> 本文由 简悦 SimpRead 转码， 原文地址 kaiwu.lagou.com

很多人学习 Vue.js，会把 Vue.js 的响应式原理误解为双向绑定。其实响应式原理是一种单向行为，它是数据到 DOM 的映射。而真正的双向绑定，除了数据变化，会引起 DOM 的变化之外，还应该在操作 DOM 改变后，反过来影响数据的变化。

那么 Vue.js 里有内置的双向绑定的实现吗？答案是有的，v-model 指令就是一种双向绑定的实现，我们在平时项目开发中，也经常会使用 v-model。

v-model 也不是可以作用到任意标签，它只能在一些特定的表单标签如 input、select、textarea 和自定义组件中使用。

那么 v-model 的实现原理到底是怎样的呢？接下来，我们从普通表单元素和自定义组件两个方面来分别分析它的实现。

## 在普通表单元素上作用 v-model

首先，我们来看在普通表单元素上作用 v-model，还是先举一个基本的示例：`<input v-model="searchText"/>`。

我们先看这个模板编译后生成的 render 函数：

```
import { vModelText as _vModelText, createVNode as _createVNode, withDirectives as _withDirectives, openBlock as _openBlock, createBlock as _createBlock } from "vue"


export function render(_ctx, _cache, $props, $setup, $data, $options) {


return _withDirectives((_openBlock(), _createBlock("input", {


"onUpdate:modelValue": $event => (_ctx.searchText = $event)


  }, null, 8 , ["onUpdate:modelValue"])), [


    [_vModelText, _ctx.searchText]


  ])


}
```

可以看到，作用在 input 标签的 v-model 指令在编译后，除了使用 withDirectives 给这个 vnode 添加了 vModelText 指令对象外，还额外传递了一个名为 onUpdate:modelValue 的 prop，它的值是一个函数，这个函数就是用来更新变量 searchText。

我们来看 vModelText 的实现：

```
const vModelText = {
```

```
  created(el, { value, modifiers: { lazy, trim, number } }, vnode) {
    el.value = value == null ? '' : value
    el._assign = getModelAssigner(vnode)
const castToNumber = number || el.type === 'number'
    addEventListener(el, lazy ? 'change' : 'input', e => {
if (e.target.composing)
return
      let domValue = el.value
if (trim) {
        domValue = domValue.trim()
      }
else if (castToNumber) {
        domValue = toNumber(domValue)
      }
      el._assign(domValue)
    })
if (trim) {
      addEventListener(el, 'change', () => {
        el.value = el.value.trim()
```

```
      })

    }

if (!lazy) {

      addEventListener(el, 'compositionstart', onCompositionStart)

      addEventListener(el, 'compositionend', onCompositionEnd)

    }

  },

  beforeUpdate(el, { value, modifiers: { trim, number } }, vnode) {

    el._assign = getModelAssigner(vnode)

if (document.activeElement === el) {

if (trim && el.value.trim() === value) {

return

      }

if ((number || el.type === 'number') && toNumber(el.value) === value) {

return

      }

    }

const newValue = value == null ? '' : value

if (el.value !== newValue) {
```

```
      el.value = newValue

    }

  }

}

const getModelAssigner = (vnode) => {

const fn = vnode.props['onUpdate:modelValue']

return isArray(fn) ? value => invokeArrayFns(fn, value) : fn

}

function onCompositionStart(e) {

  e.target.composing = true

}

function onCompositionEnd(e) {

const target = e.target

if (target.composing) {

    target.composing = false

trigger(target, 'input')

  }

}
```

那么接下来，我们就来拆解这个指令的实现。首先，这个指令实现了两个钩子函数，created 和 beforeUpdate。

我们先来看 created 部分的实现，根据上节课的分析，我们知道第一个参数 el 是节点的 DOM 对象，第二个参数是 binding 对象，第三个参数 vnode 是节点的 vnode 对象。

created 函数首先把 v-model 绑定的值 value 赋值给 el.value，这个就是数据到 DOM 的单向流动；接着通过 getModelAssigner 方法获取 props 中的 onUpdate:modelValue 属性对应的函数，赋值给 el._assign 属性；最后通过 addEventListener 来监听 input 标签的事件，它会根据是否配置 lazy 这个修饰符来决定监听 input 还是 change 事件。

我们接着看这个事件监听函数，当用户手动输入一些数据触发事件的时候，会执行函数，并通过 el.value 获取 input 标签新的值，然后调用 el._assign 方法更新数据，这就是 DOM 到数据的流动。

至此，我们就实现了数据的双向绑定，就是这么简单。接着我们来看 input v-model 支持的几个修饰符都分别代表什么含义。

### lazy 修饰符

如果配置了 lazy 修饰符，那么监听的是 input 的 change 事件，它不会在 input 输入框实时输入的时候触发，而会在 input 元素值改变且失去焦点的时候触发。

如果不配置 lazy，监听的是 input 的 input 事件，它会在用户实时输入的时候触发。此外，还会多监听 compositionstart 和 compositionend 事件。

当用户在使用一些中文输入法的时候，会触发 compositionstart 事件，这个时候设置 e.target.composing 为 true，这样虽然 input 事件触发了，但是 input 事件的回调函数里判断了 e.target.composing 的值，如果为 true 则直接返回，不会把 DOM 值赋值给数据。

然后当用户从输入法中确定选中了一些数据完成输入后，会触发 compositionend 事件，这个时候判断 e.target.composing 为 true 的话则把它设置为 false，然后再手动触发元素的 input 事件，完成数据的赋值。

### trim 修饰符

如果配置了 trim 修饰符，那么会在 input 或者 change 事件的回调函数中，在获取 DOM 的值后，手动调用 trim 方法去除首尾空格。另外，还会额外监听 change 事件执行 el.value.trim() 把 DOM 的值的首尾空格去除。

### number 修饰符

如果配置了 number 修饰符，或者 input 的 type 是 number，就会把 DOM 的值转成 number 类型后再赋值给数据。

接下来我们再来看一下 beforeUpdate 钩子函数的实现，非常简单，主要就是在组件更新前判断如果数据的值和 DOM 的值不同，则把数据更新到 DOM 上。

前面我们的分析的是文本类型的 input，如果我们对示例稍加修改：

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

你可以看到，编译的结果不同，调用的指令也不一样了，我希望你可以举一反三，去自学其他类型的表单元素的 v-model 实现。

## 在自定义组件上作用 v-model

接下来，我们来分析自定义组件上作用 v-model，看看它与表单的 v-model 有哪些不同。还是通过一个示例说明：

```
app.component('custom-input', {
```

```
  props: ['modelValue'],
  template: `
    <input v-model="value">
  `,
  computed: {
    value: {
      get() {
return this.modelValue
      },
      set(value) {
this.$emit('update:modelValue', value)
      }
    }
  }
})
```

我们先通过 app.component 全局注册了一个 custom-input 自定义组件，内部我们使用了原生的 input 并使用了 v-model 指令实现数据的绑定。

注意这里我们不能直接把 modelValue 作为 input 对应的 v-model 数据，因为不能直接对 props 的值修改，因此这里使用计算属性。

计算属性 value 对应的 getter 函数是直接取 modelValue 这个 prop 的值，而 setter 函数是派发一个自定义事件 update:modelValue。

接下来我们就可以在应用的其他的地方使用这个自定义组件了：

```
<custom-input v-model="searchText"/>
```

我们来看一下这个模板编译后生成的 render 函数：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
openBlock as _openBlock, createBlock as _createBlock } from "vue"


export function render(_ctx, _cache, $props, $setup, $data, $options) {


const _component_custom_input = _resolveComponent("custom-input")


return (_openBlock(), _createBlock(_component_custom_input, {


    modelValue: _ctx.searchText,


"onUpdate:modelValue": $event => (_ctx.searchText = $event)


  }, null, 8 , ["modelValue", "onUpdate:modelValue"]))


}
```

可以看到，编译的结果似乎和指令没有什么关系，并没有调用 withDirective 函数。

我们对示例稍做修改：

```
<custom-input :modelValue="searchText" @update:modelValue="$event=>{searchText =
$event}"/>
```

然后我们再来看它编译后生成的 render 函数：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
openBlock as _openBlock, createBlock as _createBlock } from "vue"


export function render(_ctx, _cache, $props, $setup, $data, $options) {


const _component_custom_input = _resolveComponent("custom-input")


return (_openBlock(), _createBlock(_component_custom_input, {


    modelValue: _ctx.searchText,
```

```
"onUpdate:modelValue": $event=>{_ctx.searchText = $event}

  }, null, 8 , ["modelValue", "onUpdate:modelValue"]))

}
```

我们发现，它和前面示例的编译结果是一模一样的，因为 v-model 作用于组件上本质就是一个语法糖，就是往组件传入了一个名为 modelValue 的 prop，它的值是往组件传入的数据 data，另外它还在组件上监听了一个名为 update:modelValue 的自定义事件，事件的回调函数接受一个参数，执行的时候会把参数 $event 赋值给数据 data。

正因为这个原理，所以我们想要实现自定义组件的 v-model，首先需要定义一个名为 modelValue 的 prop，然后在数据改变的时候，派发一个名为 update:modelValue 的事件。

Vue.js 3.0 关于组件 v-model 的实现和 Vue.js 2.x 实现是很类似的，在 Vue.js 2.x 中，想要实现自定义组件的 v-model，首先需要定义一个名为 value 的 prop，然后在数据改变的时候，派发一个名为 input 的事件。

总结下来，作用在组件上的 v-model 实际上就是一种打通数据双向通讯的语法糖，即外部可以往组件上传递数据，组件内部经过某些操作行为修改了数据，然后把更改后的数据再回传到外部。

v-model 在自定义组件的设计中非常常用，你可以看到 Element UI 几乎所有的表单组件都是通过 v-model 的方式完成了数据的交换。

一旦我们使用了 v-model 的方式，我们必须在组件中申明一个 modelValue 的 prop，如果不想用这个 prop，想换个名字，当然也是可以的。

Vue.js 3.0 给组件的 v-model 提供了参数的方式，允许我们指定 prop 的名称：`<custom-input v-model:text="searchText"/>`。

然后我们再来看编译后的 render 函数：

```
import { resolveComponent as _resolveComponent, createVNode as _createVNode,
openBlock as _openBlock, createBlock as _createBlock } from "vue"

export function render(_ctx, _cache, $props, $setup, $data, $options) {

const _component_custom_input = _resolveComponent("custom-input")

return (_openBlock(), _createBlock(_component_custom_input, {

    text: _ctx.searchText,

"onUpdate:text": $event => (_ctx.searchText = $event)

  }, null, 8 , ["text", "onUpdate:text"]))
```

```
}
```

可以看到，我们往组件传递的 prop 变成了 text，监听的自定义事件也变成了 @update:text 了。

显然，如果 v-model 支持了参数，那么我们就可以在一个组件上使用多个 v-model 了：

```
<ChildComponent v-model:title="pageTitle" v-model:content="pageContent" />
```

至此，我们就掌握了组件 v-model 的实现原理，它

的本质就是语法糖：**通过 prop 向组件传递数据，并监听自定义事件接受组件反传的数据并更新。**

prop 的实现原理我们之前分析过，但自定义事件是如何派发的呢？因为从模板的编译结果看，除了 modelValue 这个 prop，还多了一个 onUpdate:modelValue 的 prop，它和自定义事件有什么关系？接下来我们就来分析这部分的实现。

## 自定义事件派发

从前面的示例我们知道，子组件会执行`this.$emit('update:modelValue',value)`方法派发自定义事件，$emit 内部执行了 emit 方法，我们来看一下它的实现：

```
function emit(instance, event, ...args) {

const props = instance.vnode.props || EMPTY_OBJ

  let handlerName = `on${capitalize(event)}`

  let handler = props[handlerName]

if (!handler && event.startsWith('update:')) {

    handlerName = `on${capitalize(hyphenate(event))}`

    handler = props[handlerName]

  }

if (handler) {

    callWithAsyncErrorHandling(handler, instance, 6 , args)

  }

}
```

emit 方法支持 3 个参数，第一个参数 instance 是组件的实例，也就是执行 $emit 方法的组件实例，第二个参数 event 是自定义事件名称，第三个参数 args 是事件传递的参数。

emit 方法首先获取事件名称，把传递的 event 首字母大写，然后前面加上 on 字符串，比如我们前面派发的 update:modelValue 事件名称，处理后就变成了 onUpdate:modelValue。

接下来，通过这个事件名称，从 props 中根据事件名找到对应的 prop 值，作为事件的回调函数。

如果找不到对应的 prop 并且 event 是以 update: 开头的，则尝试把 event 名先转成连字符形式然后再处理。

找到回调函数 handler 后，再去执行这个回调函数，并且把参数 args 传入。针对 v-model 场景，这个回调函数就是拿到子组件回传的数据然后修改父元素传入到子组件的 prop 数据，这样就达到了数据双向通讯的目的。

## 总结

好的，到这里我们这一节的学习也要结束啦，通过这节课的学习，你应该要了解 v-model 在普通表单元素上以及在自定义指令上的实现原理分别是怎样的，以及了解自定义事件派发的实现原理。

最后，给你留一道思考题目，如果自定义组件不用 v-model，也不用自定义事件监听的方式，如何实现和 v-model 一样的效果，怎么做呢？欢迎你在留言区与我分享。

> 本节课的相关代码在源代码中的位置如下：
> packages/runtime-dom/src/directives/vModel.ts