

## 全部开发者教程 :三

11: 框架实现: 完成虚拟节点下的 class 和 style 的增强

## 12: 总结

## 第十章: runtime 运行时 - 构建 renderer 渲染器

## 01: 前言

## 02: 源码阅读: 初见 render 函数, ELEMENT 节点的挂载操作

### 03: 框架实现: 构建 renderer 基本架构

#### 04: 框架实现: 基于 renderer 完成 ELEMENT 节点挂载

05: 框架实现: 合并渲染架构, 得到可用的 render 函数

## 06: 源码阅读: 渲染更新, ELEMENT 节点的更新操作

## 07: 框架实现: 渲染更新, ELEMENT 节点的更新实现

## 08: 源码阅读: 新旧节点不同



Sunday • 更新于 2022-10-19

◀ 上一节 05: 框架实现: ... 07: 框架实现: ... 下一节 ▶

## 06: 源码阅读: 渲染更新, ELEMENT 节点的更新操作

我们知道对于 `render` 而言，除了有 **挂载** 操作之外，还存在 **更新** 操作。

所谓更新操作指的是：**生成一个新的虚拟 DOM 树**，运行时渲染器遍历这棵新树，将它与旧树进行比较，然后将必要的更新应用到**真实 DOM** 上去。

所以我们可以创建如下测试实例 `packages/vue/examples/imooc/runtime/render-element-update.html`：

### <> 代码块

```
1 <script>
2   const { h, render } = Vue
3
4   const vnode = h('div', {
5     class: 'test'
6   }, 'hello render')
7   // 挂载
8   render(vnode, document.querySelector('#app'))
9
10  // 延迟两秒，生成新的 vnode，进行更新操作
11  setTimeout(() => {
12    const vnode2 = h('div', {
13      class: 'active'
14    }, 'update')
15    render(vnode2, document.querySelector('#app'))
16  }, 2000);
17 </script>
```

两秒之后，我们可以发现 **DOM** 发生了更新。

那么在这样一个更新操作中，render 又是如何操作的呢？

我们知道每次的 `render` 渲染 `ELEMENT`，其实都会触发 `processElement`，所以我们可以直接在 `processElement` 中增加断点，进入 `debugger`：

1. 第一次触发 `processElement` 为 挂载 操作，可以直接 跳过
2. 第二次触发 `processElement` 为 更新操作：

1. 此时 n1 (旧的) 存在值为:

```
▼ n1:
  anchor: null
  appContext: null
  children: "hello render"
  component: null
  dirs: null
  dynamicChildren: null
  dynamicProps: null
  ▶ el: div.test
    key: null
    patchFlag: 0
  ▶ props: {class: 'test'}
```

2. n2（新的）存在值为：

```
▼ n2:
  anchor: null
  appContext: null
  children: "update"
  component: null
  dirs: null
  dynamicChildren: null
  dynamicProps: null
  el: null
  key: null
  patchFlag: 0
  ▶ props: {class: 'active'}
```

3. 代码执行 `patchElement`，即：更新操作：

1. 执行 `const el = (n2.el = n1.el!)`。使 **新旧 vnode 指向同一个 el 元素**
2. 执行 `patchChildren(...)` 方法，表示 **为子节点打补丁**：
  1. 进入 `patchChildren` 方法
  2. 执行 `c1 = xx`、`c2 = xx`，为 `c1`、`c2` 赋值，此时 `c1` 为 **旧节点的 children**，`c2` 为 **新节点的 children**
  3. 执行 `if (shapeFlag & ShapeFlags.TEXT_CHILDREN)`，我们知道此时 **子 children 的 shapeFlag = 9**，是 **可以** & `ShapeFlags.TEXT_CHILDREN`
  4. 而 `prevShapeFlag = 9`，是 **不可以** & `ShapeFlags.ARRAY_CHILDREN` 的
  5. 所以会触发 `hostSetElementText`。我们知道 `hostSetElementText` 其实是一个 **设置 text 的方法**
  6. 那么此时 **text 内容更新完成，浏览器展示的 text 会发生变化**
3. 代码继续执行
4. 执行 `patchProps(...)` 方法，表示 **为 props 打补丁**
  1. 进入 `patchProps` 方法，此时新旧 props 为：

```
▶ newProps: {class: 'active'}
▶ oldProps: {class: 'test'}
```

2. 查看代码可以发现代码执行了两次 `for` 循环操作：

1. 第一次循环执行 `for in newProps`，执行 `hostPatchProp` 方法设置新的 `props`
2. 第二次循环执行 `for in oldProps`，执行 `hostPatchProp`，配合 `!(key in newProps)` 判断，删除 **没有被指定的旧属性**，比如：

<> 代码块

```
1 // 原属性:
2 {
3   class: 'test',
4   id: 'test-id'
5 }
6 // 新属性:
7 {
8   class: 'active'
9 }
```

则 **删除** `id`

3. 至此 `props` 更新完成

4. 至此，更替更新完成

由以上代码可知：

1. 无论是 **挂载** 还是 **更新** 都会触发 `processElement` 方法，状态根据 `oldValue` 进行判定
2. `Element` 的更新操作有可能 **会在同一个 `el` 中完成**。（注意：仅限元素没有发生变化时，如果新旧元素不同，那么是另外的情况，后面会专门讲解！！）
3. 更新操作分为：

1. `children` 更新
2. `props` 更新

05：框架实现：合并渲染架构，得到可用的 r... ◀ 上一节      下一节 ▶ 07：框架实现：渲染更新，ELEMENT 节点...

✎ 我要提出意见反馈