> 本文由 简悦 SimpRead 转码， 原文地址 kaiwu.lagou.com

我们知道 Vue.js 的核心思想之一是数据驱动，数据是 DOM 的映射。在大部分情况下，你是不用操作 DOM 的，但是这并不意味着你不能操作 DOM。

有些时候，我们希望手动去操作某个元素节点的 DOM，比如当这个元素节点挂载到页面的时候通过操作底层的 DOM 来做一些事情。

为了支持这个需求，Vue.js 提供了指令的功能，它允许我们自定义指令，作用在普通的 DOM 元素上。

举个聚焦输入框的例子，我们希望在页面加载时，输入框自动获得焦点，我们可以全局注册一个 v-focus 指令：

```
import Vue from 'vue'

const app = Vue.createApp({})

app.directive('focus', {

  mounted(el) {

    el.focus()

  }

})
```

当然，我们也可以在组件内部局部注册：

```
directives: {

  focus: {

    mounted(el) {

      el.focus()

    }

  }

}
```

然后我们就可以在模板中使用这个指令了：`<input v-focus />`。

至此我们就大致了解了指令的功能和用法，那么接下来，我们就从指令的定义、指令的注册和指令的应用三个方面来一起探究它的实现原理。

指令的定义

指令本质上就是一个 JavaScript 对象，对象上挂着一些钩子函数，我们可以举个例子来说明，比如我定义一个 v-log 指令，这个指令做的事情就是在指令的各个生命周期去输出一些 log 信息：

```
const logDirective = {

  beforeMount() {

    console.log('log directive before mount')

  },

  mounted() {

     console.log('log directive mounted')

  },

  beforeUpdate() {

    console.log('log directive before update')

  },

  updated() {

    console.log('log directive updated')

  },

  beforeUnmount() {

    console.log('log directive beforeUnmount')
```

```
  },

  unmounted() {

    console.log('log directive unmounted')

  }

}
```

然后你可以在创建应用后注册它：

```
import { createApp } from 'vue'

import App from './App'

const app = createApp(App)

app.directive('log', logDirective)

app.mount('#app')
```

接着在 App 组件中使用这个指令：

```
<template>

<p v-if="flag">{{ msg }}</p>

<input v-else v-log v-model="text"/>

<button @click="flag=!flag">toggle</button>

</template>

<script>

export default {

    data() {
```

```
return {

flag: true,

msg: 'Hello Vue',

text: ''

      }

    }

  }

</script>
```

我建议你拷贝上述代码运行这个示例，你会发现，当你点击按钮后，会先执行指令定义的 beforeMount 和 mounted 钩子函数，然后你在 input 输入框中输入一些内容，会执行 beforeUpdate 和 updated 钩子函数，然后你再次点击按钮，会执行 beforeUnmount 和 unmounted 钩子函数。

所以一个指令的定义，无非就是在合适的钩子函数中编写一些相关的处理逻辑。我基于 Vue.js 3.0 写过一个简单图片懒加载的插件 vue3-lazy，你也可以去看看它的源码，了解一下一个成熟的指令插件是如何编写的。

指令的注册

所以当我们编写好指令后，在应用它之前，我们需要先注册它。所谓注册，其实就是把指令的定义保存到相应的地方，未来使用的时候我可以从保存的地方拿到它。

指令的注册和组件一样，可以全局注册，也可以局部注册。我们来分别看一下它们的实现原理。

首先，我们来了解全局注册的方式，它是通过 app.directive 方法去注册的，比如：

```
app.directive('focus', {

  mounted(el) {

    el.focus()

  }

})
```

我们来看 directive 方法的实现：

```
function createApp(rootComponent, rootProps = null) {

const context = createAppContext()

const app = {

    _component: rootComponent,

    _props: rootProps,

    directive(name, directive) {

if ((process.env.NODE_ENV !== 'production')) {

        validateDirectiveName(name)

      }

if (!directive) {

return context.directives[name]

      }

if ((process.env.NODE_ENV !== 'production') && context.directives[name]) {

        warn(`Directive "${name}" has already been registered in target app.`)

      }

      context.directives[name] = directive

return app

    }

  }
```

```
return app

}
```

directive 是 app 对象上的一个方法，它接受两个参数，第一个参数是指令的名称，第二个参数就是指令对象。

指令全局注册方法的实现非常简单，就是把指令对象注册到 app 对象创建的全局上下文 context.directives 中，并用 name 作为 key。

这里有几个细节要注意一下，validateDirectiveName 是用来检测指令名是否和内置的指令（如 v-model、v-show）冲突；如果不传第二个参数指令对象，表示这是一次指令的获取；指令重复注册会报警告。

接下来，我们来了解局部注册的方式，它是直接在组件对象中定义的，比如：

```
directives: {

  focus: {

    mounted(el) {

      el.focus()

    }

  }

}
```

因此全局注册和局部注册的区别是，一个保存在 appContext 中，一个保存在组件对象的定义中。

指令的应用

接下来，我们重点分析指令的应用过程，我们以 v-focus 指令为例，在组件中使用这个指令：`<input v-focus />`。

我们先看这个模板编译后生成的 render 函数：

```
import { resolveDirective as _resolveDirective, createVNode as _createVNode,
withDirectives as _withDirectives, openBlock as _openBlock, createBlock as
_createBlock } from "vue"


export function render(_ctx, _cache, $props, $setup, $data, $options) {
```

```
const _directive_focus = _resolveDirective("focus")

return _withDirectives((_openBlock(), _createBlock("input", null, null, 512 )),
[

    [_directive_focus]

  ])

}
```

我们再来看看如果不使用 v-focus，单个 input 编译生成后的 render 函数是怎样的：

```
import { createVNode as _createVNode, openBlock as _openBlock, createBlock as
_createBlock } from "vue"

export function render(_ctx, _cache, $props, $setup, $data, $options) {

return (_openBlock(), _createBlock("input"))

}
```

对比两个编译结果可以看到，区别在于如果元素节点使用指令，那么它编译生成的 vnode 会用 withDirectives 包装一层。

在分析 withDirectives 函数的实现之前先来看指令的解析函数 resolveDirective，因为前面我们已经了解指令的注册其实就是把定义的指令对象保存下来，那么 resolveDirective 做的事情就是根据指令的名称找到保存的对应指令对象，我们来看一下它的实现：

```
const DIRECTIVES = 'directives';

function resolveDirective(name) {

return resolveAsset(DIRECTIVES, name)

}

function resolveAsset(type, name, warnMissing = true) {

const instance = currentRenderingInstance || currentInstance
```

```
if (instance) {

const Component = instance.type

const res =

      resolve(Component[type], name) ||

      resolve(instance.appContext[type], name)

if ((process.env.NODE_ENV !== 'production') && warnMissing && !res) {

      warn(`Failed to resolve ${type.slice(0, -1)}: ${name}`)

    }

return res

  }

else if ((process.env.NODE_ENV !== 'production')) {

    warn(`resolve${capitalize(type.slice(0, -1))} ` +

      `can only be used in render() or setup().`)

  }

}

function resolve(registry, name) {

return (registry &&

    (registry[name] ||

      registry[camelize(name)] ||
```

```
      registry[capitalize(camelize(name))]))

}
```

可以看到，resolveDirective 内部调用了 resolveAsset 函数，传入的类型名称为 directives 字符串。

resolveAsset 内部先通过 resolve 函数解析局部注册的资源，由于我们传入的是 directives，所以就从组件定义对象上的 directives 属性中查找对应 name 的指令，如果查找不到则通过 instance.appContext，也就是我们前面提到的全局的 appContext，根据其中的 name 查找对应的指令。

所以 resolveDirective 的实现很简单，优先查找组件是否局部注册该指令，如果没有则看是否全局注册该指令，如果还找不到则在非生产环境下报警告，提示用户没有解析到该指令。如果你平时在开发工作中遇到这个警告，那么你很可能就是没有注册这个指令，或者是 name 写得不对。

注意，在 resolve 函数实现的过程中，它会先根据 name 匹配，如果失败则把 name 变成驼峰格式继续匹配，还匹配不到则把 name 首字母大写后继续匹配，这么做是为了让用户编写指令名称的时候可以更加灵活，所以需要多判断几步用户可能编写的指令名称的情况。

接下来，我们来分析 withDirectives 的实现：

```
function withDirectives(vnode, directives) {

const internalInstance = currentRenderingInstance

if (internalInstance === null) {

    (process.env.NODE_ENV !== 'production') && warn(`withDirectives can only be
used inside render functions.`)

return vnode

  }

const instance = internalInstance.proxy

const bindings = vnode.dirs || (vnode.dirs = [])

for (let i = 0; i < directives.length; i++) {

    let [dir, value, arg, modifiers = EMPTY_OBJ] = directives[i]

if (isFunction(dir)) {
```

```
      dir = {

        mounted: dir,

        updated: dir

      }

    }

    bindings.push({

      dir,

      instance,

      value,

      oldValue: void 0,

      arg,

      modifiers

    })

  }

return vnode

}
```

withDirectives 函数第一个参数是 vnode，第二个参数是指令构成的数组，因为一个元素节点上是可以应用多个指令的。

withDirectives 其实就是给 vnode 添加了一个 dirs 属性，属性的值就是这个元素节点上的所有指令构成的对象数组。它通过对 directives 的遍历，拿到每一个指令对象以及指令对应的值 value、参数 arg、修饰符 modifiers 等，然后构造成一个 binding 对象，这个对象还绑定了组件的实例 instance。

这么做的目的是在元素的生命周期中知道运行哪些指令相关的钩子函数，以及在运行这些钩子函数的时候，还可以往钩子函数中传递一些指令相关的参数。

那么，接下来我们就来看在元素的生命周期中是如何运行这些钩子函数的。

首先，我们来看元素挂载时候会执行哪些指令的钩子函数。通过前面章节的学习我们了解到，一个元素的挂载是通过执行 mountElement 函数完成的，我们再来回顾一下它的实现：

```
const mountElement = (vnode, container, anchor, parentComponent, parentSuspense,
isSVG, optimized) => {

  let el

const { type, props, shapeFlag, dirs } = vnode

  el = vnode.el = hostCreateElement(vnode.type, isSVG, props && props.is)

if (props) {

  }

if (shapeFlag & 8 ) {

    hostSetElementText(el, vnode.children)

  } else if (shapeFlag & 16 ) {

    mountChildren(vnode.children, el, null, parentComponent, parentSuspense,
isSVG && type !== 'foreignObject', optimized || !!vnode.dynamicChildren)

  }

if (dirs) {

    invokeDirectiveHook(vnode, null, parentComponent, 'beforeMount')

  }

  hostInsert(el, container, anchor)

if (dirs) {
```

```
    queuePostRenderEffect(()=>{

      invokeDirectiveHook(vnode, null, parentComponent, 'mounted')

    })

  }

}
```

这一次，我们添加了元素指令调用的相关代码，可以直观地看到，在元素插入到容器之前会执行指令的 beforeMount 钩子函数，在插入元素之后，会通过 queuePostRenderEffect 的方式执行指令的 mounted 钩子函数。

钩子函数的执行，是通过调用 invokeDirectiveHook 方法完成的，我们来看它的实现：

```
function invokeDirectiveHook(vnode, prevVNode, instance, name) {

const bindings = vnode.dirs

const oldBindings = prevVNode && prevVNode.dirs

for (let i = 0; i < bindings.length; i++) {

const binding = bindings[i]

if (oldBindings) {

      binding.oldValue = oldBindings[i].value

    }

const hook = binding.dir[name]

if (hook) {

      callWithAsyncErrorHandling(hook, instance, 8 , [

        vnode.el,
```

```
        binding,

        vnode,

        prevVNode

      ])

    }

  }

}
```

invokeDirectiveHook 函数有四个参数，第一个和第二个参数分别代表新旧 vnode，第三个参数是组件实例 instance，第四个参数是钩子名称 name。

invokeDirectiveHook 的实现很简单，通过遍历 vnode.dirs 数组，找到每一个指令对应的 binding 对象，然后从 binding 对象中根据 name 找到指令定义的对应的钩子函数，如果定义了这个钩子函数则执行它，并且传入一些响应的参数，包括元素的 DOM 节点 el，binding 对象，新旧 vnode，这就是我们在执行指令钩子函数的时候，可以访问到这些参数的原因。

另外我们注意到，mounted 钩子函数会用 queuePostRenderEffect 包一层执行，这么做和组件的初始化过程执行 mounted 钩子函数一样，在整个应用 render 完毕后，同步执行 flushPostFlushCbs 的时候执行元素指令的 mounted 钩子函数。

接下来，我们来看元素更新时候会执行哪些指令的钩子函数。通过前面章节的学习我们了解到，一个元素的更新是通过执行 patchElement 函数，我们再来回顾一下它的实现：

```
const patchElement = (n1, n2, parentComponent, parentSuspense, isSVG, optimized)
=> {

const el = (n2.el = n1.el)

const oldProps = (n1 && n1.props) || EMPTY_OBJ

const newProps = n2.props || EMPTY_OBJ

const { dirs } = n2

  patchProps(el, n2, oldProps, newProps, parentComponent, parentSuspense, isSVG)

const areChildrenSVG = isSVG && n2.type !== 'foreignObject'
```

```
if (dirs) {

    invokeDirectiveHook(n2, n1, parentComponent, 'beforeUpdate')

  }

  patchChildren(n1, n2, el, null, parentComponent, parentSuspense,
areChildrenSVG)

if (dirs) {

    queuePostRenderEffect(()=>{

      invokeDirectiveHook(vnode, null, parentComponent, 'updated')

    })

  }

}
```

这一次，我们添加了元素指令调用的相关代码，可以直观地看到，在更新子节点之前会执行指令的 beforeUpdate 钩子函数，在更新完子节点之后，会通过 queuePostRenderEffect 的方式执行指令的 updated 钩子函数。

最后，我们来看元素卸载时候会执行哪些指令的钩子函数。通过前面章节的学习我们了解到，一个元素的卸载是通过执行 unmount 函数，我们再来回顾一下它的实现：

```
const unmount = (vnode, parentComponent, parentSuspense, doRemove = false) => {

const { type, props, children, dynamicChildren, shapeFlag, patchFlag, dirs } =
vnode

  let vnodeHook

if ((vnodeHook = props && props.onVnodeBeforeUnmount)) {

    invokeVNodeHook(vnodeHook, parentComponent, vnode)

  }
```

```
const shouldInvokeDirs = shapeFlag & 1  && dirs

if (shapeFlag & 6 ) {

    unmountComponent(vnode.component, parentSuspense, doRemove)

  }

else {

if (shapeFlag & 128 ) {

      vnode.suspense.unmount(parentSuspense, doRemove)

return

    }

if (shouldInvokeDirs) {

      invokeDirectiveHook(vnode, null, parentComponent, 'beforeUnmount')

    }

if (dynamicChildren &&

      (type !== Fragment ||

        (patchFlag > 0 && patchFlag & 64 ))) {

      unmountChildren(dynamicChildren, parentComponent, parentSuspense)

    }

else if (shapeFlag & 16 ) {

      unmountChildren(children, parentComponent, parentSuspense)
```

```
    }

if (shapeFlag & 64 ) {

      vnode.type.remove(vnode, internals)

    }

if (doRemove) {

      remove(vnode)

    }

  }

if ((vnodeHook = props && props.onVnodeUnmounted) || shouldInvokeDirs) {

    queuePostRenderEffect(() => {

      vnodeHook && invokeVNodeHook(vnodeHook, parentComponent, vnode)

if (shouldInvokeDirs) {

        invokeDirectiveHook(vnode, null, parentComponent, 'unmounted')

      }

    }, parentSuspense)

  }

}
```

unmount 方法的主要思路就是用递归的方式去遍历删除自身节点和子节点。

可以看到，在移除元素的子节点之前会执行指令的 beforeUnmount 钩子函数，在移除子节点和当前节点之后，会通过 queuePostRenderEffect 的方式执行指令的 unmounted 钩子函数。

总结

好的，到这里我们这一节的学习也要结束啦，通过这节课的学习，你应该了解指令是如何定义、如何注册，以及如何应用的。指令无非就是给我们提供了在一个元素的生命周期中注入代码的途径，它的本身实现是很简单的。

最后，给你留一道思考题目，请实现一个 v-uid 指令，实现创建唯一的元素 id，使用方式如下：

```
<div v-uid="foo"></div>


<div v-uid="foo"></div>
```

最终会在页面上生成的 HTML 如下：

```
<div></div>


<div></div>
```

你有什么好的思路吗？欢迎你在留言区与我分享。

> 本节课的相关代码在源代码中的位置如下：
> packages/runtime-core/src/directives.ts
> packages/runtime-core/src/apiCreateApp.ts
> packages/runtime-core/src/helpers/resolveAssets.ts
> packages/runtime-core/src/renderer.ts