

# Vue组件通信方式居然有这么多？你了解几种

 [mp.weixin.qq.com/s/BK8NJwOoymnfK3vB9Fas1Q](https://mp.weixin.qq.com/s/BK8NJwOoymnfK3vB9Fas1Q)

前端新世界 2022-01-05 12:47

收录于话题 #vue 34个



喜欢就关注我们吧

作者：小科比\_

来源：<https://juejin.cn/post/6887709516616433677>

vue组件通信的方式，这是在面试中一个非常高频的问题，我刚开始找实习便经常遇到这个问题，当时只知道回到props和\$emit，后来随着学习的深入，才发现vue组件的通信方式竟然有这么多！

今天对vue组件通信方式进行一下总结，如写的有疏漏之处还请大家不吝赐教。

## 1. props/\$emit 前端开发博客

### 简介

props和\$emit相信大家十分的熟悉了，这是我们最常用的vue通信方式。

**props**：props可以是数组或对象，用于接收来自父组件通过v-bind传递的数据。当props为数组时，直接接收父组件传递的属性；当props为对象时，可以通过type、default、required、validator等配置来设置属性的类型、默认值、是否必传和校验规则。

**\$emit**：在父子组件通信时，我们通常会使用\$emit来触发父组件v-on在子组件上绑定相应事件的监听。

### 代码实例

下面通过代码来实现一下props和\$emit的父子组件通信，在这个实例中，我们都实现了以下的通信：

- 父向子传值：父组件通过 `:messageFromParent="message"` 将父组件 message 值传递给子组件，当父组件的 input 标签输入时，子组件p标签中的内容就会相应改变。

- 子向父传值：父组件通过 `@on-receive="receive"` 在子组件上绑定了 `receive` 事件的监听，子组件 `input` 标签输入时，会触发 `receive` 回调函数，通过 `this.$emit('on-receive', this.message)` 将子组件 `message` 的值赋值给父组件 `messageFromChild`，改变父组件 `p` 标签的内容。

请看代码：

```
// 子组件代码
<template>
  <div class="child">
    <h4>this is child component</h4>
    <input type="text" v-model="message" @keyup="send" />
    <p>收到来自父组件的消息：{{ messageFromParent }}</p>
  </div>
</template>
<script>
export default {
  name: 'Child',
  props: ['messageFromParent'], // 通过props接收父组件传过来的消息
  data() {
    return {
      message: '',
    }
  },
  methods: {
    send() {
      this.$emit('on-receive', this.message) // 通过$emit触发on-receive事件，调用父组件中receive回调，并将this.message作为参数
    },
  },
}
</script>
```

```
// 父组件代码
<template>
  <div class="parent">
    <h3>this is parent component</h3>
    <input type="text" v-model="message" />
    <p>收到来自子组件的消息：{{ messageFromChild }}</p>
    <Child :messageFromParent="message" @on-receive="receive" />
  </div>
</template>
<script>
import Child from './child'
export default {
  name: 'Parent',
  data() {
    return {
      message: '', // 传递给子组件的消息
      messageFromChild: '',
    }
  },
  components: {
    Child,
  },
  methods: {
    receive(msg) { // 接受子组件的信息，并将其赋值给messageFromChild
      this.messageFromChild = msg
    },
  },
}
</script>
```

## 效果预览

---

**this is parent component**

收到来自子组件的消息：

**this is child component**

收到来自父组件的消息：

@稀土掘金技术社区

## 2. v-slot前端开发博客

### 简介

v-slot是 Vue2.6 版本中新增的用于统一实现插槽和具名插槽的api，用于替代 `slot(2.6.0废弃)`、`slot-scope(2.6.0废弃)`、`scope(2.5.0废弃)` 等api。

v-slot在 template 标签中用于提供具名插槽或需要接收 prop 的插槽，如果不指定 v-slot，则取默认值 default。

### 代码实例

下面请看v-slot的代码实例，在这个实例中我们实现了：

父向子传值：父组件通过 `<template v-slot:child>{{ message }}</template>` 将父组件的message值传递给子组件，子组件通过 `<slot name="child"></slot>` 接收到相应内容，实现了父向子传值。

```
// 子组件代码
<template>
  <div class="child">
    <h4>this is child component</h4>
    <p>收到来自父组件的消息 :
      <slot name="child"></slot>  <!-- 展示父组件通过插槽传递的{{message}}-->
    </p>
  </div>
</template>

<template>
  <div class="parent">
    <h3>this is parent component</h3>
    <input type="text" v-model="message" />
    <Child>
      <template v-slot:child>
        {{ message }}  <!-- 插槽要展示的内容-->
      </template>
    </Child>
  </div>
</template>
<script>
import Child from './child'
export default {
  name: 'Parent',
  data() {
    return {
      message: '',
    }
  },
  components: {
    Child,
  },
}
</script>
```

## 效果预览

---

**this is parent component**

I

**this is child component**

收到来自父组件的消息：

@稀土掘金技术社区

### 3. *refs/parent/children/root*

#### 简介

我们也同样可以通过 `$refs/$parent/$children/$root` 等方式获取 Vue 组件实例，得到实例上绑定的属性及方法等，来实现组件之间的通信。

**\$refs**：我们通常会将 `refs` 绑定在子组件上，从而获取子组件实例。

`parent` 来获取当前组件的父组件实例（如果有的话）。

`parent`：我们可以在 Vue 中直接通过 `this`。

`children` 来获取当前组件的子组件实例的数组。但是需要注意的是，`this.$children` 数组中的元素下标并不一定对用父组件引用的子组件的顺序，例如有异步加载的子组件，可能影响其在 `children` 数组中的顺序。所以使用时需要根据一定的条件例如子组件的 `name` 去找到相应的子组件。

**\$root**：获取当前组件树的根 Vue 实例。如果当前实例没有父实例，此实例将会是其自己。通过 `$root`，我们可以实现组件之间的跨级通信。

## 代码实例

---

下面来看一个 `parent` 和 `children` 使用的实例（由于这几个api的使用方式大同小异，所以关于 `refs` 和 `root` 的使用就不在这里展开了，在这个实例中实现了：

- 父向子传值：子组件通过 `$parent.message` 获取到父组件中message的值。
- 子向父传值：父组件通过 `$children` 获取子组件实例的数组，在通过对数组进行遍历，通过实例的 `name` 获取到对应 Child1 子组件实例将其赋值给 `child1`，然后通过 `child1.message` 获取到 Child1 子组件的message。

代码如下：

```
// 子组件
<template>
  <div class="child">
    <h4>this is child component</h4>
    <input type="text" v-model="message" />
    <p>收到来自父组件的消息：{{ $parent.message }}</p>  <!--展示父组件实例的message-->
  </div>
</template>
<script>
export default {
  name: 'Child1',
  data() {
    return {
      message: '',    // 父组件通过this.$children可以获取子组件实例的message
    }
  },
}
</script>
```

```

// 父组件
<template>
  <div class="parent">
    <h3>this is parent component</h3>
    <input type="text" v-model="message" />
    <p>收到来自子组件的消息：{{ child1.message }}</p> <!--展示子组件实例的message-->
    <Child />
  </div>
</template>
<script>
import Child from './child'
export default {
  name: 'Parent',
  data() {
    return {
      message: '',
      child1: {},
    }
  },
  components: {
    Child,
  },
  mounted() {
    this.child1 = this.$children.find((child) => {
      return child.$options.name === 'Child1' // 通过options.name获取对应name的
    })
  },
}
</script>

```

## 效果预览

---



**this is parent component**

收到来自子组件的消息:

**this is child component**

收到来自父组件的消息:

@稀土掘金技术社区

#### 4. `attrs` / `listener` [前端开发博客](#)

##### 简介

`attrs` 和 `listeners` 都是 Vue2.4 中新增加的属性，主要是用来供使用者用来开发高级组件的。

**`$attrs`**：用来接收父作用域中不作为 `prop` 被识别的 `attribute` 属性，并且可以通过 `v-bind="$attrs"` 传入内部组件——在创建高级别的组件时非常有用。

试想一下，当你创建了一个组件，你要接收 `param1`、`param2`、`param3` ..... 等数十个参数，如果通过 `props`，那你需要通过 `props: ['param1', 'param2', 'param3', .....]` 等声明一大堆。如果这些 `props` 还有一些需要往更深层次的子组件传递，那将会更加麻烦。

而使用 **`$attrs`**，你不需要任何声明，直接通过 **`$attrs.param1`**、**`$attrs.param2`** .....就可以使用，而且向深层子组件传递上面也给了示例，十分方便。

`$listeners` : 包含了父作用域中的 v-on 事件监听器。它可以通过 `v-on="$listeners"` 传入内部组件——在创建更高层次的组件时非常有用，这里在传递时的使用方法和 `$attrs` 十分类似。

## 代码实例

在这个实例中，共有三个组件：A、B、C，其关系为：`[ A [ B [ C ] ] ]`，A为B的父组件，B为C的父组件。即：1级组件A，2级组件B，3级组件C。我们实现了：

- 父向子传值：1级组件A通过 `:messageFromA="message"` 将 `message` 属性传递给2级组件B，2级组件B通过 `$attrs.messageFromA` 获取到1级组件A的 `message`。
- 跨级向下传值：1级组件A通过 `:messageFromA="message"` 将 `message` 属性传递给2级组件B，2级组件B再通过 `v-bind="$attrs"` 将其传递给3级组件C，3级组件C通过 `$attrs.messageFromA` 获取到1级组件A的 `message`。
- 子向父传值：1级组件A通过 `@keyup="receive"` 在子孙组件上绑定 `keyup` 事件的监听，2级组件B在通过 `v-on="$listeners"` 来将 `keyup` 事件绑定在其 `input` 标签上。当2级组件B `input` 输入框输入时，便会触发1级组件A的 `receive` 回调，将2级组件B的 `input` 输入框中的值赋值给1级组件A的 `messageFromComp`，从而实现子向父传值。
- 跨级向上传值：1级组件A通过 `@keyup="receive"` 在子孙组件上绑定 `keyup` 事件的监听，2级组件B在通过 `<CompC v-on="$listeners" />` 将其继续传递给C。3级组件C在通过 `v-on="$listeners"` 来将 `keyup` 事件绑定在其 `input` 标签上。当3级组件C `input` 输入框输入时，便会触发1级组件A的 `receive` 回调，将3级组件C的 `input` 输入框中的值赋值给1级组件A的 `messageFromComp`，从而实现跨级向上传值。

代码如下：

```
// 3级组件C
<template>
  <div class="compC">
    <h5>this is C component</h5>
    <input name="compC" type="text" v-model="message" v-on="$listeners" /> <!--将A
组件keyup的监听回调绑在该input上-->
    <p>收到来自A组件的消息：{{ $attrs.messageFromA }}</p>
  </div>
</template>
<script>
export default {
  name: 'CompC',
  data() {
    return {
      message: '',
    }
  },
}
</script>
```

```

// 2级组件B
<template>
  <div class="compb">
    <h4>this is B component</h4>
    <input name="compB" type="text" v-model="message" v-on="$listeners" /> <!--将A
组件keyup的监听回调绑在该input上-->
    <p>收到来自A组件的消息：{{ $attrs.messageFromA }}</p>
    <CompC v-bind="$attrs" v-on="$listeners" /> <!--将A组件keyup的监听回调继续传递给C组
件，将A组件传递的attrs继续传递给C组件-->
  </div>
</template>
<script>
import CompC from './compC'
export default {
  name: 'CompB',
  components: {
    CompC,
  },
  data() {
    return {
      message: '',
    }
  },
}
</script>

```

```

// A组件
<template>
  <div class="compa">
    <h3>this is A component</h3>
    <input type="text" v-model="message" />
    <p>收到来自{{ comp }}的消息: {{ messageFromComp }}</p>
    <CompB :messageFromA="message" @keyup="receive" />  <!-- 监听子孙组件的keyup事件，
将message传递给子孙组件 -->
  </div>
</template>
<script>
import CompB from './compB'
export default {
  name: 'CompA',
  data() {
    return {
      message: '',
      messageFromComp: '',
      comp: '',
    }
  },
  components: {
    CompB,
  },
  methods: {
    receive(e) { // 监听子孙组件keyup事件的回调，并将keyup所在input输入框的值赋值给
messageFromComp
      this.comp = e.target.name
      this.messageFromComp = e.target.value
    },
  },
}
</script>

```

## 效果预览

---

**this is A component**

传递  I

收到来自的消息：

**this is B component**

收到来自A组件的消息：传递

**this is C component**

收到来自A组件的消息：传递

@稀土掘金技术社区

## 5. provide/inject [前端开发博客](#)

### 简介

provide/inject这对选项需要一起使用，以允许一个祖先组件向其所有子孙后代注入一个依赖，不论组件层次有多深，并在其上下游关系成立的时间里始终生效。如果你是熟悉React的同学，你一定会立刻想到Context这个api，二者是十分相似的。

**provide**：是一个对象，或者是一个返回对象的函数。该对象包含可注入其子孙的property，即要传递给子孙的属性和属性值。

**inject**：一个字符串数组，或者是一个对象。当其为字符串数组时，使用方式和props十分相似，只不过接收的属性由data变成了provide中的属性。当其为对象时，也和props类似，可以通过配置default和from等属性来设置默认值，在子组件中使用新的命名属性等。

## 代码实例

---

这个实例中有三个组件，1级组件A，2级组件B，3级组件C：`[ A [ B [ C ] ] ]`，A是B的父组件，B是C的父组件。实例中实现了：

- 父向子传值：1级组件A通过provide将message注入给子孙组件，2级组件B通过`inject: ['messageFromA']`来接收1级组件A中的message，并通过`messageFromA.content`获取1级组件A中message的content属性值。
- 跨级向下传值：1级组件A通过provide将message注入给子孙组件，3级组件C通过`inject: ['messageFromA']`来接收1级组件A中的message，并通过`messageFromA.content`获取1级组件A中message的content属性值，实现跨级向下传值。

代码如下：

```

// 1级组件A
<template>
  <div class="compa">
    <h3>this is A component</h3>
    <input type="text" v-model="message.content" />
    <CompB />
  </div>
</template>
<script>
import CompB from './compB'
export default {
  name: 'CompA',
  provide() {
    return {
      messageFromA: this.message, // 将message通过provide传递给子孙组件
    }
  },
  data() {
    return {
      message: {
        content: '',
      },
    }
  },
  components: {
    CompB,
  },
}
</script>

// 2级组件B
<template>
  <div class="compb">
    <h4>this is B component</h4>
    <p>收到来自A组件的消息：{{ messageFromA && messageFromA.content }}</p>
    <CompC />
  </div>
</template>
<script>
import CompC from './compC'
export default {
  name: 'CompB',
  inject: ['messageFromA'], // 通过inject接受A中provide传递过来的message
  components: {
    CompC,
  },
}
</script>

```

```
// 3级组件C
<template>
  <div class="compc">
    <h5>this is C component</h5>
    <p>收到来自A组件的消息：{{ messageFromA && messageFromA.content }}</p>
  </div>
</template>
<script>
export default {
  name: 'Compc',
  inject: ['messageFromA'], // 通过inject接受A中provide传递过来的message
}
</script>
```

注意点：

1. 可能有同学想问我上面1级组件A中的message为什么要用object类型而不是string类型，因为在vue provide 和 inject 绑定并不是可响应的。如果message是string类型，在1级组件A中通过input输入框改变message值后无法再赋值给messageFromA，如果是object类型，当对象属性值改变后，messageFromA里面的属性值还是可以随之改变的，子孙组件inject接收到的对象属性值也可以相应变化。
2. 子孙provide和祖先同样的属性，会在后代中覆盖祖先的provide值。例如2级组件B中也通过provide向3级组件C中注入一个messageFromA的值，则3级组件C中的messageFromA会优先接收2级组件B注入的值而不是1级组件A。

## 效果预览

---



**this is A component**

provide

**this is B component**

收到来自A组件的消息: provide

**this is C component**

收到来自A组件的消息: provide

@稀土掘金技术社区

## 6. EventBus

### 简介

eventBus又称事件总线，通过注册一个新的Vue实例，通过调用这个实例的 *emit* 和 *on* 等来监听和触发这个实例的事件，通过传入参数从而实现组件的全局通信。它是一个不具备 DOM 的组件，有的仅仅只是它实例方法而已，因此非常的轻便。

我们可以通过在全局Vue实例上注册:

```
// main.js
Vue.prototype.$Bus = new Vue()
```

但是当项目过大时，我们最好将事件总线抽象为单个文件,将其导入到需要使用的每个组件文件中。这样,它不会污染全局命名空间：

```
// bus.js, 使用时通过import引入
import Vue from 'vue'
export const Bus = new Vue()
```

## 原理分析

---

eventBus的原理其实比较简单，就是使用订阅-发布模式，实现 *emit* 和 *on* 两个方法即可：

```
// eventBus原理
export default class Bus {
  constructor() {
    this.callbacks = {}
  }
  $on(event, fn) {
    this.callbacks[event] = this.callbacks[event] || []
    this.callbacks[event].push(fn)
  }
  $emit(event, args) {
    this.callbacks[event].forEach((fn) => {
      fn(args)
    })
  }
}

// 在main.js中引入以下
// Vue.prototype.$bus = new Bus()
```

## 代码实例

---

在这个实例中，共包含了4个组件:[ A [ B [ C、D ] ] ]，1级组件A，2级组件B，3级组件C和3级组件D。我们通过使用eventBus实现了：

全局通信：即包括了父子组件相互通信、兄弟组件相互通信、跨级组件相互通信。4个组件的操作逻辑相同，都是在input输入框时，通过 `this.$bus.$emit('sendMessage', obj)` 触发sendMessage事件回调，将sender和message封装成对象作为参数传入；同时通过 `this.$bus.$on('sendMessage', obj)` 监听其他组件的sendMessage事件，实例当前组件示例sender和message的值。这样任一组件input输入框值改变时，其他组件都能接收到相应的信息，实现全局通信。

代码如下：

```
// main.js
Vue.prototype.$bus = new Vue()
```

```

// 1级组件A
<template>
  <div class="containerA">
    <h2>this is CompA</h2>
    <input type="text" v-model="message" @keyup="sendMessage" />
    <p v-show="messageFromBus && sender !== $options.name">
      收到{{ sender }}的消息: {{ messageFromBus }}
    </p>
    <CompB />
  </div>
</template>
<script>
import CompB from './compB'
export default {
  name: 'CompA',
  components: {
    CompB,
  },
  data() {
    return {
      message: '',
      messageFromBus: '',
      sender: '',
    }
  },
  mounted() {
    this.$bus.$on('sendMessage', (obj) => { // 通过eventBus监听sendMessage事件
      const { sender, message } = obj
      this.sender = sender
      this.messageFromBus = message
    })
  },
  methods: {
    sendMessage() {
      this.$bus.$emit('sendMessage', { // 通过eventBus触发sendMessage事件
        sender: this.$options.name,
        message: this.message,
      })
    },
  },
}
</script>

```

```

// 2级组件B
<template>
  <div class="containerB">
    <h3>this is CompB</h3>
    <input type="text" v-model="message" @keyup="sendMessage" />
    <p v-show="messageFromBus && sender !== $options.name">
      收到{{ sender }}的消息: {{ messageFromBus }}
    </p>
    <CompC />
    <CompD />
  </div>
</template>
<script>
import CompC from './compC'
import CompD from './compD'
export default {
  name: 'CompB',
  components: {
    CompC,
    CompD,
  },
  data() {
    return {
      message: '',
      messageFromBus: '',
      sender: '',
    }
  },
  mounted() {
    this.$bus.$on('sendMessage', (obj) => { // 通过eventBus监听sendMessage事件
      const { sender, message } = obj
      this.sender = sender
      this.messageFromBus = message
    })
  },
  methods: {
    sendMessage() {
      this.$bus.$emit('sendMessage', { // 通过eventBus触发sendMessage事件
        sender: this.$options.name,
        message: this.message,
      })
    },
  },
}
</script>

```

```

// 3级组件C
<template>
  <div class="containerC">
    <p>this is CompC</p>
    <input type="text" v-model="message" @keyup="sendMessage" />
    <p v-show="messageFromBus && sender !== $options.name">
      收到{{ sender }}的消息: {{ messageFromBus }}
    </p>
  </div>
</template>
<script>
export default {
  name: 'CompC',
  data() {
    return {
      message: '',
      messageFromBus: '',
      sender: ''
    }
  },
  mounted() {
    this.$bus.$on('sendMessage', (obj) => { // 通过eventBus监听sendMessage事件
      const { sender, message } = obj
      this.sender = sender
      this.messageFromBus = message
    })
  },
  methods: {
    sendMessage() {
      this.$bus.$emit('sendMessage', { // 通过eventBus触发sendMessage事件
        sender: this.$options.name,
        message: this.message,
      })
    }
  },
}
</script>

```

```
// 3级组件D
<template>
  <div class="containerD">
    <p>this is CompD</p>
    <input type="text" v-model="message" @keyup="sendMessage" />
    <p v-show="messageFromBus && sender !== $options.name">
      收到{{ sender }}的消息: {{ messageFromBus }}
    </p>
  </div>
</template>
<script>
export default {
  name: 'CompD',
  data() {
    return {
      message: '',
      messageFromBus: '',
      sender: ''
    }
  },
  mounted() {
    this.$bus.$on('sendMessage', (obj) => { // 通过eventBus监听sendMessage事件
      const { sender, message } = obj
      this.sender = sender
      this.messageFromBus = message
    })
  },
  methods: {
    sendMessage() {
      this.$bus.$emit('sendMessage', { // 通过eventBus触发sendMessage事件
        sender: this.$options.name,
        message: this.message,
      })
    }
  }
}
</script>
```

## 效果预览

---



## 7. Vuex

当项目庞大以后，在多人维护同一个项目时，如果使用事件总线进行全局通信，容易让全局的变量的变化难以预测。于是有了Vuex的诞生。

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

有关Vuex的内容，可以参考Vuex官方文档<sup>[1]</sup>，我就不在这里班门弄斧了，直接看代码。

### 代码实例

Vuex的实例和事件总线leisi，同样是包含了4个组件:[ A [ B [ C、D ] ] ]，1级组件A，2级组件B，3级组件C和3级组件D。我们在这个实例中实现了：

全局通信：代码的内容和eventBus也类似，不过要比eventBus使用方便很多。每个组件通过watch监听input输入框的变化，把input的值通过vuex的commit触发mutations，从而改变stroe的值。然后每个组件都通过computed动态获取store中的数据，从而实现全局通信。

```
// store.js
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex)
export default new Vuex.Store({
  state: {
    message: {
      sender: '',
      content: '',
    },
  },
  mutations: {
    sendMessage(state, obj) {
      state.message = {
        sender: obj.sender,
        content: obj.content,
      }
    },
  },
})
```



```

// 组件A
<template>
  <div class="containerA">
    <h2>this is CompA</h2>
    <input type="text" v-model="message" />
    <p v-show="messageFromStore && sender !== $options.name">
      收到{{ sender }}的消息: {{ messageFromStore }}
    </p>
    <CompB />
  </div>
</template>
<script>
import CompB from './compB'
export default {
  name: 'CompA',
  components: {
    CompB,
  },
  data() {
    return {
      message: '',
    }
  },
  computed: {
    messageFromStore() {
      return this.$store.state.message.content
    },
    sender() {
      return this.$store.state.message.sender
    },
  },
  watch: {
    message(newValue) {
      this.$store.commit('sendMessage', {
        sender: this.$options.name,
        content: newValue,
      })
    },
  },
}
</script>

```

同样和eventBus中一样，B，C，D组件中的代码除了引入子组件的不同，script部分都是一样的，就不再往上写了。

## 效果预览

---

# this is CompA

vuextong I

tong

1. 同 2. 通 3. 痛 4. 桶 5. tongue

< > v



## this is CompB

收到CompA的消息: vuex

## this is CompC

收到CompA的消息: vuex

## this is CompD

收到CompA的消息: vuex

@稀土掘金技术社区

## 总结 前端开发博客

上面总共提到了7种Vue的组件通信方式，他们能够进行的通信种类如下图所示：



- `props/$emit`：可以实现父子组件的双向通信，在日常的父子组件通信中一般会作为我们的最常用选择。
- `v-slot`：可以实现父子组件单向通信（父向子传值），在实现可复用组件，向组件中传入DOM节点、html等内容以及某些组件库的表格值二次处理等情况时，可以优先考虑v-slot。
- `refs/parent/children/root`：可以实现父子组件双向通信，其中\$root可以实现根组件实例向子孙组件跨级单向传值。在父组件没有传递值或通过v-on绑定监听时，父子间想要获取彼此的属性或方法可以考虑使用这些api。
- `attrs/listeners`：能够实现跨级双向通信，能够让你简单的获取传入的属性和绑定的监听，并且方便地向下级子组件传递，在构建高级组件时十分好用。
- `provide/inject`：可以实现跨级单向通信，轻量地向子孙组件注入依赖，这是你在实现高级组件、创建组件库时的不二之选。
- `eventBus`：可以实现全局通信，在项目规模不大的情况下，可以利用eventBus实现全局的事件监听。但是eventBus要慎用，避免全局污染和内存泄漏等情况。
- `Vuex`：可以实现全局通信，是vue项目全局状态管理的最佳实践。在项目比较庞大，想要集中式管理全局组件状态时，那么安装Vuex准没错！

最后，鲁迅说过：“一碗酸辣汤，耳闻口讲的，总不如亲自呷一口的明白。”

(鲁迅：这句话我真说过！)

看了这么多，不如自己亲手去敲一敲更能理解，看完可以去手动敲一敲加深理解。

(文本完)



每日分享前端插件干货，欢迎关注！



**前端新世界**

分享 JS / CSS 技术教程；Vue、React、jQuery等前端开发组件

543篇原创内容

公众号

点赞和在看就是最大的支持❤️