# 计 算属性

当模板中需要多次相同逻辑计算时，我们没必要在模板中将代码重复几遍，根据官方文档推荐，我们可 以使用计算属性来描述依赖响应式状态的复杂逻辑，例子：

<script setup>

import { reactive, computed } from 'vue'

const author = reactive({ name: 'John Doe',

]

})

// a computed ref

const publishedBooksMessage = computed(() => { return author.books.length > 0 ? 'Yes' : 'No'

})

</script>

<template>

<p>Has published books:</p>

<span>{{ publishedBooksMessage }}</span>

</template>

|  |  |  |
| --- | --- | --- |
| books:  'Vue | [  2 | - Advanced Guide', |
| 'Vue | 3 | - Basic Guide', |
| 'Vue | 4 | - The Mystery' |

author.books.length > 0 ? 'Yes' : 'No' 是一个三目运算符：[…] ? a : b ,[]里面是二元运算符，当结果是true时，返回a；当结果时false时，返回b。

reactive()创建响应式对象author，定义一个计算属性publishedBooksMessage，computed()方法用来 接收一个getter函数，返回值为一个计算属性ref。

vue使用计算属性的一个好处就是vue的计算属性会自动追踪响应式依赖，如上例子，它会检测到publishedBooksMessage依赖于author.books，当author.books发生改变时任何绑定于

publishedBooksMessage都会同步更新。

而计算属性相比于方法的不同之处在于计算属性值会基于响应式依赖被缓存，只要author.books不改变，无论访问多少次publishedBooksMessage都会直接返回先前的计算结果，不用重复执行getter函 数。

计算属性默认为只读，当想用可写属性时，可以手动提供getter和setter来创建：

<script setup>

import { ref, computed } from 'vue'

const firstName = ref('John') const lastName = ref('Doe')

const fullName = computed({

// getter get() {

return firstName.value + ' ' + lastName.value

},

// setter set(newValue) {

// 注意：我们这里使用的是解构赋值语法

[firstName.value, lastName.value] = newValue.split(' ')

}

})

</script>

**条**

## v-if

**件渲染**

用于条件性地渲染，只有当表达式为真值时才会被渲染

v-if

<h1 v-if="awesome">Vue is awesome!</h1>

## v-else

一个

v-else

## v-else-if

v-else-if

元素必须跟在一个 或者

元素后面，否则它将不会被识别。

v-else-if

和

v-else

## v-show

类似，一个使用

的元素必须紧跟在一个

或一个

元素后面。

用法和v-if 相似，不同的是v-show 会在DOM渲染中保留该元素，并且不能用于<template>标签

v-if

v-else-if

总的来说， v-if 有更高的切换开销，而 有更高的初始渲染开销。因此，如果需要频繁切换，

v-show

v-show

v-if

则使用

较好；如果在运行时绑定条件很少改变，则

会更合适。

# 列 表渲染

## v-for

例子；

const items = ref([{ message: 'Foo' }, { message: 'Bar' }])

<li v-for="item in items">

{{ item.message }}

</li>

这里是基于数组来渲染一个列表， v-for 指令的值需要使用 形式的特殊语法，其中

item in items

是源数据的数组，而 是迭代项的别名

items

item

v-for 可以和v-if 组合使用，但不是在同一个标签里面使用，否则会因为v-if 优先级更高无法访问v- for 作用域定义的变量别名

# 事 件处理

## v-on

使用v-on 监听事件（简写@）用法： v-on:click="methodName" 或

@click="handler"

## 内联事件处理器：

<script setup>

import { ref } from 'vue'

const counter = ref(0)

</script>

<template>

<button @click="counter++">Add 1</button>

<p>The button above has been clicked {{ counter }} times.</p>

</template>

counter++是内联语句，也就是说内联事件处理器用v-on 监听事件然后“ ”里面用内联语句组成内联事件处理器

在内联处理器中可以调用方法，这允许我们向方法传入自定义参数以代替原生事件：

<script setup> function say(message) {

alert(message)

}

</script>

<template>

<button @click="say('hello')">Say hello</button>

<button @click="say('bye')">Say bye</button>

</template>

可以向该处理器方法传入一个特殊的件参数：

$event

变量，或者使用内联箭头函数在内联事件处理器中访问事

<script setup>

function warn(message, event) {

// 这里可以访问原生事件

if (event) { event.preventDefault()

}

alert(message)

}

</script>

<template>

<!-- 使用特殊的 $event 变量 -->

<button @click="warn('Form cannot be submitted yet.', $event)"> Submit

</button>

<!-- 使用内联箭头函数 -->

<button @click="(event) => warn('Form cannot be submitted yet.', event)"> Submit

</button>

</template>

## 方法事件处理器：

<script setup>

import { ref } from 'vue'

const name = ref('Vue.js')

function greet(event) { alert(`Hello ${name.value}!`)

// `event` is the native DOM event if (event) {

alert(event.target.tagName)

}

}

</script>

<template>

<button @click="greet">Greet</button>

</template>

v-on 通过接受一个方法名或者对方法的调用，称为方法事件处理器，使用方法处理器的好处就是当事件逻辑越来越复杂的时候，内联事件处理器就显得过于笨重，不够优雅，此时方法事件处理器就可以更灵 活的处理事件。

方法事件处理器会自动接收原生 DOM 事件并触发执行。在上面的例子中，我们能够通过被触发事件的访问到该 DOM 元素。

event.target.tagName

## 事件修饰符：

Vue 为 提供了**事件修饰符**，以便不用在方法中调用例如event.preventDefault()（阻止事件的默认

v-on



.

行为）之类的去处理DOM事件细节，让我们更专注于数据逻辑。修饰符是用以下这些：

表示的指令后缀，包含

.stop

.prevent

.self

.capture

.once

.passive

<!-- 单击事件将停止传递 -->

<a @click.stop="doThis"></a>

<!-- 提交事件将不再重新加载页面 -->

<form @submit.prevent="onSubmit"></form>

<!-- 修饰语可以使用链式书写 -->

<a @click.stop.prevent="doThat"></a>

<!-- 也可以只有修饰符 -->

<form @submit.prevent></form>

<!-- 仅当 event.target 是元素本身时才会触发事件处理器 -->

<!-- 例如：事件处理器不来自子元素 -->

<div @click.self="doThat">...</div>

.capture 、.once 和

.passive

修饰符与原生

事件相对应：

addEventListener

<!-- 添加事件监听器时，使用 `capture` 捕获模式 -->

<!-- 例如：指向内部元素的事件，在被内部元素处理前，先被外部处理 -->

<div @click.capture="doThis">...</div>

<!-- 点击事件最多被触发一次 -->

<a @click.once="doThis"></a>

<!-- 滚动事件的默认行为 (scrolling) 将立即发生而非等待 `onScroll` 完成 -->

<!-- 以防其中包含 `event.preventDefault()` -->

<div @scroll.passive="onScroll">...</div>

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

.passive

## 按键修饰符

需要监听键盘事件时，可以通过@keyup 来实现：

<!-- 仅在 `key` 为 `Enter` 时调用 `submit` -->

<input @keyup.enter="submit" />

|  |  |
| --- | --- |
| **别名** | **实际键值** |
| .delete | delete（删除）/BackSpace（退格） |
| .tab | Tab |
| .enter | Enter（回车） |
| .esc | Esc（退出） |
| .space | Space（空格键） |
| .left | Left（左箭头） |
| .up | Up（上箭头） |
| .right | Right（右箭头） |
| .down | Down（下箭头） |
| .ctrl | Ctrl |
| .alt | Alt |
| .shift | Shift |
| .meta | (window系统下是window键，mac下是command键) |

vue支持组合写法：

<!-- Alt + Enter -->

<input @keyup.alt.enter="clear" />

<!-- Ctrl + 点击 -->

<div @click.ctrl="doSomething">Do something</div>

.exact **修饰符**

修饰符允许控制触发一个事件所需的确定组合的系统按键修饰符。

.exact

<!-- 当按下 Ctrl 时，即使同时按下 Alt 或 Shift 也会触发 -->

<button @click.ctrl="onClick">A</button>

<!-- 仅当按下 Ctrl 且未按任何其他键时才会触发 -->

<button @click.ctrl.exact="onCtrlClick">A</button>

<!-- 仅当没有按下任何系统按键时触发 -->

<button @click.exact="onClick">A</button>

# 表 单输入

v-modle 将表单输入框的内容同步给js中相应的变量（之前提到的双向绑定就是通过这个原理）

v-modle 支持各种不同类型的input、<textare>、<select>元素例如：

<p>Message is: {{ message }}</p>

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

这是一个基础的v-modle 文本语法，<textare>、<select>用法类似，但注意在 中是不支

<textarea>

持插值表达式的。请使用 来替代：

v-model

<!-- 错误 -->

<textarea>{{ text }}</textarea>

<!-- 正确 -->

<textarea v-model="text"></textarea>

**对于单选按钮、复选框和选择器选项，** v**-**modle **通常绑定的值是静态的字符串**，因此可以通过使用



v-

v-bind

来实现将该值绑定到当前组件实例上的动态数据。此外，使用字符串的数据类型：

bind

v**-**model

还可以将选项值绑定为非

<input type="radio" v-model="pick" :value="first" />

<input type="radio" v-model="pick" :value="second" />

**vue复选框有两个特有属性：** true**-**value **和**

false**-**value

## ，仅支持和

**配套使用**：

<input

type="checkbox" v-model="toggle" true-value="yes" false-value="no">

这里toggle的值被选中时设为“yes”，取消选择时是“no”。这两个属性搭配上 v-bind 可以实现绑定动态值

## 一些修饰符

（拓展）：

文本类型的

<input>

<input type="checkbox">

事件；

change

会绑定

<select>

和 <textarea> 元素会绑定和

property 并侦听

<input type="radio">

value

change

property 并侦听会绑定

事件。

value

checked

lazy

事件；

property 并侦听

input

**.lazy**：默认情况下， v-model 会在每次事件后更新数据：

input

change

事件后更新数据。可以添加

修饰符来改为在每次

<!-- 在 "change" 事件后同步更新而不是 "input" -->

<input v-model.lazy="msg" />

**.number**：让用户的输入自动转换为数字（当输入框type=“number”时自动生效）

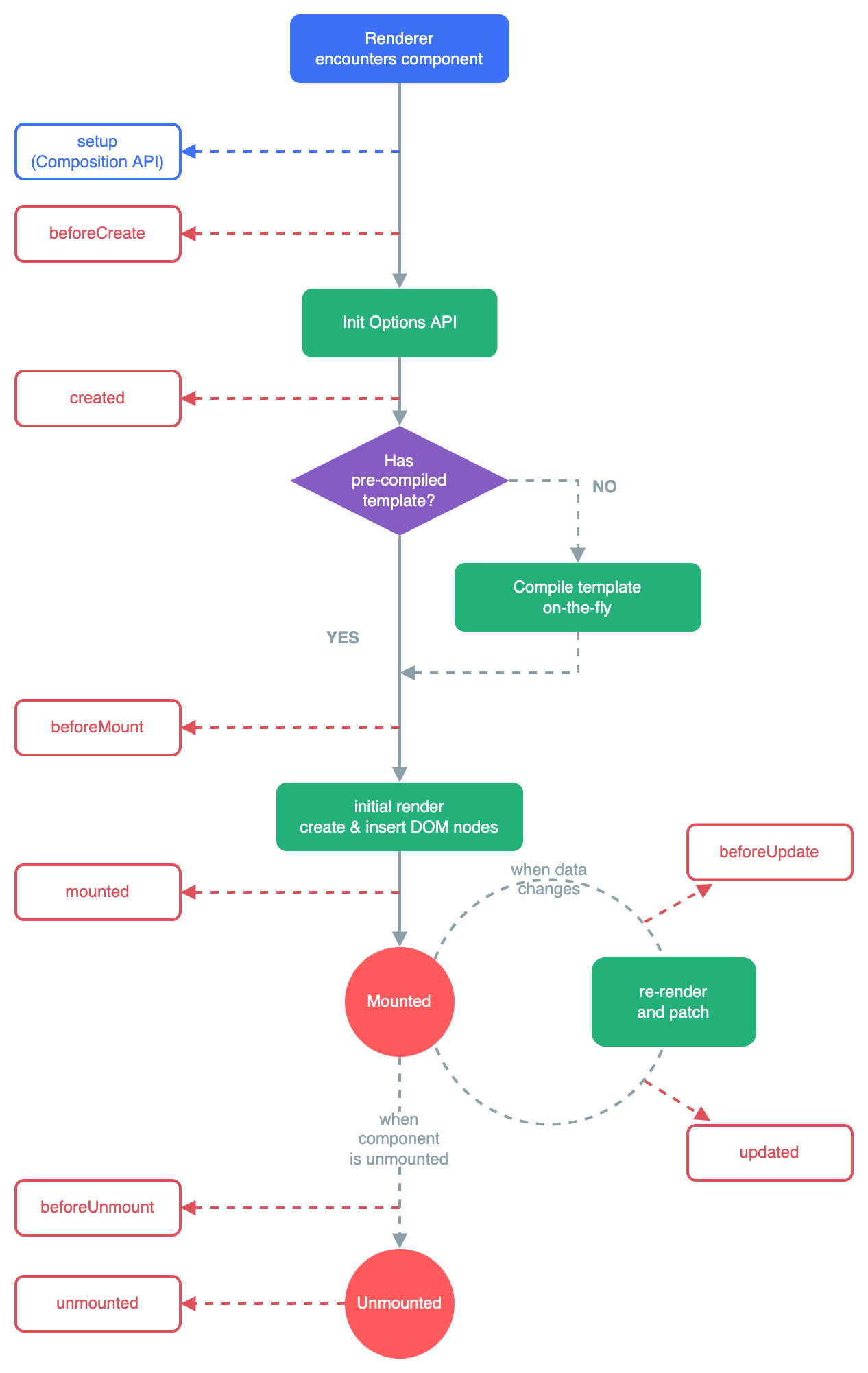
<input v-model.number="age" />

**.trim**:默认自动除去用户输入内容中两端的空格

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

# 生 命周期

以下是文档提供的生命周期的图表（详细可以参考vue3.js的[API](https://cn.vuejs.org/api/)部分）：



# 侦 听器

基本示例：

<script setup>

import { ref, watch } from 'vue'

const question = ref('')

const answer = ref('Questions usually contain a question mark. ;-)')

// 可以直接侦听一个 ref

watch(question, async (newQuestion, oldQuestion) => { if (newQuestion.indexOf('?') > -1) {

answer.value = 'Thinking...' try {

const res = await fetch('https://yesno.wtf/api') answer.value = (await res.json()).answer

} catch (error) {

answer.value = 'Error! Could not reach the API. ' + error

}

}

})

</script>

<template>

<p>

Ask a yes/no question:

<input v-model="question" />

</p>

<p>{{ answer }}</p>

</template>

该示例使用watch函数 在每次响应式状态发生改变时触发回调函数。 watch 的第一个参数可以是不同形式的“数据源”：**它可以是一个 ref (包括计算属性)、一个响应式对象、一个 getter 函数、或多个数据源组成的数组**

watch **函数是懒执行，只有当数据源发生变化时才执行回调函数**，当想要创建侦听器时立马执行一遍回调函数，可以使用watchEffect() ， watchEffect() **会立即执行一遍回调函数，如果这时函数产生了副作用，Vue 会自动追踪副作用的依赖关系，自动分析出响应源**。

这是没使用watchEffect() 函数：

const url = ref('https://...') const data = ref(null)

async function fetchData() {

const response = await fetch(url.value) data.value = await response.json()

}

// 立即获取

fetchData()

// ...再侦听 url 变化

watch(url, fetchData)

这是使用watchEffect() ：

watchEffect(async () => {

const response = await fetch(url.value) data.value = await response.json()

})

可以很明显的看出使用watchEffect() 要简洁不少。

ps：想要使用深层监听，可以设置deep：true

watch(

() => state.someObject, (newValue, oldValue) => {

// 注意：`newValue` 此处和 `oldValue` 是相等的

// \*除非\* state.someObject 被整个替换了

},

{ deep: true }

)

**总结一下** watch() **和** watchEffect() **区别**：

只追踪明确侦听的数据源。它不会追踪任何在回调中访问到的东西。另外，仅在数据源确 实改变时才会触发回调。 watch 会避免在发生副作用时追踪依赖，因此，我们能更加精确地控制回调函数的触发时机。

watch

watchEffect ，则会在副作用发生期间追踪依赖。它会在同步执行过程中，自动追踪所有能访问

到的响应式属性。这更方便，而且代码往往更简洁，但有时其响应性依赖关系会不那么明确。

## 回调的触发时机

设置flush: 'post' ，在侦听器回调中访问更新之后的DOM，因为默认情况下创建的侦听器回调都会在组件更新之前被调用

watch(source, callback, { flush: 'post'

})

watchEffect(callback, { flush: 'post'

})

// 后置刷新的 watchEffect() 有个更方便的别名 watchPostEffect() import { watchPostEffect } from 'vue'

watchPostEffect(() => {

/\* 在 Vue 更新后执行 \*/

})

## 停止侦听器

在 或

setup()

中用同步语句创建的侦听器，会自动绑定到宿主组件实例上，并且会

<script setup>

在宿主组件卸载时自动停止。

一个关键点是，侦听器必须用**同步**语句创建：如果用异步回调创建一个侦听器，那么它不会绑定到当前 组件上，你必须手动停止它，以防内存泄漏。如下方这个例子：

<script setup>

import { watchEffect } from 'vue'

// 它会自动停止

watchEffect(() => {})

// ...这个则不会！ setTimeout(() => {

watchEffect(() => {})

}, 100)

</script>

要手动停止一个侦听器，将停止：

watch

或 watchEffect 函数赋给一个变量，然后调用便可以在达到条件时

const unwatch = watchEffect(() => {})

// ...当该侦听器不再需要时unwatch()

## 模板引用

当需要直接访问底层DOM元素时，可以使用ref 属性实现：

<input ref="input">

使用组合式API获得模板引用，应该声明同名的ref

<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>

如果不是使用<script setup>，需要确保从 setup() 返回ref：

export default { setup() {

const input = ref(null)

// ... return {

input

}

}

}

当在 中使用模板引用时，对应的 ref 中包含的值是一个数组，它将在元素被挂载后包含对应整个

v-for

列表的所有元素：

<script setup>

import { ref, onMounted } from 'vue'

const list = ref([

/\* ... \*/

])

const itemRefs = ref([])

onMounted(() => console.log(itemRefs.value))

</script>

<template>

<ul>

<li v-for="item in list" ref="itemRefs">

{{ item }}

</li>

</ul>

</template>

注：此用法需要v3.2.25及以上版本

# 组 件基础

组件树：

## 定义一个组件

当使用构建步骤时，我们一般会将 Vue 组件定义在一个单独的称 SFC)：

.vue

文件中，这被叫做[单文件组件](https://cn.vuejs.org/guide/scaling-up/sfc.html) (简

<script setup>

import { ref } from 'vue'

const count = ref(0)

</script>

<template>

<button @click="count++">You clicked me {{ count }} times.</button>

</template>

## 使用组件

要使用一个子组件，我们需要在父组件中导入它。假设我们把计数器组件放在了一个叫做 的文件中，这个组件将会以默认导出的形式被暴露给外部。

ButtonCounter.vue

<script setup>

import ButtonCounter from './ButtonCounter.vue'

</script>

<template>

<h1>Here is a child component!</h1>

<ButtonCounter />

</template>

通过 <script setup> ，导入的组件都在模板中直接可用。组件可以被任意调用多次，每个组件维护自己的状态，没有相互影响

## 传递（props）

向组件中传递数据使用props 在顶层属性中使用：

<!-- BlogPost.vue -->

<script setup> props(['title'])

</script>

<template>

<h4>{{ title }}</h4>

</template>

如果你没有使用 <script setup> ，props 必须以函数的第一个参数被传入：

props

setup()

选项的方式声明，props 对象会作为

export default { props: ['title'], setup(props) {

console.log(props.title)

}

}

## 是一个仅 中可用的编译宏命令，并不需要显式地导入。声明的

defineProps

<script setup>

props 会自动暴露给模板。 defineProps **会返回一个对象，其中包含了可以传递给组件的所有**

## props：

const props = defineProps(['title']) console.log(props.title)

**监听事件**（示例代码：[BlogPost](https://sfc.vuejs.org/%23eNp9Uk1v2zAM/SuEMMApFlvLtpPhBN2A7jagwIDtUO%2BQOLSj1JYESU7nGf7vo/wRp2tTn8z3Hh8pki37onV0qpHFLLGZEdqBRVfrTSpFpZVx0ILBHDrIjaogIGlwpr6WqrhX1o1cxCfAO5IMIJWZkiTQBFpYe6vFgydaEPsYVktwwpUYQ/C9gaOqjcQGnoQ7wE9ygG45az/OWl%2BmELK4ovw0K38dGs%2BDsGAV5LUkZSp/36TysrNvSrof4i%2BODa6ITvgwDZoDBQ4rXW4d%2Bsgle3GC2LqmxHXKWsjH7Pi513sIsPLlGGUBfck0nSEEOIW5MmTh00DIYUgpm%2Bj4EZuRjcT%2BAu8fNzF9MJO3KMutKTB0%2BMeNmrmlNXyIVpN4k5z31XeYcHoY/SX8/Fy2ZMOqw2qro6NVku6k7Rc7EtRvDD3iMVq7j1N2cE7bmHObZ/4WjjZSpuD0F5laOlFhhLYKd0Y9WTRknLJ%2Bf6MHJ/CEJjQo92jQvOX5n/SFr7ellXf0lMvrfOXe95gLifdGabt4CPq5Bv5SBvyuEs7jl/Pt6auXQhP1l5KVW2tpEzuqHvp1zAdx%2BLxp2%2BFWoesSTvHI7GrnlITbrBTZIyW/Qyq/eF78hozuBgA8kPAh6%2Boyu3%2B/5lNs)）

父组件可以通过 或 来选择性地监听子组件上抛的事件，就像监听原生 DOM 事件那样：

v-on



@

<BlogPost

...

@enlarge-text="postFontSize += 0.1"

/>

子组件可以通过调用内置的 方法，通过传入事件名称来抛出一个事件：

$emit

<!-- BlogPost.vue, 省略了 <script> -->

<template>

<div class="blog-post">

<h4>{{ title }}</h4>

<button @click="$emit('enlarge-text')">Enlarge text</button>

</div>

</template>

我们可以通过 defineEmits 宏来声明需要抛出的事件：

<!-- BlogPost.vue -->

<script setup> defineProps(['title']) defineEmits(['enlarge-text'])

</script>

和 类似， defineEmits 仅可用于 之中，并且不需要导入，它返回一

defineProps

<script setup>

$emit

emit

个等同于

方法的

函数。

它可以被用于在组件的 中抛出事件，因为此处无法直接访问 $emit ：

<script setup>

<script setup>

const emit = defineEmits(['enlarge-text'])

emit('enlarge-text')

</script>

不使用<script setup>，可以通过 emits 选项自定义组件抛出的事件，可以从 函数的第二个参

setup()

数，即 setup 上下文对象上访问到 函数：

emit

export default {

emits: ['enlarge-text'], setup(props, ctx) {

ctx.emit('enlarge-text')

}

}

## 通过插槽分配内容

想和HTML元素一样向组件中传递内容，可以通过vue自定义的<slot>元素来实现

<template>

<div class="alert-box">

<strong>This is an Error for Demo Purposes</strong>

<slot />

</div>

</template>

<style scoped>

.alert-box {

/\* ... \*/

}

</style>

如上，使用 作为一个占位符，父组件传递进来的内容就会渲染在这里。

<slot>