```
$ hexo n vue-skill component-communication
```

前言

组件化也是vue的一大特点之一，组件是组成vue项目的基本单位。

一个组件可以做什么？ 这取决于组件的设计，组件一般由以下几点组成：

- 选项
- 实例属性/方法
- 生命周期
- 全局API
- 指令
- 导入的其他组件

组件是可复用的vue实例，可以复用，导入导出，组成组件化系统。

vue的组件都是保持相对独立的关系，可是又可以进行相互依赖（导入导出机制）；

这种规范下，实际开发又难免会出现组件之间**传递数据**、**转发事件**的场景。

而vue又对组件间的数据、事件做了一定的限制；

正因如此，我们需要好好探讨vue的组件通讯。

<!-- more -->

父子组件通讯

由于组件通讯，有时候是为了**传递数据**，而有时候是为了**转发事件**；

这里不对这两种场景做区分，统一要解决的场景为“组件通讯”。

其实vue组件通讯并不复杂，官方文档提供了几个方法。

- props 向子组件传递数据 传松门
- emit 向父组件抛出事件 传送门
- v-model 父子组件数据同步（带有一定的抛出事件机制） 传送门
- props+sync修饰符 传松门
- slot-scope 作用域插槽 传送门

由于官方文档介绍得很详细，也提供了**传送门**，具体实现并不详细介绍。

不过vue组件有一个设计概念还是有大家了解下的——单向数据流 传送门

单向数据流

> 所有的 prop 都使得其父子 prop 之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。额外的，每次父级组件发生更新时，子组

件中所有的 prop 都将会刷新为最新的值。这意味着你不应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器的控制台中发出警告。

而如果需要改变父组件的数据，则子组件抛出事件；父组件定义自定义事件，在自定义事件中改变自己的数据。

这些方法有所缺点，不同的业务场景需要开发者自行衡量；

不过适应的业务场景都**局限于父子组件的通讯**。

其实父子组件通讯的是最好解决的，毕竟有直接联系。

非父子组件通讯

由于非父子组件没有直接联系，只能使用间接联系。

两个组件使用相同的“代理“，由”代理“转发数据或者事件的交互。

”代理“各式各样，也有不同的实现方式。

备注：这里的“代理”是中转站的意思，为了方便理解，表述为：“代理”。

官方也提供几种解决方案：

- $root.data 传送门
- vuex 传送门
- vue-router 传送门

$root.data

我们知晓：一个vue项目由组件为单位组成。

但是，一个vue项目只有一个根组件。

且所有组件实例均可访问根组件实例this.$root。

官方文档也提供了相应的实现方式传送门。

实际开发中，并不使用这种方式进行组件通讯。

我们并不希望$root.data 挂载庞大的数据变量。

我们仅希望他只负责渲染HTML DOM元素。

vuex

vuex专用于vue项目，作为**状态管理模式**插件。（理解为集中存储全局变量的地方就好了。）

前面说到“单向数据流”理念，并不适用**多个组件共享状态**场景。

> 因此，我们为什么不把组件的共享状态抽取出来，以一个全局单例模式管理呢？在这种模式下，我们的组件树构成了一个巨大的“视图”，不管在树的哪个位置，任何组件都能获取状态或者触发行为！

通过定义和隔离状态管理中的各种概念并通过强制规则维持视图和状态间的独立性，我们的代码将会变得更结构化且易维护。

vuex的核心是store（仓库），"store"是一个容器，包含应用中大部分的状态(state)，与单纯的全局对象有两点不同：

- Vuex 的状态存储是响应式的。
- 不能直接改变 store 中的状态，若想改变，唯一途径就是显式地提交 (commit) mutation。

一个store可以包含

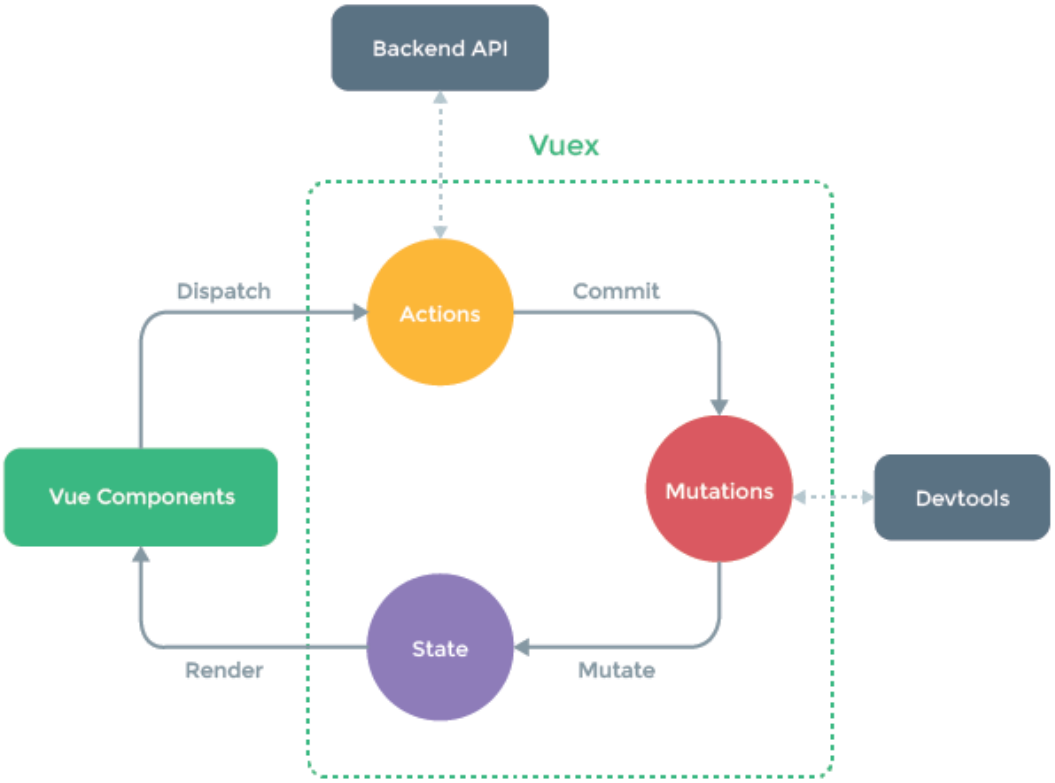
state：储存状态　　单一状态树

aciton：　提交mutation方法，可执行异步操作

mutation：更改store状态，必须是同步操作

getter：从state派生状态，返回值会根据依赖缓存

modules：将store分割成模块



简单分析下这张图：

1. State 直接渲染在vue组件
2. 组件可以可以通过Dispatch触发 Actions

3. Actions可以Commit触发Mutations
4. Mutations可以Mutate更改State
5. vue组件计算属性更新State的值

还有几个注意点：

- vuex的范围：组件外部的 State => Actions => Mutations => State
- Actions、Mutations： Actions可以执行异步（一般用于后端Api交互）、Commit Mutations；Mutations必须是同步，（一般用于更改State状态）
- 执行顺序：组件也可以跳过Actions 直接 Commit Mutations， 不过执行顺序是不可逆的。
- Actions是可以Dispatch 其他Actions的，同理一个Actions也可以Commit 多个Mutations

vue-router

vue-router也可以拥有组件通讯的能力，不过这些组件针对的是直接挂载在路由的页面组件。

- 路由元信息 传送门
- 路由组件参数 传送门
- 导航守卫 传送门
- $route.query 传送门

总线模式（bus）

总线模式需要实例化一个空Vue实例，我们把非父子组件的事件统一集中到这个空Vue实例，让这个实例监听，触发这些自定事件就可以。

Vue 其实是一个构造函数，既然是构造函数，那就可以多次实例化。

而构造函数又有原型对象prototype，原型就是为了共享。

那第一步我们可以在Vue的原型对象上实例化一个空Vue实例。

```
Vue.prototype.bus = new Vue()
```

备注：如果是大规模使用，可以在main.js直接定义原型bus属性。局部使用可以包装成一个js模块，需要时引入即可。

再看实例事件

- vm.$on
- vm.$emit

我们可以知晓：$emit 会触发 $on 就可以了。

准备一个文件夹 bus，定义三个组件：

- bus.vue

- bus-child1.vue
- bus-child2.vue

我们要做的是让child1触发child2事件。

```
<!-- bus.vue 引入 child1, child2 -->
<bus-child1 content="skill-bus-child1"></bus-child1>
<bus-child2 content="skill-bus-child2"></bus-child2>
// bus-child1.vue
export default {
  // ...
    methods: {
        handleClick () {
            this.bus.$emit('on-change', this.selfContent)
        }
    }
}
// bus-child2.vue
export default {
  methods: {
    handleChange ($event) {
            this.selfContent = $event
        }
  },
  mounted () {
        this.bus.$on('on-change', this.handleChange)
  }
}
```

bus.vue 源码

然而总线模式还是有带来一定的作用域问题，由于非父子组件的事件统一集中到空Vue实例，这导致了这些被空vue实例监听的自定义事件，在所有的组件是共享的，这种情况下，很容易产生副作用。

若再增加一个bus-child3.vue，它也监听了this.bus.$on('on-change', this.handleChange)。

而其实我们并不想bus-child3.vue响应 bus-child1.vue事件。

说白了就是作用域太广，bus是全局作用域；

又或者说没有命令空间，相同的事件名不能在两个组件定义。

其实这种解决方案之于$root.dat是换汤不换药的，都是把不同组件的数据/事件集中到一个vue实例。

emitter (dispatch / broadcast)

如果两个组件，有相同的父组件，可以使用**事件派发与广播机制**处理非父子组件通讯。

其实都会有相同的父组件的，再不济最顶层就是vue的根实例了，不过一般不需要到根实例。

派发/广播机制很依赖组件层级关系。

派发(dispatch):由本组件向上派发事件，供上层组件监听处理。

广播(broadcast)：由本组件向下广播事件，供下层组件监听处理。

Element UI 框架源码的[emitter.js](emitter.js)**就是基于派发广播机制处理非父子组件的通讯。**

源码的emitter.js把其机制封装成了一个独立的mixin，以便在各个组件方便使用。

这里就以emitter.js为例子，简单讲解是如何实现的，还有如何使用这个mixin。

```
// 简化emitter.js，只提取出结构

function broadcast (componentName, eventName, params) {
  // ...
}



// 标准mixins结构
export default {
  methods: {
      // 定义dispatch方法，需要三个参数
      // @params componentName 派发给哪个上层组件
      // @params eventName        派发的事件名
      // @params params              事件携带参数
      dispatch (componentName, eventName, params) {},
      // 定义broadcast方法，需要三个参数
      // @params broadcast           广播给哪个下层组件
      // @params eventName        广播的事件名
      // @params params              事件携带参数
      broadcast (componentName, eventName, params) {}
  }
}
```

componentName 指的是定义组件时，组件的选项name 传送门

```
export default {
  name: 'component-name'
}
```

跟总线模式一样，我们也要定义一个文件夹——emitter，三个组件

1. emitter.vue
2. child1.vue
3. child2.vue

同样让child1触发child2事件。

```
<!-- emitter.vue 引入 child1 child2 -->
<emitter-child1  content="emitter-child1"></emitter-child1>
<emitter-child2  content="emitter-child2"></emitter-child2>
```

这里child1,child2是兄弟组件，他们共同拥有父组件emitter.vue。

所以child1要触发child2事件，需要emitter.vue的帮助。

1. child1派发事件到emitter.vue
2. emitter定义监听事件，广播到child2
3. child2定义监听，接收emitter的广播事件

```
// child1.vue 派发事件到 emitter.vue
import  Emitter  from  '@/mixins/Emitter'

export default {
  mixins:  [  Emitter  ],
    methods:  {
        handleClick  ()  {
            this.dispatch('skill-emitter',  'on-child1-
change',  this.selfContent)
        }
    }
}
// emitter.vue 定义监听事件，广播到 child2.vue
import  Emitter  from  '@/mixins/emitter'

export default {
  mixins:  [  Emitter  ],
  created  ()  {
```

```
        //  代理child1派发的on-child1-change事件
        //  该事件不在此组件处理
        //  广播给child2的on-change事件处理
        this.$on('on-child1-change',  e  =>  {
            this.broadcast('skill-emitter-child2',  'on-change',  e)
        })
    }
}
// child2.vue 定义监听，接收 emitter.vue 的广播事件
export default {
  methods: {
    handleChange  ($event)  {
            this.selfContent  =  $event
        }
  },
    created  ()  {
        this.$on('on-change',  this.handleChange)
    }
}
```

emitter.vue 源码

派发/广播机制对比总线模式，限制了事件必须是具体的某个组件的，使得事件的传递更加精确可控。

emitter.vue 是兄弟组件实例，如果是上下层的组件。

不需要经过如emitter.vue的代理，直接在目标组件监听自定义事件。

不管是dispath还是broadcast方法，都很依赖componentName，还有eventName，

所以**良好的命名规范是非常重要的**，dispath/broadcast到指定的组件的xx事件。

该指定的组件this.$on监听自定义事件就可以触发。

结语

今天我们从组件通讯的解决方式出发，介绍到了多种组件通讯的解决方案。

以前觉得组件通讯无非就是这几种。

1. prop <=> emit
2. vuex
3. bus

4. dispatch / broadcast

好好的梳理出来才发现其实在很多场景我们都需要用到组件通讯。

只是我们太过习以为常，反而有点忽略了也是其解决方案之一。

在实例开发中，我们也需要结合实际场景思考，是否以上的解决方案是否适用。

一般情况下，我是支持使用dispatch/broadcast模式去处理大多数的非父子组件通讯的。

可有时候要一个组件触发各自触发n个组件的事件，这种模式可能编写起来比较繁琐。

这个时候可以考虑总线模式，又或者，为了一处比较特殊的处理，是否考虑要引入一个mixins。

这些问题都是需要开发者是在实际开发的时候好好去估量利弊的，并没有一种万能的解决方案。

（对我来讲，dispatch/broadcast 就已经很万能了。）

还有这里想吐槽一下。

Flux 架构就像眼镜：您自会知道什么时候需要它 。--Dan Abramov (Redux 的作者)

说真的，直到现在我完全不能理解这句充满诗意的话，以及为什么偏偏是像眼镜而不是心灵的窗户？！