Vue3开发与实战
一.vue3介绍
1. 官网初识
2.环境搭建
2-1 线上尝试
2-2 CDN使用
2-3 Vue CLI
2-4 Vite
二.vue3基础
1.模板语法
1-1 我的第一个vue应用
1-2 应用背后的真相
1-3 模板语法-新的皮肤来了
1-4 Todolist-来个案例
1-5 点击变心案例 - 是变色
1-6 v-html- 模板的陷阱
2.class与style
2-1 class的绑定-方法太多了
2-2 style的绑定-同上
3.条件渲染
3-1 条件渲染-生或死的选择
4.列表渲染
4-1 v-for列表渲染 - 影分身术
4-2 key设置 - 性能的保障
4-3 数组变动侦测
4-4 模糊搜索案例
5.事件处理器
5-1 事件处理器 - 告别原生事件
5-2 事件修饰符 - 事件偷懒符?
6.表单控件绑定
6-1 表单输入绑定-一根绳上的蚂蚱
6-2 购物车案例
6-3 表单修饰符
7.计算属性
7-1 计算属性 - 很聪明，会缓存
7-2 之前案例的小改造
7-3 侦听器watch的对比
8.数据请求
8-1 Fetch
8-2 axios
9.过滤器
9-1 vue3过滤器不支持了 - 怎么办?
三.vue3进阶
1.单文件组件
1-1 组件定义 - 重塑经脉，断了?
1-2 单文件组件(SFC) - 独立日
1-3 Vue-CLI创建项目 - 锅灶升级
1-4 Vite 创建项目 - 官方推荐
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
2-2 属性验证&默认属性
2-3 子传父 - 自定义事件
2-4 $refs - 父组件的强权
2-5 $Misplaced &$root - 子组件的无法无天
2-6 跨级通信- provide&inject
2-7 动态组件 - 墙头草
2-8 组件中的v-model
2-9 异步组件
3.组件插槽
3-1 插槽的基本应用
3-2 具名插槽
3-3 作用域插槽
4. 生命周期
5.组件的封装
6.自定义指令
6-1 指令写法&钩子
7.过渡效果
7-1 过渡效果
7-2 列表过渡
7-3 可复用过渡

Vue3开发与实战

作者：kerwin

版本：QF1.0

公众号: 大前端私房菜

> **千锋精品教程，好学得不像实力派！**

一.vue3介绍

1. 官网初识

渐进式

JavaScript 框架

> 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。

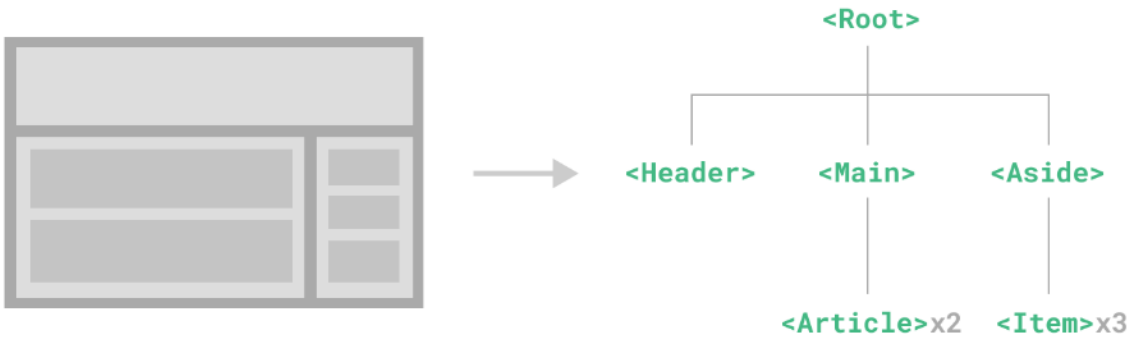
在 3.x 中，过滤器已移除，且不再支持。取而代之的是，我们建议用方法调用或计算属性来替换它们。

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：

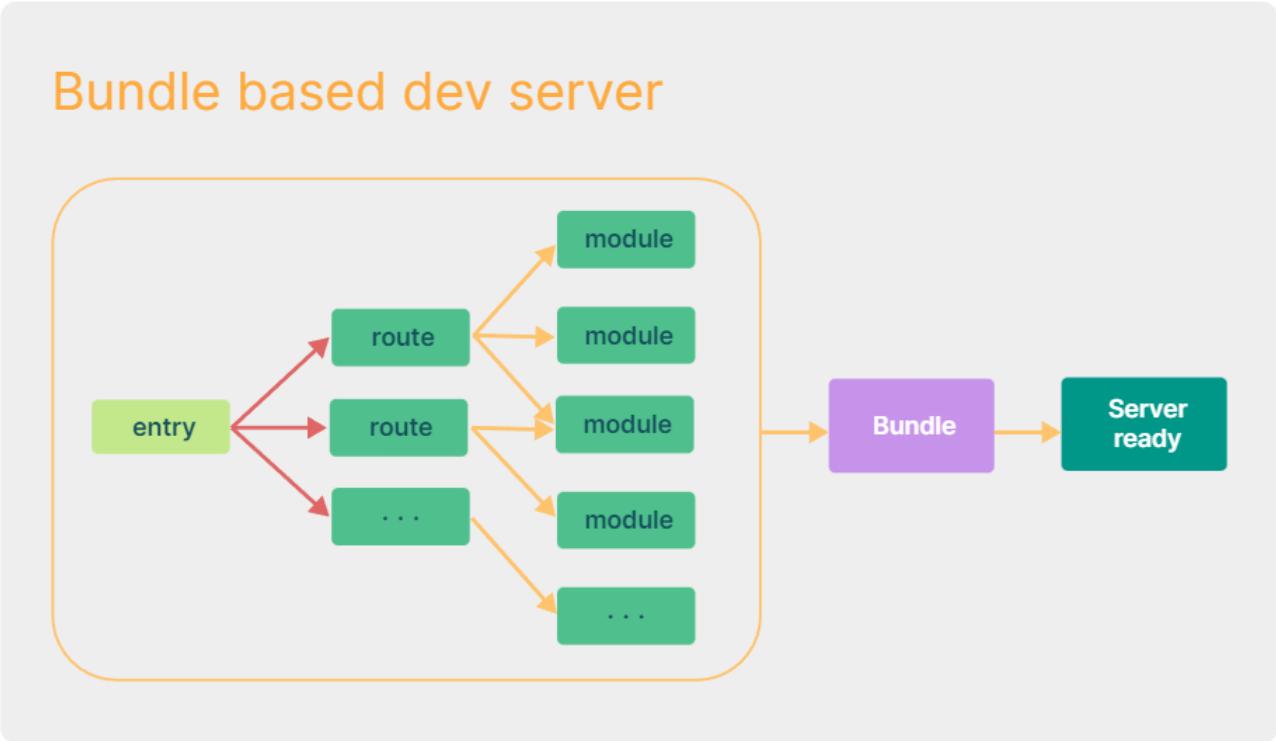
```
vue create my-project
# OR
vue ui
```

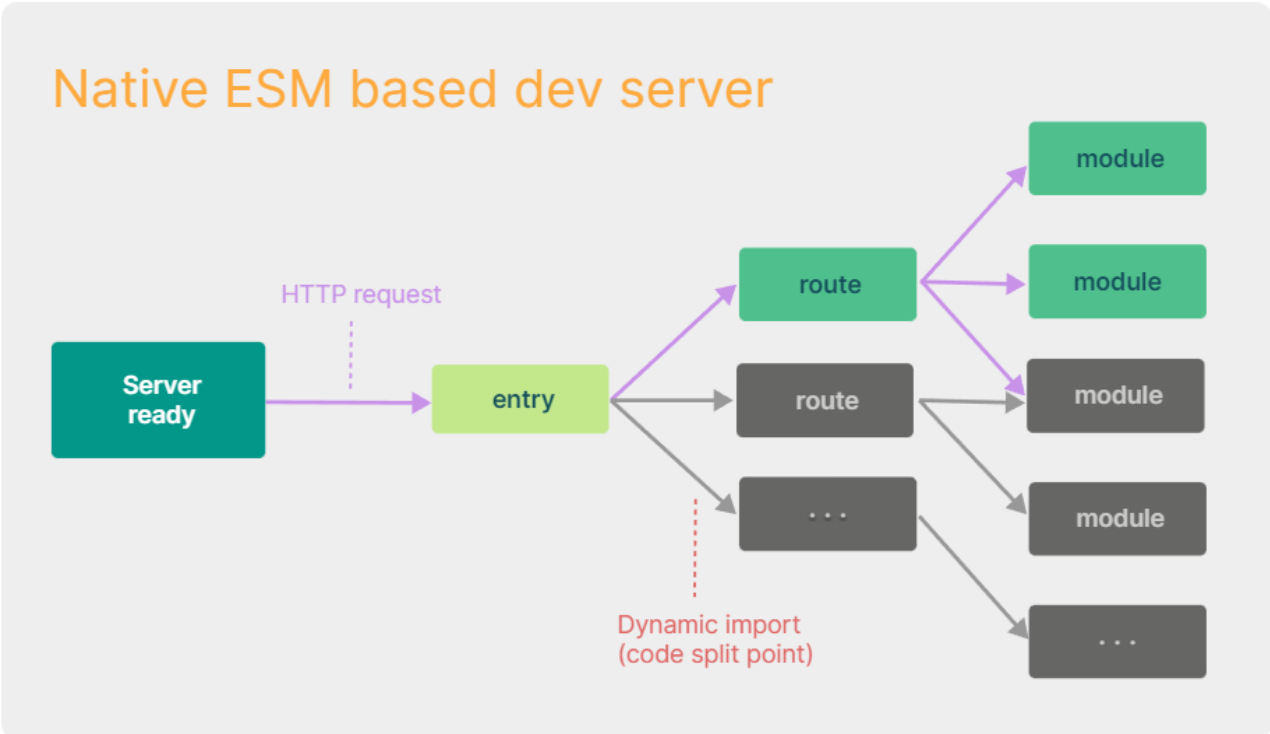
1-4 Vite 创建项目 - 官方推荐

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>
```

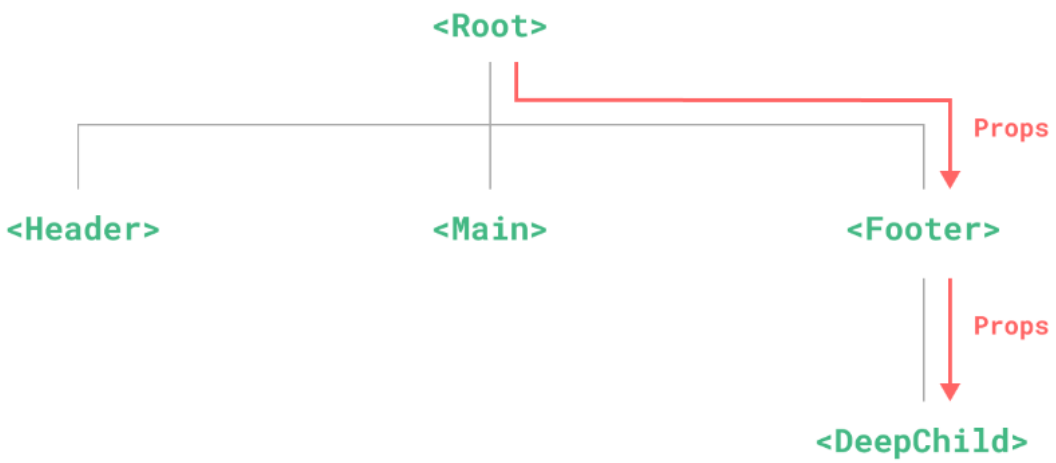
```
methods:{
      handleRegister(){
        console.log(this.$refs.myusername.value,this.$refs.mypassword.value)
      },
      handelReset(){
        this.$refs.myusername.value=""
        this.$refs.mypassword.value=""
      }
}
```

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

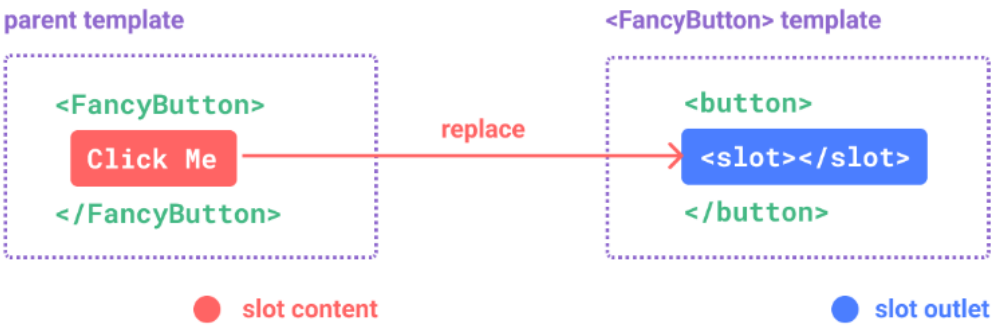
```
const AsyncComp = defineAsyncComponent({
  // 加载函数
```

```
  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。

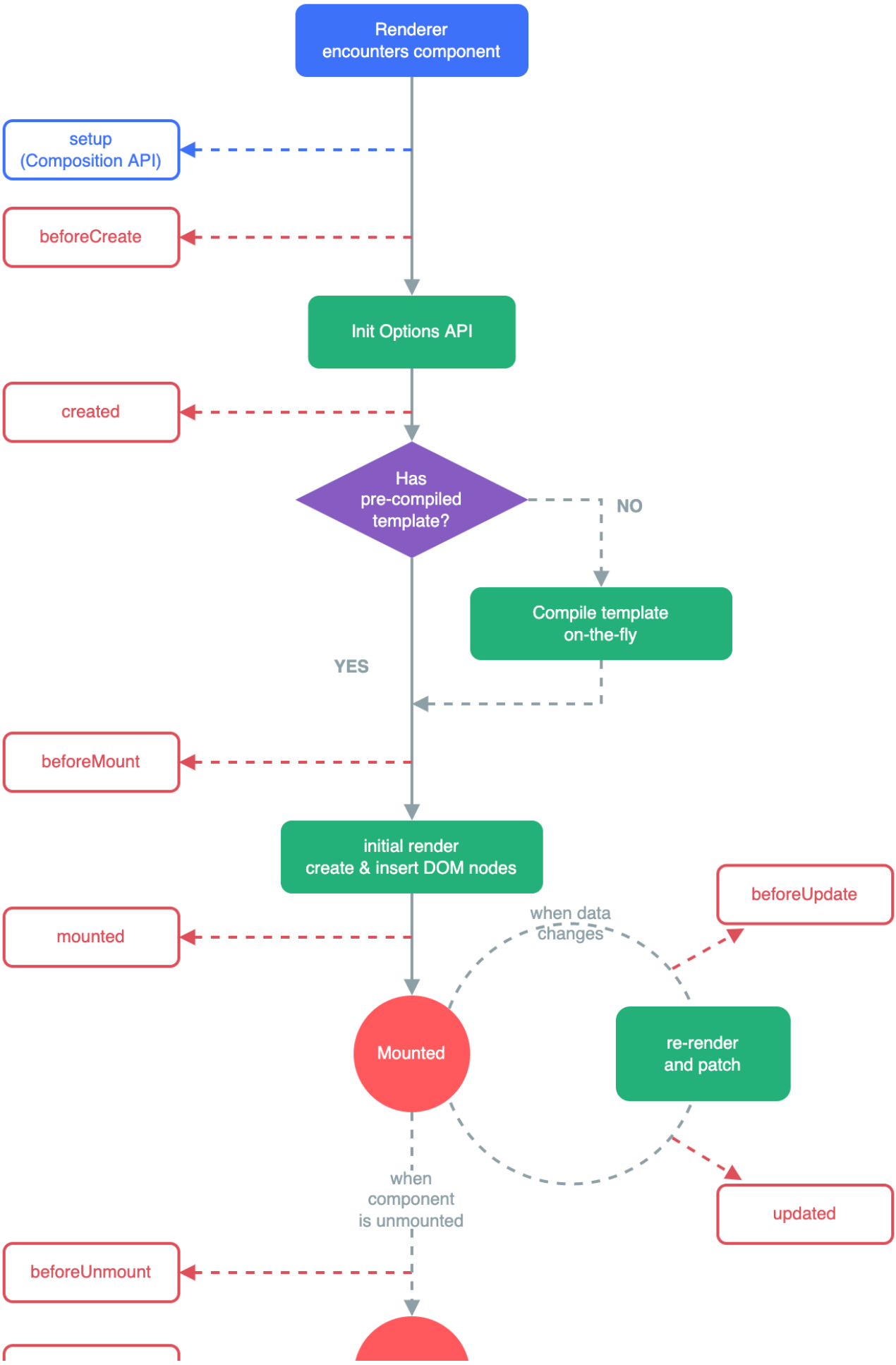
(4)mounted()所有同步子组件都已经被挂载。这个钩子通常用于执行需要访问组件所渲染的 DOM 树相关的副作用

(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>
```

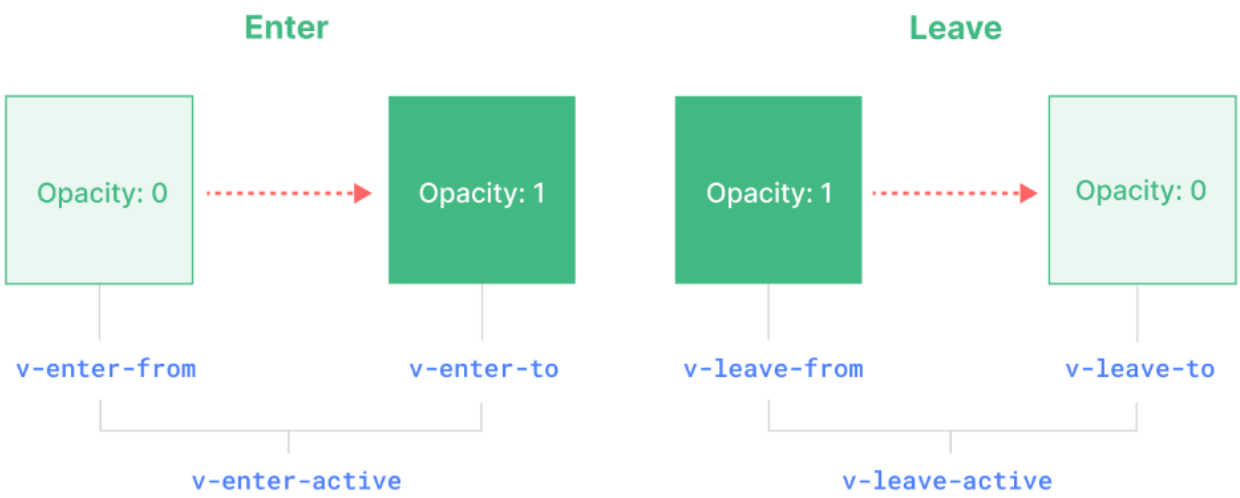
```
app.directive('color', (el, binding) => {
  // 这会在 `mounted` 和 `updated` 时都调用
  el.style.color = binding.value
})
```

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>
```