

学会使用 \$attrs 与 \$listeners，二次包装组件就靠它了

前几天产品经理给我甩过来一份管理系统的设计原型，我打开看了看，虽然内心是拒绝的，但是为了活着，还是要做的。小编看了看原型，发现系统中的大部分弹框右下角都是确定和取消两个按钮。如果使用element-ui提供的Dialog，那么每一个弹框都要手动加按钮，不但代码量增多，而且后面如果按钮UI，需求发生变化，改动量也比较大。

```
<el-dialog
  title="提示"
  :visible.sync="dialogVisible"
  width="30%"
  :before-close="handleClose"
>
  <span>这是一段信息</span>
  <span slot="footer" class="dialog-footer">
    <el-button @click="dialogVisible = false">取 消</el-button>
    <el-button type="primary" @click="dialogVisible = false">
      确 定
    </el-button>
  </span>
</el-dialog>
```

如果可以将Dialog进行二次封装，将按钮封装到组件内部，就可以不用重复去写了。说干就干。

定义基本弹框代码

```
<template>
  <el-dialog :visible.sync="visibleDialog">
    <!--内容区域的默认插槽-->
    <slot></slot>
    <!--使用弹框的footer插槽添加按钮-->
    <template #footer>
      <!--对外继续暴露footer插槽，有个别弹框按钮需要自定义-->
      <slot name="footer">
        <!--将取消与确定按钮集成到内部-->
        <span>
          <el-button @click="$_handleCancel">取 消</el-button>
          <el-button type="primary" @click="$_handleConfirm">
            确 定
          </el-button>
        </span>
      </slot>
    </template>
  </el-dialog>
</template>
```

```

<script>
export default {
  props: {
    // 对外暴露visible属性，用于显示隐藏弹框
    visible: {
      type: Boolean,
      default: false
    }
  },
  computed: {
    // 通过计算属性，对.sync进行转换，外部也可以直接使用visible.sync
    visibleDialog: {
      get() {
        return this.visible;
      },
      set(val) {
        this.$emit("update:visible",val);
      }
    }
  },
  methods: {
    // 对外抛出cancel事件
    $_handleCancel() {
      this.$emit("cancel");
    },
    // 对外抛出 confirm事件
    $_handleConfirm() {
      this.$emit("confirm");
    }
  }
};
</script>
复制代码

```

通过上面的代码，我们已经将按钮封装到组件内部了，效果如下图所示：

```

<!--外部使用方式 confirm cancel 是自定义的事件 opened是包装el-dialog的事件，通过
$listeners传入到el-dialog里面-->
<custom-dialog :visible.sync="visibleDialog" @opened="$_handleOpened"
@confirm="$_handleConfirm" @cancel="$_handleCancel">这是一段内容</custom-dialog>
复制代码

```

效果图



但上面的代码存在一个问题，无法将 Dialog 自身的属性和事件暴露到外部（虽然可以通过 props 及 \$emit 一个一个添加，但是很麻烦），这时候就可以使用 \$attrs 与 \$listeners

使用 \$attrs 与 \$listeners

`$attrs`: 当组件在调用时传入的属性没有在 `props` 里面定义时, 传入的属性将被绑定到 `$attrs` 属性内 (`class` 与 `style` 除外, 他们会挂载到组件最外层元素上)。并可通过 `v-bind="$attrs"` 传入到内部组件中

`$listeners`: 当组件被调用时, 外部监听的这个组件的所有事件都可以通过 `$listeners` 获取到。并可通过 `v-on="$listeners"` 传入到内部组件中。

修改弹框代码

```
<!--使用了v-bind与v-on监听属性与事件-->
<template>
  <el-dialog :visible.sync="visibleDialog" v-bind="$attrs" v-on="$listeners">
    <!--其他代码不变-->
  </el-dialog>
</template>
<script>
  export default {
    //默认情况下父作用域的不被认作 props 的 attribute 绑定 (attribute bindings)
    //将会“回退”且作为普通的 HTML attribute 应用在子组件的根元素上。
    //通过设置 inheritAttrs 到 false, 这些默认行为将会被去掉
    inheritAttrs: false
  }
</script>

<!--外部使用方式-->
<custom-dialog
  :visible.sync="visibleDialog"
  title="测试弹框"
  @opened="$_handleOpened"
>
  这是一段内容
</custom-dialog>
复制代码
```

对于 `$attrs`, 我们也可以使用 `$props` 来代替, 实现代码如下

```
<template>
  <el-dialog :visible.sync="visibleDialog" v-bind="$props" v-on="$listeners">
    <!--其他代码不变-->
  </el-dialog>
</template>
<script>
import { Dialog } from 'element-ui'
export default {
  props: {
    // 将Dialog的props通过扩展运算符展开到props属性里面
    ...Dialog.props
  }
}
</script>
复制代码
```

但上面的代码存在一定的缺陷, 有些组件存在非 `props` 的属性, 比如对于一些封装的表单组件, 我们可能需要给组件传入原生属性, 但实际原生属性并没有在组件的 `props` 上面定义, 这时候, 如果通过上面的方式去包装组件, 那么这些原生组件将无法传递到内部组件里面。

使用 `require.context` 实现前端工程自动化

`require.context` 是一个 webpack 提供的 Api, 通过执行 `require.context` 函数获取一个特定的上下文, 主要是用于实现自动化导入模块。

什么时候用? 当一个 js 里面需要手动引入过多的其他文件夹里面的文件时, 就可以使用。

在 Vue 项目开发过程中, 我们可能会遇到这些可能会用到 `require.context` 的场景

1. 当我们路由页面比较多的时候, 可能会将路由文件拆分成多个, 然后再通过 `import` 引入到 `index.js` 路由主入口文件中
2. 当使用 svg symbol 时候, 需要将所有的 svg 图片导入到系统中 (建议使用 `svg-sprite-loader`)
3. 开发了一系列基础组件, 然后把所有组件都导入到 `index.js` 中, 然后再放入一个数组中, 通过遍历数组将所有组件进行安装。

对于上述的几个场景, 如果我们需要导入的文件比较少的情况下, 通过 `import` 一个一个去导入还可以接受, 但对于量比较大的情况, 就变成了纯体力活, 而且每次修改增加都需要在主入口文件内进行调整。这时候我们就可以通过 `require.context` 去简化这个过程。

现在以上述第三条为例, 来说明 `require.context` 的用法

常规用法



组件通过常规方式安装

```
1 import CustomDialog from './custom -dialog.vue'
2 import CustomGrid from './custom -grid.vue'
3 import CustomInput from './custom -input.vue'
4 import CustomLoading from './custom -loading.vue'
5 import CustomSelect from './custom -select.vue'
6
7 const components = [
8   CustomDialog ,
9   CustomGrid ,
10  CustomInput ,
11  CustomLoading ,
12  CustomSelect
13 ]
14
15 components .forEach ( component => {
16   Vue .component ( component . name, component )
17 })
18
```

require.context 基本语法

```
/**
 * directory: 要扫描的目录
 * useSubdirectories: 是否扫描所有的子级文件夹
 * regExp: 要扫描的文件，用正则进行匹配
 */
require.context(directory, useSubdirectories = false, regExp = /^\.\/$/)
```

通过 require.context 安装 Vue 组件

```
/**
 * directory=./ 扫描 当前目录下面的所有文件
 * useSubdirectories=false, 表示不需要地柜扫描所有的子文件夹
 * regExp=/\.vue$/所有以.vue结束的文件
 */
const context = require.context('./', false, /\.vue$/)

/**
 * context.keys() 返回所有匹配到的文件的路径
 */
context.keys().forEach(key => {
  // 通过context(key)可以获取到对应的文件 .default表示 export default导出的内容
  component = context(key).default
  // 安装vue组件
  Vue.component(component.name, component)
})
```

自定义 v-model, 原来这么简单

在用Vue开发前端时，不论使用原生还是封装好的UI库，对于表单组件，一般都会使用到 `v-model`。虽然 `v-model` 是一个语法糖，但是吃到嘴里挺甜的啊。学会自定义 `v-model`，还是很有必要的。

基本用法

一个组件上的 `v-model` 默认是通过在组件上面定义一个名为 `value` 的 props,同时对外暴露一个名为 `input` 的事件。

源码：

```
<template>
  <div class="custom-input">
    <input :value="value" @change="$_handleChange" />
  </div>
</template>
<script>
export default {
  props: {
    // 定义一个名为value的属性
    value: {
      type: String,
      default: ""
    }
  },
  methods: {
    $_handleChange(e) {
      // 对外暴露一个input事件
      this.$emit("input", e.target.value);
    }
  }
};
</script>
```

使用方式：

```
<custom-input v-model="text"></custom-input>
```

自定义属性与事件

通常情况下，使用 `value` 属性与 `input` 事件没有问题，但是有时候有些组件会将 `value` 属性或 `input` 事件用于不同的目的，比如对于单选框、复选框等类型的表单组件的 `value` 属性就有其他用处，参考 (developer.mozilla.org/en-US/docs/...)。或者希望属性名称或事件名称与实际行为更贴切，比如 `active`、`checked` 等属性名。

```

<template>
  <!-- 开关组件，通过样式控制开关组件的状态-->
  <div
    :class="['custom-switch', active && 'custom-switch__active']"
    @click="$_handleClick"
  >
    <!-- 开关组件内部的开关-->
    <span class="custom-switch__core"></span>
  </div>
</template>
<script>
export default {
  // 通过model 可以自定义 属性名 和事件名
  model: {
    event: "change",
    prop: "active"
  },
  props: {
    // 定义一个名为active的属性
    active: {
      type: Boolean,
      default: false
    }
  },
  methods: {
    $_handleClick() {
      // 对外暴露一个change事件
      this.$emit("change", !this.active);
    }
  }
};
</script>

```

使用 `.sync`, 更优雅的实现数据双向绑定

在 `vue` 中, `props` 属性是单向数据传输的, 父级的 `prop` 的更新会向下流动到子组件中, 但是反过来不行。可是有些情况, 我们需要对 `prop` 进行“双向绑定”。上文中, 我们提到了使用 `v-model` 实现双向绑定。但有时候我们希望一个组件可以实现多个数据的“双向绑定”, 而 `v-model` 一个组件只能有一个 (Vue3.0 可以有多个), 这时候就需要使用到 `.sync`。

`.sync` 与 `v-model` 的异同

相同点:

- 两者的本质都是语法糖, 目的都是实现组件与外部数据的双向绑定
- 两个都是通过属性+事件来实现的

不同点(个人观点, 如有不对, 麻烦下方评论指出, 谢谢):

- 一个组件只能定义一个 `v-model`, 但可以定义多个 `.sync`
- `v-model` 与 `.sync` 对于的事件名称不同, `v-model` 默认事件为 `input`, 可以通过配置 `model` 来修改, `.sync` 事件名称固定为 `update: 属性名`

自定义 .sync

在开发业务时，有时候需要使用一个遮罩层来阻止用户的行为（更多会使用遮罩层+loading动画），下面通过自定义 .sync 来实现一个遮罩层

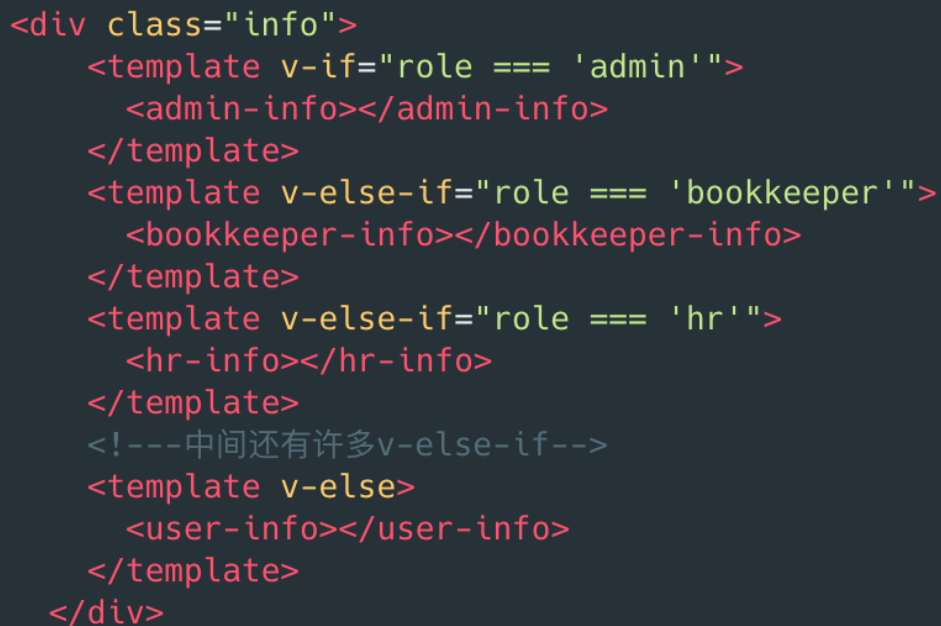
```
<template>
  <!--遮罩层-->
  <div class="custom-overlay" v-show="visible" @click="$_handleClick"></div>
</template>
<script>
export default {
  props: {
    // 定义控制遮罩层显示隐藏的属性
    visible: {
      type: Boolean,
      default: false
    }
  },
  methods: {
    $_handleClick() {
      // 通过update:visible 事件修改外部传入的visible属性的值
      this.$emit("update:visible", false);
    }
  }
};
</script>
```

```
<!--调用方式-->
<template>
  <custom-overlay :visible.sync="visible" />
</template>

<script>
export default {
  data() {
    return {
      visible: false
    }
  }
}
</script>
复制代码
```

动态组件，让页面渲染更灵活

前两天产品经理来了新的需求了，告诉我，需要根据用户的权限不同，页面上要显示不同的内容，然后我就哼哼唧唧的将不同权限对应的组件写了出来，然后再通过 v-if 来判断要显示哪个组件，就有了下面的代码



```

<div class="info">
  <template v-if="role === 'admin'">
    <admin-info></admin-info>
  </template>
  <template v-else-if="role === 'bookkeeper'">
    <bookkeeper-info></bookkeeper-info>
  </template>
  <template v-else-if="role === 'hr'">
    <hr-info></hr-info>
  </template>
  <!--中间还有许多v-else-if-->
  <template v-else>
    <user-info></user-info>
  </template>
</div>

```

但是看到上面代码的那一长串 `v-if`, `v-else-if`, 我感觉我的代码洁癖症要犯了, 不行, 这样 `code review` 过不了关, 我连自己这一关都过不了, 这时候就改动态组件发挥作用了。

```

<template>
  <div class="info">
    <component :is="roleComponent" v-if="roleComponent" />
  </div>
</template>
<script>
import AdminInfo from './admin-info'
import BookkeeperInfo from './bookkeeper-info'
import HrInfo from './hr-info'
import UserInfo from './user-info'
export default {
  components: {
    AdminInfo,
    BookkeeperInfo,
    HrInfo,
    UserInfo
  },
  data() {
    return {
      roleComponents: {
        admin: AdminInfo,
        bookkeeper: BookkeeperInfo,
        hr: HrInfo,

```

```

      user: UserInfo
    },
    role: 'user',
    roleComponent: undefined
  }
},
created() {
  const { role, roleComponents } = this
  this.roleComponent = roleComponents[role]
}
}
</script>
复制代码

```

mixins，更高效的实现组件内容的复用

`mixins` 是 `vue` 提供的一种混合机制，用来更高效的实现组件内容的复用。怎么去理解混入呢，我觉得和 `Object.assign`，但实际与 `Object.assign` 又有所不同。

基本示例

在开发 `echarts` 图表组件时，需要在窗口尺寸发生变化时，重置图表的大小，此时如果在每个组件里面都去实现一段监听代码，代码重复太多了，此时就可以使用混入来解决这个问题

```

// 混入代码 resize-mixins.js
import { debounce } from 'lodash'
const resizeChartMethod = '$__resizeChartMethod'

export default {
  data() {
    // 在组件内部将图表init的引用映射到chart属性上
    return {
      chart: null
    }
  },
  created() {
    window.addEventListener('resize', this[resizeChartMethod])
  },
  beforeDestroy() {
    window.removeEventListener('reisz', this[resizeChartMethod])
  },
  methods: {
    // 通过lodash的防抖函数来控制resize的频率
    [resizeChartMethod]: debounce(function() {
      if (this.chart) {
        this.chart.resize()
      }
    }, 100)
  }
}

```

复制代码

```

<!-- 图表组件代码 -->
<template>
  <div class="chart"></div>
</template>
<script>

```

```
import echartsMixins from './echarts-mixins'
export default {
  // mixins属性用于导入混入，是一个数组，数组可以传入多个混入对象
  mixins: [echartsMixins],
  data() {
    return {
      chart: null
    }
  },
  mounted() {
    this.chart = echarts.init(this.$el)
  }
}
</script>
复制代码
```

不同位置的混入规则

在vue中，一个混入对象可以包含任意组件选项，但是对于不同的组件选项，会有不同的合并策略。

1. data 对于data,在混入时会进行递归合并，如果两个属性发生冲突，则以组件自身为主，如上例中的chart属性
2. 生命周期钩子函数

对于生命周期钩子函数，混入时会将同名钩子函数加入到一个数组中，然后在调用时依次执行。混入对象里面的钩子函数会优先于组件的钩子函数执行。如果一个组件混入了多个对象，对于混入对象里面的同名钩子函数，将按照数组顺序依次执行，如下代码：

```
const mixin1 = {
  created() {
    console.log('我是第一个输出的')
  }
}

const mixin2 = {
  created() {
    console.log('我是第二个输出的')
  }
}

export default {
  mixins: [mixin1, mixin2],
  created() {
    console.log('我是第三个输出的')
  }
}
复制代码
```

1. 其他选项 对于值为对象的选项，如methods, components, filter, directives, props等等，将被合并为同一个对象。两个对象键名冲突时，取组件对象的键值对。

全局混入

混入也可以进行全局注册。一旦使用全局混入，那么混入的选项将在所有的组件内生效，如下代码所示：

```
Vue.mixin({
```

```

methods: {
  /**
   * 将埋点方法通过全局混入添加到每个组件内部
   *
   * 建议将埋点方法绑定到vue的原型链上面，如： vue.prototype.$track = () => {}
   * */
  track(message) {
    console.log(message)
  }
}
})

```

复制代码

请谨慎使用全局混入，因为它会影响每个单独创建的 Vue 实例 (包括第三方组件)。大多数情况下，只应当应用于自定义选项，

插槽，相信每一位 `vue` 都有使用过，但是如何更好的去理解插槽，如何去自定义插槽，今天小编为你带来更形象的说明。

默认插槽

大学毕业刚上班，穷鬼一个，想着每个月租房还要掏房租，所以小编决定买一个一居室，东拼西凑借了一堆债，终于凑够了首付，买了一个小小的毛坯房。我们可以把这个一居室的毛坯房想想成一个组件，这个房子的户型，面积，楼层都是固定的，但是室内如何装修，摆什么家具，这个却是由你来说定的，房间内部就可以理解为插槽，允许用户去自定义内容。

1. 开发商终于将一居室开发完交房了

```

<template>
  <!--这是一个一居室-->
  <div class="one-bedroom">
    <!--添加一个默认插槽，用户可以在外部随意定义这个一居室的内容-->
    <slot></slot>
  </div>
</template>

```

复制代码

2. 小编要开始装修了

```

<template>
  <!--这里一居室-->
  <one-bedroom>
    <!--将家具放到房间里面，组件内部就是上面提供的默认插槽的空间-->
    <span>先放一个小床，反正没有女朋友</span>
    <span>再放一个电脑桌，在家还要加班写bug</span>
  </one-bedroom>
</template>
<script>
import OneBedroom from '../components/one-bedroom'
export default {
  components: {
    OneBedroom
  }
}
</script>

```

复制代码

具名插槽

过了几年，小编有了女朋友，准备结婚了，一居室房间肯定不行啊，丈母娘嫌小不同意，没办法，只能又凑钱买大房子，买了一个两居室（穷逼一个），因为是两居室，所以有了主卧和次卧之分，装修是否也不能把主卧和次卧装修的一模一样，所以就需要进行区分。将房子想想成组件，那么组件就有两个插槽，并且需要起名字进行区分。

1. 开发商终于开发完交房了

```
<template>
  <div class="two-bedroom">
    <!--这是主卧-->
    <div class="master-bedroom">
      <!--主卧使用默认插槽-->
      <slot></slot>
    </div>
    <!--这是次卧-->
    <div class="secondary-bedroom">
      <!--次卧使用具名插槽-->
      <slot name="secondard"></slot>
    </div>
  </div>
</template>
```

复制代码

2. 小编要卖血攒钱装修了

```
<template>
  <two-bedroom>
    <!--主卧使用默认插槽-->
    <div>
      <span>放一个大床，要结婚了，嘿嘿嘿</span>
      <span>放一个衣柜，老婆的衣服太多了</span>
      <span>算了，还是放一个电脑桌吧，还要写bug</span>
    </div>
    <!--次卧，通过v-slot:secondard 可以指定使用哪一个具名插槽， v-slot:secondard 也可以简写为 #secondard-->
    <template v-slot:secondard>
      <div>
        <span>父母要住，放一个硬一点的床，软床对腰不好</span>
        <span>放一个衣柜</span>
      </div>
    </template>
  </two-bedroom>
</template>
<script>
import TwoBedroom from '../components/slot/two-bedroom'
export default {
  components: {
    TwoBedroom
  }
}
```

```
</script>
```

复制代码

作用域插槽

装修的时候，装修师傅问我洗衣机是要放到卫生间还是阳台，一般情况下开发商会预留放洗衣机的位置。而这个位置可以理解成插槽传的参数，这个就是作用域插槽。

1. 看一下卫生间插槽传了什么参数

```
<template>
  <div class="two-bedroom">
    <!--其他内容省略-->
    <div class="toilet">
      <!--通过v-bind 可以向外传递参数，告诉外面卫生间可以放洗衣机-->
      <slot name="toilet" v-bind="{ washer: true }"></slot>
    </div>
  </div>
</template>
```

复制代码

2. 把洗衣机放到卫生间

```
<template>
  <two-bedroom>
    <!--其他省略-->
    <!--卫生间插槽，通过v-slot="scope"可以获取组件内部通过v-bind传的值-->
    <template v-slot:toilet="scope">
      <!--判断是否可以放洗衣机-->
      <span v-if="scope.washer">这里放洗衣机</span>
    </template>
  </two-bedroom>
</template>
```

复制代码

插槽默认值

小编的同事不想等期房，所以就买了二手房，二手房前业主都装修好了，可以直接入住。当然也可以重新装修，下面是同事买的二手房。

1. 这是装修好的二手房

```
<template>
  <div class="second-hand-house">
    <div class="master-bedroom">
      <!--插槽可以指定默认值，如果外部调用组件时没有修改插槽内容，则使用默认插槽-->
      <slot>
        <span>这里有一张水床，玩的够嗨</span>
        <span>还有一个衣柜，有点旧了</span>
      </slot>
    </div>
    <!--这是次卧-->
    <div class="secondary-bedroom">
```

```
<!--次卧使用具名插槽-->
<slot name="secondard">
  <span>这里有一张婴儿床</span>
</slot>
</div>
</div>
</template>
```

复制代码

2. 同事决定先把主卧装修了，以后结婚用

```
<second-hand-house>
  <!--主卧使用默认插槽，只装修主卧-->
  <div>
    <span>放一个大床，要结婚了，嘿嘿嘿</span>
    <span>放一个衣柜，老婆的衣服太多了</span>
    <span>算了，还是放一个电脑桌吧，还要写bug</span>
  </div>
</second-hand-house>
```

复制代码

了解选项合并策略,自定义生命周期钩子函数

当你使用 `vue` 的 `mixins` 的时候，是否有发现，如果混入的 `methods` 里面的方法与组件的方法同名，则会被组件方法覆盖，但是生命周期函数如果重名，混入的与组件自身的都会被执行，且执行顺序是先混入和自身，这是怎么做到的呢？

1. 了解 `vue` 合并策略

在 `vue` 中，不同的选项有不同的合并策略，比如 `data`, `props`, `methods` 是同名属性覆盖合并，其他直接合并，而生命周期钩子函数则是将同名的函数放到一个数组中，在调用的时候依次调用，具体可参考小编前面的一篇文章[绝对干货~！学会这些Vue小技巧，可以早点下班和女神约会了](#)

在 `vue` 中，提供了一个 `api`，`vue.config.optionMergeStrategies`，可以通过这个 `api` 去自定义选项的合并策略。

在代码中打印

```
console.log(Vue.config.optionMergeStrategies)
```

复制代码

控制台打印内容控制台打印内容

通过上图可以看到 `vue` 所有选项的合并策略函数，我们可以通过覆盖上面的方法，来自定义合并策略函数，不过一般用不到。

2. 通过合并策略自定义生命周期函数

背景

最近客户给领导反馈，我们的系统用一段时间，浏览器就变得有点卡，不知道为什么。问题出来了，本来想甩锅到后端，但是浏览器问题，没法甩锅啊，那就排查吧。

后来发现页面有许多定时器，ajax 轮询还有动画，打开一个浏览器页签没法问题，打开多了，浏览器就变得卡了，这时候我就想如果能在用户切换页签时候将这些都停掉，不久解决了。百度里面上下检索，找到了一个事件 `visibilitychange`，可以用来判断浏览器页签是否显示。

有方法了，就写呗

```
export default {
  created() {
    window.addEventListener('visibilitychange', this.$_handlevisiblityChange)
    // 此处用了hookEvent，可以参考小编前一篇文章
    this.$on('hook:beforeDestroy', () => {
      window.removeEventListener(
        'visibilitychange',
        this.$_handlevisiblityChange
      )
    })
  },
  methods: {
    $_handlevisiblityChange() {
      if (document.visibilityState === 'hidden') {
        // 停掉那一堆东西
      }
      if (document.visibilityState === 'visible') {
        // 开启那一堆东西
      }
    }
  }
}
```

复制代码

通过上面的代码，可以看到在每一个需要监听处理的文件都要写一堆事件监听，判断页面是否显示的代码，一处两处还可以，文件多了就头疼了，这时候小编突发奇想，定义一个页面显示隐藏的生命周期钩子，把这些判断都封装起来，哪里需要点哪里，so easy（点读机记得广告费）。

自定义生命周期钩子函数

定义生命周期函数 `pageHidden` 与 `pageVisible`

```
import vue from 'vue'

// 通知所有组件页面状态发生了变化
const notifyvisiblityChange = (lifeCycleName, vm) => {
  // 生命周期函数会存在$options中，通过$options[lifeCycleName]获取生命周期
  const lifeCycles = vm.$options[lifeCycleName]
  // 因为使用了created的合并策略，所以是一个数组
  if (lifeCycles && lifeCycles.length) {
    // 遍历 lifeCycleName对应的生命周期函数列表，依次执行
    lifeCycles.forEach(lifecycle => {
      lifecycle.call(vm)
    })
  }
  // 遍历所有的子组件，然后依次递归执行
  if (vm.$children && vm.$children.length) {
    vm.$children.forEach(child => {
      notifyvisiblityChange(lifeCycleName, child)
    })
  }
}
```



```

    }
  }

  /**
   * 添加生命周期钩子函数
   * @param {*} rootVm vue 根实例，在页面显示隐藏时候，通过root向下通知
   */
  export function init() {
    const optionMergeStrategies = Vue.config.optionMergeStrategies
    /**
     * 定义了两个生命周期函数 pageVisible, pageHidden
     * 为什么要赋值为 optionMergeStrategies.created呢
     * 这个相当于指定 pageVisible, pageHidden 的合并策略与 created的相同（其他生命周期函数都一样）
     */
    optionMergeStrategies.pageVisible = optionMergeStrategies.beforeCreate
    optionMergeStrategies.pageHidden = optionMergeStrategies.created
  }

  /**
   * 将事件变化绑定到根节点上面
   * @param {*} rootVm
   */
  export function bind(rootVm) {
    window.addEventListener('visibilitychange', () => {
      // 判断调用哪个生命周期函数
      let lifecycleName = undefined
      if (document.visibilityState === 'hidden') {
        lifecycleName = 'pageHidden'
      } else if (document.visibilityState === 'visible') {
        lifecycleName = 'pageVisible'
      }
      if (lifecycleName) {
        // 通过所有组件生命周期发生了变化
        notifyVisibilityChange(lifecycleName, rootVm)
      }
    })
  }
}

```

复制代码

应用

1. 在 main.js 主入口文件引入

```

import { init, bind } from './utils/custom-life-cycle'

// 初始化生命周期函数，必须在vue实例化之前确定合并策略
init()

const vm = new Vue({
  router,
  render: h => h(App)
}).$mount('#app')

// 将rootVm 绑定到生命周期函数监听里面
bind(vm)

```

复制代码

1. 在需要的地方监听生命周期函数

```
export default {
  pageVisible() {
    console.log('页面显示出来了')
  },
  pageHidden() {
    console.log('页面隐藏了')
  }
}
```

复制代码

provide与inject，不止父子传值，祖宗传值也可以

vue 相关的面试经常会被面试官问道，vue 父子之间传值的方式有哪些，通常会回答，props 传值，\$emit 事件传值，vuex 传值，还有 eventbus 传值等等，今天再加一种 provide 与 inject 传值，离 offer 又近了一步。（对了，下一节还有一种）

使用过 React 的同学都知道，在 React 中有一个上下文 Context，组件可以通过 Context 向任意后代传值，而 vue 的 provide 与 inject 的作用于 Context 的作用基本一样

先举一个例子

使用过 element-ui 的同学一定对下面的代码感到熟悉

```
<template>
  <el-form :model="formData" size="small">
    <el-form-item label="姓名" prop="name">
      <el-input v-model="formData.name" />
    </el-form-item>
    <el-form-item label="年龄" prop="age">
      <el-input-number v-model="formData.age" />
    </el-form-item>
    <el-button>提交</el-button>
  </el-form>
</template>
<script>
export default {
  data() {
    return {
      formData: {
        name: '',
        age: 0
      }
    }
  }
}
</script>
```

复制代码

看了上面的代码，貌似没啥特殊的，天天写啊。在 `el-form` 上面我们指定了一个属性 `size="small"`，然后有没有发现表单里面的所有表单元素以及按钮的 `size` 都变成了 `small`，这个是怎么做到的？接下来我们自己手写一个表单模拟一下

自己手写一个表单

我们现在模仿 `element-ui` 的表单，自己自定义一个，文件目录如下



自定义表单 `custom-form.vue`

```
<template>
  <form class="custom-form">
    <slot></slot>
  </form>
</template>
<script>
export default {
  props: {
    // 控制表单元素的大小
    size: {
      type: String,
      default: 'default',
      // size 只能是下面的四个值
      validator(value) {
        return ['default', 'large', 'small', 'mini'].includes(value)
      }
    },
    // 控制表单元素的禁用状态
    disabled: {
      type: Boolean,
      default: false
    }
  },
  // 通过provide将当前表单实例传递到所有后代组件中
  provide() {
    return {
      customForm: this
    }
  }
}
</script>
```

复制代码

在上面代码中，我们通过 `provide` 将当前组件的实例传递到后代组件中，`provide` 是一个函数，函数返回的是一个对象

自定义表单项 `custom-form-item.vue`

没有什么特殊的，只是加了一个 `label`，`element-ui` 更复杂一些

```
<template>
  <div class="custom-form-item">
    <label class="custom-form-item__label">{{ label }}</label>
```

```

      <div class="custom-form-item__content">
        <slot></slot>
      </div>
    </div>
  </template>
  <script>
  export default {
    props: {
      label: {
        type: String,
        default: ''
      }
    }
  }
  </script>

```

复制代码

自定义输入框 custom-input.vue

```

<template>
  <div
    class="custom-input"
    :class="[
      `custom-input--${getSize}`,
      getDisabled && `custom-input--disabled`
    ]"
  >
    <input class="custom-input__input" :value="value" @input="$_handleChange" />
  </div>
</template>
<script>
/* eslint-disable vue/require-default-prop */
export default {
  props: {
    // 这里用了自定义v-model
    value: {
      type: String,
      default: ''
    },
    size: {
      type: String
    },
    disabled: {
      type: Boolean
    }
  },
  // 通过inject 将form组件注入的实例添加进来
  inject: ['customForm'],
  computed: {
    // 通过计算组件获取组件的size，如果当前组件传入，则使用当前组件的，否则是否form组件的
    getSize() {
      return this.size || this.customForm.size
    },
    // 组件是否禁用
    getDisabled() {
      const { disabled } = this

```

```

        if (disabled !== undefined) {
          return disabled
        }
        return this.customForm.disabled
      }
    },
    methods: {
      // 自定义v-model
      $_handleChange(e) {
        this.$emit('input', e.target.value)
      }
    }
  }
}
</script>

```

复制代码

在 `form` 中，我们通过 `provide` 返回了一个对象，在 `input` 中，我们可以通过 `inject` 获取 `form` 中返回对象中的项，如上代码 `inject: ['customForm']` 所示，然后就可以在组件内通过 `this.customForm` 调用 `form` 实例上面的属性与方法了

在上面代码中我们使用了自定义 `v-model`，关于自定义 `v-model` 可以阅读小编前面的文章[绝对干货~！学会这些Vue小技巧，可以早点下班和女神约会了](#)

在项目中使用

```

<template>
  <custom-form size="small">
    <custom-form-item label="姓名">
      <custom-input v-model="formData.name" />
    </custom-form-item>
  </custom-form>
</template>
<script>
import CustomForm from '../components/custom-form'
import CustomFormItem from '../components/custom-form-item'
import CustomInput from '../components/custom-input'
export default {
  components: {
    CustomForm,
    CustomFormItem,
    CustomInput
  },
  data() {
    return {
      formData: {
        name: '',
        age: 0
      }
    }
  }
}
</script>

```

复制代码

执行上面代码，运行结果为：

```
<form class="custom-form">
  <div class="custom-form-item">
    <label class="custom-form-item__label">姓名</label>
    <div class="custom-form-item__content">
      <!--size=small已经添加到指定的位置了-->
      <div class="custom-input custom-input--small">
        <input class="custom-input__input">
      </div>
    </div>
  </div>
</form>
```

复制代码

通过上面的代码可以看到，`input` 组件已经设置组件样式为 `custom-input--small` 了

inject 格式说明

除了上面代码中所使用的 `inject:['customForm']` 写法之外，`inject` 还可以是一个对象。且可以指定默认值

修改上例，如果 `custom-input` 外部没有 `custom-form`，则不会注入 `customForm`，此时为 `customForm` 指定默认值

```
{
  inject: {
    customForm: {
      // 对于非原始值，和props一样，需要提供一个工厂方法
      default: () => ({
        size: 'default'
      })
    }
  }
}
```

复制代码

如果我们希望 `inject` 进来的属性的名字不叫 `customForm`，而是叫 `parentForm`，如下代码

```
inject: {
  // 注入的属性名称
  parentForm: {
    // 通过 from 指定从哪个属性注入
    from: 'customForm',
    default: () => ({
      size: 'default'
    })
  }
},
computed: {
  // 通过计算组件获取组件的size，如果当前组件传入，则使用当前组件的，否则是否form组件的
  getSize() {
    return this.size || this.parentForm.size
  }
}
```

复制代码

使用限制

1. `provide` 和 `inject` 的绑定不是可响应式的。但是，如果你传入的是一个可监听的对象，如上面的 `customForm: this`，那么其对象的属性还是可响应的。
2. `vue` 官网建议 `provide` 和 `inject` 主要在开发高阶插件/组件库时使用。不推荐用于普通应用程序代码中。因为 `provide` 和 `inject` 在代码中是不可追溯的(`ctrl + f`可以搜)，建议可以使用 `vuex` 代替。但是，也不是说不能用，在局部功能有时候用了作用还是比较大的。

dispatch 和 broadcast ,这是一种有历史的组件通信方式

`$dispatch` 与 `$broadcast` 是一种有历史的组件通信方式，为什么是有历史的，因为他们是 `vue1.0` 提供的一种方式，在 `vue2.0` 中废弃了。但是废弃了不代表我们不能自己手动实现，像许多UI库内部都有实现。本文以 `element-ui` 实现为基础进行介绍。同时看完本节，你会对组件的 `$parent`, `$children`, `$options` 有所了解。

方法介绍

`$dispatch`: `$dispatch` 会向上触发一个事件，同时传递要触发的祖先组件的名称与参数，当事件向上传递到对应的组件上时会触发组件上的事件侦听器，同时传播会停止。

`$broadcast`: `$broadcast` 会向所有的后代组件传播一个事件，同时传递要触发的后代组件的名称与参数，当事件传递到对应的后代组件时，会触发组件上的事件侦听器，同时传播会停止（因为向下传递是树形的，所以只会停止其中一个叶子分支的传递）。

`$dispatch` 实现与应用

1. 代码实现

```
/**
 * 向上传播事件
 * @param {*} eventName 事件名称
 * @param {*} componentName 接收事件的组件名称
 * @param {...any} params 传递的参数,可以有多个
 */
function dispatch(eventName, componentName, ...params) {
  // 如果没有$parent，则取$root
  let parent = this.$parent || this.$root
  while (parent) {
    // 组件的name存储在组件的$options.componentName 上面
    const name = parent.$options.name
    // 如果接收事件的组件是当前组件
    if (name === componentName) {
      // 通过当前组件上面的$emit触发事件,同事传递事件名称与参数
      parent.$emit.apply(parent, [eventName, ...params])
      break
    } else {
      // 否则继续向上判断
      parent = parent.$parent
    }
  }
}

// 导出一个对象，然后在需要用到的地方通过混入添加
export default {
```

```

    methods: {
      $dispatch: dispatch
    }
  }
}

```

复制代码

2. 代码应用

在子组件中通过 `$dispatch` 向上触发事件

```

import emitter from '../mixins/emitter'
export default {
  name: 'Chart',
  // 通过混入将$dispatch加入进来
  mixins: [emitter],
  mounted() {
    // 在组件渲染完之后，将组件通过$dispatch将自己注册到Board组件上
    this.$dispatch('register', 'Board', this)
  }
}

```

复制代码

在 Board 组件上通过 `$on` 监听要注册的事件

```

export default {
  name: 'Board',
  created() {
    this.$on('register', (component) => {
      // 处理注册逻辑
    })
  }
}

```

复制代码

`$broadcast` 实现与应用

1. 代码实现

```

/**
 * 向下传播事件
 * @param {*} eventName 事件名称
 * @param {*} componentName 要触发组件的名称
 * @param {...any} params 传递的参数
 */
function broadcast(eventName, componentName, ...params) {
  this.$children.forEach(child => {
    const name = child.$options.name
    if (name === componentName) {
      child.$emit.apply(child, [eventName, ...params])
    } else {
      broadcast.apply(child, [eventName, componentName, ...params])
    }
  })
}

```



```
// 导出一个对象，然后在需要用到的地方通过混入添加
export default {
  methods: {
    $broadcast: broadcast
  }
}
```

复制代码

2. 代码应用

在父组件中通过 `$broadcast` 向下触发事件

```
import emitter from '../mixins/emitter'
export default {
  name: 'Board',
  // 通过混入将$dispatch加入进来
  mixins: [emitter],
  methods: {
    //在需要的时候，刷新组件
    $_refreshChildren(params) {
      this.$broadcast('refresh', 'Chart', params)
    }
  }
}
```

复制代码

在后代组件中通过 `$on` 监听刷新事件

```
export default {
  name: 'Chart',
  created() {
    this.$on('refresh', (params) => {
      // 刷新事件
    })
  }
}
```

复制代码

总结

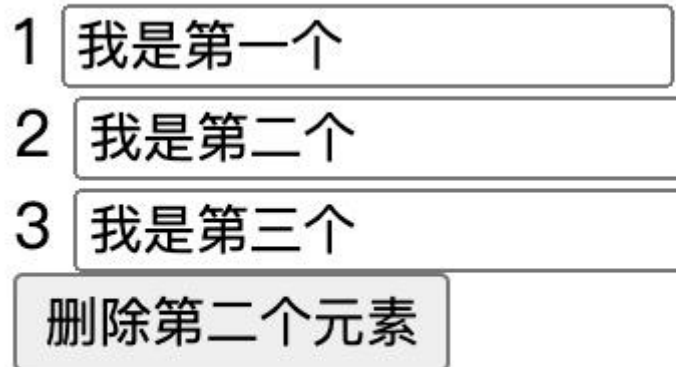
通过上面的例子，同学们应该都能对 `$dispatch` 和 `$broadcast` 有所了解，但是为什么 `vue2.0` 要放弃这两个方法呢？官方给出的解释是：“因为基于组件树结构的事件流方式实在是让人难以理解，并且在组件结构扩展的过程中会变得越来越脆弱。这种事件方式确实不太好，我们也不希望在以后让开发者们太痛苦。并且 `$dispatch` 和 `$broadcast` 也没有解决兄弟组件间的通信问题。”

确实如官网所说，这种事件流的方式确实不容易让人理解，而且后期维护成本比较高。但是在小编看来，不管黑猫白猫，能抓老鼠的都是好猫，在许多特定的业务场景中，因为业务的复杂性，很有可能使用到这样的通信方式。但是使用归使用，但是不能滥用，小编一直就在项目中有使用。

v-for设置键值

提到 `v-for` 需要设置键值，许多人第一反应会从 `diff` 算法的角度去讲原因，我更喜欢举一个例子来演示一下原因

假设有这样的一个页面，页面的列表是通过遍历数组得来的，如下图所示



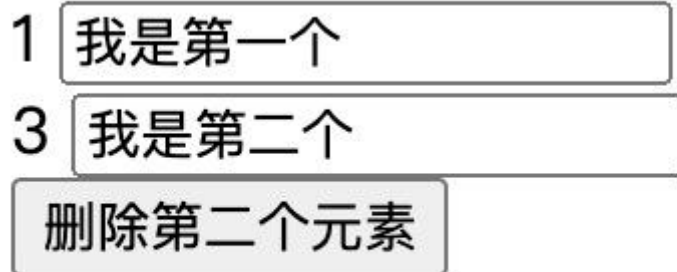
示例代码如下

```
<!--模板部分-->
<div id="app">
  <div v-for="item in arr">
    {{item}}
    <input/>
  </div>

  <button @click="deleteData">删除第二个元素</button>
</div>
复制代码
// js 部分
new Vue({
  el: '#app',
  data() {
    return {
      arr: [1,2,3]
    }
  },
  methods:{
    deleteData() {
      this.arr.splice(1,1)
    }
  }
})
复制代码
```

现在需要删除第二个元素。下面我们分别在渲染列表是 不使用key,使用索引作为key, 使用唯一值id作为key,看三种场景删除第二个元素之后的效果

- `v-for` 不使用 key [点击查看代码演示](#)



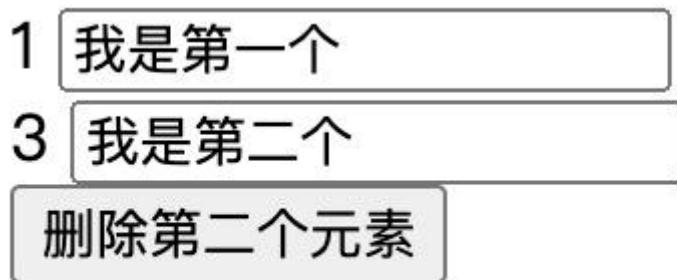
可以看到，不使用key，删除第二个元素之后，输入框前面的数字显示正确的，但是数字3后面的输入框的内容显示错了，应该显示 *我是第三个*

- `v-for`

使用索引作为

`key`

[点击查看代码演示](#)



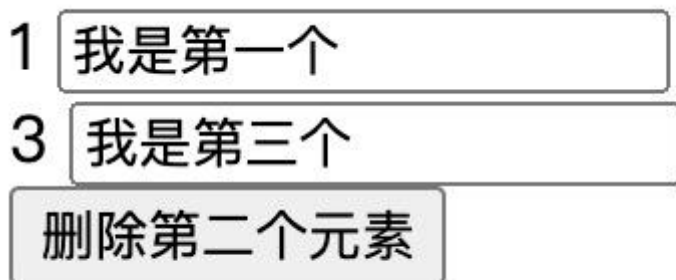
可以看到，使用索引作为

`key`

之后，与不使用key的效果一样，删除第二个元素之后，输入框前面的数字显示正确的，但是数字3后面的输入框的内容显示错了，应该显示

我是第三个

- `v-for` 使用唯一值id作为 `key` [点击查看代码演示](#)



使用id作为 key,显示正确

为什么 v-for 需要设置key, 原因很简单。对比数组 [1,2,3]和[1,3], 我们很容易发现删掉了2, 但是计算机不是这样的逻辑

1. 计算机对比新旧数组, 发现1===1, 保持不变
2. 然后再对比2, 发现2变成了3, 那么就把2修改为3, 原来第二行的元素都可以复用, 只把数字改一下就可以了
3. 然后在对比3与undefined, 发现3被删了, 索引把第三行的元素删掉

那么为什么不能用索引作为key呢? 当删掉[1,2,3]中的2之后, 数组的长度由3变成了2, 那么原来数字3的索引就变成了数字2的索引了。

1. 计算机对比key为0的值, 发现都是1, 保持不变
2. 计算机对比key为1的值, 发现从2变成了3, 元素复用, 修改元素上面的文字
3. 计算机对比key为2的值, 发现被删掉了, 所以删掉第三行元素

而对于使用id作为key,那么每条数据都有了唯一的标识, 当删掉 [{id: '1',value: 1},{id: '2',value: 2}, {id: '3', value:3}] 中的第二个, 整个过程如下

1. 计算机取出新数据第一项的id, 然后在原来数据里面寻找, 发现存在相同id的数据, 而且数据没有变化, 所以保持不变
2. 计算机继续取第二项的id, 发现是3, 然后从原来数据里面也找到了3, 所以3保留
3. 这时候旧数据里面剩了id为2的数据, 而新的里面没有了, 所以删掉。

没得问题嘛!!!

模板中的复杂逻辑使用计算属性代替

vue在模板可以使用表达式是非常方便的, 但表达式在设计之初是为了进行简单逻辑处理的, 如果在模板中使用太多或太复杂的逻辑, 会让模板的可读性和可维护性变得很差, 整个模板显得很臃肿。

Bad

```
<!--v-if使用了一连串的条件判断,可读性比较差-->
<button
  v-if="
    user.roles &&
    user.roles.includes('workflowbeheer') &&
    data.userId === user.id &&
    data.status === 1
  "
>
  删除
</button>
```

复制代码

Good

```
<template>
  <button v-if="deletable">
    删除
  </button>
</template>
<script>
export default {
```

```

computed: {
  // 判断是否可以删除
  deletable() {
    const { data, user } = this
    // 如果当前用户不是流程管理员，则不能编辑
    if (user.roles && user.roles.includes('workflowbeheer')) {
      // 如果当前用户为流程发起者且状态为未启动，则可以删除
      return data.userId === user.id && data.status === 1
    }
    return false
  }
}
}
</script>
复制代码

```

避免v-for与v-if混用

永远不要将 *v-for* 和 *v-if* 同时用在同一个元素上。

在开发vue项目中，大家可能会遇到这样的代码

```

<ul>
  <li v-for="item in list" v-if="item.visible" :key="item.id">
    {{ item.name }}
  </li>
</ul>
复制代码

```

如果在项目中启用了 `eslint`，则可能会看到下面这样的异常提示(需要启用 `eslint vue/no-use-v-if-with-v-for` 规则)

```

The 'list' variable inside 'v-for' directive should be replaced with a computed
property that returns filtered array instead. You should not mix 'v-for' with
'v-if'.
复制代码

```

在vue处理指令的时候，`v-for` 比 `v-if` 会有更高的优先级，那么上述的代码用js可以模拟为

```

list.map(item => {
  if(item.visible) {
    return item.name
  }
})
复制代码

```

通过上述代码可以看到，即使大部分数据的visible都是false,也会将整个list全部遍历一次。如果每一次都需要遍历整个数组，将会影响速度，尤其是当之需要渲染很小一部分的时候。

对于上述的问题，可以使用计算属性来处理

```

<ul>
  <li v-for="item in getList" :key="item.id">
    {{ item.name }}

```

```

    </li>
  </ul>

  computed: {
    getList() {
      return this.list.filter(item => {
        return item.visible
      })
    }
  }
}

```

复制代码

通过上述的代码，我们可以获得以下好处

- 过滤后的列表只会在 list 数组发生相关变化时才被重新运算，过滤更高效。
- 使用 `v-for="item in list"` 之后，我们在渲染的时候只遍历需要显示的数据，渲染更高效。
- 解耦渲染层的逻辑,可维护性比较高。

尽量使用私有属性/方法

在开发vue组件的时候，我们可以通过组件的 `ref` 来调用组件对外提供的方法，但是一个组件内部有些方法是内部私有的，不应该被外部调用，但实际上js中并没有像其他语言一样有私有方法（比如 `java` 的 `private`），所以在js中一般约定使用 `_` 开头的属性或者方法表示私有的。

```

{
  // 公共的
  selectRows(rows) {},
  // 私有的 外部虽然可以调用到，但是因为是以`_`开头，所以按照约定不应该去调用
  _select(rows){}
}

```

复制代码

在vue中定义私有属性/方法又与js常规约定有所不同。在Vue内部，已经使用了 `_` 开头去定义Vue原型上面的私有属性/方法，如果在组件内上面继续使用 `_` 开头去定义私有属性/方法可能会出现覆盖实例上面属性/方法的情况,比如下面这种情况:

```

methods: {
  // 初始化组件的数据方法
  _init() {
    fetch().then(data => {

    })
  }
}

```

复制代码

上面的代码看似没有问题，实际上运行的时候会报错，因为 `_init` 方法会覆盖 `vue.prototype` 上面的同名方法,如下图为Vue原型链的方法，第一个便是 `_init`



在Vue2.0风格指南中，建议使用 `$_` 来定义私有方法，可以确保不会和Vue自身发生冲突。修改上例为

```
methods: {
  // 初始化组件的数据方法
  $_init() {
    fetch().then(data => {

    })
  }
}
```

复制代码

组件数据必须是一个函数，并返回一个对象

在说为什么组件的数据必须返回一个函数之前，我们先来了解一下js中的基本类型与引用类型。

1. 基本类型

在es2020发布了bigint类型之后，js中的基本类型一种包含七种，分别是

- string 字符类型
- number 数值类型
- boolean 布尔类型
- undefined
- null
- Symbol
- BigInt

基本类型的特点包括

- 基本类型的值是存放到栈内存里面的
- 基本类型的比较是它们的值的比较
- 基本类型的值是不可变的，对值的修改会在栈内存中开辟新的空间
- 基本类型上面不能挂载新的属性

```
let a = 2
let b = a
// 对a的值的修改，会在栈内存开辟新的空间，所以不会影响到b的值
a = 3
// 输出 3 2
console.log(a,b)

// 不能给基本类型上面挂载新的属性
a.testProp = '挂载的属性'
// 输出undefined
console.log(a.testProp)
```

复制代码

1. 引用类型 在js中，除了八种基本类型，其他都属于引用类型，像 `Object`, `Array`, `Function`, `RegExp`, `Date` 等等

引用类型的特点包括

- 引用类型的值保存在堆内存中，而引用保存到栈内存中
- 引用类型的值是按引用访问的
- 引用类型的值是可变的（在堆内存中直接修改）
- 引用类型上面可以挂载新的属性

```
let obj1 = {a: 1, b: 2}
let obj2 = obj1
// 因为引用类型的值是保存到堆内存的，obj1与obj2引用的是同一块堆内存空间，所以对obj1的值进行
// 修改，会直接影响到obj2的值
obj1.a = 3
// 输出 3
console.log(obj2.a)

// 挂载新的属性
obj1.testProp = '挂载的新属性值'
// 输出 "挂载的新属性值"
console.log(obj1.testProp)
复制代码
```

通过上面的对比，我想大家其实也清楚了为什么vue的数据必须返回一个函数了。

假设我们现在开发了一个组件，组件上面的data是一个普通的对象，那么当我们实例化多个组件的时候，所有的实例将共享引用同一个数据对象，任何一个实例对数据的修改都会影响到其他实例。而将组件上面的数据定义为一个函数之后，当实例化多个组件的时候，每个实例通过调用data函数，从而返回初始数据的一个全新副本数据对象，这时候就避免了对象引用。

为组件样式设置作用域

在前端发展日新月异的今天，所有的一切都在飞速的发展，前端项目规模越来越大，而css作为一个只有全局作用域的语言，样式冲突会带来很多麻烦。JS语言模块已经标准化，CSS还是在不断探索，同时这也是一个急需解决的问题。现在人们提出了许多为css添加作用域的解决方法，比如BEM样式规范，比如css module。

在Vue中，使用了通过给元素添加 `scoped attribute` 的方式为css添加作用域，具体代码如下

```
<template>
  <button class="button">按钮</button>
</template>
<!--给style标签添加scoped属性-->
<style scoped>
.button {
  width: 50px;
  height: 40px;
}
</style>
复制代码
```

编译之后的结果如下

```
<!--html 添加了一个新属性 data-v-039c5b43，对于组件内的所有元素，都会添加同一个属性data-v-039c5b43,这样保证了同一个组件内所有元素都在同一个作用域内-->
<button data-v-039c5b43="" class="button">按钮</button>
复制代码
/*通过为样式添加属性选择器，去限制样式的作用域*/
.button[data-v-039c5b43] {
  width: 50px;
  height: 40px;
}
复制代码
```


虽然我们建议为组件样式添加作用域，但是不一定必须使用vue提供的 `attribute scoped`，对于组件库之类可能需要在外部覆盖样式，如果使用 `attribute scoped`，因为属性名不确定，且样式权重较高，导致样式覆盖很困难

这时候更建议使用类似BEM之类的命名规范来约束，这让覆写内部样式更容易，使用了常人可理解的 `class` 名称且没有太高的选择器优先级，而且不太会导致冲突。比如 `element ui` 和 `vant` 均使用了BEM

将复杂页面拆分成多个多个组件文件

你有没有见过一个Vue文件里面有一大坨密密麻麻的模板代码，模板代码里面还加载了大量的 `v-if`, `v-for`, `v-show` 之类的指令，我不知道你看到之后感觉怎么样，对于小编来说，这无疑是地狱，各种逻辑耦合到一起，改bug比蜀道还要难 对于一个复杂的页面，我们建议将页面按照模块/功能进行拆分，然后写成多个小的，单一的组件，然后在主入口文件中引用。比如，对于一个复杂的页面，我们可以拆分成

```
header.vue` `main.vue` `footer.vue
```

三个文件，然后在三个文件内完成各自的逻辑，最后通过将三个组件都引入主入口文件，来实现页面的拼装。这样做的好处包括

- 将复杂的逻辑进行解耦，代码结构更清晰，逻辑更简单，可读性更强
- 对功能进行组件化抽取抽象，组件复用变得更简单
- 便于多人协作开发，不同的人可以同时开发一个复杂的页面

prop应该尽量详细

对比下面的两段代码

```
// 第一段
export default {
  props:['status','data']
}
// 第二段
export default {
  props:{
    status: {
      type: Boolean,
      default: true
    },
    data:{
      type: Array,
      required: true
    }
  }
}
```

复制代码

对比上面两段代码，通过第二段代码我们可以很清楚的知道组件需要什么属性，属性的类型是什么，默认值是什么，是否是必须的，这样做的好处包括：

- 详细的定义了属性的各方面信息，所以很容易看懂组件的用法；
- 在开发环境下，如果向一个组件提供格式不正确的 `prop`，Vue将会得到警告，可以更快的发现潜在的问题。

组件名应该由多个单词组成

对于组件名应该由多个单词组成的必要性，我想到了自己曾经见过的一段代码

```
<header class="header">
  <!-- 栏目 -->
  <ul>
    <li>首页</li>
    <li>关于</li>
  </ul>
</header>
```

复制代码

看到这段代码，然后感觉很正常，没啥毛病，然后我看了一眼界面，诶，为什么header左侧有一个logo呢？我笑着说，这一定是样式里面加的咯，然后看了一眼样式，wtf,什么鬼，样式里面也没有加啊，这是怎么做到的，好神奇。后来就看到了这样的一段代码

```
import Header from '@/components/header'
export default {
  components: {
    Header
  }
}
```

作者：前端进击者

链接：<https://juejin.im/post/5edafece51882542e3023545>

来源：掘金

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。