

Vue3快速上手



1.Vue3简介

- 2020年9月18日，Vue.js发布3.0版本，代号：One Piece（海贼王）
- 耗时2年多、[2600+次提交](#)、[30+个RFC](#)、[600+次PR](#)、[99位贡献者](#)
- github上的tags地址：<https://github.com/vuejs/vue-next/releases/tag/v3.0.0>

2.Vue3带来了什么

1.性能的提升

- 打包大小减少41%
- 初次渲染快55%，更新渲染快133%
- 内存减少54%

.....

2.源码的升级

- 使用Proxy代替defineProperty实现响应式
- 重写虚拟DOM的实现和Tree-Shaking

.....

3.拥抱TypeScript

- Vue3可以更好的支持TypeScript

4.新的特性

1. Composition API（组合API）

- setup配置
- ref与reactive
- watch与watchEffect

- provide与inject
 -
2. 新的内置组件
- Fragment
 - Teleport
 - Suspense
3. 其他改变
- 新的生命周期钩子
 - data 选项应始终被声明为一个函数
 - 移除keyCode支持作为 v-on 的修饰符
 -

一、创建Vue3.0工程

1.使用 vue-cli 创建

官方文档: <https://cli.vuejs.org/zh/guide/creating-a-project.html#vue-create>

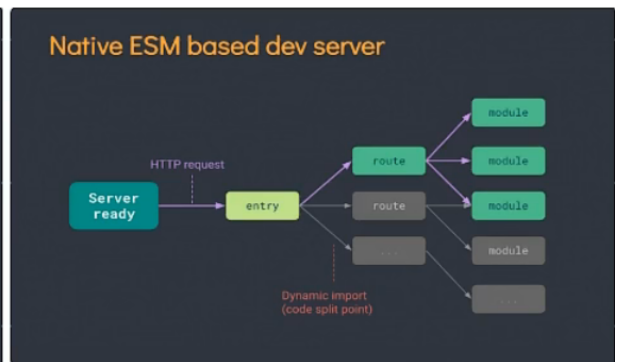
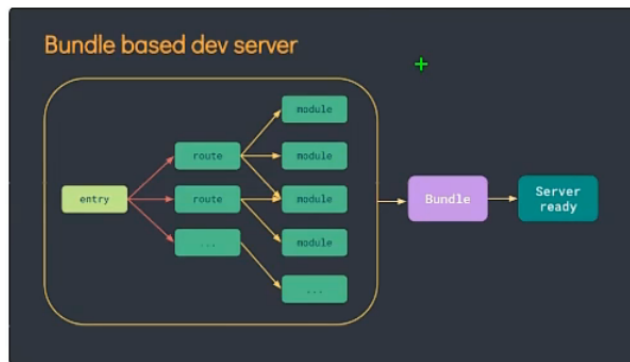
```
1  ## 查看@vue/cli版本, 确保@vue/cli版本在4.5.0以上
2  vue --version
3  ## 安装或者升级你的@vue/cli
4  npm install -g @vue/cli
5  ## 创建
6  vue create vue_test
7  ## 启动
8  cd vue_test
9  npm run serve
```

2.使用 vite 创建

官方文档: <https://v3.cn.vuejs.org/guide/installation.html#vite>

vite官网: <https://vitejs.cn>

- 什么是vite? —— 新一代前端构建工具。
- 优势如下:
 - 开发环境中, 无需打包操作, 可快速的冷启动。
 - 轻量快速的热重载 (HMR) 。
 - 真正的按需编译, 不再等待整个应用编译完成。
- 传统构建 与 vite构建对比图



```

1  ## 创建工程
2  npm init vite-app <project-name>
3  ## 进入工程目录
4  cd <project-name>
5  ## 安装依赖
6  npm install
7  ## 运行
8  npm run dev

```

二、常用 Composition API

官方文档: <https://v3.cn.vuejs.org/guide/composition-api-introduction.html>

1. 拉开序幕的setup

1. 理解: Vue3.0中一个新的配置项, 值为一个函数。
2. setup是所有**Composition API (组合API)** “表演的舞台”。
3. 组件中所用到的: 数据、方法等等, 均要配置在setup中。
4. setup函数的两种返回值:
 1. 若返回一个对象, 则对象中的属性、方法, 在模板中均可以直接使用。(重点关注!)
 2. 若返回一个渲染函数: 则可以自定义渲染内容。(了解)
5. 注意点:
 1. 尽量不要与Vue2.x配置混用
 - Vue2.x配置 (data、methos、computed...) 中**可以访问到**setup中的属性、方法。
 - 但在setup中**不能访问到**Vue2.x配置 (data、methos、computed...) 。
 - 如果有重名, setup优先。
 2. setup不能是一个async函数, 因为返回值不再是return的对象, 而是promise, 模板看不到return对象中的属性。(后期也可以返回一个Promise实例, 但需要Suspense和异步组件的配合)

2.ref函数

- 作用: 定义一个响应式的数据
- 语法: `const xxx = ref(initValue)`
 - 创建一个包含响应式数据的**引用对象 (reference对象, 简称ref对象)**。

- JS中操作数据: `xxx.value`
- 模板中读取数据: 不需要.value, 直接: `<div>{{xxx}}</div>`
- 备注:
 - 接收的数据可以是: 基本类型、也可以是对象类型。
 - 基本类型的数据: 响应式依然是靠 `Object.defineProperty()` 的 `get` 与 `set` 完成的。
 - 对象类型的数据: 内部“求助”了Vue3.0中的一个新函数—— `reactive` 函数。

3.reactive函数

- 作用: 定义一个**对象类型**的响应式数据 (基本类型不要用它, 要用 `ref` 函数)
- 语法: `const 代理对象= reactive(源对象)` 接收一个对象 (或数组), 返回一个**代理对象 (Proxy的实例对象, 简称proxy对象)**
- `reactive`定义的响应式数据是“深层次的”。
- 内部基于 ES6 的 Proxy 实现, 通过代理对象操作源对象内部数据进行操作。

4.Vue3.0中的响应式原理

vue2.x的响应式

- 实现原理:
 - 对象类型: 通过 `Object.defineProperty()` 对属性的读取、修改进行拦截 (数据劫持)。
 - 数组类型: 通过重写更新数组的一系列方法来实现拦截。 (对数组的变更方法进行了包裹)。

```
1 Object.defineProperty(data, 'count', {
2   get () {},
3   set () {}
4 })
```

- 存在问题:
 - 新增属性、删除属性, 界面不会更新。
 - 直接通过下标修改数组, 界面不会自动更新。

Vue3.0的响应式

- 实现原理:
 - 通过Proxy (代理): 拦截对象中任意属性的变化, 包括: 属性值的读写、属性的添加、属性的删除等。
 - 通过Reflect (反射): 对源对象的属性进行操作。
 - MDN文档中描述的Proxy与Reflect:
 - Proxy: https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy
 - Reflect: https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Reflect

```
1 new Proxy(data, {
2   // 拦截读取属性值
```

```

3      get (target, prop) {
4          return Reflect.get(target, prop)
5      },
6      // 拦截设置属性值或添加新属性
7      set (target, prop, value) {
8          return Reflect.set(target, prop, value)
9      },
10     // 拦截删除属性
11     deleteProperty (target, prop) {
12         return Reflect.deleteProperty(target, prop)
13     }
14 })
15
16 proxy.name = 'tom'

```

5.reactive对比ref

- 从定义数据角度对比：
 - ref用来定义：**基本类型数据**。
 - reactive用来定义：**对象（或数组）类型数据**。
 - 备注：ref也可以用来定义**对象（或数组）类型数据**，它内部会自动通过 `reactive` 转为**代理对象**。
- 从原理角度对比：
 - ref通过 `Object.defineProperty()` 的 `get` 与 `set` 来实现响应式（数据劫持）。
 - reactive通过使用**Proxy**来实现响应式（数据劫持），并通过**Reflect**操作**源对象**内部的数据。
- 从使用角度对比：
 - ref定义的数据：操作数据**需要** `.value`，读取数据时模板中直接读取**不需要** `.value`。
 - reactive定义的数据：操作数据与读取数据：**均不需要** `.value`。

6.setup的两个注意点

- setup执行的时机
 - 在beforeCreate之前执行一次，this是undefined。
- setup的参数
 - props: 值为对象，包含：组件外部传递过来，且组件内部声明接收了的属性。
 - context: 上下文对象
 - attrs: 值为对象，包含：组件外部传递过来，但没有在props配置中声明的属性，相当于 `this.$attrs`。
 - slots: 收到的插槽内容，相当于 `this.$slots`。
 - emit: 分发自定义事件的函数，相当于 `this.$emit`。

7.计算属性与监视

1.computed函数

- 与Vue2.x中computed配置功能一致

- 写法

```
1 import {computed} from 'vue'
2
3 setup(){
4   ...
5   //计算属性—简写
6   let fullName = computed(()=>{
7     return person.firstName + '-' + person.lastName
8   })
9   //计算属性—完整
10  let fullName = computed({
11    get(){
12      return person.firstName + '-' + person.lastName
13    },
14    set(value){
15      const nameArr = value.split('-')
16      person.firstName = nameArr[0]
17      person.lastName = nameArr[1]
18    }
19  })
20 }
```

2.watch函数

- 与Vue2.x中watch配置功能一致
- 两个小“坑”：
 - 监视reactive定义的响应式数据时：oldValue无法正确获取、强制开启了深度监视（deep配置失效）。
 - 监视reactive定义的响应式数据中某个属性时：deep配置有效。

```
1 //情况一：监视ref定义的响应式数据
2 watch(sum, (newValue, oldValue)=>{
3   console.log('sum变化了', newValue, oldValue)
4 }, {immediate:true})
5
6 //情况二：监视多个ref定义的响应式数据
7 watch([sum, msg], (newValue, oldValue)=>{
8   console.log('sum或msg变化了', newValue, oldValue)
9 })
10
11 /* 情况三：监视reactive定义的响应式数据
12     若watch监视的是reactive定义的响应式数据，则无法正确获得oldValue！！
13     若watch监视的是reactive定义的响应式数据，则强制开启了深度监视
14 */
15 watch(person, (newValue, oldValue)=>{
16   console.log('person变化了', newValue, oldValue)
17 }, {immediate:true, deep:false}) //此处的deep配置不再奏效
18
19 //情况四：监视reactive定义的响应式数据中的某个属性
20 watch(()=>person.job, (newValue, oldValue)=>{
```

```

21     console.log('person的job变化了',newValue,oldValue)
22 },{immediate:true,deep:true})
23
24 //情况五: 监视reactive定义的响应式数据中的某些属性
25 watch([()=>person.job,()=>person.name],(newValue,oldValue)=>{
26     console.log('person的job变化了',newValue,oldValue)
27 },{immediate:true,deep:true})
28
29 //特殊情况
30 watch(()=>person.job,(newValue,oldValue)=>{
31     console.log('person的job变化了',newValue,oldValue)
32 },{deep:true}) //此处由于监视的是reactive素定义的对象中的某个属性, 所以deep配置有效

```

3.watchEffect函数

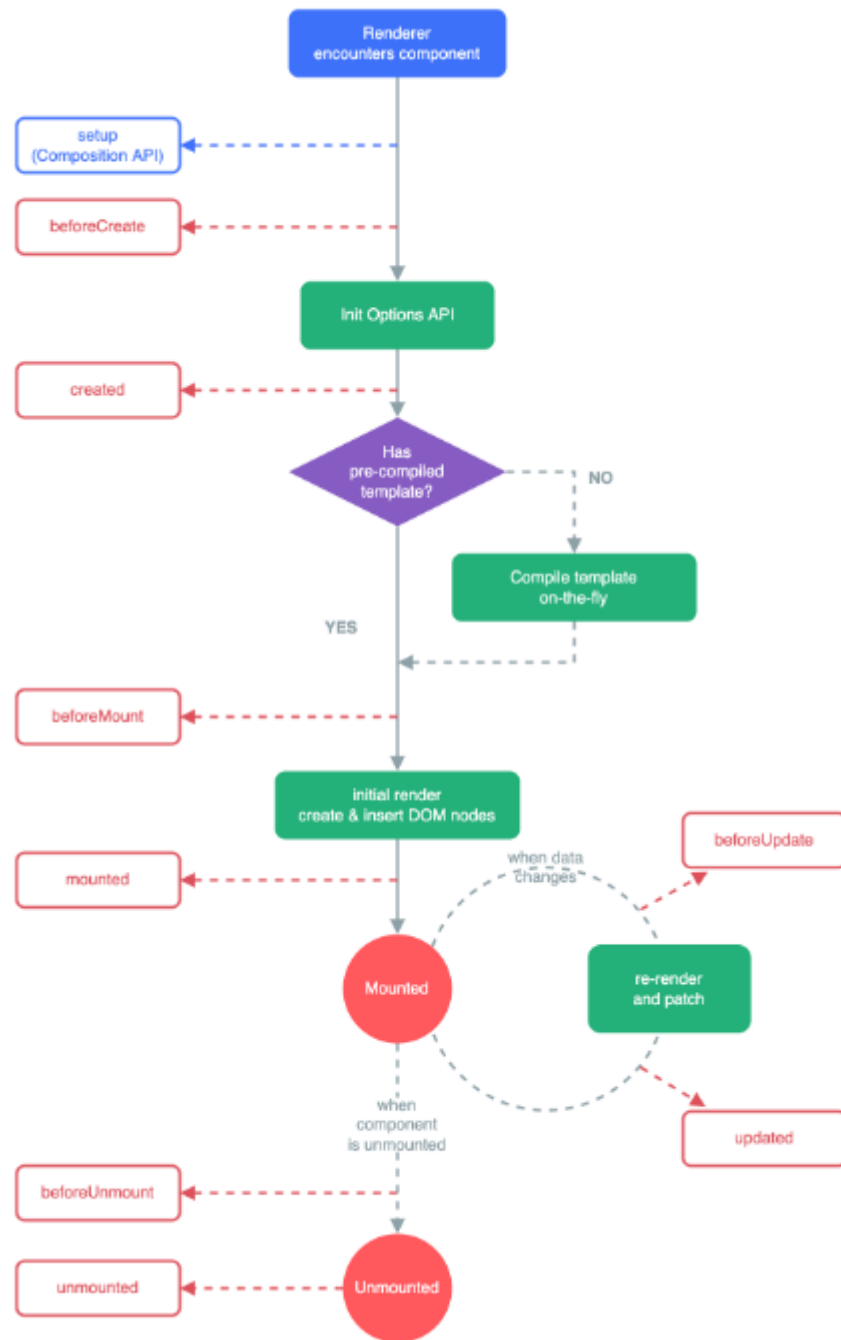
- watch的套路是：既要指明监视的属性，也要指明监视的回调。
- watchEffect的套路是：不用指明监视哪个属性，监视的回调中用到哪个属性，那就监视哪个属性。
- watchEffect有点像computed：
 - 但computed注重的计算出来的值（回调函数的返回值），所以必须要写返回值。
 - 而watchEffect更注重的是过程（回调函数的函数体），所以不用写返回值。

```


1 //watchEffect所指定的回调中用到的数据只要发生变化, 则直接重新执行回调。
2 watchEffect(()=>{
3     const x1 = sum.value
4     const x2 = person.age
5     console.log('watchEffect配置的回调执行了')
6 })

```

8.生命周期



- Vue3.0中可以继续使用Vue2.x中的生命周期钩子，但有有两个被更名：
 - `beforeDestroy` 改名为 `beforeUnmount`
 - `destroyed` 改名为 `unmounted`
- Vue3.0也提供了 Composition API 形式的生命周期钩子，与Vue2.x中钩子对应关系如下：
 - `beforeCreate` ==> `setup()`
 - `created` ==> `setup()`
 - `beforeMount` ==> `onBeforeMount`
 - `mounted` ==> `onMounted`
 - `beforeUpdate` ==> `onBeforeUpdate`
 - `updated` ==> `onUpdated`
 - `beforeUnmount` ==> `onBeforeUnmount`

○ `unmounted`  `onUnmounted`

9.自定义hook函数

- 什么是hook? —— 本质是一个函数, 把setup函数中使用的Composition API进行了封装。
- 类似于vue2.x中的mixin。
- 自定义hook的优势: 复用代码, 让setup中的逻辑更清楚易懂。

10.toRef

- 作用: 创建一个 ref 对象, 其value值指向另一个对象中的某个属性。
- 语法: `const name = toRef(person, 'name')`
- 应用: 要将响应式对象中的某个属性单独提供给外部使用时。
- 扩展: `toRefs` 与 `toRef` 功能一致, 但可以批量创建多个 ref 对象, 语法: `toRefs(person)`

三、其它 Composition API

1.shallowReactive 与 shallowRef

- shallowReactive: 只处理对象最外层属性的响应式(浅响应式)。
- shallowRef: 只处理基本数据类型的响应式, 不进行对象的响应式处理。
- 什么时候使用?
 - 如果有一个对象数据, 结构比较深, 但变化时只是外层属性变化 ==> shallowReactive。
 - 如果有一个对象数据, 后续功能不会修改该对象中的属性, 而是生新的对象来替换 ==> shallowRef。

2.readonly 与 shallowReadonly

- readonly: 让一个响应式数据变为只读的(深只读)。
- shallowReadonly: 让一个响应式数据变为只读的(浅只读)。
- 应用场景: 不希望数据被修改时。

3.toRaw 与 markRaw

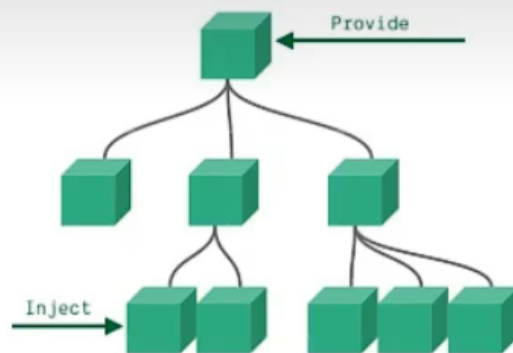
- toRaw:
 - 作用: 将一个由 `reactive` 生成的**响应式对象**转为**普通对象**。
 - 使用场景: 用于读取响应式对象对应的普通对象, 对这个普通对象的所有操作, 不会引起页面更新。
- markRaw:
 - 作用: 标记一个对象, 使其永远不会再成为响应式对象。
 - 应用场景:
 1. 有些值不应被设置为响应式的, 例如复杂的第三方类库等。
 2. 当渲染具有不可变数据源的大列表时, 跳过响应式转换可以提高性能。

4.customRef

- 作用：创建一个自定义的 ref，并对其依赖项跟踪和更新触发进行显式控制。
- 实现防抖效果：

```
1 <template>
2   <input type="text" v-model="keyword">
3   <h3>{{keyword}}</h3>
4 </template>
5
6 <script>
7   import {ref,customRef} from 'vue'
8   export default {
9     name: 'Demo',
10    setup(){
11      // let keyword = ref('hello') //使用Vue准备好的内置ref
12      //自定义一个myRef
13      function myRef(value,delay){
14        let timer
15        //通过customRef去实现自定义
16        return customRef((track,trigger)=>{
17          return{
18            get(){
19              track() //告诉Vue这个value值是需要被“追踪”的
20              return value
21            },
22            set(newValue){
23              clearTimeout(timer)
24              timer = setTimeout(()=>{
25                value = newValue
26                trigger() //告诉Vue去更新界面
27              },delay)
28            }
29          }
30        })
31      }
32      let keyword = myRef('hello',500) //使用程序员自定义的ref
33      return {
34        keyword
35      }
36    }
37  }
38 </script>
```

5.provide 与 inject



- 作用：实现**祖与后代组件间**通信
- 套路：父组件有一个 `provide` 选项来提供数据，后代组件有一个 `inject` 选项来开始使用这些数据
- 具体写法：
 1. 祖组件中：

```
1  setup(){
2      .....
3      let car = reactive({name:'奔驰',price:'40万'})
4      provide('car',car)
5      .....
6  }
```

2. 后代组件中：

```
1  setup(props,context){
2      .....
3      const car = inject('car')
4      return {car}
5      .....
6  }
```

6.响应式数据的判断

- `isRef`: 检查一个值是否为一个 `ref` 对象
- `isReactive`: 检查一个对象是否是由 `reactive` 创建的响应式代理
- `isReadonly`: 检查一个对象是否是由 `readonly` 创建的只读代理
- `isProxy`: 检查一个对象是否是由 `reactive` 或者 `readonly` 方法创建的代理

四、Composition API 的优势

1.Options API 存在的问题

使用传统OptionsAPI中，新增或者修改一个需求，就需要分别在data，methods，computed里修改。

2.Composition API 的优势

我们可以更加优雅的组织我们的代码，函数。让相关功能的代码更加有序的组

织在一起。

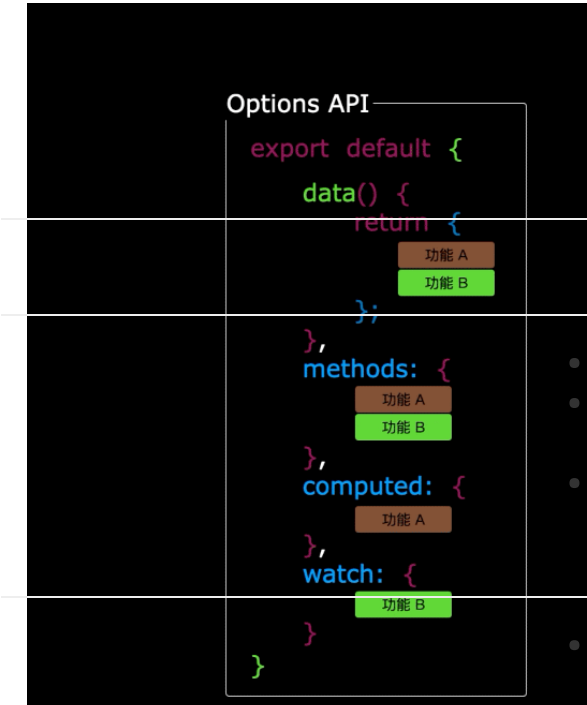
五、新的组件

1.Fragment

- 在Vue2中: 组件必须有一个根标签
- 在Vue3中: 组件可以没有根标签, 内部会将多个标签包含在一个Fragment虚拟元素中
- 好处: 减少标签层级, 减小内存占用

2.Teleport

- 什么是Teleport? —— Teleport 是一种能够将我们的**组件html结构**移动到指定位置的技术。



```

1 <teleport to="移
  动位置">
2   <div v-
    if="isShow"
    class="mask">
3     <div
        class="dialog">
4       <h3>
        我是一个弹窗</h3>
5
6       <button
        @click="isShow =
        false">关闭弹窗
        </button>
7     </div>
8   </div>
  </teleport>

```

3.Suspense

- 等待异步组件时渲染一些额外内容，让应用有更好的用户体验
- 使用步骤：

- 异步引入组件

```

1 import {defineAsyncComponent} from 'vue'
2 const Child = defineAsyncComponent(()=>import('./components/Child.vue'))

```

- 使用 `Suspense` 包裹组件，并配置好 `default` 与 `fallback`

```

1 <template>
2   <div class="app">
3     <h3>我是App组件</h3>
4     <Suspense>
5       <template v-slot:default>
6         <Child/>
7       </template>
8       <template v-slot:fallback>
9         <h3>加载中.....</h3>
10      </template>
11    </Suspense>
12  </div>
13 </template>

```

六、其他

1.全局API的转移

- Vue 2.x 有许多全局 API 和配置。
 - 例如：注册全局组件、注册全局指令等。

```
1 //注册全局组件
2 Vue.component('MyButton', {
3   data: () => ({
4     count: 0
5   }),
6   template: '<button @click="count++">Clicked {{ count }} times.</button>'
7 })
8
9 //注册全局指令
10 Vue.directive('focus', {
11   inserted: el => el.focus()
12 }
```

- Vue3.0中对这些API做出了调整：
 - 将全局的API，即： `Vue.xxx` 调整到应用实例（ `app` ）上

2.x 全局 API (<code>Vue</code>)	3.x 实例 API (<code>app</code>)
<code>Vue.config.xxxx</code>	<code>app.config.xxxx</code>
<code>Vue.config.productionTip</code>	移除
<code>Vue.component</code>	<code>app.component</code>
<code>Vue.directive</code>	<code>app.directive</code>
<code>Vue.mixin</code>	<code>app.mixin</code>
<code>Vue.use</code>	<code>app.use</code>
<code>Vue.prototype</code>	<code>app.config.globalProperties</code>

2.其他改变

- data选项应始终被声明为一个函数。
- 过度类名的更改：
 - Vue2.x写法

```

1  .v-enter,
2  .v-leave-to {
3    opacity: 0;
4  }
5  .v-leave,
6  .v-enter-to {
7    opacity: 1;
8  }

```

◦ Vue3.x写法

```

1  .v-enter-from,
2  .v-leave-to {
3    opacity: 0;
4  }
5
6  .v-leave-from,
7  .v-enter-to {
8    opacity: 1;
9  }

```

- **移除**keyCode作为 v-on 的修饰符，同时也不再支持 `config.keyCodes`

- **移除** `v-on.native` 修饰符

◦ 父组件中绑定事件

```

1  <my-component
2    v-on:close="handleComponentEvent"
3    v-on:click="handleNativeClickEvent"
4  />

```

◦ 子组件中声明自定义事件

```

1  <script>
2    export default {
3      emits: ['close']
4    }
5  </script>

```

- **移除**过滤器 (filter)

过滤器虽然这看起来很方便，但它需要一个自定义语法，打破大括号内表达式是“只是JavaScript”的假设，这不仅有学习成本，而且有实现成本！建议用方法调用或计算属性去替换过滤器。

-