

2. Vue3 新特性

课程源码：



vue3-demo.zip

136.00KB



Vue3新特性.zip

73.52KB



目录导航

Composition API 【重点掌握哦】

面对 Composition API 这一全新概念，我们完全用拥抱变化的心态去接受。

在此前，Vue 的基本语法给我们的感觉就是很规整，规整得有些写法显得略微有些臃肿，这种写法我们把它称之为选项式 API。这样的语法我们以前经常会使用到，比如我们设计一个 Select 组件，那么他的选项值我们通过 options 来确定，就像 React 中：

```
<Select
  placeholder="请选择"
  styles={{
    root: {
      width: 200
    }
  }}
  options={[
    { label: '男', value: 'man' },
    { label: '女', value: 'woman' }
  ]}
/>
```

这里的 options 我们就可以理解为选项式的，因为是通过 Object JSON 配置。

如果我们不想这样子，那么我们可以选择声明式的方式进行组件书写，在 Vue 其实我们通常就是这样做的，还是上面的例子，我们还能这样子：

```

<Select
  placeholder="请选择"
  styles={{
    root: {
      width: 200
    }
  }}
>
  <Select.Option value="nan">男</Select.Option>
  <Select.Option value="nv">女</Select.Option>
</Select>

```

那我们回到 Vue，在 Vue2 我们定义一个组件，基本模板应该是：

```

<script>
export default {
  // data() 返回的属性将会成为响应式的状态
  // 并且暴露在 `this` 上
  data() {
    return {
      count: 0
    }
  },

  // methods 是一些用来更改状态与触发更新的函数
  // 它们可以在模板中作为事件处理器绑定
  methods: {
    increment() {
      this.count++
    }
  },

  // 生命周期钩子会在组件生命周期的各个不同阶段被调用
  // 例如这个函数就会在组件挂载完成后被调用
  mounted() {
    console.log(`The initial count is ${this.count}.`)
  }
}
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>

```

```
</template>
```

如果我们选用 Vue3 的组合式方式呢？

```
<script setup>
import { ref, onMounted } from 'vue'

// 响应式状态
const count = ref(0)

// 用来修改状态、触发更新的函数
function increment() {
  count.value++
}

// 生命周期钩子
onMounted(() => {
  console.log(`The initial count is ${count.value}.`)
})
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

注意到以上最重要的区别了吗？

那就是：

组合形式的写法，必须要在 script 中指定 setup，这样 Vue 在编译转换等阶段都会采用一种全新的方式进行。其实你也完全可以在选型式子 API 的方式下处理组合 API 的内容，只需要这样：

```
export default {
  // `setup` 是一个专门用于组合式 API 的特殊钩子函数
  setup() {
    const state = reactive({ count: 0 })

    // 暴露 state 到模板
    return {
      state
    }
  }
}
```

但是这样太过繁琐，我们一般都还是选用 `<script setup>` 的形式。

什么是 Composition API?

选项式 API 与 Composition API 属于两种心智模型，**选项式**就是我们所说的配置式，它通过前期的内容约定，结构规范保证程序健壮性及可读性。**组合式**显然比选项式灵活，除此以外，组合式实现了 UI 复用与状态逻辑复用的分离。他们在底层编译和转换时区别很大，具体原理与区别我们会在后续 [Vue 源码课](#)进行讲解。

选项式 vs 组合式状态与事件

选项式声明的方式相信大家都非常熟悉了，就像这样

```
<!--
这个示例展示了如何通过 v-on 指令处理用户输入。
-->

<script>
export default {
  data() {
    return {
      message: 'Hello World!'
    }
  },
  methods: {
    reverseMessage() {
      this.message = this.message.split('').reverse().join('')
    },
    notify() {
      alert('navigation was prevented.')
    }
  }
}
</script>

<template>
  <!--
    注意我们不需要在模板中写 .value，
    因为在模板中 ref 会自动“解包”。
  -->
  <h1>{{ message }}</h1>

  <!--
```

绑定到一个方法/函数。

这个 @click 语法是 v-on:click 的简写。

-->

```
<button @click="reverseMessage">Reverse Message</button>
```

<!-- 也可以写成一个内联表达式语句 -->

```
<button @click="message += '!'">Append "!"</button>
```

<!--

Vue 也为一些像 e.preventDefault() 和 e.stopPropagation() 这样的常见任务提供了修饰符。

-->

```
<a href="https://vuejs.org" @click.prevent="notify">
```

A link with e.preventDefault()

```
</a>
```

```
</template>
```

```
<style>
```

```
button, a {
```

```
  display: block;
```

```
  margin-bottom: 1em;
```

```
}
```

```
</style>
```

如果换为组合式方式呢？

<!--

这个示例展示了如何通过 v-on 指令处理用户输入。

-->

```
<script setup>
```

```
import { ref } from 'vue'
```

```
const message = ref('Hello World!')
```

```
function reverseMessage() {
```

```
  // 通过其 .value 属性
```

```
  // 访问/修改一个 ref 的值。
```

```
  message.value = message.value.split('').reverse().join('')
```

```
}
```

```
function notify() {
```

```
  alert('navigation was prevented.')
```

```
}
```

```
</script>
```

```

<template>
  <!--
    注意我们不需要在模板中写 .value,
    因为在模板中 ref 会自动“解包”。
  -->
  <h1>{{ message }}</h1>

  <!--
    绑定到一个方法/函数。
    这个 @click 语法是 v-on:click 的简写。
  -->
  <button @click="reverseMessage">Reverse Message</button>

  <!-- 也可以写成一个内联表达式语句 -->
  <button @click="message += '!'>Append "!"</button>

  <!--
    Vue 也为一些像 e.preventDefault() 和 e.stopPropagation()
    这样的常见任务提供了修饰符。
  -->
  <a href="https://vuejs.org" @click.prevent="notify">
    A link with e.preventDefault()
  </a>
</template>

<style>
button, a {
  display: block;
  margin-bottom: 1em;
}
</style>

```

常用 Composition API

先问几个问题：

1. 说说你对 setup 的理解
 - a. 组合式 API 的入口
2. ref 和 reactive 有什么区别？
 - a. ref 内部使用了 reactive

```
export const toReactive = <T extends unknown>(value: T): T =>
  isObject(value) ? reactive(value) : value
```

```
ref(obj) === reactive({ value: obj })
```

3. ref 和 shallowRef 的区别，以及 reactive 和 shallowReactive 的区别？

- a. shallow 表示浅层，这里均是指的响应值作用在第一层，即 `.value`，不过我们可以使用 `triggerRef(xxx)` 来在深层内容变更后，手动触发更新，需要注意的是 shallowReactive 没有对应方法

4. watchEffect 与 watch 的区别

- a. 懒执行副作用；
- b. 更加明确是应该由哪个状态触发侦听器重新执行；
- c. 可以访问所侦听状态的前一个值和当前值。

需要我们完全掌握的 API：

setup

`setup()` 钩子是在组件中使用组合式 API 的入口，通常只在以下情况下使用：

1. 需要在非单文件组件中使用组合式 API 时。
2. 需要在基于选项式 API 的组件中集成基于组合式 API 的代码时。

```
import { h, ref } from 'vue'

export default {
  setup(props, { expose }) {
    const count = ref(0)
    const increment = () => ++count.value

    // 透传 Attributes (非响应式的对象，等价于 $attrs)
    console.log(context.attrs)

    // 插槽 (非响应式的对象，等价于 $slots)
    console.log(context.slots)

    // 触发事件 (函数，等价于 $emit)
    console.log(context.emit)
```

```

    // 暴露公共属性（函数）
    console.log(context.expose)

    expose({
      increment
    })

    return () => h('div', count.value)
  }
}

```

从这个例子我们可以很清晰看到，vue 的 setup 函数可以就收两个参数，分别为 props、context。context 中包含属性、插槽、触发事件等内容。

在组件内部可以通过 compose 方法，将内部方法暴露到模板引用上以供父组件使用。

ref

注意，这里有两个概念需要区分，一个是 ref 响应式方法，另一个是模板引用

```

<script setup>
import { ref, onMounted } from 'vue'

// 声明一个 ref 来存放该元素的引用
// 必须和模板里的 ref 同名
const input = ref(null)

onMounted(() => {
  input.value.focus()
})
</script>

<template>
  <input ref="input" />
</template>

```

这个例子我们可以看到，`ref` 函数 用于定义一个响应式对象，这个对象用于存储 input ref 应用，即为 dom 对象。

其实 ref 函数最终也跟 reactive 有这紧密联系，我们不妨看看这段源码：

<https://github.com/vuejs/core/blob/507f3e7a16c98398a661c150ce89d36b1441f6cc/packages/reactivity/src/ref.ts#L157>

github.com

reactive

用法同 ref

```
const obj = reactive({ count: 0 })
obj.count++
```

computed

```
const count = ref(1)
const plusOne = computed(() => count.value + 1)

console.log(plusOne.value) // 2

plusOne.value++ // 错误
```

或者通过对象 get、set 指定的方式

```
const count = ref(1)
const plusOne = computed({
  get: () => count.value + 1,
  set: (val) => {
    count.value = val - 1
  }
})

plusOne.value = 1
console.log(count.value) // 0
```

watchEffect、watchPostEffect、watchSyncEffect

后两者是前者的语法糖，就是将第二个参数中的 flush，指定为对应值，分别为：`flush?: 'pre'`
`| 'post' | 'sync' // 默认: 'pre'`

简单使用

```
const count = ref(0)

watchEffect(() => console.log(count.value))
// -> 输出 0

count.value++
// -> 输出 1
```

具有清除与停止侦听的功能

```
const stop = watchEffect(async (onCleanup) => {
  const { response, cancel } = doAsyncWork(id.value)
  // `cancel` 会在 `id` 更改时调用
  // 以便取消之前
  // 未完成的请求
  onCleanup(cancel)
  data.value = await response
})

// 什么时候需要停止的话，那就
stop()
```

watch

侦听一个 getter 函数：

```
const state = reactive({ count: 0 })
watch(
  () => state.count,
  (count, prevCount) => {
    /* ... */
  }
)
```

侦听一个 ref：

```
const count = ref(0)
watch(count, (count, prevCount) => {
  /* ... */
})
```

当侦听多个来源时，回调函数接受两个数组，分别对应来源数组中的新值和旧值：

```
watch([fooRef, barRef], ([foo, bar], [prevFoo, prevBar]) => {
  /* ... */
})
```

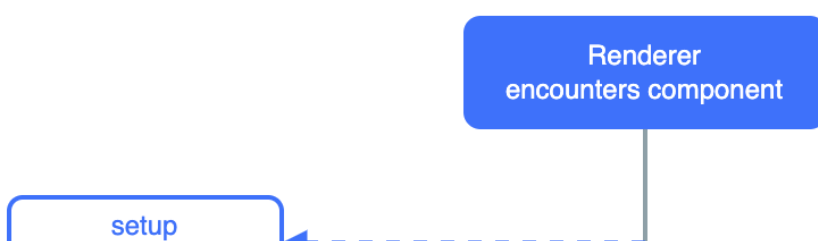
当使用 `getter` 函数作为源时，回调只在此函数的返回值变化时才会触发。如果你想让回调在深层级变更时也能触发，你需要使用 `{ deep: true }` 强制侦听器进入深层级模式。在深层级模式时，如果回调函数由于深层级的变更而被触发，那么新值和旧值将是同一个对象。

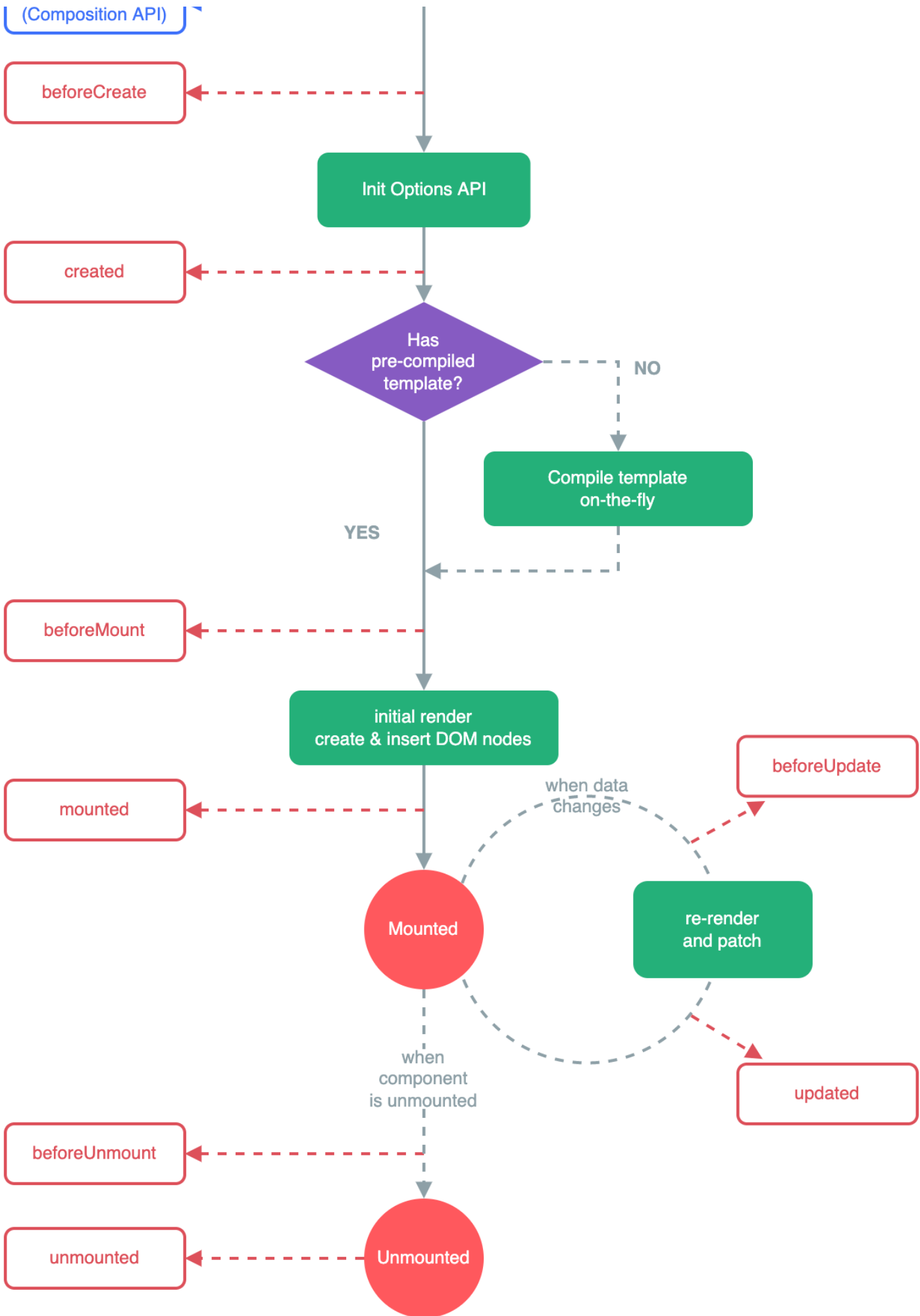
```
const state = reactive({ count: 0 })
watch(
  () => state,
  (newValue, oldValue) => {
    // newValue === oldValue
  },
  { deep: true }
)
```

当直接侦听一个响应式对象时，侦听器会自动启用深层模式：

```
const state = reactive({ count: 0 })
watch(state, () => {
  /* 深层级变更状态所触发的回调 */
})
```

生命周期






vue2 和 vue3 关于生命周期的对比

Vue2	Vue3
beforeCreate	setup ，在 beforeCreate 和 created 前，因此一般在组合式 api 中使用它做一些前置处理。
created	
beforeMount	onBeforeMount
mounted	onMounted
beforeUpdate	onBeforeUpdate
updated	onUpdated
beforeDestroy	onBeforeUnmount
destroyed	onUnmounted

异步组件

什么是异步组件

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。换言之，我们的组件可能不再是同步导入或者组件需要等待 `Promise` `resolve` 完成后才被渲染。这样的组件我们称为**异步组件**

 如果你是 React 开发者，相比对这个概念并不陌生，异步组件的导入是类似的，组件通过 `lazy` 包装实现。

```
const OtherComponent = React.lazy(() => import('./OtherComponent'))
function MyComponent() {
  return (
    <div>
      {' '}
      <OtherComponent />{' '}
    </div>
  )
}
```

Vue 提供了 `defineAsyncComponent` 方法来实现此功能：

```
import { defineAsyncComponent } from 'vue'
const AsyncComp = defineAsyncComponent(() => {
  return new Promise((resolve, reject) => {
    // ...从服务器获取组件
    resolve(/* 获取到的组件 */)
  })
}) // ... 像使用其他一般组件一样使用 `AsyncComp`
```

如你所见，`defineAsyncComponent` 方法接收一个返回 Promise 的加载函数。这个 Promise 的 `resolve` 回调方法应该在从服务器获得组件定义时调用。你也可以调用 `reject(reason)` 表明加载失败。

[ES 模块动态导入](#) 也会返回一个 Promise，所以多数情况下我们会将它和 `defineAsyncComponent` 搭配使用。类似 Vite 和 Webpack 这样的构建工具也支持此语法 (并且会将它们作为打包时的代码分割点)，因此我们也可以用它来导入 Vue 单文件组件：

```
import { defineAsyncComponent } from 'vue'

const AsyncComp = defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
)
```

最后得到的 `AsyncComp` 是一个外层包装过的组件，仅在页面需要它渲染时才会调用加载内部实际组件的函数。它会接收到的 props 和插槽传给内部组件，所以你可以使用这个异步的包装组件无缝地替换原始组件，同时实现延迟加载。

与普通组件一样，异步组件可以使用 `app.component()` [全局注册](#)：

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

const AdminPage = defineAsyncComponent(() =>
  import('./components/AdminPageComponent.vue')
)
</script>

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

通常会与 Suspense 配合

`<Suspense>` 是一个内置组件，用来在组件树中协调对异步依赖的处理。它让我们可以在组件树上层等待下层的多个嵌套异步依赖项解析完成，并可以在等待时渲染一个加载状态。

不过暂时最好不要在生产环境大量使用这一特性，因为该特性目前还不是稳定版本，后续可能会有变更。

检测一下异步组件带来的价值【学会分析】

安装 `rollup-plugin-visualizer` 进行打包产物分析

对比组件通过异步导入和不通过异步导入，构建产物的区别

自定义指令

我们都知道指令是为了增强组件的，我们常见的指令有：`v-if`、`v-show`、`v-model`、`v-bind:value`、`v-on:click` 等。

自定义指令其实非常简单，我们需要始终关注以下几个问题：

1. 指令的钩子函数，有点类似生命周期函数钩子
2. 指令钩子函数中的参数
3. 指令的逻辑处理

一个基本的 v-focus 指令

例如，我们想要 `input` 组件在初始化渲染时，就聚焦，那么我们可以这样：

```
<script setup>
// 在模板中启用 v-focus
const vFocus = {
  mounted: (el) => el.focus()
}
</script>

<template>
  <input v-focus />
</template>
```

```
</template>
```

如果你是用的选项式 API，那就这样：这里不细讲了

```
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) {}
}
```

参数详解

指令的钩子会传递以下几种参数：

- `el`：指令绑定到的元素。这可以用于直接操作 DOM。
- `binding`：一个对象，包含以下属性。
 - `value`：传递给指令的值。例如在 `v-my-directive="1 + 1"` 中，值是 `2`。
 - `oldValue`：之前的值，仅在 `beforeUpdate` 和 `updated` 中可用。无论值是否更改，它都可用。
 - `arg`：传递给指令的参数 (如果有的话)。例如在 `v-my-directive:foo` 中，参数是 `"foo"`。
 - `modifiers`：一个包含修饰符的对象 (如果有的话)。例如在 `v-my-directive.foo.bar` 中，修饰符对象是 `{ foo: true, bar: true }`。
 - `instance`：使用该指令的组件实例。
 - `dir`：指令的定义对象。
- `vnode`：代表绑定元素的底层 VNode。
- `prevNode`：之前的渲染中代表指令所绑定元素的 VNode。仅在 `beforeUpdate` 和 `updated` 钩子中可用。

举例来说，像下面这样使用指令：

```
<div v-example:foo.bar="baz">
```

`binding` 参数会是一个这样的对象：

```
{
  arg: 'foo',
  modifiers: { bar: true },
  value: /* `baz` 的值 */,
  oldValue: /* 上一次更新时 `baz` 的值 */
}
```

和内置指令类似，自定义指令的参数也可以是动态的。举例来说：

```
<div v-example:[arg]="value"></div>
```

这里指令的参数会根据组件的 `arg` 数据属性响应式地更新。

最后聊聊 v-model

vuejs/core



Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web.

449

Contributors

681

Issues

1k

Discussions

43k

Stars

8k

Forks




<https://github.com/vuejs/core/blob/507f3e7a16c98398a661c150ce89d36b1441f6cc/packages/compiler-core/src/transforms/vModel.ts> at 507f3e7a16c98398a661c150ce89d36b1441f6cc · vuejs/core

Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web.

在实现一个输入组件双向绑定需求时，我们先抛开这个概念，如果我们使用原始 api 方式实现 input 的值状态记录，并且输入时更新这一状态，当外界有操作更新了这一状态的数据，input 也会更新，我们怎么做？

那自然是需要用到两个指令：`v-bind:value` 和 `v-on:input`

 `compiler-core` 模块是 Vue3 编译器的核心实现，负责将模板编译为渲染函数，它包含了一些基础的编译器功能，如 AST 的生成、指令和表达式的处理、优化和代码生成等。可以运行在各种 JavaScript 环境下的，不仅限于浏览器环境，因此它可以用于开发基于 Vue3 的跨平台应用程序，如桌面应用程序、移动应用程序

`compiler-dom` 模块是 Vue3 编译器在浏览器环境下的实现，它扩展了 `compiler-core` 模块的功能，以适应浏览器环境下的特殊需求，例如对 DOM 元素的属性、事件等进行编译。生成的代码是直接在浏览器中执行的，因此它会生成特定于浏览器环境的代码

Teleport【多用用，挺简单】

该特性允许你将组件内的某个子组件挂载到任意 HTML 节点上，这个特性像极了 React 中的 `createPortal`。

这个特性到底有什么作用呢？我们一般用在哪？思考这个问题之前，我们不妨一起来看看这个问题：页面中有一个按钮，当按钮点击时，会弹出 modal。看到这个需求，我们很容易就能想到实现方案：

不使用 Teleport

```
// App.vue
<!--
可定制插槽和 CSS 过渡效果的模态框组件。
-->

<script setup>
import Modal from './Modal.vue'
import { ref } from 'vue'
```

```

const showModal = ref(false)
</script>

<template>
  <button id="show-modal" @click="showModal = true">Show Modal</button>
  <modal :show="showModal" @close="showModal = false">
    <template #header>
      <h3>custom header</h3>
    </template>
  </modal>
</template>

```

然后在同级创建 Modal 组件，相关代码如下：

```

<script setup>
const props = defineProps({
  show: Boolean
})
</script>

<template>
  <Transition name="modal">
    <div v-if="show" class="modal-mask">
      <div class="modal-container">
        <div class="modal-header">
          <slot name="header">default header</slot>
        </div>

        <div class="modal-body">
          <slot name="body">default body</slot>
        </div>

        <div class="modal-footer">
          <slot name="footer">
            default footer
            <button
              class="modal-default-button"
              @click="$emit('close')"
            >OK</button>
          </slot>
        </div>
      </div>
    </div>
  </Transition>
</template>

```

```

<style>
.modal-mask {
  position: fixed;
  z-index: 9998;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(0, 0, 0, 0.5);
  display: flex;
  transition: opacity 0.3s ease;
}

.modal-container {
  width: 300px;
  margin: auto;
  padding: 20px 30px;
  background-color: #fff;
  border-radius: 2px;
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.33);
  transition: all 0.3s ease;
}

.modal-header h3 {
  margin-top: 0;
  color: #42b983;
}

.modal-body {
  margin: 20px 0;
}

.modal-default-button {
  float: right;
}

/*
 * 对于 transition="modal" 的元素来说
 * 当通过 Vue.js 切换它们的可见性时
 * 以下样式会被自动应用。
 *
 * 你可以简单地通过编辑这些样式
 * 来体验该模态框的过渡效果。
 */

.modal-enter-from {

```

```

    opacity: 0;
  }

  .modal-leave-to {
    opacity: 0;
  }

  .modal-enter-from .modal-container,
  .modal-leave-to .modal-container {
    -webkit-transform: scale(1.1);
    transform: scale(1.1);
  }
</style>

```

使用 Teleport

`Modal.vue` 内容不变，调整 `App.vue` 的内容，如下：

```

<!-- App.vue -->
<!--
可定制插槽和 CSS 过渡效果的模态框组件。
-->

<script setup>
import Modal from './Modal.vue'
import { ref } from 'vue'

const showModal = ref(false)
</script>

<template>
  <button id="show-modal" @click="showModal = true">Show Modal</button>

  <Teleport to="body">
    <!-- 使用这个 modal 组件，传入 prop -->
    <modal :show="showModal" @close="showModal = false">
      <template #header>
        <h3>custom header</h3>
      </template>
    </modal>
  </Teleport>
</template>

```

Teleport 原理简单介绍

<https://github.com/vuejs/core/blob/507f3e7a16c98398a661c150ce89d36b1441f6cc/packages/runtime-core/src/componen...>
github.com

- Teleport 组件渲染的时候会调用 `patch` 方法，`patch` 方法会判断如果 `shapeFlag` 是一个 Teleport 组件,则会调用它的 `process` 方法。`process` 方法包含了Teleport组件创建和组件更新的逻辑。
- Teleport 组件创建
 - 首先会在在主视图里插入注释节点或者空白文本节点
 - 接着获取目标元素节点
 - 最后调用 `mount` 方法创建子节点往目标元素插入 Teleport 组件的子节点
- Teleport 组件更新首先会更新子节点，处理 `disabled` 属性变化的情况，处理 `to` 属性变化的情况。
- 最后 Teleport 组件挂载会调用 `unmount` 方法，会判断如果 `shapeFlag` 是一个 Teleport 组件，则会执行它的 `remove` 方法。
- `remove` 方法 会调用`hostRemove`方法移除文本节点，然后遍历子节点循环调用 `unmount` 方法挂载子节点。

用武之地

作用与 react 的 `createPortal` 类似

- 弹出层
- Popover 等
- tooltip

自定义 Hooks 【比较复杂，但是理解了用起来是真舒服】

自定义 Hook 的价值

我们通过自定义 Hook，可以将组件的状态与 UI 实现分离，虽然这个 api 和早期的 `mixin` 非常像，但是他的设计思想实在先进太多。

自定义 Hook 示例

假设我们需要封装一个计数器，该计数器用于实现数字的增加或者减少，并且我们可以指定数字可最大和最小值，如果我们使用 `vue3 composition` 封装，会是怎样的呢？

我们先设想一下使用方法：

```
<template>
  <div>
    <p>{{ current }} [max: 10; min: 1;]</p>
    <div class="contain">
      <button @click="inc()">
        Inc()
      </button>
      <button @click="dec()" style="margin-left: 8px">
        Dec()
      </button>
      <button @click="set(3)" style="margin-left: 8px">
        Set(3)
      </button>
      <button @click="reset()" style="margin-left: 8px">
        Reset()
      </button>
    </div>
  </div>
</template>

<script lang="ts" setup>
  import { useCounter } from './useCounter'
  const [current, { inc, dec, set, reset }] = useCounter(20, { min: 1, max: 10
})
</script>
```

看到了使用方法，我们可以尝试定义这个 hook 函数，这里我们新建一个文件，用于编写 `useCounter` 相关代码。

```
import { Ref, readonly, ref } from 'vue'

// 判断是否为数字
const isNumber = (value: unknown): value is number => typeof value === 'number'

export interface UseCounterOptions {
  /**
   * Min count
   */
  min?: number

  /**
   * Max count
   */
}
```

```

    */
    max?: number
}

export interface UseCounterActions {
  /**
   * Increment, default delta is 1
   * @param delta number
   * @returns void
   */
  inc: (delta?: number) => void

  /**
   * Decrement, default delta is 1
   * @param delta number
   * @returns void
   */
  dec: (delta?: number) => void

  /**
   * Set current value
   * @param value number | ((c: number) => number)
   * @returns void
   */
  set: (value: number | ((c: number) => number)) => void

  /**
   * Reset current value to initial value
   * @returns void
   */
  reset: () => void
}

export type ValueParam = number | ((c: number) => number)

function getTargetValue(val: number, options: UseCounterOptions = {}) {
  const { min, max } = options
  let target = val
  if (isNumber(max)) {
    target = Math.min(max, target)
  }
  if (isNumber(min)) {
    target = Math.max(min, target)
  }
  return target
}

```



```
function useCounter(  
  initialValue = 0,  
  options: UseCounterOptions = {},  
) : [Ref<number>, UseCounterActions] {  
  const { min, max } = options  
  
  const current = ref(  
    getTargetValue(initialValue, {  
      min,  
      max,  
    }),  
  )  
  
  const setValue = (value: ValueParam) => {  
    const target = isNumber(value) ? value : value(current.value)  
    current.value = getTargetValue(target, {  
      max,  
      min,  
    })  
    return current.value  
  }  
  
  const inc = (delta = 1) => {  
    setValue(c => c + delta)  
  }  
  
  const dec = (delta = 1) => {  
    setValue(c => c - delta)  
  }  
  
  const set = (value: ValueParam) => {  
    setValue(value)  
  }  
  
  const reset = () => {  
    setValue(initialValue)  
  }  
  
  return [  
    readonly(current),  
    {  
      inc,  
      dec,  
      set,  
      reset,  
    },  
  ]  
}
```

```
}
```

```
export default useCounter
```

拓展

一定一定看看这个：

<https://github.com/InhiblabCore/vue-hooks-plus>

github.com