> `hookEvent`,原来可以这样监听组件生命周期
>
> ”

1. 内部监听生命周期函数

今天产品经理又给我甩过来一个需求，需要开发一个图表，拿到需求，瞄了一眼，然后我就去`echarts`官网复制示例代码了，复制完改了改差不多了，改完代码长这样

```
<template>
  <div class="echarts"></div>
</template>
<script>
export default {
  mounted() {
    this.chart = echarts.init(this.$el)
    // 请求数据，赋值数据 等等一系列操作...
    // 监听窗口发生变化，resize组件
    window.addEventListener('resize', this.$_handleResizeChart)
  },
  updated() {
    // 干了一堆活
  },
  created() {
    // 干了一堆活
  },
  beforeDestroy() {
    // 组件销毁时，销毁监听事件
    window.removeEventListener('resize', this.$_handleResizeChart)
  },
  methods: {
    $_handleResizeChart() {
      this.chart.resize()
    },
    // 其他一堆方法
  }
}
</script>
复制代码
```

功能写完开开心心的提测了，测试没啥问题，产品经理表示做的很棒。然而code review时候，技术大佬说了，这样有问题。

```
 大佬：这样写不是很好，应该将监听`resize`事件与销毁`resize`事件放到一起，现在两段代码分开而且相隔几百行代码，可读性比较差

 我：那我把两个生命周期钩子函数位置换一下，放到一起?

 大佬： `hook`听过没?

 我：`Vue3.0`才有啊，咋，咱要升级`Vue`?
复制代码
```

然后技术大佬就不理我了,并向我扔过来一段代码

```
export default {
  mounted() {
    this.chart = echarts.init(this.$el)
    // 请求数据，赋值数据 等等一系列操作...

    // 监听窗口发生变化，resize组件
    window.addEventListener('resize', this.$_handleResizeChart)
    // 通过hook监听组件销毁钩子函数，并取消监听事件
    this.$once('hook:beforeDestroy', () => {
      window.removeEventListener('resize', this.$_handleResizeChart)
    })
  },
  updated() {},
  created() {},
  methods: {
    $_handleResizeChart() {
      // this.chart.resize()
    }
  }
}
复制代码
```

看完代码，恍然大悟，大佬不愧是大佬，原来`Vue`还可以这样监听生命周期函数。

在`Vue`组件中，可以用过`$on`,`$once`去监听所有的生命周期钩子函数，如监听组件的`updated`钩子函数可以写成`this.$on('hook:updated', () => {})`

2. 外部监听生命周期函数

今天同事在公司群里问，想在外部监听组件的生命周期函数，有没有办法啊?

为什么会有这样的需求呢，原来同事用了一个第三方组件，需要监听第三方组件数据的变化，但是组件又没有提供`change`事件，同事也没办法了，才想出来要去在外部监听组件的`updated`钩子函数。查看了一番资料，发现`Vue`支持在外部监听组件的生命周期钩子函数。

```
<template>
  <!--通过@hook:updated监听组件的updated生命钩子函数-->
  <!--组件的所有生命周期钩子都可以通过@hook:钩子函数名 来监听触发-->
  <custom-select @hook:updated="$_handleSelectUpdated" />
</template>
<script>
import CustomSelect from '../components/custom-select'
export default {
  components: {
    CustomSelect
  },
  methods: {
    $_handleSelectUpdated() {
      console.log('custom-select组件的updated钩子函数被触发')
    }
  }
}
</script>

复制代码
```

> “
>
> ## 小项目还用Vuex?用Vue.observable手写一个状态管理吧
>
> ”

在前端项目中，有许多数据需要在各个组件之间进行传递共享，这时候就需要有一个状态管理工具，一般情况下，我们都会使用Vuex，但对于小型项目来说，就像Vuex官网所说：“如果您不打算开发大型单页应用，使用 Vuex 可能是繁琐冗余的。确实是如此——如果您的应用够简单，您最好不要使用 Vuex”。这时候我们就可以使用Vue2.6提供的新API Vue.observable手动打造一个Vuex

创建 store

```
import Vue from 'vue'

// 通过Vue.observable创建一个可响应的对象
export const store = Vue.observable({
  userInfo: {},
  roleIds: []
})

// 定义 mutations, 修改属性
export const mutations = {
  setUserInfo(userInfo) {
    store.userInfo = userInfo
  },
  setRoleIds(roleIds) {
    store.roleIds = roleIds
  }
}

复制代码
```

在组件中引用

```
<template>
  <div>
    {{ userInfo.name }}
  </div>
</template>
<script>
import { store, mutations } from '../store'
export default {
  computed: {
    userInfo() {
      return store.userInfo
    }
  },
  created() {
    mutations.setUserInfo({
      name: '子君'
    })
  }
}
```

```
</script>
复制代码
```

> “
>
> ## 开发全局组件，你可能需要了解一下`Vue.extend`
>
> ”

`Vue.extend`是一个全局Api,平时我们在开发业务的时候很少会用到它，但有时候我们希望可以开发一些全局组件比如`Loading`,`Notify`,`Message`等组件时，这时候就可以使用`Vue.extend`。

同学们在使用`element-ui`的`loading`时，在代码中可能会这样写

```
// 显示loading
const loading = this.$loading()
// 关闭loading
loading.close()
复制代码
```

这样写可能没什么特别的，但是如果你这样写

```
const loading = this.$loading()
const loading1 = this.$loading()
setTimeout(() => {
  loading.close()
}, 1000 * 3)
复制代码
```

这时候你会发现，我调用了两次loading,但是只出现了一个，而且我只关闭了`loading`，但是`loading1`也被关闭了。这是怎么实现的呢？我们现在就是用`Vue.extend` + 单例模式去实现一个`loading`

开发`loading`组件

```
<template>
  <transition name="custom-loading-fade">
    <!--loading蒙版-->
    <div v-show="visible" class="custom-loading-mask">
      <!--loading中间的图标-->
      <div class="custom-loading-spinner">
        <i class="custom-spinner-icon"></i>
        <!--loading上面显示的文字-->
        <p class="custom-loading-text">{{ text }}</p>
      </div>
    </div>
  </transition>
</template>
<script>
export default {
  props: {
  // 是否显示loading
    visible: {
      type: Boolean,
```

```
      default: false
    },
    // loading上面的显示文字
    text: {
      type: String,
      default: ''
    }
  }
}
</script>
复制代码
```

开发出来`loading`组件之后，如果需要直接使用，就要这样去用

```
<template>
  <div class="component-code">
    <!--其他一堆代码-->
    <custom-loading :visible="visible" text="加载中" />
  </div>
</template>
<script>
export default {
  data() {
    return {
      visible: false
    }
  }
}
</script>
复制代码
```

但这样使用并不能满足我们的需求

1. 可以通过js直接调用方法来显示关闭
2. `loading`可以将整个页面全部遮罩起来

通过`Vue.extend`将组件转换为全局组件

1. 改造`loading`组件，将组件的`props`改为`data`

```
export default {
  data() {
    return {
      text: '',
      visible: false
    }
  }
}
复制代码
```

2. 通过`Vue.extend`改造组件

```
// loading/index.js
import Vue from 'vue'
import LoadingComponent from './loading.vue'
```

```
// 通过Vue.extend将组件包装成一个子类
const LoadingConstructor = Vue.extend(LoadingComponent)

let loading = undefined

LoadingConstructor.prototype.close = function() {
  // 如果loading 有引用，则去掉引用
  if (loading) {
    loading = undefined
  }
  // 先将组件隐藏
  this.visible = false
  // 延迟300毫秒，等待loading关闭动画执行完之后销毁组件
  setTimeout(() => {
    // 移除挂载的dom元素
    if (this.$el && this.$el.parentNode) {
      this.$el.parentNode.removeChild(this.$el)
    }
    // 调用组件的$destroy方法进行组件销毁
    this.$destroy()
  }, 300)
}

const Loading = (options = {}) => {
  // 如果组件已渲染，则返回即可
  if (loading) {
    return loading
  }
  // 要挂载的元素
  const parent = document.body
  // 组件属性
  const opts = {
    text: '',
    ...options
  }
  // 通过构造函数初始化组件 相当于 new Vue()
  const instance = new LoadingConstructor({
    el: document.createElement('div'),
    data: opts
  })
  // 将loading元素挂在到parent上面
  parent.appendChild(instance.$el)
  // 显示loading
  Vue.nextTick(() => {
    instance.visible = true
  })
  // 将组件实例赋值给loading
  loading = instance
  return instance
}

export default Loading
```


3. 在页面使用loading

```
import Loading from './loading/index.js'
export default {
  created() {
    const loading = Loading({ text: '正在加载。。。' })
    // 三秒钟后关闭
    setTimeout(() => {
      loading.close()
    }, 3000)
  }
}
复制代码
```

通过上面的改造，loading已经可以在全局使用了，如果需要像`element-ui`一样挂载到`Vue.prototype`上面，通过`this.$loading`调用，还需要改造一下

将组件挂载到`Vue.prototype`上面

```
Vue.prototype.$loading = Loading
// 在export之前将Loading方法进行绑定
export default Loading

// 在组件内使用
this.$loading()
复制代码
```

> “
>
> ## 自定义指令，从底层解决问题
>
> ”

什么是指令？指令就是你女朋友指着你说，“那边搓衣板，跪下，这是命令！”。开玩笑啦，程序员哪里会有女朋友。

通过上一节我们开发了一个`loading`组件，开发完之后，其他开发在使用的时候又提出来了两个需求

1. 可以将`loading`挂载到某一个元素上面，现在只能是全屏使用
2. 可以使用指令在指定的元素上面挂载`loading`

有需求，咱就做，没话说

开发`v-loading`指令

```
import Vue from 'vue'
import LoadingComponent from './loading'
// 使用 Vue.extend构造组件子类
const LoadingContructor = Vue.extend(LoadingComponent)

// 定义一个名为loading的指令
Vue.directive('loading', {
  /**
   * 只调用一次，在指令第一次绑定到元素时调用，可以在这里做一些初始化的设置
   * @param {*} el 指令要绑定的元素
   * @param {*} binding 指令传入的信息，包括 {name:'指令名称', value: '指令绑定的
值',arg: '指令参数 v-bind:text 对应 text'}
```

```
   */
  bind(el, binding) {
    const instance = new LoadingContructor({
      el: document.createElement('div'),
      data: {}
    })
    el.appendChild(instance.$el)
    el.instance = instance
    Vue.nextTick(() => {
      el.instance.visible = binding.value
    })
  },
  /**
   * 所在组件的 VNode 更新时调用
   * @param {*} el
   * @param {*} binding
   */
  update(el, binding) {
    // 通过对比值的变化判断loading是否显示
    if (binding.oldValue !== binding.value) {
      el.instance.visible = binding.value
    }
  },
  /**
   * 只调用一次，在 指令与元素解绑时调用
   * @param {*} el
   */
  unbind(el) {
    const mask = el.instance.$el
    if (mask.parentNode) {
      mask.parentNode.removeChild(mask)
    }
    el.instance.$destroy()
    el.instance = undefined
  }
})
```
复制代码

在元素上面使用指令

```
<template>
  <div v-loading="visible"></div>
</template>
<script>
export default {
  data() {
    return {
      visible: false
    }
  },
  created() {
    this.visible = true
    fetch().then(() => {
      this.visible = false
    })
  }
}
```

```
</script>

复制代码
```

项目中哪些场景可以自定义指令

1. 为组件添加`loading`效果
2. 按钮级别权限控制 `v-permission`
3. 代码埋点,根据操作类型定义指令
4. input输入框自动获取焦点
5. 其他等等。。。

> “
>
> ## 深度`watch`与`watch`立即触发回调,我可以监听到你的一举一动
>
> ”

在开发Vue项目时，我们会经常性的使用到`watch`去监听数据的变化，然后在变化之后做一系列操作。

基础用法

比如一个列表页，我们希望用户在搜索框输入搜索关键字的时候，可以自动触发搜索,此时除了监听搜索框的`change`事件之外，我们也可以通过`watch`监听搜索关键字的变化

```
<template>
  <!--此处示例使用了element-ui-->
  <div>
    <div>
      <span>搜索</span>
      <input v-model="searchValue" />
    </div>
    <!--列表，代码省略-->
  </div>
</template>
<script>
export default {
  data() {
    return {
      searchValue: ''
    }
  },
  watch: {
    // 在值发生变化之后，重新加载数据
    searchValue(newValue, oldValue) {
      // 判断搜索
      if (newValue !== oldValue) {
        this.$_loadData()
      }
    }
  },
  methods: {
    $_loadData() {
      // 重新加载数据，此处需要通过函数防抖
    }
  }
```

```
}
</script>
复制代码
```

立即触发

通过上面的代码，现在已经可以在值发生变化的时候触发加载数据了，但是如果要在页面初始化时候加载数据，我们还需要在`created`或者`mounted`生命周期钩子里面再次调用`$_loadData`方法。不过，现在可以不用这样写了，通过配置`watch`的立即触发属性，就可以满足需求了

```
// 改造watch
export default {
  watch: {
    // 在值发生变化之后，重新加载数据
    searchValue: {
    // 通过handler来监听属性变化，初次调用 newValue为""空字符串， oldValue为 undefined
      handler(newValue, oldValue) {
        if (newValue !== oldValue) {
          this.$_loadData()
        }
      },
      // 配置立即执行属性
      immediate: true
    }
  }
}
复制代码
```

深度监听（我可以看到你内心的一举一动）

一个表单页面，需求希望用户在修改表单的任意一项之后，表单页面就需要变更为被修改状态。如果按照上例中`watch`的写法，那么我们就需要去监听表单每一个属性，太麻烦了，这时候就需要用到`watch`的深度监听`deep`

```
export default {
  data() {
    return {
      formData: {
        name: '',
        sex: '',
        age: 0,
        deptId: ''
      }
    }
  },
  watch: {
    // 在值发生变化之后，重新加载数据
    formData: {
      // 需要注意，因为对象引用的原因， newValue和oldValue的值一直相等
      handler(newValue, oldValue) {
        // 在这里标记页面编辑状态
      },
      // 通过指定deep属性为true，watch会监听对象里面每一个值的变化
      deep: true
    }
  }
```

```
}
```
复制代码

随时监听，随时取消，了解一下 `$watch`

有这样一个需求，有一个表单，在编辑的时候需要监听表单的变化，如果发生变化则保存按钮启用，否则保存按钮禁用。这时候对于新增表单来说，可以直接通过`watch`去监听表单数据(假设是`formData`)，如上例所述，但对于编辑表单来说，表单需要回填数据，这时候会修改`formData`的值，会触发`watch`，无法准确的判断是否启用保存按钮。现在你就需要了解一下`$watch`

```
export default {
  data() {
    return {
      formData: {
        name: '',
        age: 0
      }
    }
  },
  created() {
    this.$_loadData()
  },
  methods: {
    // 模拟异步请求数据
    $_loadData() {
      setTimeout(() => {
        // 先赋值
        this.formData = {
          name: '子君',
          age: 18
        }
        // 等表单数据回填之后，监听数据是否发生变化
        const unwatch = this.$watch(
          'formData',
          () => {
            console.log('数据发生了变化')
          },
          {
            deep: true
          }
        )
        // 模拟数据发生了变化
        setTimeout(() => {
          this.formData.name = '张三'
        }, 1000)
      }, 1000)
    }
  }
}
```
复制代码

根据上例可以看到，我们可以在需要的时候通过`this.$watch`来监听数据变化。那么如何取消监听呢，上例中`this.$watch`返回了一个值`unwatch`,是一个函数，在需要取消的时候，执行`unwatch()`即可取消

本文使用 mdnice 排版

> “
>
> ## 函数式组件，函数是组件?
>
> ”

什么是函数式组件？函数式组件就是函数是组件，感觉在玩文字游戏。使用过`React`的同学，应该不会对函数式组件感到陌生。函数式组件，我们可以理解为没有内部状态，没有生命周期钩子函数，没有`this`(不需要实例化的组件)。

在日常写bug的过程中，经常会开发一些纯展示性的业务组件，比如一些详情页面，列表界面等，它们有一个共同的特点是只需要将外部传入的数据进行展现，不需要有内部状态，不需要在生命周期钩子函数里面做处理，这时候你就可以考虑使用函数式组件。

先来一个函数式组件的代码

```
export default {
  // 通过配置functional属性指定组件为函数式组件
  functional: true,
  // 组件接收的外部属性
  props: {
    avatar: {
      type: String
    }
  },
  /**
   * 渲染函数
   * @param {*} h
   * @param {*} context 函数式组件没有this, props, slots等都在context上面挂着
   */
  render(h, context) {
    const { props } = context
    if (props.avatar) {
      return <img src={props.avatar}></img>
    }
    return <img src="default-avatar.png"></img>
  }
}
复制代码
```

在上例中，我们定义了一个头像组件，如果外部传入头像，则显示传入的头像，否则显示默认头像。上面的代码中大家看到有一个render函数，这个是`Vue`使用`JSX`的写法，关于`JSX`,小编将在后续文章中会出详细的使用教程。

为什么使用函数式组件

1. 最主要最关键的原因是函数式组件不需要实例化，无状态，没有生命周期，所以渲染性能要好于普通组件
2. 函数式组件结构比较简单，代码结构更清晰

函数式组件与普通组件的区别

1. 函数式组件需要在声明组件是指定functional
2. 函数式组件不需要实例化，所以没有`this`,`this`通过`render`函数的第二个参数来代替
3. 函数式组件没有生命周期钩子函数，不能使用计算属性，watch等等

4. 函数式组件不能通过$emit对外暴露事件，调用事件只能通过`context.listeners.click`的方式调用外部传入的事件
5. 因为函数式组件是没有实例化的，所以在外部通过`ref`去引用组件时，实际引用的是`HTMLElement`
6. 函数式组件的`props`可以不用显示声明，所以没有在`props`里面声明的属性都会被自动隐式解析为`prop`,而普通组件所有未声明的属性都被解析到`$attrs`里面，并自动挂载到组件根元素上面(可以通过`inheritAttrs`属性禁止)

我不想用`JSX`，能用函数式组件吗?

在`Vue2.5`之前，使用函数式组件只能通过`JSX`的方式，在之后，可以通过模板语法来生命函数式组件

```
<!--在template 上面添加 functional属性-->
<template functional>
  <img :src="props.avatar ? props.avatar : 'default-avatar.png'" />
</template>
<!--根据上一节第六条，可以省略声明props-->
```