Sunday • 更新于 2022-10-19

上一节 01：前言 03：基于 templ… 下一节

02：基于 render 渲染的 createApp 的构建逻辑

本小节我们先完成第一步，最终期望的渲染逻辑为：

代码块

```
<script>
  const { createApp, h } = Vue
  // 构建组件实例
  const APP = {
    render() {
      return h('div', 'hello world')
    }
  }

  // 通过 createAPP 标记挂载组件
  const app = createApp(APP)
  // 挂载位置
  app.mount('#app')
</script>
```

对于以上代码而言， `createApp` 和 `mount` 这两个方法我们是不熟悉的，我们可以先看下之前的渲染逻辑，然后倒推一下 `createAPP` 和 `mount` 都做了什么。

以下为之前的渲染逻辑：

代码块

```
<script>
  const { h, render } = Vue

  const component = {
    render() {
      return h('div', 'hello component')
    }
  }
    // 生成 vnode
  const vnode = h(component)
  // 挂载
  render(vnode, document.querySelector('#app'))
</script>
```

由以上代码我们知道，想要挂载一个组件，那么必须经历 **生成** `vnode` 、 `render` **挂载** 的过程。

那么对比两次的实例，我们由此可以推断出：

1. `createApp` 中，必然要生成对应的 `vnode`
2. `mount` 方法，必然要触发 `render` ，生成 `vnode`

明确好了这样的逻辑之后，下面我们去实现对应的实现代码就比较简单了。

1. 在 `packages/runtime-dom/src/index.ts` 中构建 `createApp` 方法：

代码块

```
/**
```

```
 */
export const createApp = (...args) => {
    const app = ensureRenderer().createApp(...args)

    return app
}
```

1. 其中 `ensureRenderer` 方法会返回一个 `renderer` 实例，我们之前实现过对应的代码，可以看一下：

   代码块

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

   function baseCreateRenderer(options: RendererOptions): any {
       ....
       return {
           render
       }
   }
   ```

2. 不知道大家还记不记得，之前我们在实现 `baseCreateRenderer` 时，返回的对象中，其实需要包含三个属性：

   代码块

   ```
   return {
     render,
     hydrate,
     createApp: createAppAPI(render, hydrate)
   }
   ```

3. 我们之前只实现了一个 `render`，那么现在是时候实现 `createApp` 了。

2. 创建 `packages/runtime-core/src/apiCreateApp.ts` 模块，实现 `createAppAPI` 函数：

   代码块

   ```
   /**
    * 创建 app 实例，这是一个闭包函数
    */
   export function createAppAPI<HostElement>(render) {
       return function createApp(rootComponent, rootProps = null) {
           const app = {
               _component: rootComponent,
               _container: null,
               // 挂载方法
               mount(rootContainer: HostElement): any {
                   // 直接通过 createVNode 方法构建 vnode
                   const vnode = createVNode(rootComponent, rootProps)
                   // 通过 render 函数进行挂载
                   render(vnode, rootContainer)
               }
           }

           return app
       }
   }
   ```

3. 在 `baseCreateRenderer` 中，配置 `createAPP` 属性：

   代码块

   ```
   return {
       render,
       createApp: createAppAPI(render)
   }
   ```

但是此时，我们虽然已经可以通过 `createApp` 方法获取到 `app` 实例了，但是还存在一个问题，那就是 `mount` 挂载时，我们期望传递一个 `#app` 的字符串，但是我们查看 `mount` 函数，会发现他期望得到的应该是一个 `element` 对象。

所以我们需要对 `mount` **进行重构**：

1. 在 `packages/runtime-dom/src/index.ts` 的 `createApp` 方法中：

代码块

```
export const createApp = (...args) => {
    const app = ensureRenderer().createApp(...args)

    // 获取到 mount 挂载方法
    const { mount } = app
    // 对该方法进行重构，标准化 container，在重新触发 mount 进行挂载
    app.mount = (containerOrSelector: Element | string) => {
        const container = normalizeContainer(containerOrSelector)
        if (!container) return
        mount(container)
    }

    return app
}

/**
 * 标准化 container 容器
 */
function normalizeContainer(container: Element | string): Element | null {
    if (isString(container)) {
        const res = document.querySelector(container)
        return res
    }
    return container
}
```

2. 那么至此，我们就成功的完成了 `mount` 挂载操作。

接下来，我们就导出 `createApp` 函数，以便直接通过 `const {} = Vue` 的形式进行访问。

1. 查看 `packages/vue/src/index.ts` 模块，我们可以发现现在已经导出了很多方法了，所以我们可以直接使用 `*` 通配符，简化一下对应代码量：

代码块

```
export * from '@vue/reactivity'
export * from '@vue/runtime-core'

export * from '@vue/runtime-dom'

export * from '@vue/vue-compat'

export * from '@vue/shared'
```

至此，整个 `render` 渲染逻辑完成。

01：前言 ◂ 上一节　　下一节 ▸ 03：基于 template 渲染的 createApp 的构...

我要提出意见反馈