Composition API（三）
高级语法补充

王红元 coderwhy

生命周期钩子

- 我们前面说过 setup 可以用来替代 data 、 methods 、 computed 、watch 等等这些选项，也可以替代 生命周期钩子。
- 那么setup中如何使用生命周期函数呢？
  - 可以使用直接导入的 onX 函数注册生命周期钩子；

```
onMounted(() => {
  console.log("onMounted")
})

onUpdated(() => {
  console.log('onUpdate')
})

onUnmounted(() => {
  console.log('onUnmounted')
})
```

选项式 API	Hook inside `setup`
`beforeCreate`	Not needed*
`created`	Not needed*
`beforeMount`	`onBeforeMount`
`mounted`	`onMounted`
`beforeUpdate`	`onBeforeUpdate`
`updated`	`onUpdated`
`beforeUnmount`	`onBeforeUnmount`
`unmounted`	`onUnmounted`
`activated`	`onActivated`
`deactivated`	`onDeactivated`

TIP

因为 `setup` 是围绕 `beforeCreate` 和 `created` 生命周期钩子运行的，所以不需要显式地定义它们。换句话说，在这些钩子中编写的任何代码都应该直接在 `setup` 函数中编写。

Provide函数

- 事实上我们之前还学习过Provide和Inject，Composition API也可以替代之前的 Provide 和 Inject 的选项。
- 我们可以通过 provide来提供数据：
  - 可以通过 provide 方法来定义每个 Property；
  - provide可以传入两个参数：
    - ✓ name：提供的属性名称；
    - ✓ value：提供的属性值；

```
let counter = 100
let info = {
  name: "why",
  age: 10
}

provide("counter", counter)
provide("info", info)
```

Inject函数

- 在 后代组件 中可以通过 inject 来注入需要的属性和对应的值：
  - 可以通过 inject 来注入需要的内容；
  - inject可以传入两个参数：
    - ✓ 要 inject 的 property 的 name；
    - ✓ 默认值；

```
const counter = inject("counter")
const info = inject("info")
```

数据的响应式

- 为了增加 provide 值和 inject 值之间的响应性，我们可以在 provide 值时使用 ref 和 reactive。

```
let counter = ref(100)
let info = reactive({
  name: "why",
  age: 18
})
provide("counter", counter)
provide("info", info)
```

修改响应式Property

- 如果我们需要修改可响应的数据，那么最好是在数据提供的位置来修改：
  - 我们可以将修改方法进行共享，在后代组件中进行调用；

```
const changeInfo = () => {
  info.name = "coderwhy"
}
provide("changeInfo", changeInfo)
```

useCounter

- 我们先来对之前的counter逻辑进行抽取：

```
import { ref } from 'vue'

export function useCounter() {
  const counter = ref(0);

  const increment = () => counter.value++
  const decrement = () => counter.value--

  return {
    counter,
    increment,
    decrement
  }
}
```

useTitle

- 我们编写一个修改title的Hook：

```
import { ref, watch } from 'vue'

export function useTitle(title = '默认值') {
  const titleRef = ref(title);

  watch(titleRef, (newValue) => {
    document.title = newValue;
  }, {
    immediate: true
  })

  return titleRef;
}
```

useScrollPosition

- 我们来完成一个监听界面滚动位置的Hook：

```
import { ref } from "vue";

export function useScrollPosition() {
  const scrollX = ref(0)
  const scrollY = ref(0)

  document.addEventListener('scroll', () => {
    scrollX.value = window.scrollX
    scrollY.value = window.scrollY
  })

  return { scrollX, scrollY }
}
```

useMousePosition

- 我们来完成一个监听鼠标位置的Hook：

```
import { ref } from "vue";

export function useMousePosition() {
  const mouseX = ref(0)
  const mouseY = ref(0)

  window.addEventListener('mousemove', (event) => {
    mouseX.value = event.pageX
    mouseY.value = event.pageY
  })

  return { mouseX, mouseY }
}
```

useLocalStorage

- 我们来完成一个使用 localStorage 存储和获取数据的Hook：

```
import { ref, watch } from "vue"

export function useLocalStorage(key, defaultValue) {
  const data = ref(defaultValue)

  if (defaultValue) {
    window.localStorage.setItem(key, JSON.stringify(defaultValue))
  } else {
    data.value = JSON.parse(window.localStorage.getItem(key))
  }

  watch(data, () => {
    window.localStorage.setItem(key, JSON.stringify(data.value))
  })

  return data;
}
```

认识h函数

- Vue推荐在绝大数情况下**使用模板**来创建你的HTML，然后一些特殊的场景，你真的需要**JavaScript的完全编程的能力**，这个时候你可以使用 **渲染函数** ，它**比模板更接近编译器**；
  - 前面我们讲解过VNode和VDOM的改变：
  - Vue在生成真实的DOM之前，会将我们的节点转换成VNode，而VNode组合在一起形成一颗树结构，就是虚拟DOM（VDOM）；
  - 事实上，我们之前编写的 template 中的HTML 最终也是使用**渲染函数**生成对应的VNode；
  - 那么，如果你想充分的利用JavaScript的编程能力，我们可以自己来编写 createVNode 函数，生成对应的VNode；
- 那么我们应该怎么来做呢？**使用 h()函数：**
  - h() 函数是一个用于创建 vnode 的一个函数；
  - 其实更准备的命名是 createVNode() 函数，但是为了简便在Vue将之简化为 h() 函数；

h()函数 如何使用呢？

■ h()函数 如何使用呢？它接受三个参数：

```
// {String | Object | Function} tag
// 一个 HTML 标签名、一个组件、一个异步组件、或
// 一个函数式组件。
//
// 必需的。
'div',
```

```
// {Object} props
// 与 attribute、prop 和事件相对应的对象。
// 我们会在模板中使用。
//
// 可选的。
{},
```

```
// {String | Array | Object} children
// 子 VNodes，使用 `h()` 构建，
// 或使用字符串获取 "文本 Vnode" 或者
// 有插槽的对象。
//
// 可选的。
[
  'Some text comes first.',
  h('h1', 'A headline'),
  h(MyComponent, {
    someProp: 'foobar'
  })
]
```

■ 注意事项：

- 如果没有props，那么通常可以将children作为第二个参数传入；
- 如果会产生歧义，可以将null作为第二个参数传入，将children作为第三个参数传入；

h函数的基本使用

- h函数可以在两个地方使用：
  - render函数选项中；
  - setup函数选项中（setup本身需要是一个函数类型，函数再返回h函数创建的VNode）；

```
import { h } from 'vue'

export default {
  render() {
    return h('div', {class: "app"}, "Hello App")
  }
}
```

```
import { h } from 'vue'

export default {
  setup() {
    return () => h('div', {class: "app"}, "Hello App")
  }
}
```

h函数计数器案例

```
data() {
  return {
    counter: 0
  }
},
render() {
  return h(
    'div',
    {class: "app"},
    [
      h("h2", null, `当前计数:${this.counter}`),
      h("button", {
        onClick: () => this.counter++
      }, "+1"),
      h("button", {
        onClick: () => this.counter--
      }, "-1")
    ]
  )
}
```

函数组件和插槽的使用

```vue
<script>
  import { h } from "vue"

  export default {
    render() {
      return h(
        'div',
        {class: "hello-world"},
        [
          h("h2", null, "Hello World"),
          this.$slots.default ? this.$slots.default({info: "hahaha"})
                              : h("span", null, "我是默认值")
        ]
      )
    }
  }
</script>
```

```vue
export default {
  data() {
    return {
      counter: 0
    }
  },
  render() {
    return h(
      'div',
      {class: "app"},
      [
        h("h2", null, "app component"),
        h(HelloWorld, null, {
          default: props => h('span', `app 传入: ${props.info}`)
        })
      ]
    )
  }
}
```

jsx的babel配置

- 如果我们希望**在项目中使用jsx**，那么我们**需要添加对jsx的支持**：
  - jsx我们通常会通过Babel来进行转换（React编写的jsx就是通过babel转换的）；
  - 对于Vue来说，我们只需要在Babel中配置对应的插件即可；
- 安装**Babel支持Vue的jsx插件**：

```
npm install @vue/babel-plugin-jsx -D
```

- 在**babel.config.js配置文件**中配置插件：

```
module.exports = {
  presets: [
    '@vue/cli-plugin-babel/preset'
  ],
  plugins: [
    "@vue/babel-plugin-jsx"
  ]
}
```

```
<script>
  export default {
    render() {
      return (
        <div>Hello Jsx</div>
      )
    }
  }
</script>
```

jsx计数器案例

```jsx
export default {
  setup() {
    const counter = ref(0);
    const increment = () => counter.value++;
    const decrement = () => counter.value--;

    return {
      counter,
      increment,
      decrement
    }
  },

  render() {
    return (
      <div>
        <h2>当前计数: {this.counter}</h2>
        <button onClick={this.increment}>+1</button>
        <button onClick={this.decrement}>-1</button>
      </div>
    )
  }
}
```

jsx组件的使用

jsx语法 > HelloWorld.jsx > default

```jsx
export default {
  setup() {

  },
  render() {
    return (
      <div>
        <h2>Hello World</h2>
        <div className="content">
          {this.$slots.default ?
            this.$slots.default({name: "coderwhy"})
            : <span>我是默认值</span>}
        </div>
      </div>
    )
  }
}
```

```jsx
import HelloWorld from './HelloWorld.jsx';

export default {

  render() {
    return (
      <div>
        <HelloWorld>
          {{default: props => <button>{props.name}</button>}}
        </HelloWorld>
      </div>
    );
  },
};
```

认识自定义指令

- 在Vue的模板语法中我们学习过各种各样的指令：v-show、v-for、v-model等等，除了使用这些指令之外，**Vue也允许我们来自定义自己的指令。**
  - 注意：在Vue中，代码的复用和抽象主要还是通过组件；
  - 通常在某些情况下，你需要对DOM元素进行底层操作，这个时候就会用到自定义指令；
- **自定义指令分为两种：**
  - 自定义局部指令：组件中通过 directives 选项，只能在当前组件中使用；
  - 自定义全局指令：app的 directive 方法，可以在任意组件中被使用；
- **比如我们来做一个非常简单的案例：当某个元素挂载完成后可以自定获取焦点**
  - 实现方式一：如果我们使用默认的实现方式；
  - 实现方式二：自定义一个 v-focus 的局部指令；
  - 实现方式三：自定义一个 v-focus 的全局指令；

实现方式一：聚焦的默认实现

```vue
<template>
  <div>
    <input type="text" ref="inputRef">
  </div>
</template>

<script>
  import { ref, onMounted } from "vue";

  export default {
    setup() {
      const inputRef = ref(null);

      onMounted(() => {
        inputRef.value.focus();
      })

      return {
        inputRef
      }
    }
  }
</script>
```

实现方式二：局部自定义指令

- 实现方式二：自定义一个 v-focus 的局部指令
  - 这个自定义指令实现非常简单，我们只需要在组件选项中使用 directives 即可；
  - 它是一个对象，在对象中编写我们自定义指令的名称（注意：这里不需要加v-）；
  - 自定义指令有一个生命周期，是在组件挂载后调用的 mounted，我们可以在其中完成操作；

```
<script>
  export default {
    directives: {
      focus: {
        mounted(el) {
          el.focus()
        },
      }
    }
  }
</script>
```

方式三：自定义全局指令

- 自定义一个**全局的v-focus指令**可以让我们在任何地方直接使用

```
app.directive("focus", {
  mounted(el) {
    el.focus()
  }
})
```

指令的生命周期

- **一个指令定义的对象，Vue提供了如下的几个钩子函数：**
- created：在绑定元素的 attribute 或事件监听器被应用之前调用；
- beforeMount：当指令第一次绑定到元素并且在挂载父组件之前调用；
- mounted：在绑定元素的父组件被挂载后调用；
- beforeUpdate：在更新包含组件的 VNode 之前调用；
- updated：在包含组件的 VNode **及其子组件的 VNode** 更新后调用；
- beforeUnmount：在卸载绑定元素的父组件之前调用；
- unmounted：当指令与元素解除绑定且父组件已卸载时，只调用一次；

指令的参数和修饰符

- 如果我们指令需要**接受一些参数或者修饰符**应该如何操作呢？
  - info是参数的名称；
  - aaa-bbb是修饰符的名称；
  - 后面是传入的具体的值；
- 在我们的生命周期中，我们可以**通过 bindings 获取到对应的内容**：

```
<button v-why:info.aaa.bbb="{name: 'coderwhy', age: 18}">{{counter}}</button>
```

```
▼{dir: {…}, instance: Proxy, value: {…}, oldValue: undefined, arg: "info", …}
  arg: "info"
  ▶dir: {created: ƒ, beforeMount: ƒ, mounted: ƒ, beforeUpdate: ƒ, updated: ƒ, …}
  ▶instance: Proxy {…}
  ▶modifiers: {aaa: true, bbb: true}
  oldValue: undefined
  ▶value: {name: "coderwhy", age: 18}
  ▶__proto__: Object
```

自定义指令练习

- **自定义指令案例：时间戳的显示需求：**
  - 在开发中，大多数情况下从服务器获取到的都是时间戳；
  - 我们需要将时间戳转换成具体格式化的时间来展示；
  - 在Vue2中我们可以通过过滤器来完成；
  - 在Vue3中我们可以通过 计算属性（computed） 或者 自定义一个方法（methods） 来完成；
  - 其实我们还可以通过一个自定义的指令来完成；
- 我们来实现一个可以自动对**时间格式化的指令v-format-time**：
  - 这里我封装了一个函数，在首页中我们只需要调用这个函数并且传入app即可；
- 代码见下页：

时间格式化指令

```
import dayjs from 'dayjs';

export default function(app) {
  let format = "YYYY-MM-DD HH:mm:ss"
  app.directive("format-time", {
    created(el, bingings) {
      if (bingings.value) {
        format = bingings.value;
      }
    },
    mounted(el) {
      const textContent = el.textContent;
      let timestamp = parseInt(el.textContent);
      if (textContent.length === 10) {
        timestamp = timestamp * 1000;
      }
      console.log(timestamp);
      el.textContent = dayjs(timestamp).format(format);
    }
  });
}
```

认识Teleport

- 在组件化开发中，我们**封装一个组件A**，在**另外一个组件B中使用**：
  - 那么组件A中template的元素，会被挂载到组件B中template的某个位置；
  - 最终我们的应用程序会形成**一颗DOM树结构**；
- 但是某些情况下，我们希望**组件不是挂载在这个组件树上**的，可能是**移动到Vue app之外的其他位置**：
  - 比如移动到body元素上，或者我们有其他的div#app之外的元素上；
  - 这个时候我们就可以通过teleport来完成；
- **Teleport是什么呢？**
  - 它是一个Vue提供的内置组件，类似于react的Portals；
  - teleport翻译过来是心灵传输、远距离运输的意思；
    - ✓ 它有两个属性：
      - ➢ to：指定将其中的内容移动到的目标元素，可以使用选择器；
      - ➢ disabled：是否禁用 teleport 的功能；

我们来看下面代码的效果：

```
<template>
  <div class="my-app">
    <teleport to="body">
      <h2>coderwhy</h2>
    </teleport>
  </div>
</template>
```

```
<body> == $0
  <noscript>…</noscript>
  <div id="app" data-v-app>…</div>
  <!-- built files will be auto injected -->
  <script type="text/javascript" src="/js/chunk-vendors.js"></scr
  <script type="text/javascript" src="/js/app.js"></script>
  <h2 data-v-16338632>coderwhy</h2>
</body>
```

和组件结合使用

- 当然，**teleport也可以和组件结合一起来使用：**
  - 我们可以在 teleport 中使用组件，并且也可以给他传入一些数据；

```
<template>
  <div class="my-app">
    <teleport to="body">
      <hello-world message="我是App中的message"></hello-world>
    </teleport>
  </div>
</template>
```

```
<body> == $0
  <noscript>…</noscript>
  <div id="app" data-v-app>…</div>
  <!-- built files will be auto injected -->
  <script type="text/javascript" src="/js/chunk-vendors.js"></sc
  <script type="text/javascript" src="/js/app.js"></script>
  <div data-v-6d1ebc5f data-v-16338632>
    <h2 data-v-6d1ebc5f>HelloWorld</h2>
    <p data-v-6d1ebc5f>我是coderwhy, 哈哈哈</p>
    <p data-v-6d1ebc5f>App传入:我是App中的message</p>
  </div>
</body>
```

多个teleport

- 如果我们将**多个teleport应用**到**同一个目标上（to的值相同）**，那么这些**目标会进行合并**：

```
<template>
  <teleport to="#why">
    <h2>coderwhy</h2>
  </teleport>
  <teleport to="#why">
    <hello-world message="我是App中的message"></hello-world>
  </teleport>
</template>
```

- 实现效果如下：

```
▼ <div id="why">
    <h2 data-v-16338632>coderwhy</h2> == $0
  ▼ <div data-v-6d1ebc5f data-v-16338632>
      <h2 data-v-6d1ebc5f>HelloWorld</h2>
      <p data-v-6d1ebc5f>我是coderwhy, 哈哈哈</p>
      <p data-v-6d1ebc5f>App传入:我是App中的message</p>
    </div>
  </div>
```

认识Vue插件

- 通常我们**向Vue全局添加一些功能**时，会采用**插件的模式，它有两种编写方式**：
  - 对象类型：一个对象，但是必须包含一个 install 的函数，该函数会在安装插件时执行；
  - 函数类型：一个function，这个函数会在安装插件时自动执行；
- 插件可以**完成的功能没有限制**，比如下面的几种都是可以的：
  - 添加全局方法或者 property，通过把它们添加到 config.globalProperties 上实现；
  - 添加全局资源：指令/过滤器/过渡等；
  - 通过全局 mixin 来添加一些组件选项；
  - 一个库，提供自己的 API，同时提供上面提到的一个或多个功能；

插件的编写方式

对象类型的写法

plugins > JS plugin_01.js > ...

```
export default {
  name: "why",
  install(app, options) {
    console.log("插件被安装:", app, options);
    console.log(this.name);
  }
}
```

函数类型的写法

gins > JS plugin_02.js > ...

```
export default function(app, options) {
  console.log("插件被安装:", app, options);
}
```