



下载APP



27 | 自定义渲染器：如何实现Vue的跨端渲染？

2021-12-22 大圣

《玩转Vue 3全家桶》

课程介绍 >

**讲述：大圣**

时长 07:35 大小 6.95M



你好，我是大圣。


上一讲我们讲完了组件库的核心知识点，这一讲我们来学习一个叫 Vue 3 的进阶知识点：自定义渲染器，这个功能可以自定义 Vue 渲染的逻辑。

在给你讲清楚原理之后，我还会带你一起实现一个 Canvas 的渲染器实际上手体验一下。

什么是渲染器



我们都知道，Vue 内部的组件是以虚拟 dom 形式存在的。下面的代码就是一个很常见的虚拟 Dom，用对象的方式去描述一个项目。相比 dom 标签相比，这种形式可以让整个 Vue 项目脱离浏览器的限制，更方便地实现 Vuejs 的跨端。

 复制代码


```
1 {  
2   tag: 'div',  
3   props: {  
4     id: 'app'  
5   },  
6   children: [  
7     {  
8       tag: Container,  
9       props: {  
10        className: 'el-container'  
11      },  
12      children: [  
13        '哈喽小老弟!!!'  
14      ]  
15    }  
16  ]  
17 }
```

渲染器是围绕虚拟 Dom 存在的。在浏览器中，我们把虚拟 Dom 渲染成真实的 Dom 对象，Vue 源码内部把一个框架里所有和平台相关的操作，抽离成了独立的方法。所以，我们只需要实现下面这些方法，就可以实现 Vue 3 在一个平台的渲染。

首先用 `createElement` 创建标签，还有用 `createText` 创建文本。创建之后就需要用 `insert` 新增元素，通过 `remove` 删除元素，通过 `setText` 更新文本和 `patchProps` 修改属性。然后再实现 `parentNode`、`nextSibling` 等方法实现节点的查找关系。完成这些工作，理论上就可以在一个平台内实现一个应用了。

在 Vue 3 中的 `runtime-core` 模块，就对外暴露了这些接口，`runtime-core` 内部基于这些函数实现了整个 Vue 内部的所有操作，然后在 `runtime-dom` 中传入以上所有方法。

下面的代码就是 Vue 代码提供浏览器端操作的函数，这些 DOM 编程接口完成了浏览器端增加、添加和删除操作，这些 API 都是浏览器端独有的，如果一个框架强依赖于这些函数，那就只能在浏览器端运行。

 复制代码

```
1 export const nodeOps: Omit<RendererOptions<Node, Element>, 'patchProp'> = {  
2   //插入元素  
3   insert: (child, parent, anchor) => {  
4     parent.insertBefore(child, anchor || null)  
5   },
```

```
6 // 删除元素
7 remove: child => {
8   const parent = child.parentNode
9   if (parent) {
10     parent.removeChild(child)
11   }
12 },
13 // 创建元素
14 createElement: (tag, isSVG, is, props): Element => {
15   const el = isSVG
16     ? doc.createElementNS(svgNS, tag)
17     : doc.createElement(tag, is ? { is } : undefined)
18
19   if (tag === 'select' && props && props.multiple !== null) {
20     ;(el as HTMLSelectElement).setAttribute('multiple', props.multiple)
21   }
22
23   return el
24 }
25 //...其他操作函数
26 }
```

如果一个框架想要实现跨端的功能，那么渲染器本身不能依赖任何平台下特有的接口。

在后面的代码中，我们通过 `createRenderer` 函数区创建了一个渲染器。通过参数 `options` 获取增删改查所有的函数以后，在内部的 `render`、`mount`、`patch` 等函数中，需要去渲染一个元素的时候，就可以通过 `option.createElement` 和 `option.insert` 来实现。

这段代码给你展现的是核心逻辑，完整版本你可以看一下 [Vue 3 的源码](#)。

[复制代码](#)

```
1 export default function createRenderer(options) {
2   const {
3     insert: hostInsert,
4     remove: hostRemove,
5     patchProp: hostPatchProp,
6     createElement: hostCreateElement,
7     createText: hostCreateText,
8     createComment: hostCreateComment,
9     setText: hostSetText,
10    setElementText: hostSetElementText,
11    parentNode: hostParentNode,
12    nextSibling: hostNextSibling,
13    setScopeId: hostSetScopeId = NOOP,
```

```
14     cloneNode: hostCloneNode,
15     insertStaticContent: hostInsertStaticContent
16   } = options
17
18   function render(vnode, container) { }
19
20   function mount(vnode, container, isSVG, refNode) { }
21
22   function mountElement(vnode, container, isSVG, refNode) { }
23
24   function mountText(vnode, container) { }
25
26   function patch(prevVNode, nextVNode, container) { }
27
28   function replaceVNode(prevVNode, nextVNode, container) { }
29   function patchElement(prevVNode, nextVNode, container) { }
30   function patchChildren(
31     prevChildFlags,
32     nextChildFlags,
33     prevChildren,
34     nextChildren,
35     container
36   ) { }
37
38   function patchText(prevVNode, nextVNode) { }
39   function patchComponent(prevVNode, nextVNode, container) { }
40
41   return { render }
42 }
```

在每个函数实现的内部，比如 `mountElemnt`，我们之前的实现方式是调用浏览器的 API 创建。

[复制代码](#)

```
1 function mountElement(vnode, container, isSVG, refNode) {
2   const el = isSVG
3     ? document.createElementNS(...)
4     : document.createElement(vnode.tag)
5 }
6
```

对比一下，经过渲染器抽离之后，内部的 `mountElmenet` 就会把所有 `document` 的操作全部换成 `options` 传递进来的 `hostCreate` 函数。

[复制代码](#)

```
1 function mountElement(vnode, container, isSVG, refNode) {  
2   const el = hostCreateElement(vnode.tag, isSVG)  
3 }
```

然后，我们使用后面的代码创建一个具体平台的渲染器，这也是 Vue 3 中的 runtime-dom 包主要做的事。了解了 Vue 中自定义渲染器的实现方式后，我们还可以基于 Vue 3 的 runtime-core 包封装其他平台的渲染器，让其他平台也能使用 Vue 内部的响应式和组件化等优秀的特性。

[复制代码](#)

```
1 const { render } = createRenderer({  
2   nodeOps: {  
3     createElement() { },  
4     createText() { }  
5     // more...  
6   },  
7   patchData  
8 })  
9
```

自定义渲染

说完了渲染器创建，我们再来看看自定义渲染。

自定义渲染器让 Vue 脱离了浏览器的限制，我们只需要实现平台内部的增删改查函数后，就可以直接对接 Vue 3。比方说，我们可以把 Vue 渲染到小程序平台，实现 Vue 3-minipp；也可以渲染到 Canvas，实现 vue 3-canvas，把虚拟 dom 渲染成 Canvas；甚至还可以尝试把 Vue 3 渲染到 three.js 中，在 3D 世界使用响应式开发。

接下来，我们一起尝试实现一个 Canvas 的渲染器。具体操作是这样的，我们在项目的 src 目录下新建 renderer.js，通过这个文件实现一个简易的 Canvas 渲染逻辑。Canvas 平台中操作的方式相对简单，没有太多节点的概念，我们可以把整个 Canvas 维护成一个对象，每次操作的时候直接把 Canvas 重绘一下就可以了。

[复制代码](#)

```
1 import { createRenderer } from '@vue/runtime-core'  
2 const { createApp: originCa } = createRenderer({  
3   insert: (child, parent, anchor) => {  
4     },
```

```
5   createElement(type, isSVG, isCustom) {
6   },
7   setElementText(node, text) {
8   },
9   patchProp(el, key, prev, next) {
10  },
11  });
```

下面的代码中我们实现了 draw 函数，这里我们就是用 Canvas 的操作方法**递归**地把 Canvas 对象渲染到 Canvas 标签内部。

[复制代码](#)

```
1  let ctx
2  function draw(ele, isChild) {
3    if (!isChild) {
4      ctx.clearRect(0, 0, 500, 500)
5    }
6
7    ctx.fillStyle = ele.fill || 'white'
8    ctx.fillRect(...ele.pos)
9    if (ele.text) {
10     ctx.fillStyle = ele.color || 'white'
11     ele.fontSize = ele.type == "h1" ? 20 : 12
12     ctx.font = (ele.fontSize || 18) + 'px serif'
13     ctx.fillText(ele.text, ele.pos[0] + 10, ele.pos[1] + ele.fontSize)
14   }
15   ele.child && ele.child.forEach(c => {
16     console.log('child:::', c)
17     draw(c, true)
18   })
19
20 }
```

由于我们主体需要维护的逻辑就是对于对象的操作，所以创建和更新操作直接操作对象即可。新增 insert 需要维护 parent 和 child 元素。另外，插入的时候也需要调用 draw 函数，并且需要监听 onclick 事件。

[复制代码](#)

```
1  const { createApp: originCa } = createRenderer({
2    insert: (child, parent, anchor) => {
3      if (typeof child == 'string') {
4        parent.text = child
5      } else {
6        child.parent = parent
```

```
7     if (!parent.child) {
8       parent.child = [child]
9     } else {
10      parent.child.push(child)
11    }
12  }
13  if (parent.nodeName) {
14    draw(child)
15    if (child.onClick) {
16      ctx.canvas.addEventListener('click', () => {
17        child.onClick()
18        setTimeout(() => {
19          draw(child)
20        })
21      }, false)
22    }
23  }
24 },
25 createElement(type, isSVG, isCustom) {
26   return {
27     type
28   }
29 },
30 setElementText(node, text) {
31   node.text = text
32 },
33 patchProp(el, key, prev, next) {
34   el[key] = next
35 },
36
37 });
```

现在我们来到 `src/main.js` 中，这时候就不能直接从 `vue` 中引入 `createApp` 了，而是需要从 `runtime-core` 中导入 `createRenderer`。

接下来，通过 `createRenderer` 用我们自己定义的 `renderer` 去创建 `createApp`，并且重写 `mount` 函数。在 `Canvas` 的 `mount` 中，我们需要创建 `Canvas` 标签并且挂载到 `App` 上。

[复制代码](#)

```
1 import { createRenderer } from '@vue/runtime-core'
2 const { createApp: originCa } = createRenderer({
3
4 })
5 function createApp(...args) {
6   const app = originCa(...args)
7   return {
8     mount(selector) {
```

```
9     const canvas = document.createElement('canvas')
10    canvas.width = window.innerWidth
11    canvas.height = window.innerHeight
12    document.querySelector(selector).appendChild(canvas)
13    ctx = canvas.getContext('2d')
14    app.mount(canvas)
15  }
16 }
17 }
```

下一步进入 `src/App.vue` 中，我们就可以在 Vue 组件中使用 `ref` 等响应式的写法了。我们实现了通过 `ref` 返回的响应式对象，渲染 Canvas 内部的文字和高度，并且点击的时候还可以修改文字。完成上面的操作，我们就实现了 Canvas 平台的基本渲染。

[复制代码](#)

```
1 <template>
2 <div @click="setName('vue3真棒')" :pos="[10,10,300,300]" fill="#eee">
3   <h1 :pos="[20,20,200,100]" fill="red" color="#000">累加器{{count}}</h1>
4   <span :pos="pos" fill="black" >哈喽{{name}}</span>
5 </div>
6
7
8
9 </template>
10
11 <script setup>
12
13 import {ref} from 'vue'
14 const name = ref('vue3入门')
15 const pos = ref([20,120,200,100])
16 const count = ref(1)
17 const setName = (n)=>{
18   name.value = n
19   pos.value[1]+=20
20   count.value+=2
21 }
22 </script>
```

上面的代码在浏览器里就会有下图的显示效果。我们点击 Canvas 后，文案就会显示为“哈喽 vue3 真棒”，并且黑色方块和红色方块的距离也会变大。

累加器5

哈喽vue3真棒

基于这个原理，我们其实可以做很多有意思的尝试，社区也也有越来越多开源的 Vue 3 的自定义渲染器，比如小程序跨端框架 uni-app，Vugel 可以使用 Vue 渲染 WebGL 等，你也可以动手多多体验。

比如下面的代码中，我们对 three.js 进行一个渲染的尝试。它的实现逻辑和 Canvas 比较类似，通过对于对象的维护和 draw 函数实现最终的绘制。在 draw 函数内部，我们调用 three.js 的操作方法去创建 camera，scene，geometry 等概念，最后对外暴露 three.js 的 createApp 函数。

复制代码

```
1 import { createRenderer } from '@vue/runtime-core'
2 import * as THREE from 'three'
3 import { nextTick } from '@vue/runtime-core'
4
5 let renderer
6
7 function draw(obj) {
8   const { camera, cameraPos, scene, geometry, geometryArg, material, mesh, meshY, m
9   if([camera, cameraPos, scene, geometry, geometryArg, material, mesh, meshY, mesh
10     return
11   }
12   let cameraObj = new THREE[camera]( 40, window.innerWidth / window.innerHei
13   Object.assign(cameraObj.position, cameraPos)
14
15   let sceneObj = new THREE[scene]()
16
17   let geometryObj = new THREE[geometry]( ...geometryArg)
18   let materialObj = new THREE[material]()
```

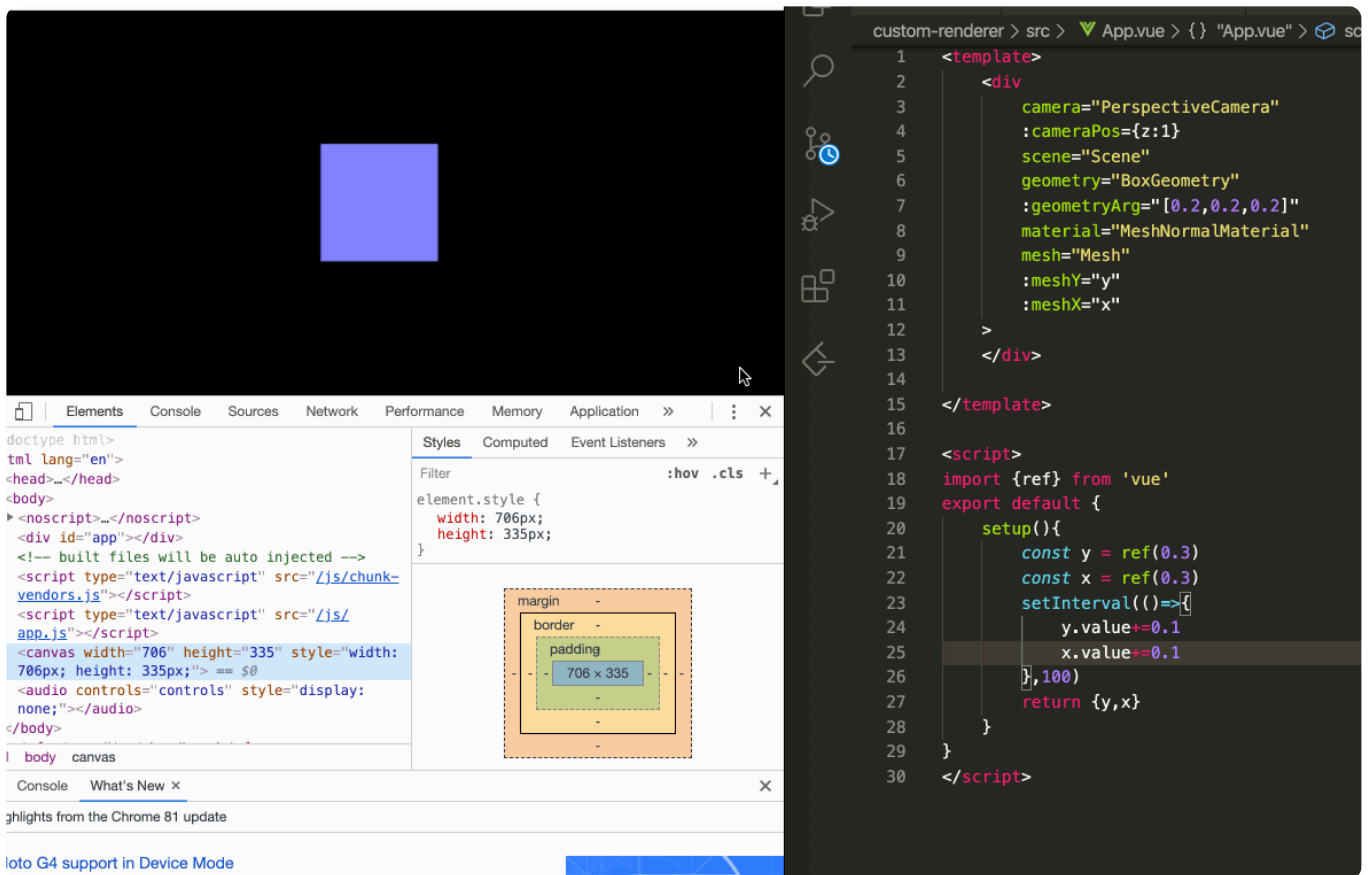
```
19
20   let meshObj = new THREE[mesh]( geometryObj, materialObj )
21   meshObj.rotation.x = meshX
22   meshObj.rotation.y = meshY
23   sceneObj.add( meshObj )
24   renderer.render( sceneObj, cameraObj );
25
26 }
27
28 const { createApp: originCa } = createRenderer({
29   insert: (child, parent, anchor) => {
30     if(parent.domElement){
31       draw(child)
32     }
33   },
34   createElement(type, isSVG, isCustom) {
35     return {
36       type
37     }
38   },
39   setElementText(node, text) {
40   },
41   patchProp(el, key, prev, next) {
42     el[key] = next
43     draw(el)
44   },
45   parentNode: node => node,
46   nextSibling: node => node,
47   createText: text => text,
48   remove: node => node
49 });
50
51 function createApp(...args) {
52   const app = originCa(...args)
53   return {
54     mount(selector) {
55       renderer = new THREE.WebGLRenderer( { antialias: true } );
56       renderer.setSize( window.innerWidth, window.innerHeight );
57       document.body.appendChild( renderer.domElement );
58       app.mount(renderer)
59     }
60   }
61 }
62 export { createApp }
63
64
65
```

然后我们在 App.vue 中，使用下面的代码渲染出一个立方体，并且通过 ref 响应式对象控制立方体偏移的监督，再通过 setInterval 实现立方体的动画，实现下图的反转效果。

[复制代码](#)

```
1 <template>
2   <div
3     camera="PerspectiveCamera"
4     :cameraPos={z:1}
5     scene="Scene"
6     geometry="BoxGeometry"
7     :geometryArg="[0.2,0.2,0.2]"
8     material="MeshNormalMaterial"
9     mesh="Mesh"
10    :meshY="y"
11    :meshX="x"
12  >
13  </div>
14
15 </template>
16
17 <script>
18 import {ref} from 'vue'
19 export default {
20   setup(){
21     const y = ref(0.3)
22     const x = ref(0.3)
23     setInterval(()=>{
24       y.value+=0.3
25       x.value+=0.5
26     },100)
27     return {y,x}
28   }
29 }
30 </script>
```

反转效果演示如下：



我们还可以在 Canvas 的封装上更进一步，并且实现对一些 Canvas 已有框架 Pixi.js 的封装，这样就可以通过 Vue 3 的响应式的开发方式，快速开发一个小游戏。

下面的代码中就是针对 Pixi.js 实现的封装函数，你可以看一下。

复制代码

```
1 import {Graphics} from "PIXI.js";
2
3 export const getNodeOps = (app) => {
4   return {
5     insert: (child, parent, anchor) => {
6       parent.addChild(child);
7     },
8
9     remove: (child) => {
10       const parent = child.parentNode;
11       if (parent) {
12         parent.removeChild(child);
13       }
14     },
15
16     createElement: (tag, isSVG, is) => {
17       let element;
18       if (tag === "Rectangle") {
19         // 创建一个矩形
```

```
20     element = new window.PIXI.Graphics();
21     element.lineStyle(4, 0xff3300, 1);
22     element.beginFill(0x66ccff);
23     element.drawRect(0, 0, 64, 64);
24     element.endFill();
25     element.x = 0;
26     element.y = 0;
27     // Opt-in to interactivity
28     element.interactive = true;
29
30     // Shows hand cursor
31     element.buttonMode = true;
32   } else if (tag === "Sprite") {
33     element = new window.PIXI.Sprite();
34     element.x = 0;
35     element.y = 0;
36   } else if (tag === "Container") {
37     element = new window.PIXI.Container();
38     element.x = 0;
39     element.y = 0;
40   }
41
42   return element;
43 },
44
45 createText: (text) => doc.createTextNode(text),
46
47 createComment: (text) => {
48   //   console.log(text);
49 },
50
51 setText: (node, text) => {
52   node.nodeValue = text;
53 },
54
55 setElementText: (el, text) => {
56   el.textContent = text;
57 },
58
59 parentNode: (node) => node.parentNode,
60
61 nextSibling: (node) => node.nextSibling,
62
63 querySelector: (selector) => doc.querySelector(selector),
64
65 setScopeId(el, id) {
66   el.setAttribute(id, "");
67 },
68
69 cloneNode(el) {
70   return el.cloneNode(true);
71 },
```

```
72   };  
73
```

Pixi 中的属性修改可以使用下面的代码，判断 x、y、width 和 on 属性不同的操作，就是用响应式包裹了 Pixi 的对象。关于 Vue 3 和 Pixi 实现的代码效果，你可以在 [🔗 GitHub](#) 看到全部的源码。

[📄 复制代码](#)

```
1  export const patchProp = (  
2    el,  
3    key,  
4    prevValue,  
5    nextValue,  
6    isSVG = false,  
7  ) => {  
8    switch (key) {  
9      case "x":  
10     case "y":  
11     case "width":  
12     case "height":  
13       el[key] = nextValue;  
14       break;  
15     case "on":  
16       Object.keys(nextValue).forEach((eventName) => {  
17         const callback = nextValue[eventName];  
18         el.on(eventName, callback);  
19       });  
20       break;  
21     case "texture":  
22       let texture = PIXI.Texture.from(nextValue);  
23       el.texture = texture;  
24       break;  
25     }  
26   };  
27
```

总结

今天聊的内容到此就结束了，我们来总结一下今天学到的知识点。

首先我们了解了自定义渲染器的原理，就是**把所有的增删改查操作暴露出去，使用的时候不需要知道内部的实现细节，我们只需要针对每个平台使用不同的 API 即可。**

你可以这样理解，就像武侠小说中高手可以通过给你传输内力的方式控制你进行比武。我们打出去的每招每式都是来源于背后的高手，只不过自己做了简单的适配。在 Vue 渲染器的设计中就把 document 所有的操作都抽离成了 nodeOps，并且通过调用 Vue 的 createRenderer 函数创建平台的渲染器。

这样一来，只要我们实现了 Canvas 平台的增删改查，就可以在 Canvas 的世界中使用 Vue 的响应式语法控制绘图和做游戏，Vue 生态中对小程序和原生 app 的支持原理也是基于自定义渲染器实现的。

其实，自定义渲染器也代表着适配器设计模式的一个实践。除了自定义渲染器 API 的学习，我们也要反思一下自己现在负责的项目中，有哪些地方为了不同的接口或者平台写了太多的判断代码，是否也可以使用类似自定义渲染器的逻辑和模式，把多个组件、平台、接口之间不同的操作方式封装成一个核心模块，去进行单独函数的扩展。

思考题

最后留个思考题给你，Vue 如何在 node 环境中渲染呢？欢迎在评论区分享你的答案，我们下一讲再见！

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 26 | 文档：如何给你的组件库设计一个可交互式文档？

下一篇 28 | 响应式：万能的面试题，怎么手写响应式系统

更多课程推荐

跟月影学可视化

系统掌握图形学与可视化核心原理

月影

奇虎 360 奇舞团团长

可视化 UI 框架 SpriteJS 核心开发者



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言 (4)

💬 写留言



|| 神

2021-12-22

开眼界的一课。平时工作中涉及相关内容较少，所以这部分的内容还需课下多练几遍。结合之前所学，写几点感想，首先有以下几种“过程”；

1. html --> dom --> 浏览器
2. html --> v-dom --> dom --> 浏览器
3. svg, canvas, webgl --> v-dom --> dom --> 浏览器...

展开 ∨



👍 2



无双

2021-12-22

想深入了解一下three.js和vue的结合，有没有这方面的资料，或者最佳实践？



Johnson

2021-12-22

这一讲的跨端原理讲解太实用啦！👍



1



海阔天空

2021-12-22

有很多都不是很懂。。。

编辑回复：哪里不懂留言讨论啊~

