第04课：.sync 王者回归，v-model 使命将至

上一章我们已经对 $emit和v-on 如何进行数据和行为的交互做了讲解，但如果只是简单用来数据传递改变的话 .sync和v-model 再适合不过了。如果用过1.0的 Vue 的开发者，我相信 .sync 会让你用起来非常便捷，通过双向绑定很简单就能实双，父子组件的双向绑定，2.0为了保持单向数据流的良好性，去除了 .sync 的功能。

官方解释：

1.0 Props 现在只能单向传递。为了对父组件产生反向影响，子组件需要显式地传递一个事件而不是依赖于隐式地双向绑定。

推荐使用
* 自定义组件事件
* 自定义输入组件 (使用组件事件)
* 全局状态管理

通过大量观察，在初期2.0版本中，因为 .sync 并没有回归，只是在2.3进行回归，在组件库中进行数据双向绑定，几乎都是通过 v-model 来进行的。但是无论从语意上还是感观上，给代码维护的感就是不直观，v-model 在开发通常都是结合 Input 输入框来结合进行一个数据绑定，进行父子组件双向绑定，但是相比自定义 v-on 组件事件，无论从代码量，还是用法上更加简洁。

在 Vue 中，有许多方法和 Angular 相似，这主要是因为 Angular 是 Vue 早期开发的灵感来源。然而 Angular 中存在许多问题，在 Vue 中已经得到解决。

官方解释

自定义事件可以用来创建自定义的表单输入组件，使用 v-model 来进行数据双向绑定。

```
<input v-model="something">
```

这不过是以下示例的语法糖：

```
<input
  v-bind:value="something"
  v-on:input="something = $event.target.value">
```

v-model 其实也是一个语法糖，想要理解这些代码，你要先知道 Input元素 上本身有个 oninput事件 ，这是 HTML5新增加 的，类似 onchange，每当输入框内容发生变化的时

候，就会触发 Input事件 ，然后把 Input 输入框中 value 值再次传递给 something。

此时 value 运用在一个 Input 元素上，用：v-bind:value=’something’，意义上面只是把 Input 输入框中的 value 值与 something 作为一一对应的双向绑定，这就像一个循环操作，当再次触发 Input 事件时，`input($event.target)` 对象中的value值会再次改变 something 。

这里我们对 v-model 绑定在 Input 元素上进行语法糖上的解析。

既然在元素上能进行双向绑定，那在组件中进行双向绑定又如何实现，原理其实都是一样的，只是应用在自定义的组件上时，拿的并不是 `$event.target.value` ，因为我此时不作用在 Input 输入框上。

在组件中使用时，它相当于下面的简写：

```
<custom-input
  v-bind:value="something"
  v-on:input="something = arguments[0]">
</custom-input>
```


通过以上简写，通过自定事件让 v-model 进行一个父子组件双向绑定的话。

- v-bind:value=’something’ 此时 value 是作为子组件接收的 Props

接收的只能是 value 吗？必须是，因为 v-model 是基于 Input 输入框定制的，其中value 值是为 Input 内部定制的

```
 v-on:input="something = arguments[0]"
```

此时作用在组件上时，v-on 监听的语法糖也会有所改动，监听的并不是 `$event.target.value` ，而是 回调函数中的第一个参数 。

父组件

```
<template>
  <div class="hello">
    <button @click="show=true">打开model</button>
    <demo v-model="show"></demo>
  </div>
</template>

<script>
import Demo from './Demo.vue'
```

```
export default {
  name: 'hello',
  components: {
    Demo
  },
  data () {
    return {
      show: false
    }
  }
}
</script>
```

子组件

```
<template>
   <div v-show="value">
      <div>
         <p>这是一个Model框</p>
         <button @click="close">关闭model</button>
      </div>
   </div>
</template>

<script>
export default {
  props: ['value'],
  methods: {
    close () {
      this.$emit('input',false)
    }
  }
}
</script>
```


这是一个模态框的基本雏形，可以在父组件通过 v-model 来进行 model 框和父组件之间的显示交互。

通过子组件看出 通过props接收了value值 ， 当点击关闭的时候还是通过 $emit事件触发input事件 ， 然后通过传入 false 参数。

父组件隐式 v-on:input=”something = arguments[0]” 进行了监听，一但 Input 事件触发，父组件就会执行监听回调，从而做到了双向绑定。

> checkbox 和 radio 原理

```
 <input type="checkbox" :checked="status" @change="status =
$event.target.checked" />

  <input type="radio" :checked="status" @change="status =
$event.target.checked" />
```

通过绑定 checked 属性，同样的监听的是 change 事件，无论是 checkbox 还是 radio 在操作的时候都会隐式自动触发一个 change 事件，跟 Input 通过 value 值，Input 触发事件原理绑定是一样的。

定制组件 v-model

定制组件，我们就可以 重写v-model里的Props 和 event ，默认情况下，一个组件的 v-model 会使用 value 属性 和 input 事件 ，往往有些时候，value 值被占用了，或者表单的和自定议 v-model的$emit('input')事件发生冲突 ，为了避免这种冲突，可以定制组件 v-model，冲突示例。

子组件


```
<template>
   <div v-show="value">
      <div>
         <p>这是一个Model框</p>
        <input type="text" v-model="value">
        {{value}}
         <button @click="close">关闭model</button>
      </div>
   </div>
</template>

<script>
export default {
  props: ['value'],
  methods: {
    close () {
      this.$emit('input',false)
    }
  }
}
</script>
```

父组件

```
<template>
  <div class="hello">
    <button @click="show=true">打开model</button>
    <demo v-model="show"></demo>
  </div>
</template>

<script>
  import Demo from './Demo.vue'
  export default {
    name: 'hello',
    components: {
      Demo
    },
    data () {
      return {
        show: false
      }
    }
  }
</script>
```

上面例子可以发现，在子组件中 input中v-model 和 model显示的操作数据共同占用的 props 中的（value），同样两者也共同占用了 emit('input') 触发事件，Input 输入框的事件是自动出发，而 model 显示消失是手动触发。

初始化的时候，Input 输入框的值的会被 value 传入的 false 值给自动加上，当改变 Input 输入框的时候，因为冲突而导致报错。

定制 v-model， 通过 model 选项改变 props 和 event 的值，从而解除两者的冲突。

- props代替掉原本 value 的值，可以自定义值
- event代表掉原本 input 的触发事件，可以自定义触发事件

子组件

```
<template>
   <div v-show="show">
      <div>
         <p>这是一个Model框</p>
        <input type="text" v-model="value">
        {{value}}
         <button @click="closeModel">关闭model</button>
      </div>
   </div>
</template>

<script>
```

```
export default {
  model: {
    prop: 'show',
    event: 'close'
  },
  props: ['show'],
  data () {
     return {
       value: 10
     }
  },
  methods: {
    closeModel () {
      this.$emit('close',false)
    }
  }
}
</script>
```

父组件


```
<template>
  <div class="hello">
    <button @click="show=true">打开model</button>
    <demo v-model="show" ></demo>
  </div>
</template>

<script>
  import Demo from './Demo.vue'
  export default {
    name: 'hello',
    components: {
      Demo
    },
    data () {
      return {
        show: false
      }
    }
  }
</script>
```

通过 model 选项的改变，把 props 从原本的 value换成了show， input触发的事件换成了 close ，从而两者都不相互依赖，解决了冲突的问题。

有些时候通过父组件中的子组件模板中想传递 value 值，也会导致同样的冲突。

在不用定制组件的情况下，以下的写法，也会同样导致冲突，导致同用一个 value。

```
<demo v-model="show" value="some value"></demo>


props:['value']
```

王者回归 .sync

在一些情况下，我们可能会需要对一个 prop 进行『双向绑定』。事实上，这正是 Vue 1.x 中的 .sync 修饰符所提供的功能。当一个子组件改变了一个 prop 的值时，这个变化也会同步到父组件中所绑定的值。这很方便，但也会导致问题，因为它破坏了『单向数据流』的假设。由于子组件改变 prop 的代码和普通的状态改动代码毫无区别，当光看子组件的代码时，你完全不知道它何时悄悄地改变了父组件的状态。这在 debug 复杂结构的应用时会带来很高的维护成本。

在2.0发布一段之后，无论在业务组件还是在功能组件库上面的，大量的子组件改变父子组件的数据和组件库中可能达到大功率的复用，但是在2.3中回归，重新引入了 .sync 修饰符，这次它只是作为一个编译时的语法糖存在。它会被扩展为一个自动更新父组件属性的 v-on 侦听器。

之前的例子中，v-model 毕竟不是给组件与组件之间通信而设计的双向绑定，无论从语意上和代码写法上都没有 .sync 直观和方便。

无论从 v-model 还是 .sync 修饰符来看，都离不开 $emit v-on 语法糖的封装，主要目的还是为了保证数据的正确单向流动与显示流动。

```
<demo :foo.sync="something"></demo>
```

语法糖的扩展：

```
<demo :foo="something" @update:foo="val => something = val">
</demo>
```

- foo 则是 demo 子组件需要从父组件 props 接收的数据
- 通过事件显示监听 update:foo (foo则是 props 显示监听的数据)，通过箭头函数执行回调，把参数传给 something，则就形成了一种双向绑定的循环链条

当子组件需要更新 foo 的值时，它需要显式地触发一个更新事件：

```
this.$emit('update:foo', newValue)
```

同时父组件 @update:foo 也是依赖于子组件的显示触发，这样就可以很轻松的捕捉到了数据的正确的流动 。

第一个参数则是 update 是显示更新的事件，跟在后面的： foo 则是需要 改变对应的 props值 。

第二个参数传入的是你 希望父组件foo数据里将要变化的值 ， 以 用于父组件接收update时更新数据 。

子组件

```
<template>
    <div v-show="show">
        <p>这是一个Model框</p>
        <button @click="closeModel">关闭model</button>
    </div>
</template>
<script>
export default {
  props: ['show'],
  methods: {
    closeModel () {
      this.$emit('update:show',false)
    }
  }
}
</script>
```


父组件

```
<template>
  <div class="hello">
    <button @click="show=true">打开model</button>
    <demo :show.sync="show" ></demo>
  </div>
</template>

<script>
  import Demo from './Demo.vue'
  export default {
    name: 'hello',
    components: {
      Demo
    },
    data () {
      return {
```

```
        show: false
      }
    }
  }
</script>
```

上面的 case 同样也解决了 model 显示交互操作，从代码的语意上看上去让开发者一目了然，同样也做了 v-model 做不了的事，基于 props 的原子化，对传入的 props 进行多个数据双向绑定 .sync 也能轻松做到。

父组件

```
<template>
  <div class="hello">
    <button @click="show=true">打开model</button>
    <demo :show.sync="show" :msg.sync="msg"></demo>
  </div>
</template>

<script>
  import Demo from './Demo.vue'
  export default {
    name: 'hello',
    components: {
      Demo
    },
    data () {
      return {
        show: false,
        msg: '这是一个model'
      }
    }
  }
</script>
```


子组件

```
<template>
   <div v-show="show">
      <p>{{msg}}</p>
      <button @click="closeModel">关闭model</button>
     <button @click="$emit('update:msg','改变了model文案')">改变文案
</button>
   </div>
</template>
<script>
```

```
export default {
  props: ['show', 'msg'],
  methods: {
    closeModel () {
      this.$emit('update:show',false)
    }
  }
}
</script>
```

父组件向子组件 props 里传递了 msg 和 show 两个值，都用了.sync 修饰符，进行双向绑定。

warn

子组件改变父组件的数据时，update 冒号后面的参数和父组件传递进来的值是同步的，想改变那个，则冒号后面的值对应的那个，两者是一一对应的，同时也是必填的。

同样还可以在组件 template 里点击执行 click 后不但可以支持回调函数，还可以写入表达式，只是一种直观的表现还是推荐这种写法的。

.sync 修饰符给我们开发中带来了很大的方便，同时在2.0的初期的组件库中大量的 v-model 给开发者用起来还是很别扭，在.sync 回归后同时也会慢慢向.sync 进行一个版本的迁移。

> 下篇课程导读：

不基于大量行为操作，只是进行一个或多个数据双向组件的时候，可以轻松用 .sync 与 v-model 去化解，往往组件通信并不是你想像的那么轻松简单，在项目复杂的时候，组件如何合理的拆分，会让业务代码的 清晰度 ， 复用率 ， 后续维护都会降低成本 ，有利必有困难，同样会造成组件与组件的深层次传递，那我们如何进行通信呢？第一个想到的办法必然是 Vuex。Vuex 理解其实本质上并不是处理跨度深层次组件而使用的，往往这样会导致大家会滥用 Vuex，而 `$attrs` `$listeners` 这对兄弟可以很好的帮助你进行深组件的通信。