

### 13: 框架实现: 响应性数据改变, 触发组件的响应性变化



◀ 上一节 07: 框架实现: ... 09: 框架实现: ... 下一节 ▶

在前面我们查看《有状态的响应性组件挂载逻辑》时，其实已经在源码中查看到了对应的一些生命周期处理逻辑。

我们知道 `vue` 把生命周期叫做生命周期回调钩子，说白了就是一个：在指定时间触发的回调方法。

我们查看 `packages/runtime-core/src/component.ts` 中 第213行 可以看到 `ComponentInternalInstance` 接口，该接口描述了组件的所有选项，其中包含：

### <> 代码块

```

1  /**
2   * @internal
3   */
4   [LifecycleHooks.BEFORE_CREATE]: LifecycleHook
5   /**
6   * @internal
7   */
8   [LifecycleHooks.CREATED]: LifecycleHook
9   /**
10  * @internal
11  */
12  [LifecycleHooks.BEFORE_MOUNT]: LifecycleHook
13  /**
14  * @internal
15  */
16  [LifecycleHooks.MOUNTED]: LifecycleHook
17  /**
18  * @internal
19  */
20  [LifecycleHooks.BEFORE_UPDATE]: LifecycleHook
21  /**
22  * @internal
23  */
24  [LifecycleHooks.UPDATED]: LifecycleHook
25  /**
26  * @internal
27  */
28  [LifecycleHooks.BEFORE_UNMOUNT]: LifecycleHook
29  /**
30  * @internal
31  */
32  [LifecycleHooks.UNMOUNTED]: LifecycleHook
33  /**
34  * @internal
35  */
36  [LifecycleHooks.RENDER_TRACKED]: LifecycleHook
37  /**
38  * @internal
39  */
40  [LifecycleHooks.RENDER_TRIGGERED]: LifecycleHook
41  /**
42  * @internal
43  */
44  [LifecycleHooks.ACTIVATED]: LifecycleHook

```

## 索引目录

## 08: 源码阅读: 维



```

47     */
48     [LifecycleHooks.DEACTIVATED]: LifecycleHook
49     /**
50     * @internal
51     */
52     [LifecycleHooks.ERROR_CAPTURED]: LifecycleHook
53     /**
54     * @internal
55     */
56     [LifecycleHooks.SERVER_PREFETCH]: LifecycleHook<() => Promise<unknown>>
57

```

以上全部都是 vue 生命周期回调钩子的选项描述，大家可以在 [官方文档](#) 中查看到详细的生命周期钩子描述。

这些生命周期全部都指向 LifecycleHooks 这个 enum 对象：

<> 代码块

```

1  export const enum LifecycleHooks {
2    BEFORE_CREATE = 'bc',
3    CREATED = 'c',
4    BEFORE_MOUNT = 'bm',
5    MOUNTED = 'm',
6    BEFORE_UPDATE = 'bu',
7    UPDATED = 'u',
8    BEFORE_UNMOUNT = 'bum',
9    UNMOUNTED = 'um',
10   DEACTIVATED = 'da',
11   ACTIVATED = 'a',
12   RENDER_TRIGGERED = 'rtg',
13   RENDER_TRACKED = 'rtc',
14   ERROR_CAPTURED = 'ec',
15   SERVER_PREFETCH = 'sp'
16 }

```

在 LifecycleHooks 中，对生命周期的钩子进行了简化的描述，比如：created 被简写为 c。即：c 方法触发，就意味着 created 方法被回调。

那么明确好了这个之后，我们来看一个测试实例 `packages/vue/examples/imooc/runtime/redner-component-hook.html`：

<> 代码块

```

1  <script>
2    const { h, render } = Vue
3
4    const component = {
5      data() {
6        return {
7          msg: 'hello component'
8        }
9      },
10     render() {
11       return h('div', this.msg)
12     },
13     // 组件初始化完成之后
14     beforeCreate() {
15       alert('beforeCreate')
16     },
17     // 组件实例处理完所有与状态相关的选项之后
18     created() {
19       alert('created')
20     },
21     // 组件被挂载之前
22     beforeMount() {
23       alert('beforeMount')
24     },
25     // 组件被挂载之后
26     mounted() {
27       alert('mounted')
28     }
29   }

```

[意见反馈](#)

[收藏教程](#)

[标记书签](#)



```

29     }
30
31     const vnode = h(component)
32     // 挂载
33     render(vnode, document.querySelector('#app'))
34 </script>

```

我们知道对于组件的挂载其实会触发 `mountComponent` 方法，所以本次，我们直接从该方法进行 `debugger`，**注意：** 本次我们仅关心生命周期回调的逻辑：

1. 进入 `mountComponent` 方法

2. 触发 `setupComponent(instance)` 方法

1. 进入 `setupComponent(instance)` 方法

2. 触发 `setupStatefulComponent` 方法

1. 进入 `setupStatefulComponent` 方法

2. 触发 `finishComponentSetup` 方法

1. 进入 `finishComponentSetup` 方法

2. 触发 `applyOptions(instance)` 方法

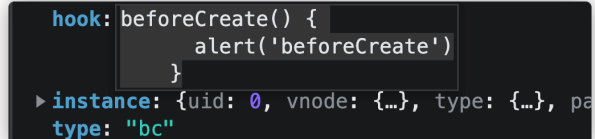
1. 进入 `applyOptions` 方法

2. 执行 `if (options.beforeCreate)`

1. 我们知道 `beforeCreate` 是生命周期回调钩子，我们当前是存在这个回调钩子的

2. 所以接下来会执行 `callHook(options.beforeCreate, instance, LifecycleHooks.BEFORE_CREATE)`，我们进入到该方法来看一下

1. 进入 `callHook`，此时的参数为：



```

hook: beforeCreate() {
  alert('beforeCreate')
}
instance: {uid: 0, vnode: {...}, type: {...}, parent: null, type: "bc"}

```

1. 在参数中，我们可以很清楚的看到 `hook` 的值就是我们写入到 `beforeCreate` 函数

2. 接下来触发 `callWithErrorHandling`

1. 对于该方法我们是熟悉的，它本质上就是：**通过 `try...catch` 捕获函数执行**的一个方法

2. 所以我们可以直接理解为 **在组件初始化完成之后，触发了 `beforeCreate` 方法**

3. 代码继续向下进行，此时触发了 `beforeCreate`，**进行 `alert` 打印**

4. 接下来代码触发 `if (created) {...}`

1. 和刚才的 `beforeCreate` 触发一样

2. 此时 **在组件实例处理完所有与状态相关的选项之后，触发了 `create` 生命周期回调**

3. 至此，我们在 `applyOptions` 方法中，触发了 `beforeCreate` 和 `created`

4. 代码继续执行~~~

5. 触发 `registerLifecycleHook(onBeforeMount, beforeMount)` 方法

```
hook: beforeMount() {  
    alert('beforeMount')  
}
```

2. 执行 `register((hook as Function).bind(publicThis))`

1. 这段代码可以分成两块来看：

1. `(hook as Function).bind(publicThis)`：我们知道对于 `bind` 方法而言，它本身是可以改变 `this` 指向，并且返回一个新的函数
2. `register(新的函数)`：该方法从名字来看是注册的意思。那么我们进入这个方法，看看它内部做了什么事情：

1. 进入 `register`
2. 进入之后可以发现，它本质上是触发了 `createHook` 方法

1.

07: 框架实现：有状态的响应性组件挂载逻辑 ◀ 上一节      下一节 ▶ 09: 框架实现：组件生命周期回调处理逻辑

 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](#)

Copyright © 2022 imooc.com All Rights Reserved | 京ICP备 12003892号-11      京公网安备11010802030151号



 意见反馈

 收藏教程

 标记书签