# 第03课： $on， $emit， v-on 三者关系

## 用家庭来描述 $emit， $on 的关系

每个 Vue 实例都实现了事件接口：

- 使用 $on(eventName) 监听事件
- 使用 $emit(eventName) 触发事件

如果把 Vue 看成一个家庭（相当于一个单独的 components），女主人一直在家里指派（$emit）男人做事，而男人则一直监听（$on）着女士的指派（$emit）里eventName 所触发的事件消息，一旦 $emit 事件一触发， $on 则监听到 $emit 所派发的事件，派发出的命令和执行派执命令所要做的事都是一一对应的。

Api 中的解释：

```
vm.$emit( event, […args] )
```



参数：
* {string} event
* [...args]

触发当前实例上的事件。附加参数都会传给监听器回调。

---

```
vm.$on( event, callback )
```

参数：

{string | Array<string>} event (数组只在 2.2.0+ 中支持)
{Function} callback

用法：

监听当前实例上的自定义事件。事件可以由 vm.$emit 触发。回调函数会接收所有传入事件触发函数的额外参数。

---

```
<template>
  <div>
```

```
        <p @click='emit'>{{msg}}</p>
    </div>
</template>

<script>
export default {
    name: 'demo',
    data () {
        return {
            msg : '点击后女人派发事件'
        }
    },
    created () {
        this.$on('wash_Goods',(arg)=> {
            console.log(arg)
        })
    },
    methods : {
        emit () {
            this.$emit('wash_Goods',['fish',true,
{name:'vue',verison:'2.4'}])
        }
    }
}
</script>
```



以上案例说了什么呢？在文章开始的时候说了 `$emit` 的（eventName）是与 `$on(eventName)` 是一一对应的，再结合以上两人在组成家庭的之前，女人会给男人列一个手册，告诉男人我会派发 `$(emit)` 那些事情，男人则会在家庭组成之前 `$on(eventName)` 后应该如何做那些事情。

通过以上说明我来进一步解释一下官方 Api 的意思。

---

```
vm.$emit( event, […args] )
```

参数：

```
* {string} event
```

- 第一个参数则是所要派发的事件名，必须是 String 类型的。
- 故事中就是要告诉男人所需要执行的事情。
    - `[...args]`
- 第二个参数是一个任何数据类型，如果我们需要传入多个不同的数据类型，则可以写入数组中，像这样[object,Boolean,function,string,...]，只要传一个参数，我们则可以直接写入

```
this.$emit('wash_Goods','fish')
```

- 故事中就是给男人的一个手册，告诉男人东西放在哪里，会需要到什么工具等等。

---

`vm.$on( event, callback )`

参数：

- `{string | Array<string>}` event (数组只在 `2.2.0+` 中支持)
    - 第一个参数是相对于 `$emit（eventName）`
      一一对应的 `$on（eventName）`，两者是并存的、必须是 String 类型的。
    - （数组只在 2.2.0+ 中支持）或者是 `Array<String>` 数组中必须包含的是 String 项，后面再具体说。
- 故事中就是男人在组件一个家庭 (components) 的时候所监听的事件名。
- `{Function} callback`
    - 第二个参数则是一个 function，同样也被叫作之前回调函数，里面可以接收到由 $emit 触发时所传入的参数（如果是单个参数）。
    - 故事中是男人在接收到女人派发的事情该去做那些事情。



---

> {string | Array} event (数组只在 `2.2.0+` 中支持)

在2.2中新增这个 Api 牵扯了另一种方式，也存在这其它的独特用法。

继续延续故事，当女人派发的事情多了，我相信作为男人也会觉得很烦，一旦听到事件的时候肯定会很烦躁，总会抱怨两句。

如果女人在组成家庭之前，告诉男人将要监听那些事情，如果做一件事就抱怨一次，启不是多此一举，所以我们可以通过 `Array<string> event` 把事件名写成一个数组，在数组里写入你所想监听的那些事件，使用共享原则去执行某些派发事件。

```
<template>
  <div>
      <p @click='emit'>{{msg}}</p>
      <p @click='emitOther'>{{msg2}}</p>
  </div>
</template>

<script>
```

```
export default {
  name: 'demo',
  data () {
      return {
         msg : '点击后女人派发事件',
         msg2 : '点击后女人派发事件2',
      }
  },
  created () {

      this.$on(['wash_Goods','drive_Car'],(arg)=> {
          console.log('事真多')
      })
      this.$on('wash_Goods',(arg)=> {
          console.log(arg)
      })
      this.$on('drive_Car',(...arg)=> {
          console.log(BMW,Ferrari)
      })
  },
  methods : {
      emit () {
         this.$emit('wash_Goods','fish')
      },
      emitOther () {
         this.$emit('drive_Car',['BMW','Ferrari'])
      }
  }
}
</script>
```



以上案例说明了当女人无论是派发 drive_Car 或者是 wash_Goods 事件，都会打印出 事真多 ，再执行一一对应监听的事件。

通常情况下，以上用法是毫无意思的。在平常业务中，这种用法也用不到，通常在写组件的时候，让 $emit在父级作用域中 进行一个触发，通知子组件的进行执行事情。接下来，可以看一个通过在父级组件中，拿到子组件的实例进行派发事件，然而在子组件中事先进行好派好事件监听的准备，接收到一一对应的事件进行一个回调，同样也可以称之为封装组件向父组件暴露的接口。

## DEMO 下拉加载 infinite-scroll

```
<template>
    <div>
        <slot name="list"></slot>

        <div class="list-donetip" v-show="!isLoading && isDone">
            <slot>没有更多数据了</slot>
```

```
        </div>

        <div class="list-loading" v-show="isLoading">
            <slot>加载中</slot>
        </div>
    </div>
</template>

<script type="text/babel">

    export default {
        data() {
            return {
                isLoading: false,
                isDone: false,
            }
        },
        props: {
            onInfinite: {
                type: Function,
                required: true
            },
            distance : {
                type : Number,
                default: 100
            }
        },
        methods: {
            init() {
                this.$on('loadedDone', () => {
                    this.isLoading = false;
                    this.isDone = true;
                });

                this.$on('finishLoad', () => {
                    this.isLoading = false;
                });
            },
            scrollHandler() {
                if (this.isLoading || this.isDone) return;
                let baseHeight = this.scrollview == window ?
document.body.offsetHeight : this.scrollview.offsetHeight
                let moreHeight = this.scrollview == window ?
document.body.scrollHeight : this.scrollview.scrollHeight;
                let scrollTop = this.scrollview == window ?
document.body.scrollTop : this.scrollview.scrollTop

                if (baseHeight + scrollTop + this.distance >
moreHeight) {
                    this.isLoading = true;
                    this.onInfinite()
                }
```

```
            }
        },
        mounted() {
            this.scrollview = window
            this.scrollview.addEventListener('scroll',
this.scrollHandler, false);
            this.$nextTick(this.init);
        },
    }
</script>
```

对下拉组件加载加更的组件进行了一个简单的封装：

`data` 参数解释：
* isLoading `false` 代表正在执行下拉加载获取更多数据的标识， `true`代表数据加载完毕
* isDone `false` 代表数据没有全完加载完毕， `true` 代表数据已经全部加载完毕

`props` 参数解释：
* onInfinite 父组件向子组件传入当滚动到底部时执行加载数据的函数
* distance 距离滚动到底部的设定值

从此组件中，我们进行每一步的分析



- 在 `mounted` 的时候，对 `window` 对像进行了一个滚动监听，监听的函数为 `scrollHandler`
    - 当 `isLoading`，`isDone` 任何一个为true时则退出
        - `isloading` 为 `true` 时防止多次同样加载，必须等待加载完毕
        - `isDone` 为 `true` 时说明所有数据已经加载完成，没有必要再执行 `scrollHandler`
    - 同时在$nextTick中进行了初始化监听
        - `loadedDone` 一旦组件实例$emit('loadedDone')事件时，执行回调，放开加载权限
        - `finishLoad` 一旦组件实例$emit('finishLoad')事件时，执行回调，放开加载权限
- 再看看 scrollHandler函数里发生了什么
    - `if (this.isLoading || this.isDone) return;` 一旦一者为true，则退出，原因在mounted已经叙述过了
    - `if (baseHeight + scrollTop + this.distance > moreHeight)` 当在window对象上监听scroll事件时，当滚动到底部的时候执行
        - `this.isLoading = true;` 防止重复监听
        - `this.onInfinite()` 执行加载数据函数

> 父组件中调用 infinite-scroll 组件

```
<template>
      <div>
          <infinite-scroll :on-infinite='loadData'
ref='infinite'>
                <ul slot='list'>
                   <li v-for='n in Number'></li>
                </ul>
          </infinite-scroll>
      </div>
</template>

<script type="text/babel">
import 'InfiniteScroll' from '.......' //引入infinitescroll.vue文
件
    export default {
         data () {
           return {
              Number : 10
           }
         },
         methods : {
           loadData () {
             setTimeout(()=>{
               this.Number = 20
               this.$refs.infinite.$emit('loadDone')
             },1000)
           }
         }
    }
</script>
```



在父组件中引入 infinite-scroll 组件

当滑到底部的时候，infinite-scroll 组件组件内部会执行传入的 :on-infinite='loadData' 函数
同时在内部也会把 Loading 设置为 true，防止重复执行。

在这里用 this.$refs.infinite 拿到 infinite-scroll 组件的实例，同时触发事件之前在组件中 $on 已经监听着的事件，在一秒后进行改变数据，同时发出 loadDone 事情，告诉组件内部去执行 loadDone 的监听回调，数据已经全部加载完毕，设置 this.isDone = true;
一旦 isDone 或者 isLoading 一者为 true，则一直保持 return退出状态 。

**$emit 和 $on 必须都在实例上进行触发和监听。**

## v-on 使用自定义绑定事件

第一阶段 $emit 和 $on 的两者之间的关系讲完了，接下来该说说 v-on 与 $emit 的关系。

**另外，父组件可以在使用子组件的引入模板直接用 v-on 来监听子组件触发的事件。**

v-on 用接着故事直观的说法就是，在家里装了一个电话，父母随一直听着电话，同样也有一本小册子，在组成家庭之前，也知识要去监听那些事。

## Warn

不能用 $on 侦听子组件释放的事件，而必须在模板里直接用 v-on 绑定 。

上面 Warn 的意思是 $emit和$on只能作用在一一对应的同一个组件实例 ， 而 v-on只能作用在父组件引入子组件后的模板上 。

就像下面这样：

```
<children v-on:eventName="callback"></children>
```

就拿官方的这个例子说吧，其实还是很直观的：



```
<div id="counter-event-example">
  <p>{{ total }}</p>
  <button-counter v-on:increment="incrementTotal"></button-
counter>
  <button-counter v-on:increment="incrementTotal"></button-
counter>
</div>
Vue.component('button-counter', {
  template: '<button v-on:click="incrementCounter">{{ counter }}
</button>',
  data: function () {
    return {
      counter: 0
    }
  },
  methods: {
    incrementCounter: function () {
      this.counter += 1
      this.$emit('increment')
    }
  },
})
new Vue({
  el: '#counter-event-example',
  data: {
    total: 0
  },
  methods: {
```

```
    incrementTotal: function () {
      this.total += 1
    }
  }
})
```

这样的好处在哪里？虽然 Vue 是进行数据单向流的，但是子组件不能直接改变父组件的数据，(也不是完全不能，但不推荐用)，标准通用明了的用法，则是通过父组件在子组件模板上进行一个 v-on 的绑定监听事件，同时再写入监听后所要执行的回调。

在 counter-event-example 父组件里，声明了两个 button-count 的实列，通过 data 用闭包的形式，让两者的数据都是单独享用的，而且 v-on 所监听的 eventName 都是当前自己实列中的 $emit 触发的事件，但是回调都是公用的一个 incrementTotal 函数，因为个实例所触发后都是执行一种操作！

如果你只是想进行简单的进行父子组件基础单个数据进行双向通信的话，在模板上通过 v-on 和所在监听的模板实例上进行 $emit 触发事件的话，未免有点多余。通常来说通过 v-on 来进行监听子组件的触发事件的话，我们会进行一些多步操作。

## 子组件



```
<template>
  <div>
      <p @click='emit'>{{msg}}</p>
  </div>
</template>

<script>
export default {
  name: 'demo',
  data () {
      return {
         msg : '点击后改变数据',
      }
  },
  methods : {
      emit () {
         this.$emit('fromDemo')
      },
  }
}
</script>
```

## 父组件

```
<template>
  <div class="hello">
     <p>hello {{msg}}</p>
     <demo v-on:fromDemo='Fdemo'></demo>
  </div>
</template>
<script>
import Demo from './Demo.vue'
export default {
  name: 'hello',
  data () {
    return {
       msg: '数据将在一秒后改变'
    }

  },
  methods: {
    waitTime() {
      return new Promise(resolve=>{
        setTimeout(()=> {
            this.msg = '数据一秒后改变了'
            resolve(1)
        },1000)
      })
    },
    async Fdemo () {
        let a = await this.waitTime();
        console.log(a)
    }
  },
  components : {
     Demo
  }
}
</script>
```



从上面 demo 可以看出当子组件触发了 fromDemo 事件，同时父组件也进行着监听。

当父组件接收到子组件的事件触发的时候，执行了 async 的异步事件，通过一秒钟的等秒改变 msg，再打印出回调后通过 promise 返回的值。

接下来想通的此例子告诉大家，这种方法通常是通过监听子组件的事件，让父组件去执行一些多步操作，如果我们只是简单的示意父组件改变传递过来的值用此方法就显的多余了。

我们进行一些的改动：

children

```
<template>
  <div>
      <p @click='emit'>{{msg}}</p>
  </div>
</template>

<script>
export default {
  name: 'demo',
  props: [ 'msg' ],
  methods : {
      emit () {
         this.$emit('fromDemo','数据改变了')
      },
  }
}
</script>
```

parent

```
<template>
  <div class="hello">
     <demo v-on:fromDemo='Fdemo' :msg='msg'></demo>
  </div>
</template>
<script>
import Demo from './Demo.vue'
export default {
  name: 'hello',
  data () {
    return {
       msg: '数据没有改变'
    }
  },
  methods: {
    Fdemo (arg) {
      this.msg = arg
    }
  },
  components : {
     Demo
  }
}
</script>
```



上面 demo 中子组件从父组件接收一个 msg 数据，但是想点击按钮的时候，改变父组件的 msg，进行父组件的数据改动，同时再次改变子组件的 msg，但是最简便的方法则是直接改变 prop 里 msg 的数据。但是数据驱动都是单向数据流，为了不造成数据传递的混

乱，我们只能依靠一些其它手段去完成，一个小小的传递数据就显得很复杂的了，所以后续我们会讲讲如何去用更简便的 Api 做对应的事。

> 下篇课程导读：

在2.0初期 .sync 被砍了，v-model 承担起了双向绑定的职责，毕竟 v-model 不是为组件与组件之间数据双向绑定而设计的，用起来总有蹩脚的时候。2.3版本的回归，启用了显示通知的形式让双向绑定又活了，.sync 或者 v-model 比 `$emit` 与 v-on 只是进行简单的父子组件数据交互更加便捷。