结束索引都减 1。

情形三：当老 VNode 节点的 start 和新 VNode 节点的 end 是同一节点时，这说明这次数据更新后 oldStartVnode 已经跑到了 oldEndVnode 后面去了。这时候在 patchVnode 后，还需要将当前真实 dom 节点移动到 oldEndVnode 的后面，同时老 VNode 节点开始索引加 1，新 VNode 节点的结束索引减 1。

情形四：当老 VNode 节点的 end 和新 VNode 节点的 start 是同一节点时，这说明这次数据更新后 oldEndVnode 跑到了 oldStartVnode 的前面去了。这时候在 patchVnode 后，还需要将当前真实 dom 节点移动到 oldStartVnode 的前面，同时老 VNode 节点结束索引减 1，新 VNode 节点的开始索引加 1。

while 循环的退出条件是直到老节点或者新节点的开始位置大于结束位置。

情形一：如果在循环中，oldStartIdx大于oldEndIdx了，那就表示oldChildren比newChildren先循环完毕，那么newChildren里面剩余的节点都是需要新增的节点，把[newStartIdx, newEndIdx]之间的所有节点都插入到DOM中

情形二：如果在循环中，newStartIdx大于newEndIdx了，那就表示newChildren比oldChildren先循环完毕，那么oldChildren里面剩余的节点都是需要删除的节点，把[oldStartIdx, oldEndIdx]之间的所有节点都删除

36. Vue组件中的Data为什么是函数，根组件却是对象呢?

综上可知，如果data是一个函数的话，这样每复用一次组件，就会返回一份新的data，类似于给每个组件实例创建一个私有的数据空间，让各个组件实例维护各自的数据。而单纯的写成对象形式，就使得所有组件实例共用了一份data，就会造成一个变了全都会变的结果。

所以说vue组件的data必须是函数。这都是因为js的特性带来的，跟vue本身设计无关。

37. Vue的组件通信

1、props和$emit

父组件向子组件传递数据是通过prop传递的，子组件传递数据给父组件是通过$emit触发事件

2、$attrs$和 listeners

3、中央事件总线 bus

上面两种方式处理的都是父子组件之间的数据传递，而如果两个组件不是父子关系呢？这种情况下可以使用中央事件总线的方式。新建一个Vue事件bus对象，然后通过bus.$emit$触发事件，bus.on监听触发的事件。

4、provide和inject

父组件中通过provider来提供变量，然后在子组件中通过inject来注入变量。不论子组件有多深，只要调用了inject那么就可以注入provider中的数据。而不是局限于只能从当前父组件的prop属性来获取数据，只要在父组件的生命周期内，子组件都可以调用。

5、v-model

父组件通过v-model传递值给子组件时，会自动传递一个value的prop属性，在子组件中通过this.$emit('input',val)自动修改v-model绑定的值

6、$parent$和 children

7、boradcast和dispatch

8、vuex处理组件之间的数据交互 如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候才有上面这一些方法可能不利于项目的维护，vuex的做法就是将这一些公共的数据抽离出来，然后其他组件就可以对这个公共数据进行读写操作，这样达到了解耦的目的。

38. 什么情况下使用 Vuex

> 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

39. Vuex可以直接修改state的值吗?

> 可以直接修改，但是极其不推荐，state的修改必须在mutation来修改，否则无法被devtool所监测，无法监测数据的来源，无法保存状态快照，也就无法实现时间漫游/回滚之类的操作。

40. 为什么Vuex的mutation不能做异步操作

> Vuex中所有的状态更新的唯一途径都是mutation，异步操作通过 Action 来提交 mutation实现，这样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。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，否则无法被devtools所监测。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。

41. 怎么修改Vuex中的状态？Vuex中有哪些方法

- 通过**this.$store.state.属性** 的方法来访问状态
- 通过**this.$store.commit('mutation中的方法')** 来修改状态

42. Vuex的缺点

> 如果您不打算开发大型单页应用，使用 Vuex 可能是繁琐冗余的，并且state中的值会伴随着浏览器的刷新而初始化，无缓存。

43. 什么是 Vue.nextTick()?

1、$nextTick 是在下次DOM更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的DOM，意思是 等你dom加载完毕以后再去调用nextTick()里面的数据内容

44. nextTick知道吗、实现的原理是什么？是宏任务还是微任务?

微任务

原理：

nextTick方法主要是使用了宏任务和微任务，定义了一个异步方法，多次调用nextTick会将方法存入队列中，通过这个异步方法清空队列。

作用： nextTick用于下次Dom更新循环结束之后执行延迟回调，在修改数据之后使用nextTick用于下次Dom更新循环结束之后执行延迟回调，在修改数据之后使用nextTick用于下次Dom更新循环结束之后执行延迟回调，在修改数据之后使用nextTick,则可以在回调中获取更新后的DOM。

45. 虚拟 dom 为什么会提高性能?

虚拟DOM其实就是一个JavaScript对象。通过这个JavaScript对象来描述真实DOM，真实DOM的操作，一般都会对某块元素的整体重新渲染，采用虚拟DOM的话，当数据变化的时候，只需要局部刷新变化的位置就好了，

虚拟 `dom` 相当于在 `js` 和真实 `dom` 中间加了一个缓存，利用 `dom diff` 算法避免了没有必要的 `dom` 操作，从而提高性能

具体实现步骤如下

- 用 `JavaScript` 对象结构表示 `DOM` 树的结构；然后用这个树构建一个真正的 `DOM` 树，插到文档当中
- 当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异
- 把2所记录的差异应用到步骤1所构建的真正的 `DOM` 树上，视图就更新

46. 你做过哪些Vue的性能优化?

1、首屏加载优化

2、路由懒加载

```
{
  path: '/',
  name: 'home',
  component: () => import('./views/home/index.vue'),
  meta: { isShowHead: true }
}
```

3、开启服务器 Gzip

开启 Gzip 就是一种压缩技术，需要前端提供压缩包，然后在服务器开启压缩，文件在服务器压缩后传给浏览器，浏览器解压后进行再进行解析。首先安装 webpack 提供的 `compression-webpack-plugin` 进行压缩,然后在 vue.config.js：

```
const CompressionWebpackPlugin = require('compression-webpack-plugin')
const productionGzipExtensions = ['js', 'css']……plugins: [
  new CompressionWebpackPlugin(
    {
      algorithm: 'gzip',
      test:     new RegExp('\\.(' + productionGzipExtensions.join('|') +                ')$'),
      threshold: 10240,
      minRatio: 0.8
       }
)]….
```

4、启动 CDN 加速

我们继续采用 cdn 的方式来引入一些第三方资源，就可以缓解我们服务器的压力，原理是将我们的压力分给其他服务器点。

5、代码层面优化

- computed 和 watch 区分使用场景

  > computed： 是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值。当我们需要进行数值计算，并且依赖于其它数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算；
  > watch：类似于某些数据的监听回调 ，每当监听的数据变化时都会执行回调进行后续操作；当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，使用 watch 选项允许我们执行异步操作 (访问一个 API)，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

- v-if 和 v-show 区分使用场景 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 则适用于需要非常频繁切换条件的场景。这里要说的优化点在于减少页面中 dom 总数，我比较倾向于使用 v-if，因为减少了 dom 数量。
- v-for 遍历必须为 item 添加 key，且避免同时使用 v-if v-for 遍历必须为 item 添加 key，循环调用子组件时添加 key，key 可以唯一标识一个循环个体，可以使用例如 item.id 作为 key 避免同时使用 v-if，v-for 比 v-if 优先级高，如果每一次都需要遍历整个数组，将会影响速度。

6、Webpack 对图片进行压缩

7、避免内存泄漏

8、减少 ES6 转为 ES5 的冗余代码

47. Vue的常用修饰符

一、v-model修饰符

1、.lazy：

输入框改变，这个数据就会改变，lazy这个修饰符会在光标离开input框才会更新数据：

```
<input type="text" v-model.lazy="value">
```

2、.trim：

输入框过滤首尾的空格：

```
<input type="text" v-model.trim="value">
```

3、.number：

先输入数字就会限制输入只能是数字，先字符串就相当于没有加number，注意，不是输入框不能输入字符串，是这个数据是数字：

```
<input type="text" v-model.number="value">
```

二、事件修饰符

4、.stop：

阻止事件冒泡，相当于调用了event.stopPropagation()方法：

```
<button @click.stop="test">test</button>
```

5、.prevent：

阻止默认行为，相当于调用了event.preventDefault()方法，比如表单的提交、a标签的跳转就是默认事件：

```
<a @click.prevent="test">test</a>
```

6、.self：

只有元素本身触发时才触发方法，就是只有点击元素本身才会触发。比如一个div里面有个按钮，div和按钮都有事件，我们点击按钮，div绑定的方法也会触发，如果div的click加上self，只有点击到div的时候才会触发，变相的算是阻止冒泡：

```
<div @click.self="test"></div>
```

7、.once：

事件只能用一次，无论点击几次，执行一次之后都不会再执行

```
<div @click.once="test"></div>
```

8、.capture：

事件的完整机制是捕获-目标-冒泡，事件触发是目标往外冒泡

9、.sync

对prop进行双向绑定

10、.keyCode：

监听按键的指令，具体可以查看vue的键码对应表

48. Vue 中 template 的编译过程

vue template模板编译的过程经过parse()生成ast(抽象语法树),optimize对静态节点优化，generate()生成render字符串 之后调用new Watcher()函数，用来监听数据的变化，render 函数就是数据监听的回调所调用的，其结果便是重新生成 vnode。 当这个 render 函数字符串在第一次 mount、或者绑定的数据更新的时候，都会被调用，生成 Vnode。 如果是数据的更新，那么 Vnode 会与数据改变之前的 Vnode 做 diff，对内容做改动之后，就会更新到 我们真正的 DOM

49. 谈谈你对Vue3.0有什么了解?

六大亮点

1. 性能比vue2.x快1.2~2倍
2. 支持tree-shaking，按需编译，体积比vue2.x更小
3. 支持组合API
4. 更好的支持TS
5. 更先进的组件

性能比vue2.x快1.2~2倍如何实现的呢

1.diff算法更快

vue2.0是需要全局去比较每个节点的，若发现有节点发生变化后，就去更新该节点

vue3.0是在创建虚拟dom中，会根据DOM的的内容会不会发生内容变化，添加静态标记， 谁有flag！比较谁。

2、静态提升

vue2中无论元素是否参与更新，每次都会重新创建，然后再渲染 vue3中对于不参与更新的元素，会做静态提升，只被创建一次，在渲染时直接复用即可

3、事件侦听缓存

默认情况下，onclick为动态绑定，所以每次都会追踪它的变化，但是因为是同一函数，没有必要追踪变化，直接缓存复用即可

在之前会添加静态标记8 会把点击事件当做动态属性 会进行diff算法比较， 但是在事件监听缓存之后就没有静态标记了，就会进行缓存复用

为什么vue3.0体积比vue2.x小

在vue3.0中创建vue项目 除了vue-cli，webpack外还有 一种创建方法是Vite Vite是作者开发的一款有意取代webpack的工具，其实现原理是利用ES6的import会发送请求去加载文件的特性，拦截这些请求，做一些预编译，省去webpack冗长的打包时间

50. vue3.0组合API

说一说vue3.0的组合API跟之前vue2.0在完成业务逻辑上的区别：

在vue2.0中： 主要是往data 和method里面添加内容，一个业务逻辑需要什么data和method就往里面添加，而组合API就是 有一个自己的方法，里面有自己专注的data 和method。

再说一下组合API的本质是什么： 首先composition API（组合API） 和 Option API（vue2.0中的data和method） 可以共用 composition API（组合API）本质就是把内容添加到Option API中进行使用

51. ref和reactive的简单理解

1.ref和reactive都是vue3的监听数据的方法，本质是proxy 2.ref 基本类型复杂类型都可以监听(我们一般用ref监听基本类型)，reactive只能监听对象（arr，json） 3.ref底层还是reactive，ref是对reactive的二次包装， ref定义的数据访问的时候要多一个.value

52. Vuex和redux有什么区别？他们的共同思想。

Redux和Vuex区别

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

共同思想

- 单一的数据源
- 变化可以预测
- `本质上:` Redux和Vuex都是对MVVM思想的服务，将数据从视图中抽离的一种方案
- `形式上:` Vuex借鉴了Redux，将store作为全局的数据中心，进行数据管理

53. 简单说一下 微信小程序 与 Vue 的区别

1、生命周期：

`小程序` 的钩子函数要简单得多 。 `vue` 的钩子函数在跳转新页面时，钩子函数都会触发，但是 `小程序` 的钩子函数，页面不同的跳转方式，触发的钩子并不一样。

在页面加载请求数据时，两者钩子的使用有些类似， `vue` 一般会在 `created` 或者 `mounted` 中请求数据，而在 `小程序` ，会在 `onLoad` 或者 `onShow` 中请求数据。

2、数据绑定：

vue动态绑定一个变量的值为元素的某个属性的时候，会在变量前面加上冒号：

```
<img :src="imgSrc"/>
```

小程序 绑定某个变量的值为元素属性时，会用两个大括号括起来，如果不加括号，为被认为是字符串

```
<image src="{{imgSrc}}"></image>
```

3、列表循环

4、显示与隐藏元素

`vue` 中，使用 `v-if` 和 `v-show` 控制元素的显示和隐藏

`小程序` 中，使用 `wx-if` 和 `hidden` 控制元素的显示和隐藏

5、事件处理

`vue` ：使用 `v-on:event` 绑定事件，或者使用 `@event` 绑定事件

`小程序` 中，全用 `bindtap(bind+event)` ，或者 `catchtap(catch+event)` 绑定事件

6、数据的双向绑定

在 `vue` 中,只需要再 `表单` 元素上加上 `v-model` ,然后再绑定 `data` 中对应的一个值，当表单元素内容发生变化时， `data` 中对应的值也会相应改变 。

当表单内容发生变化时，会触发表单元素上绑定的方法，然后在该方法中，通过 `this.setData({key:value})` 来将表单上的值赋值给 `data` 中的对应值 。

7、绑定事件传参

在 `vue` 中，绑定事件传参挺简单，只需要在触发事件的方法中，把需要传递的数据作为形参传入就可以了

在 `小程序` 中，不能直接在绑定事件的方法中传入参数，需要将参数作为属性值，绑定到元素上的 `data-` 属性上，然后在方法中，通过 `e.currentTarget.dataset.*` 的方式获取

8、父子组件通信

父组件向子组件传递数据，只需要在子组件通过 `v-bind` 传入一个值，在子组件中，通过 `props` 接收，即可完成数据的传递

父组件向子组件通信和 `vue` 类似，但是 `小程序` 没有通过 `v-bind`，而是直接将值赋值给一个变量 在子组件 `properties` 中，接收传递的值