组件的通信 2：派发与广播——自行实现 dispatch 和 broadcast 方法

上一讲的 provide / inject API 主要解决了跨级组件间的通信问题，不过它的使用场景，主要是子组件获取上级组件的状态，跨级组件间建立了一种主动提供与依赖注入的关系。然后有两种场景它不能很好的解决：

- 父组件向子组件（支持跨级）传递数据；
- 子组件向父组件（支持跨级）传递数据。

这种父子（含跨级）传递数据的通信方式，Vue.js 并没有提供原生的 API 来支持，而是推荐使用大型数据状态管理工具 Vuex，而我们之前已经介绍过 Vuex 的场景与在独立组件（或库）中使用的限制。本小节则介绍一种在父子组件间通信的方法 dispatch 和 broadcast。

$on 与 $emit

如果您使用过较早的 Vue.js 1.x 版本，肯定对 *$dispatch* 和 *$broadcast* 这两个内置的方法很熟悉，不过它们都在 Vue.js 2.x 里废弃了。在正式介绍主角前，我们先看看 $on 与 $emit 这两个 API，因为它们是本节内容的基础。

$emit 会在**当前组件**实例上触发自定义事件，并传递一些参数给监听器的回调，一般来说，都是在父级调用这个组件时，使用 @on 的方式来监听自定义事件的，比如在子组件中触发事件：

```
// child.vue，部分代码省略
export default {
  methods: {
    handleEmitEvent () {
      this.$emit('test', 'Hello Vue.js');
    }
  }
}
```

在父组件中监听由 *child.vue* 触发的自定义事件 **test**：

```
<!-- parent.vue，部分代码省略-->
<template>
  <child-component @test="handleEvent">
</template>
<script>
  export default {
    methods: {
      handleEvent (text) {
        console.log(text);  // Hello Vue.js
      }
    }
  }
</script>
```

这里看似是在父组件 *parent.vue* 中绑定的自定义事件 **test** 的处理句柄，然而事件 test 并不是在父组件上触发的，而是在子组件 *child.vue* 里触发的，只是通过 v-on 在父组件中监听。既然是子组件自己触发的，那它自己也可以监听到，这就要使用 $on 来监听实例上的事件，换言之，组件使用 $emit 在自己实例上触发事件，并用 $on 监听它。

听起来这种神（sāo）操作有点多此一举，我们不妨先来看个示例：

（也可通过在线链接 https://run.iviewui.com/ggsomfHM (https://run.iviewui.com/ggsomfHM) 直接运行该示例）

```
<template>
  <div>
    <button @click="handleEmitEvent">触发自定义事件
</button>
  </div>
</template>
<script>
  export default {
    methods: {
      handleEmitEvent () {
        // 在当前组件上触发自定义事件 test，并传值
        this.$emit('test', 'Hello Vue.js')
      }
    },
    mounted () {
      // 监听自定义事件 test
      this.$on('test', (text) => {
        window.alert(text);
      });
    }
  }
</script>
```

$on 监听了自己触发的自定义事件 test，因为有时不确定何时会触发事件，一般会在 mounted 或 created 钩子中来监听。

仅上面的示例，的确是多此一举的，因为大可在 handleEmitEvent 里直接写 window.alert(text)，没必要绕一圈。

之所以多此一举，是因为 handleEmitEvent 是当前组件内的 <button> 调用的，如果这个方法不是它自己调用，而是其它组件调用的，那这个用法就大有可为了。

了解了 $on 和 $emit 的用法后，我们再来看两个“过时的” API。

Vue.js 1.x 的 $dispatch 与 $broadcast

虽然 Vue.js 1.x 已经成为过去时，但为了充分理解本节通信方法的使用场景，还是有必要来了解一点它的历史。

在 Vue.js 1.x 中，提供了两个方法：$dispatch 和 $broadcast，前者用于向上级派发事件，只要是它的父级（一级或多级以上），都可以在组件内通过 $on （或 events，2.x 已废弃）监听到，后者相反，是由上级向下级广播事件的。

来看一个简单的示例：

```
<!-- 注意：该示例为 Vue.js 1.x 版本 -->
<!-- 子组件 -->
<template>
  <button @click="handleDispatch">派发事件</button>
</template>
<script>
export default {
  methods: {
    handleDispatch () {
      this.$dispatch('test', 'Hello, Vue.js');
    }
  }
}
</script>
```

```
<!-- 父组件，部分代码省略 -->
<template>
  <child-component></child-component>
</template>
<script>
  export default {
    mounted () {
      this.$on('test', (text) => {
        console.log(text);  // Hello, Vue.js
      });
    }
  }
</script>
```

$broadcast 类似，只不过方向相反。这两种方法一旦发出事件后，任何组件都是可以接收到的，就近原则，而且会在第一次接收到后停止冒泡，除非返回 true。

这两个方法虽然看起来很好用，但是在 Vue.js 2.x 中都废弃了，官方给出的解释是：

> 因为基于组件树结构的事件流方式有时让人难以理解，
> 并且在组件结构扩展的过程中会变得越来越脆弱。

虽然在业务开发中，它没有 Vuex 这样专门管理状态的插件清晰好用，但对独立组件（库）的开发，绝对是福音。因为独立组件一般层级并不会很复杂，并且剥离了业务，不会变的难以维护。

知道了 *$dispatch* 和 *$broadcast* 的前世今生，接下来我们就在 Vue.js 2.x 中自行实现这两个方法。

自行实现 dispatch 和 broadcast 方法

自行实现的 dispatch 和 broadcast 方法，不能保证跟 Vue.js 1.x 的 *$dispatch* 和 *$broadcast* 具有完全相同的体验，但基本功能是一样的，都是解决父子组件（含跨级）间的通信问题。

通过目前已知的信息，我们要实现的 dispatch 和 broadcast 方法，将具有以下功能：

- 在子组件调用 dispatch 方法，向上级指定的组件实例（最近的）上触发自定义事件，并传递数据，且该上级组件已预先通过 $on 监听了这个事件；
- 相反，在父组件调用 broadcast 方法，向下级指定的组件实例（最近的）上触发自定义事件，并传递数据，且该下级组件已预先通过 $on 监听了这个事件。

实现这对方法的关键点在于，如何正确地向上或向下找到对应的组件实例，并在它上面触发方法。在设计一个新功能（features）时，可以先确定这个功能的 API 是什么，也就是说方法名、参数、使用样例，确定好 API，再来写具体的代码。

因为 Vue.js 内置的方法，才是以 $ 开头的，比如 `$nextTick`、`$emit` 等，为了避免不必要的冲突并遵循规范，这里的 dispatch 和 broadcast 方法名前不加 $。并且该方法可能在很多组件中都会使用，复用起见，我们封装在混合（mixins）里。那它的使用样例可能是这样的：

```
// 部分代码省略
import Emitter from '../mixins/emitter.js'

export default {
  mixins: [ Emitter ],
  methods: {
    handleDispatch () {
      this.dispatch();  // ①
    },
    handleBroadcast () {
      this.broadcast();  // ②
    }
  }
}
```

上例中行 ① 和行 ② 的两个方法就是在导入的混合 **emitter.js** 中定义的，这个稍后我们再讲，先来分析这两个方法应该传入什么参数。一般来说，为了跟 Vue.js 1.x 的方法一致，第一个参数应当是自定义事件名，比如 “test”，第二个参数是传递的数据，比如 “Hello, Vue.js”，但在这里，有什么问题呢？只通过这两个参数，我们没办法知道要在哪个组件上触发事件，因为自行实现的这对方法，与 Vue.js 1.x 的原生方法机理上是有区别的。上文说到，实现这对方法的关键点在于准确地**找到组件实例**。那在寻找组件实例上，我们的“惯用伎俩”就是通过遍历来匹配组件的 name 选项，在独立组件（库）里，每个组件的 name 值应当是唯一的，name 主要用于递归组件，在后面小节会单独介绍。

先来看下 **emitter.js** 的代码：

```
function broadcast(componentName, eventName,
params) {
  this.$children.forEach(child => {
    const name = child.$options.name;
```

```
    if (name === componentName) {
      child.$emit.apply(child,
[eventName].concat(params));
    } else {
      broadcast.apply(child, [componentName,
eventName].concat([params]));
    }
  });
}
export default {
  methods: {
    dispatch(componentName, eventName, params) {
      let parent = this.$parent || this.$root;
      let name = parent.$options.name;

      while (parent && (!name || name !==
componentName)) {
        parent = parent.$parent;

        if (parent) {
          name = parent.$options.name;
        }
      }
      if (parent) {
        parent.$emit.apply(parent,
[eventName].concat(params));
      }
    },
    broadcast(componentName, eventName, params) {
      broadcast.call(this, componentName,
eventName, params);
    }
```

```
  }
};
```

因为是用作 mixins 导入，所以在 methods 里定义的 dispatch 和 broadcast 方法会被混合到组件里，自然就可以用 `this.dispatch` 和 `this.broadcast` 来使用。

这两个方法都接收了三个参数，第一个是组件的 name 值，用于向上或向下递归遍历来寻找对应的组件，第二个和第三个就是上文分析的自定义事件名称和要传递的数据。

可以看到，在 dispatch 里，通过 *while* 语句，不断向上遍历更新当前组件（即上下文为当前调用该方法的组件）的父组件实例（变量 parent 即为父组件实例），直到匹配到定义的 componentName 与某个上级组件的 name 选项一致时，结束循环，并在找到的组件实例上，调用 `$emit` 方法来触发自定义事件 eventName。broadcast 方法与之类似，只不过是向下遍历寻找。

来看一下具体的使用方法。有 **A.vue** 和 **B.vue** 两个组件，其中 B 是 A 的子组件，中间可能跨多级，在 A 中向 B 通信：

```
<!-- A.vue -->
<template>
    <button @click="handleClick">触发事件</button>
</template>
<script>
  import Emitter from '../mixins/emitter.js';

  export default {
    name: 'componentA',
    mixins: [ Emitter ],
    methods: {
      handleClick () {
        this.broadcast('componentB', 'on-
message', 'Hello Vue.js');
      }
    }
  }
</script>
```

```
// B.vue
export default {
  name: 'componentB',
  created () {
    this.$on('on-message', this.showMessage);
  },
  methods: {
    showMessage (text) {
      window.alert(text);
    }
  }
}
```

同理，如果是 B 向 A 通信，在 B 中调用 dispatch 方法，在 A 中使用 $on 监听事件即可。

以上就是自行实现的 dispatch 和 broadcast 方法，相比 Vue.js 1.x，有以下不同：

- 需要额外传入组件的 name 作为第一个参数；
- 无冒泡机制；
- 第三个参数传递的数据，只能是一个（较多时可以传入一个对象），而 Vue.js 1.x 可以传入多个参数，当然，你对 emitter.js 稍作修改，也能支持传入多个参数，只是一般场景传入一个对象足以。

结语

Vue.js 的组件通信到此还没完全结束，如果你想“趁热打铁”一口气看完，可以先阅读第 6 节组件的通信 3。亦或按顺序看下一节的实战，来进一步加深理解 provide / inject 和 dispatch / broadcast 这两对通信方法的使用场景。

注：本节部分代码参考 iView (https://github.com/iview/iview/blob/2.0/src/mixins/emitter