Sunday • 更新于 2022-10-19

上一节 05：局部总结：… 07：框架实现：… 下一节

06：源码阅读：有状态的响应性组件挂载逻辑

和之前一样，我们先创建一个 **有状态的组件** `packages/vue/examples/imooc/runtime/` render-component-data.html：

代码块

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

  const component = {
    data() {
      return {
        msg: 'hello component'
      }
    },
    render() {
      return h('div', this.msg)
    }
  }

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

该组件存在一个 `data` 选项， `data` 选项对外 `return` 了一个包含 `msg` 数据的对象。然后我们可以在 `render` 中通过 `this.msg` 来访问到 `msg` 数据。

这样的一种包含 `data` 选项的组件，我们就把它叫做有状态的组件。

那么下面，我们对当前实例进行 `debugger` 操作。

剔除掉之前的重复逻辑，我们从 `mountComponent` 方法开始进入 `debugger`：

1. 进入 `mountComponent` 方法，此时的参数为：

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

2. 通过 `createComponentInstance` 方法，生成 `instance` 实例
3. 代码执行 `setupComponent` 方法，我们知道这个方法是用来初始化组件实例 `instance` 的，我们进入到这个方法来看一下
    1. 进入 `setupComponent` 方法
    2. 执行 `const isStateful = isStatefulComponent(instance)` ，判断当前是否是一个有状态的组件。那么它是怎么进行判定的呢？
        1. 进入 `isStatefulComponent` 方法

代码块

```
1       return instance.vnode.shapeFlag & ShapeFlags.STATEFUL_COMPONENT
```

3. 即：直接通过 `shapeFlag` 进行 **位与运算** 即可

3. 因为我们知道当前是有状态的，此时得到 **isStateful = 4**
4. 跳过 `props` 和 `slots`
5. 因为当前 `isStateful = 4`，所以会执行 `setupStatefulComponent` 方法
    1. 进入 `setupStatefulComponent` 方法
    2. 执行 `const Component = instance.type`，得到 `Component` 实例为：

代码块

```
1   {
2       data() {
3         return {
4           msg: 'hello component'
5         }
6       },
7       render() {
8         return h('div', this.msg)
9       }
10    }
```

    3. 执行 `const { setup } = Component`，从上面的 `Component` 可以看出，我们并不存在 `setup` 函数
    4. 进入 `if` 判断逻辑，将执行 `else` 操作
        1. 执行 `finishComponentSetup(instance, isSSR)`
            1. 进入 `finishComponentSetup` 函数
            2. 同样执行 `const Component = instance.type`，得到 `Component` 实例。
            3. 执行 `instance.render = (Component.render || NOOP)`，为组件实例的 `render` 赋值
            4. 代码继续执行，触发 `applyOptions` 方法
                1. 对 `options` 进行解构，解构之后，得到两个关键属性：
                    1. `dataOptions`：

```
dataOptions: data() {
               return {
                 msg: 'hello component'
               }
             }
```

                    2. `render`：

```
render2: render() {
           return h('div', this.msg)
         }
```

                2. 因为 `dataOptions` 存在，所以 `if (dataOptions)`，被判定为 `true`，处理 `data` 相关逻辑
                    1. 执行 `const data = dataOptions.call(publicThis, publicThis)` 方法
                        1. 我们知道 call 方法会改变 `this` 指向，即把 `dataOptions` 中的

2. 而 `dataOptions` 的值，我们已经知道了（见 `4-4-1-1`）

2. 所以此时的 `data` 值为 `{msg: 'hello component'}`。即：`data` 函数返回值
3. 因为 `data` 当前是一个对象，所以会执行 `instance.data = reactive(data)`

1. 即：通过 `reactive` 方法，构建 `data` 为 `proxy` 实例
2. 此时 `instance.data` 的值为 `proxy` 实例，它的被代理对象为 `{msg: 'hello component'}`

4. 至此 `setupComponent` 完成。**完成之后 `instance` 将具备 `data` 属性，值为 `proxy`，被代理对象为 `{msg: 'hello component'}`**
5. 代码继续执行，触发 `setupRenderEffect` 方法，我们知道该方法为组件的渲染方法

1. 进入 `setupRenderEffect` 方法
2. 创建 `ReactiveEffect` 实例 `effect`
3. 最后触发 `update`，我们知道 `update` 的触发，本质上是 `componentUpdateFn` 的触发。
4. 所以，此时代码会进入 `componentUpdateFn`

1. 进入 `componentUpdateFn`
2. 执行 `const subTree = (instance.subTree = renderComponentRoot(instance))`，获取 `subTree`

1. 我们知道此时 `subTree` 即为 **真实渲染的节点**
2. 那么 `render` 函数的值为：

```
render2: render() {
            return h('div', this.msg)
        }
```

3. 所以真实渲染节点时，我们必须要**把 `this.msg` 替换为 `hello component`**
4. 那么这一步怎么做的呢？
5. 进入 `renderComponentRoot` 方法，我们来一探究竟：

1. 进入 `renderComponentRoot` 方法，此时 `instance` 中：

1. `data` 的值为：

```
▶ data: Proxy {msg: 'hello component'}
```

2. `render` 的值为：

```
render: render() {
            return h('div', this.msg)
        }
```

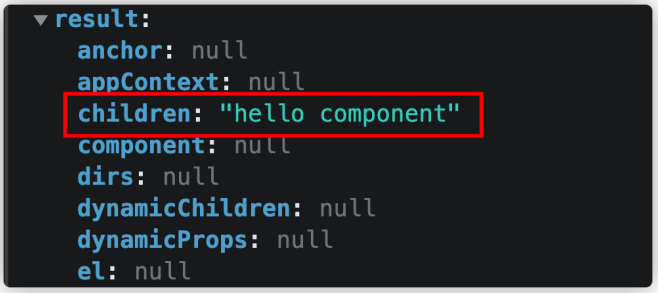
2. 代码执行：

代码块

```
1   result = normalizeVNode(
2           render!.call(
```

```
            renderCache,
            props,
            setupState,
            data,
            ctx
          )
        )
```

3. 这个代码我们之前是见过的，大家应该眼熟。
4. 在这里使用了一个 `call` 方法，对于 `call` 现在大家应该已经熟悉了：**它会改变 `this` 指向**
5. 那么我们期望 `this` 指向改变为什么呢？
    1. 我们期望 `this.msg` 变为 `hello component`
    2. 那么 `this` **的指向是不是就应该为** `data`？
6. 所以，该代码执行完成之后， `result` 的值为：



7. 至此，**我们已经成功解析了** `render`，**把** `this.msg` **成功替换为了** `hello componen t`
8. 后面的逻辑，就与 **无状态组件** 挂载完全相同了。

至此，代码解析完成。

由以上代码可知：

1. 有状态的组件渲染，核心的点是：**让** `render` **函数中的** `this.xx` **得到真实数据**
2. 那么想要达到这个目的，我们就必须要 **改变** `this` **的指向**
3. 改变的方式就是在：生成 `subTree` 时，通过 `call` 方法，指定 `this`

05：局部总结：无状态组件的挂载、更新、... ‹ 上一节　　下一节 › 07：框架实现：有状态的响应性组件挂载逻

我要提出意见反馈

企业服务　网站地图　网站首页　关于我们　联系我们　讲师招募　帮助中心　意见反馈　代码托管

意见反馈　收藏教程　标记书签