Sunday • 更新于 2022-10-19

上一节 02：源码阅读：... 04：框架实现：... 下一节

# 03：框架实现：构建 renderer 基本架构

在上一小节中，我们明确了 `render` 渲染 `Element | Text_Children` 的场景，那么接下来我们就可以根据阅读的源码来实现对应的框架渲染器了。

实现渲染器的过程我们将分为两部分：

1. 搭建出 `renderer` 的基本架构：我们知道对于 `renderer` 而言，它内部分为 **core** 和 **dom** 两部分，那么这两部分怎么交互，我们都会在基本架构这里处理
2. 处理具体的 `processElement` 方法逻辑

那么这一小节，我们就先做第一部分：搭建出 `renderer` 的基本架构：

整个 **基本架构** 应该分为 **三部分** 进行处理：

1. `renderer` 渲染器本身，我们需要构建出 `baseCreateRenderer` 方法
2. 我们知道所有和 `dom` 的操作都是与 `core` 分离的，而和 `dom` 的操作包含了 **两部分**：
    1. `Element` 操作：比如 `insert`、`createElement` 等，这些将被放入到 `runtime-dom` 中
    2. `props` 操作：比如 **设置类名**，这些也将被放入到 `runtime-dom` 中

## renderer 渲染器本身

1. 创建 `packages/runtime-core/src/renderer.ts` 文件：

代码块

```ts
import { ShapeFlags } from 'packages/shared/src/shapeFlags'
import { Fragment } from './vnode'

/**
 * 渲染器配置对象
 */
export interface RendererOptions {
    /**
     * 为指定 element 的 prop 打补丁
     */
    patchProp(el: Element, key: string, prevValue: any, nextValue: any): void
    /**
     * 为指定的 Element 设置 text
     */
    setElementText(node: Element, text: string): void
    /**
     * 插入指定的 el 到 parent 中，anchor 表示插入的位置，即：锚点
     */
    insert(el, parent: Element, anchor?): void
    /**
     * 创建指定的 Element
     */
    createElement(type: string)
}

/**
 * 对外暴露的创建渲染器的方法
```

```ts
export function createRenderer(options: RendererOptions) {
    return baseCreateRenderer(options)
}

/**
 * 生成 renderer 渲染器
 * @param options 兼容性操作配置对象
 * @returns
 */
function baseCreateRenderer(options: RendererOptions): any {
    /**
     * 解构 options，获取所有的兼容性方法
     */
    const {
        insert: hostInsert,
        patchProp: hostPatchProp,
        createElement: hostCreateElement,
        setElementText: hostSetElementText
    } = options

    const patch = (oldVNode, newVNode, container, anchor = null) => {
        if (oldVNode === newVNode) {
            return
        }

        const { type, shapeFlag } = newVNode
        switch (type) {
            case Text:
                // TODO: Text
                break
            case Comment:
                // TODO: Comment
                break
            case Fragment:
                // TODO: Fragment
                break
            default:
                if (shapeFlag & ShapeFlags.ELEMENT) {
                    // TODO: Element
                } else if (shapeFlag & ShapeFlags.COMPONENT) {
                    // TODO: 组件
                }
        }
    }

    /**
     * 渲染函数
     */
    const render = (vnode, container) => {
        if (vnode == null) {
            // TODO: 卸载
        } else {
            // 打补丁（包括了挂载和更新）
            patch(container._vnode || null, vnode, container)
        }
        container._vnode = vnode
    }
    return {
        render
    }
}
```

这样我们就构建出了渲染器框架本身。

### 封装 Element 操作

1. 创建 `packages/runtime-dom/src/nodeOps.ts` 模块，对外暴露 `nodeOps` 对象：

代码块

```
const doc = document

export const nodeOps = {
    /**
     * 插入指定元素到指定位置
     */
    insert: (child, parent, anchor) => {
        parent.insertBefore(child, anchor || null)
    },

    /**
     * 创建指定 Element
     */
    createElement: (tag): Element => {
        const el = doc.createElement(tag)

        return el
    },

    /**
     * 为指定的 element 设置 textContent
     */
    setElementText: (el, text) => {
        el.textContent = text
    }
}
```

### 封装 props 操作

1. 创建 `packages/runtime-dom/src/patchProp.ts` 模块，暴露 `patchProp` 方法：

代码块

```
import { isOn } from '@vue/shared'
import { patchClass } from './modules/class'

/**
 * 为 prop 进行打补丁操作
 */
export const patchProp = (el, key, prevValue, nextValue) => {
    if (key === 'class') {
        patchClass(el, nextValue)
    } else if (key === 'style') {
        // TODO: style
    } else if (isOn(key)) {
        // TODO: 事件
    } else {
        // TODO: 其他属性
    }
}
```

2. 创建 `packages/runtime-dom/src/modules/class.ts` 模块，暴露 `patchClass` 方法：

代码块

```
/**
 * 为 class 打补丁
 */
export function patchClass(el: Element, value: string | null) {
    if (value == null) {
        el.removeAttribute('class')
    } else {
        el.className = value
    }
}
```

3. 在 `packages/shared/src/index.ts` 中，写入 `isOn` 方法：

代码块

```
const onRE = /^on[^a-z]/
/**
 * 是否 on 开头
 */
export const isOn = (key: string) => onRE.test(key)
```

**三大块** 全部完成，标记着整个 `renderer` 架构设计完成。

02：源码阅读：初见 render 函数，ELEME... 上一节 下一节 04：框架实现：基于 renderer 完成 ELEME...

我要提出意见反馈