Sunday • 更新于 2022-10-19

上一节 01：前言 03：框架实现：… 下一节

02：源码阅读：无状态基础组件挂载逻辑

> Vue 中通常把 **状态** 比作 **数据** 的意思。我们所谓的无状态，指的就是 **无数据** 的意思。

我们先来定一个目标：本小节我们 **仅关注无状态基础组件挂载逻辑**，而忽略掉其他所有。

基于以上目标我们创建对应测试实例 `packages/vue/examples/imooc/runtime/render-component.html`：

代码块

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

  const component = {
    render() {
      return h('div', 'hello component')
    }
  }

  const vnode = h(component)
  // 挂载
  render(vnode, document.querySelector('#app'))
</script>
```

基于以上代码我们知道，`vue` 的渲染逻辑，都会从 `render` 函数，进入 `patch` 函数，所以我们可以直接在 `patch` 函数中进入 `debugger`：

1. 进入 `patch` 函数
2. 触发 `switch`，执行 `if (shapeFlag & ShapeFlags.COMPONENT)`，触发 `processComponent` 方法。该方法即为 **组件渲染** 方法：
    1. 进入 `processComponent`，此时各参数为：

```
const processComponent = (
  n1: VNode | null,  n1 = null
  n2: VNode,  n2 = {__v_isVNode: true, __v_skip: true, type: {
  container: RendererElement,  container = div#app {align: '',
  anchor: RendererNode | null,  anchor = null
  parentComponent: ComponentInternalInstance | null,  parentCo
  parentSuspense: SuspenseBoundary | null,  parentSuspense = n
  isSVG: boolean,  isSVG = false
  slotScopeIds: string[] | null,  slotScopeIds = null
  optimized: boolean  optimized = false
) => {
```

    2. 该函数内部逻辑分为三块：
        1. `Keep alive`
        2. 组件挂载
        3. 组件更新
    3. 我们当前处于 **组件挂载** 状态，所以代码会进入 `mountComponent` 方法

```
const mountComponent: MountComponentFn = (
  initialVNode,  initialVNode = {__v_isVNode: true, __v_skip:
  container,  container = div#app {align: '', title: '', lang:
  anchor,  anchor = null
  parentComponent,  parentComponent = null
  parentSuspense,  parentSuspense = null
  isSVG,  isSVG = false
  optimized  optimized = false
) => {
```

2. 代码执行：

代码块

```
    const instance: ComponentInternalInstance =
          compatMountInstance ||
          (initialVNode.component = createComponentInstance(
            initialVNode,
            parentComponent,
            parentSuspense
          ))
```

该代码通过 `createComponentInstance` 方法生成了 `instance` 实例，我们来看一下 `createComponentInstance` 方法：

1. 进入 `createComponentInstance` 方法
2. 该方法中，最重要的内容就是生成了 `instance` 实例：

   代码块

   ```
       const instance: ComponentInternalInstance = { ... }
   ```

3. **instance** 实例就是 **component** 组件实例

3. 通过以上代码，我们 **生成了** `component` **组件实例**，并且把 **组件实例绑定到了** `vnode.component` **中，即：** `initialVNode.component = instance = 组件实例`
4. 执行 `setupComponent`，该方法主要为了初始化组件的各个数据，比如 `props、slot、render`
   1. 进入 `setupComponent` 方法，**仅关注** `render`
   2. 执行 `setupStatefulComponent(instance, isSSR)`
      1. 进入 `finishComponentSetup`，因为我们当前没有 `setup` 函数
      2. 所以会执行 `finishComponentSetup`
         1. 进入 `finishComponentSetup`：
         2. 查看当前 `instance`，因为不存在 `render`，所以我们需要为 `instance` 的 `render` 赋值
         3. 执行：

            代码块

            ```
                instance.render = (Component.render || NOOP) as InternalRenderFu
            ```

      3. 至此 `instance` 组件实例，具备 `render` 属性
5. 执行 `setupRenderEffect` 方法，这个方法 **非常重要**，我们进入来看一下：
   1. 进入 `setupRenderEffect` 方法，此时参数为：

```
const setupRenderEffect: SetupRenderEffectFn = (
  instance,  instance = {uid: 0, vnode: {…}, type: {…}, parent
  initialVNode,  initialVNode = {__v_isVNode: true, __v_skip:
  container,  container = div#app {align: '', title: '', lang:
  anchor,  anchor = null
  parentSuspense,  parentSuspense = null
  isSVG,  isSVG = false
  optimized  optimized = false
) => {
```

2. 首先：创建 `componentUpdateFn` 函数。
    1. 这个函数因为现在没有执行，所以我们先不需要去管它。但是我们需要知道：**我们创建了一个函数** `componentUpdateFn`
3. 第二段：创建了 `ReactiveEffect` 实例。
    1. `ReactiveEffect` 我们应该是了解的，它可以帮助我们生成一个 **响应性的** `effect` **实例。**
        1. 当执行 `run` 时，会触发 `fn` 。即：第一个参数
        2. 提供了 `scheduler` 调度器的功能。
    2. 明确好了以上内容，我们来看这段代码：

        代码块

        ```
        const effect = (instance.effect = new ReactiveEffect(
            componentUpdateFn,
            () => queueJob(update),
            instance.scope // track it in component's effect scope
        ))
        ```

        1. 在这段代码中，我们把 `componentUpdateFn` 作为第一个参数传入，它将承担 `fn` 的作用。
        2. 第二个参数：是一个匿名函数，它将承担 `scheduler` 调度器的功能。
            1. 其中的 `queueJob` 方法，我们是遇见过的，它是 `packages/runtime-core/src/scheduler.ts` 中的函数，是一个基于 `Promise.resolve()` 的 **微任务队列处理** 的函数，因为通过 `Set` 构建的队列，所以具备去重的能力。
            2. 那么 `update` 是什么呢？我们来看下一行代码
4. 第三段：创建 `update` 对象： `const update: SchedulerJob = (instance.update = () => effect.run())`
    1. 通过以上代码可以看出：我们把 `update` 和 `instance.update` 绑定到了同一块内存空间。
    2. 它们都指向一块函数，即 `() => effect.run()` ，而 `run` 函数的触发，其实是 `fn` 的触发。而 `fn` 又是 `componentUpdateFn` 。
    3. 所以，通过以上的代码我们也知道：当 `update` 函数被触发时，其实触发的是 `componentUpdateFn` 函数。
5. 第四段：触发 `update()` 函数。
    1. 根据刚才所说， `udpate` 的触发，标志着 `componentUpdateFn` 的触发。
    2. 所以此时代码会 **进入** `componentUpdateFn` 函数。
        1. 进入 `componentUpdateFn` 函数。
        2. 观察函数代码可以发现，整个 `componentUpdateFn` 的代码被分成了两部分：
            1. `if(!instance.isMounted)`
            2. `else {}`

3. 我们知道在 `vue` 中存在一个生命周期钩子 mounted ，根据这个可以理解为：

    1. `instance.isMounted === false` 时：表示 **组件挂载前**
    2. `instance.isMounted === true` 时：表示 **组件挂载后**

4. 我们此时处于 `instance.isMounted === false` ，所以为 **挂载前** 状态。接下来要进行的就是 **挂载逻辑**

5. 忽略 **与挂载无关** 的逻辑之后，代码最终执行：

    代码块

    ```
    patch(...)
    ```

6. 对于 `patch` 函数，我们应该是熟悉的，它是一个 **打补丁** 函数

    1. 我们知道 `render` 函数其实就是触发了 `patch` 来完成的渲染。
    2. 对于 `patch` 函数我们知道，它的第二个参数表示为 **新节点**。即：要渲染的内容。
    3. 那么大家想一下对于当前测试实例的组件而言，它要渲染的内容是什么呢?

        1. 是不是就是 **组件** `render` **函数的返回值啊**

    4. 那么在这里 **第二个参数为** `subTree` 。那么我们来看看这个 `subTree` 是什么，能不能验证我们的猜想。

7. `subTree` 变量的创建在 `patch` 函数上面：

    代码块

    ```
    const subTree = (instance.subTree = renderComponentRoot(instance)
    ```

    1. `subTree` 通过 `renderComponentRoot` 方法创建。
    2. 我们再次 `debugger` ，进入这个方法：

        1. 进入 `renderComponentRoot` 方法

            1. 进入之后可以发现，这个函数相当复杂。但是不用担心，记住我们的目标：**我们只关注组件挂载** 相关的。
            2. 根据我们刚才的猜想（6-4），我们知道：组件挂载本质上是 `render` **函数返回值的挂载**，所以我们只关心 `render` 函数。

        2. 代码首先创建了两个变量：

            1. `render` ：这是从 `instance` 组件实例中结构出来的。
            2. `result` ：这是该函数的返回值，即： `subTree`

        3. 代码执行：

            代码块

            ```
            result = normalizeVNode(
              render!.call(
                proxyToUse,
                proxyToUse!,
                renderCache,
                props,
                setupState,
                data,
                ctx
              )
            )
            ```

1. 在这个代码中，我们触发了两个函数：

1.

01：前言 上一节 下一节 03：框架实现：完成无状态基础组件的挂载