- 组件基础
  - 什么是组件
  - 局部组件
  - 全局组件
  - 组件通信
    - 父传子prop
    - 子传父$emit
    - 平行组件
    - 其它组件通信方式
  - 插槽
    - 匿名插槽
    - 具名插槽
    - 作用域插槽
    - 作用域插槽应用
- 生命周期
  - 什么是生命周期
  - 干活满满
  - 生命周期钩子
    - beforCreate
    - created
    - beforeMount
    - mounted
    - beforeUpdate和updated
    - beforeDestroy和destroyed
    - activated和deactivated
- 组件进阶
    - 获取DOM和子组件对象
    - nextTick的用法
    - nextTick的应用
- 混入mixin偷懒

组件基础

什么是组件

其实突然出来的这个名词,会让您不知所以然,如果大家使用过bootstrap的同学一定会对这个名词不陌生,我们其实在很早的时候就接触这个名词

button(html+css+js)

通常一个应用会以一颗嵌套的组件树的形式来表示:

局部组件

使用局部组件的打油诗: 建子 挂子 用子

> 注意:在组件中这个data必须是一个函数,返回一个对象

```
<div id="app">
        <!-- 4.使用子组件 -->
    <App></App>
```

```
</div>
<script>
//2.创建子组件
const Vheader = {
    data(){
      return {
        msg:'hello 组件'
      }
    },
    template:`<div>我是头部 {{msg}}</div>`
  }
//1.创建父组件   .vue
const App = {
    //必须是一个函数
    data() {
        return {
            msg: '我是App组件'
        }
    },
    components: {
        //3.挂载子组件
        Vheader
    },
    template: `
    <div>
      <Vheader></Vheader>
```

```
    </div>
    `,
  methods:{

  }
}
const app = Vue.createApp(App)
app.moute('#app')
</script>
```

全局组件

通过 `Vue.component(组件名,{})` 创建全局组件,此时该全局组件可以在任意模板(template)中使用(不需要挂载)

```
const app = Vue.createApp({})
app.component('Child',{
    template:`
    <div>
        <h3>我是一个子组件</h3>
    </div>
`
})
```

组件通信

父传子prop

如果一个网页有一个博文组件,但是如果你不能向这个组件传递某一篇博文的标题和内容之类想展示的数据的话,它是没有办法使用的.这也正是prop的由来

父组件往子组件通信:**通过Prop向子组件传递数据**

```
app.component('Child',{
    template:`
    <div>
        <h3>我是一个子组件</h3>
        <h4>{{childData}}</h4>
    </div>
`,
    props:['childData']
})
const App = {
    data() {
        return {
            msg: '我是父组件传进来的值'
        }
    },
    template: `
    <div>
      <Child :childData = 'msg'></Child>
    </div>
```

```
  `,
    computed: {

    }
}
```

1. 在子组件中声明props接收在父组件挂载的属性
2. 可以在子组件的template中任意使用
3. 在父组件绑定自定义的属性

子传父$emit

网页上有一些功能可能要求我们和父组件组件进行沟通

子组件往父组件通信：**监听子组件事件,使用事件抛出一个值**

```
app.component('Child', {
    template: `
  <div>
        <h3>我是一个子组件</h3>
        <h4>{{childData}}</h4>
        <input type="text" @input =
'handleInput'/>
  </div>
`,
    props: ['childData'],
    emits:['inputHandler']
    methods:{
        handleInput(e){
```

```
            const val = e.target.value;
            //使用$emit触发子组件的事件
            this.$emit('inputHandler',val);
        }
    },
})

const App = {
    data() {
        return {
            msg: '我是父组件传进来的值',
            newVal:''
        }
    },
    methods:{
        input(newVal){
            // console.log(newVal);
            this.newVal = newVal;
        }
    },
    template: `
    <div>
        <div class='father'>
        数据:{{newVal}}
        </div>
    <!--子组件监听事件-->
```

```
        <Child :childData = 'msg' @inputHandler = 'input'></Child>
    </div>
`,
    computed: {

    }
}
```

1. 在父组件中 子组件上绑定自定义事件
2. 在子组件中 触发原生的事件 在事件函数通过this.$emit触发自定义的事件

平行组件

在开发中,可能会存在没有关系的组件通信,比如有个博客内容显示组件,还有一个表单提交组件,我们现在提交数据到博客内容组件显示,这显示有点费劲.

为了解决这种问题,在vue中我们可以使用bus,创建中央事件总线

```
const bus = new Vue();
// 中央事件总线 bus
app.component('B', {
    data() {
        return {
            count: 0
        }
    },
    template: `
<div>{{count}}</div>
```

```
`,
    created(){
        // $on 绑定事件
        bus.$on('add',(n)=>{
            this.count+=n;
        })
    }
})

app.component('A', {
    data() {
        return {

        }
    },
    template: `
    <div>
     <button @click='handleClick'>加入购物车</button>
    </div>
`,
    methods:{
        handleClick(){
            // 触发绑定的函数 // $emit 触发事件
            bus.$emit('add',1);
        }
```

```
    }
})
```

其它组件通信方式

父组件 provide来提供变量,然后再子组件中通过inject来注入变量.无论组件嵌套多深

```
app.component('B', {
    data() {
        return {
            count: 0
        }
    },
    inject:['msg'],
    created(){
        console.log(this.msg);

    },
    template: `
    <div>
        {{msg}}
    </div>
`,
})

Vue.component('A', {
```

```
20      data() {
21          return {
22
23          }
24      },
25      created(){
26          // console.log(this.$parent.$parent);
27          // console.log(this.$children);
28          console.log(this);
29
30
31      },
32      template: `
33  <div>
34    <B></B>
35  </div>
36  `
37  })
38  new Vue({
39      el: '#app',
40      data: {
41
42      },
43      components: {
44          // 2.挂载子组件
45          App
```

```
    }

})
```

插槽

匿名插槽

子组件定义 slot 插槽，但并未具名，因此也可以说是默认插槽。只要在父元素中插入的内容，默认加入到这个插槽中去

```
app.component('MBtn', {
    template: `
    <button>
      <slot></slot>
    </button>
`,
    props: {
        type: {
            type: String,
            defaultValue: 'default'
        }
    },
})

const App = {
    data() {
```

```
        return {

        }
    },
    template: `
    <div>
        <m-btn>登录</m-btn>
        <m-btn>注册</m-btn>
        <m-btn>提交</m-btn>
    </div>
`,
}
new Vue({
    el: '#app',
    data: {

    },
    components: {
        // 2.挂载子组件
        App
    }

})
```

具名插槽

具名插槽可以出现在不同的地方，不限制出现的次数。只要匹配了 name 那么这些内容就会被插入到这个 name 的槽中去

在向具名插槽提供内容的时候，我们可以在一个 `<template>` 元素上使用 `v-slot` 指令，并以 `v-slot` 的参数的形式提供其名称

```
app.component('MBtn',{
    template:`
    <button :class='type' @click='clickHandle'>
        <slot name='register'></slot>
        <slot name='login'></slot>
        <slot name='submit'></slot>
    </button>
`,
    props:{
        type:{
            type: String,
            defaultValue: 'default'
        }
    },
    methods:{
        clickHandle(){
            this.$emit('click');
        }
    }
})
```

```
21
22 const App = {
23     data() {
24         return {
25
26         }
27     },
28     methods:{
29         handleClick(){
30             alert(1);
31         },
32         handleClick2(){
33             alert(2);
34         }
35     },
36     template: `
37 <div>
38   <MBtn type='default' @click='handleClick'>
39     <template v-slot:header>
40       注册
41     </template>
42   </MBtn>
43     <MBtn type='success' @click='handleClick2'>
44         <template v-slot:login>
45             登录
46         </template>
```

```
    </MBtn>
    <MBtn type='danger'>
      <template v-slot:submit>
        提交
      </template>
    </MBtn>
</div>
`,
}
new Vue({
    el: '#app',
    data: {

    },
    components: {
        App
    }

})
```

作用域插槽

通常情况下普通的插槽是父组件使用插槽过程中传入东西决定了插槽的内容。但有时我们需要获取到子组件提供的一些数据，那么作用域插槽就排上用场了

```
app.component('MyComp', {
    data(){
```

```
        return {
            obj:{
                username:'小马哥'
            }
        }
    },
    template: `
    <div>
        <slot :data = 'obj'></slot>
        <slot :data = 'obj' name='one'></slot>
    </div>
  `
})

const App = {
    data() {
        return {

        }
    },
    template: `
    <div>
        <MyComp>
```

```
26            <!--默认的插槽 default可以省略,注意:默认插槽的缩写语法不能和具名插槽混用,因为它会导致作用域不明确,只要出现多个插槽,为所有的插槽使用完整的基于<template>的语法-->
27            <!--具名插槽 v-slot:default 可以缩写为:#default-->
28            <template v-slot:default='slotScope'>
29            {{slotScope.data.username}}
30          </template>
31
32          </MyComp>
33          <MyComp>
34            <!--与具名插槽配合使用-->
35             <template v-slot:one='user'>
36             {{user.data.username}}
37          </template>
38          </MyComp>
39      </div>
40  `,
41  }
42  new Vue({
43      el: '#app',
44      data: {
45
46      },
47      components: {
```

```
48          App
49      }
50
51  })
```

作用域插槽应用

先说一下我们假设的应用常用场景，我们已经开发了一个代办事项列表的组件，很多模块在用，现在要求在**不影响已测试通过的模块功能和展示**的情况下，给**已完成的代办项增加一个对勾效果**。

也就是说，代办事项列表组件要满足一下几点

1. 之前数据格式和引用接口不变，正常展示
2. 新的功能模块增加对勾

```
1  const todoList = {
2      data(){
3          return {
4
5          }
6      },
7      props:{
8          todos:Array,
9          defaultValue:[]
10     },
11     template:`
12     <ul>
13       <li v-for='item in todos' :key='item.id'>
```

```
14          <slot :item='item'>
15          {{item.title}}
16          </slot>
17        </li>
18      </ul>
19  `
20  }
21
22  const App = {
23      data() {
24          return {
25              todoList: [
26                  {
27                      title: '大哥你好么',
28                      isComplate:true,
29                      id: 1
30                  },
31                  {
32                      title: '小弟我还行',
33                      isComplate:false,
34                      id: 2
35                  },
36                  {
37                      title: '你在干什么',
38                      isComplate:false,
39                      id: 3
```

```
                },
                {
                    title: '抽烟喝酒烫头',
                    isComplate:true,
                    id: 4
                }
            ]
        }
    },
    components:{
        todoList
    },
    template: `
    <todoList :todos='todoList'>
      <!--解构用法-->
      <template v-slot='{item}'>
        <input type='checkbox' v-
model='data.itemValue.isComplate'/>
        {{itemValue.title}}
      </template>
    </todoList>
`,
}
new Vue({
    el: '#app',
    data: {
```

```
65
66      },
67      components: {
68          App
69      }
70
71  })
```

具名插槽缩写

```
1  <template #header></template>
2  <template #default></template>
3
4  //等价于
5  <template v-slot:header></template>
6  <template v-slot:default></template>
7
```

生命周期

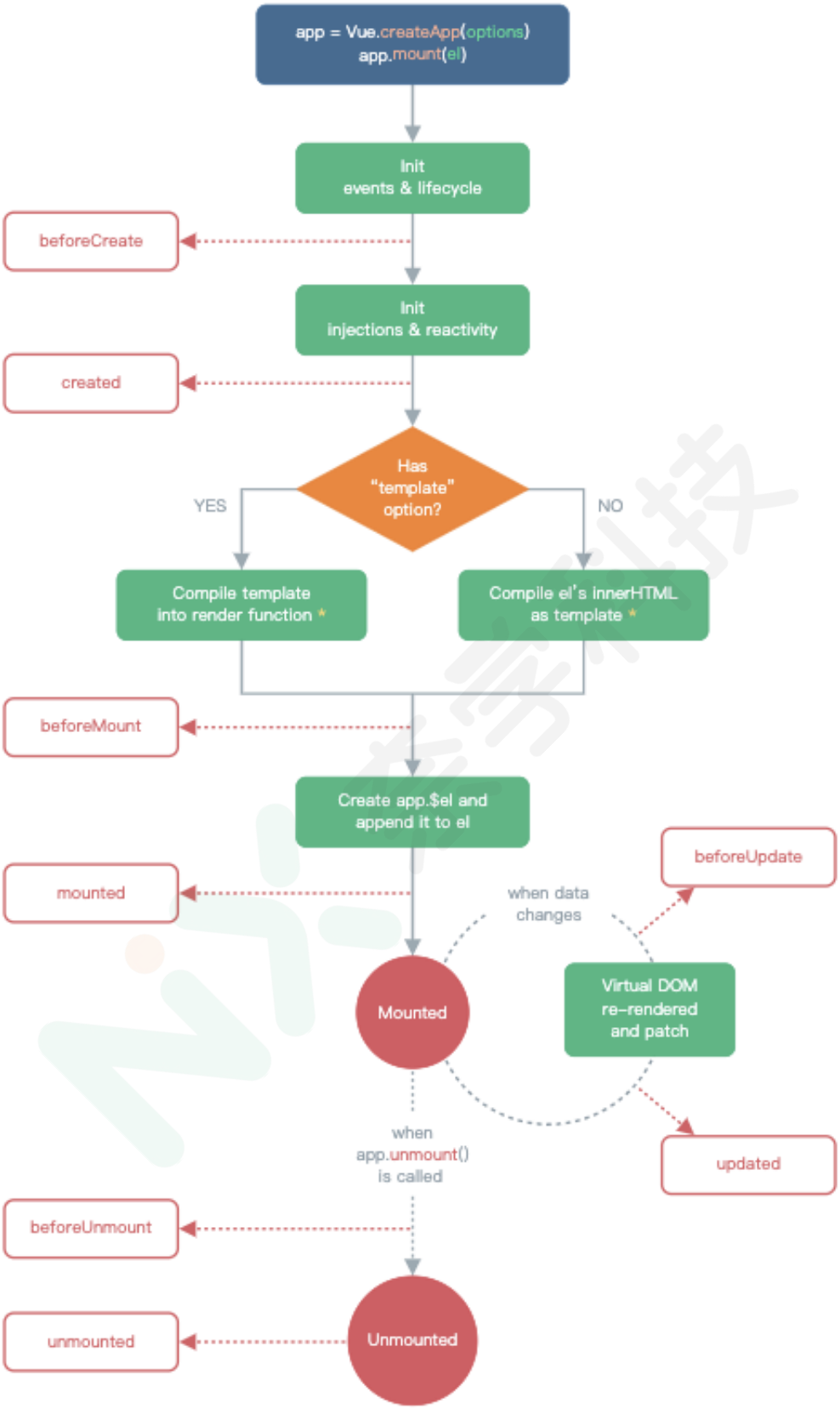
“你不需要立马弄明白所有的东西，不过随着你的不断学习和使用，它的参考价值会越来越高。

当你在做项目过程中,遇到了这种问题的时候,再回过头来看这张图

什么是生命周期

每个 Vue 实例在被创建时都要经过一系列的初始化过程。
例如：从开始创建、初始化数据、编译模板、挂载Dom、数据变化时更新DOM、卸载等一系列过程。
我们称 **这一系列的过程** 就是Vue的生命周期。
通俗说就是Vue实例从创建到销毁的过程，就是生命周期。
同时在这个过程中也会运行一些叫做**生命周期钩子**的函数，这给了用户在不同阶段添加自己的代码的机会，利用各个钩子来完成我们的业务代码。

干活满满



* Template compilation is performed ahead-of-time if using a build step, e.g., with single-file components.

生命周期钩子

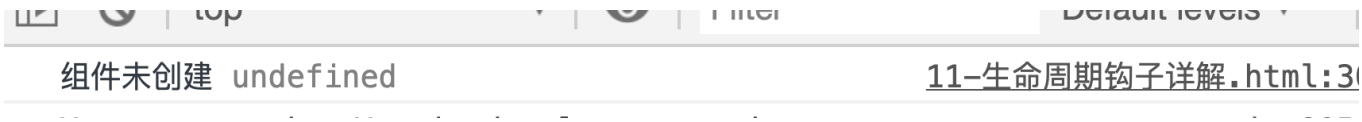
beforCreate

实例初始化之后、创建实例之前的执行的钩子事件

```
Vue.component('Test',{
    data(){
        return {
            msg:'小马哥'
        }
    },
    template:`
    <div>
      <h3>{{msg}}</h3>
    </div>
  `,
    beforeCreate:function(){
        // 组件创建之前
        console.log(this.$data);//undefined

    }
})
```

效果：



> 创建实例之前，数据观察和事件配置都没好准备好。也就是数据也没有、DOM也没生成

created

实例创建完成后执行的钩子

```
created() {
    console.log('组件创建', this.$data);
}
```

效果：

```
组件创建 ▶{__ob__: Observer}                    11-生命周期钩子详解.html:34
```

> 实例创建完成后，我们能读取到数据data的值，但是DOM还没生成,可以在此时发起ajax

beforeMount

将编译完成的html挂载到对应的**虚拟DOM**时触发的钩子 此时页面并没有内容。即此阶段解读为：即将挂载

```
beforeMount(){
    // 挂载数据到 DOM之前会调用


    console.log('DOM挂载之前',document.getElementById('app'));
}
```

效果：

mounted

编译好的html挂载到页面完成后所执行的事件钩子函数

```
mounted() {
    console.log('DOM挂载完成',document.getElementById('app'));
}
```

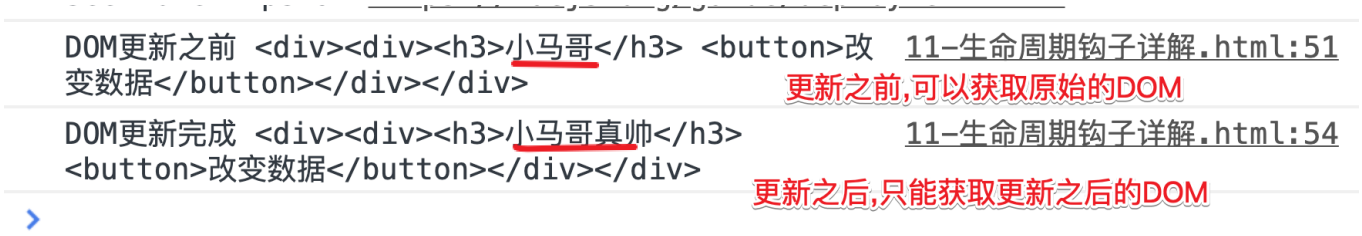
效果：

```
▼<div id="app">                                    11-生命周期钩子详解.html:41
  ▼<div>
    ▼<div>
        <h3>小马哥</h3>
      </div>
    </div>
DOM挂载完成  </div>
```

beforeUpdate和updated

```
1 beforeUpdate() {
2     // 在更新DOM之前 调用该钩子，应用：可以获取原始的DOM
3     console.log('DOM更新之前', document.getElementById('app').innerHTML);
4 },
5 updated() {
6     // 在更新DOM之后调用该钩子，应用：可以获取最新的DOM
7   console.log('DOM更新完成', document.getElementById('app').innerHTML);
8 }
```

效果：



beforeDestroy和destroyed

当子组件在v-if的条件切换时,该组价处于创建和销毁的状态

```
beforeDestroy() {
    console.log('beforeDestroy');
},
destroyed() {
    console.log('destroyed');
},
```

activated和deactivated

当配合vue的内置组件 `<keep-alive>` 一起使用的时候,才会调用下面此方法

`<keep-alive>` 组件的作用它可以缓存当前组件

```
activated() {
    console.log('组件被激活了');
},
deactivated() {
   console.log('组件被停用了');

},
```

组件进阶

获取DOM和子组件对象

尽管存在 prop 和事件，有的时候你仍可能需要在 JavaScript 里直接访问一个子组件。为了达到这个目的，你可以通过 `ref` 特性为这个子组件赋予一个 ID 引用。例如：

```
const Test = {
    template: `<div class='test'>我是测试组件</div>`
}
const App = {
    data() {
        return {

        }
```

```
    },
    created() {
        console.log(this.$refs.test);
//undefined

    },
    mounted() {
        // 如果是组件挂载了ref 获取是组件对象,如果是标签挂载了ref,则获取的是DOM元素
        console.log(this.$refs.test);
        console.log(this.$refs.btn);

        // 加载页面 让input自动获取焦点
        this.$refs.input.focus();

    },
    components: {
        Test
    },
    template: `
    <div>
        <button ref = 'btn'></button>
        <input type="text" ref='input'>
        <Test ref = 'test'></Test>
    </div>
  `
```

```
}
new Vue({
    el: '#app',
    data: {

    },
    components: {
        App
    }
})
```

nextTick的用法

将回调延迟到下次 DOM 更新循环之后执行。在修改数据之后立即使用它，然后等待 DOM 更新

有些事情你可能想不到,vue在更新DOM时是**异步**执行的.只要侦听到数据变化,Vue将开启一个队列,并缓存在同一事件循环中发生的所有数据变更.如果同一个wather被多次触发,只会被推入到队列中一次.这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际（已去重的）工作

nextTick的应用

> 有个需求:
>
> 在页面拉取一个接口，这个接口返回一些数据，这些数据是这个页面的一个浮层组件要依赖的，然后我在接口一返回数据就展示了这个浮层组件，展示的同时，上报一些数据给后台（这些数据就是父组件从接口拿的），这个时候，神奇的事情发生了，虽然我拿到数据了，但是浮层展现的时候，这些数

> 据还未更新到组件上去,上报失败

```
const Pop = {
    data() {
        return {
            isShow:false
        }
    },
    template:`
    <div v-show = 'isShow'>
      {{name}}
    </div>
  `,
    props:['name'],
    methods: {
        show(){
            this.isShow = true;
            alert(this.name);
        }
    },
}
const App = {
    data() {
        return {
            name:''
        }
```

```
    },
    created() {
        // 模拟异步请求的数据
        setTimeout(() => {
            this.name = '小马哥',
            this.$refs.pop.show();
        }, 2000);
    },
    components:{
        Pop
    },
    template: `<pop ref='pop' :name='name'>
</pop>`
}
const vm = new Vue({
    el: '#app',
    components: {
        App
    }
})
```

完美解决：

```
created() {
    // 模拟异步请求的数据
    setTimeout(() => {

        this.name = '小马哥',
         this.$nextTick(()=>{
              this.$refs.pop.show();
         })
    }, 2000);
},
```

混入mixin偷懒

> 混入(mixin)提供了一种非常灵活的方式,来分发Vue组件中的可复用功能.一个混入对象可以包含任意组件选项.
>
> 一个混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被“混合”进入该组件本身的选项。

```
<div id="app">
  {{msg}}
</div>
<script src="./vue.js"></script>
<script>
  const myMixin = {
```

```
    data(){
      return {
        msg:'123'
      }
    },
    created() {
      this.sayHello()
    },
    methods: {
      sayHello(){
        console.log('hello mixin')
      }
    },
  }
  const app = Vue.createApp({
    mixins: [myMixin]
  })

  app.mount('#app')

```

mixin应用

有一种很难常见的情况:有两个非常相似的组件,他们共享同样的基本函数,并且他们之间也有足够的不同,这时你站在了一个十字路口：我是把它拆分成两个不同的组件？还是只使用一个组件，创建足够的属性来改变不同的情况。

这些解决方案都不够完美：如果你拆分成两个组件，你就不得不冒着如果功能变动你要在两个文件中更新它的风险，这违背了 DRY 前提。另一方面，太多的属性会很快会变得混乱不堪，对维护者很不友好，甚至是你自己，为了使用它，需要理解一大段上下文，这会让你感到失望。

使用混合。Vue 中的混合对编写函数式风格的代码很有用，因为函数式编程就是通过减少移动的部分让代码更好理解。混合允许你封装一块在应用的其他组件中都可以使用的函数。如果被正确的使用，他们不会改变函数作用域外部的任何东西，所以多次执行，只要是同样的输入你总是能得到一样的值。这真的很强大。

我们有一对不同的组件,他们的作用是切换一个状态布尔值,一个模态框和一个提示框.这些提示框和模态框除了在功能,没有其它共同点:**它们看起来不一样,用法不一样,但是逻辑一样**

```
<div id="app">
  <App></App>
</div>
<script src="./vue.js"></script>
<script>
  const app = Vue.createApp({})
  // 全局混入 要格外小心 每次实例创建 都会调用
  app.mixin({
    created(){
      console.log('hello from mixin!!');

    }
  })
  // 抽离
  const toggleShow = {
    data() {
```

```
17        return {
18          isShow: false
19        }
20      },
21      methods: {
22        toggleShow() {
23          this.isShow = !this.isShow
24        }
25      }
26    }
27    const Modal = {
28      template: `<div v-if='isShow'><h3>模态框组件
    </h3></div>`,
29      data() {
30        return {
31
32        }
33      },
34      mixins:[toggleShow]
35
36    }
37    const ToolTip = {
38      data() {
39        return {
40        }
41      },
```

```
42      template: `<div v-if='isShow'><h3>提示组件
    </h3></div>`,
43      mixins:[toggleShow]
44
45    }
46    const App = {
47      data() {
48        return {
49
50        }
51      },
52      template: `
53  <div>
54  <button @click='handleModel'>模态框</button>
55  <button @click='handleToolTip'>提示框</button>
56  <Modal ref='modal'></Modal>
57  <ToolTip ref="toolTip"></ToolTip>
58    </div>
59  `,
60      components: {
61        Modal,
62        ToolTip
63      },
64      methods: {
65        handleModel() {
66          this.$refs.modal.toggleShow()
```

```
67        },
68        handleToolTip() {
69          this.$refs.toolTip.toggleShow()
70        }
71      },
72    }
73    app.mount('#app')
```