第3章 Vue3整体实现

CHAPTER 3

本章将通过对源码的调试，介绍 Vue3 的整体实现。本章内容是第 2 章内容的延伸，核心逻辑与第 2 章基本相同。读者可以将两章内容对照进行学习。读者可以跟随本章的内容学习源码调试方法，简单理解 Vue3 的运行原理。

观看视频速览本章主要内容。

3.1 源码调试

（1）基础环境：npm、node、vscode；

（2）下载源码；

```
git clone git@github.com:vuejs/vue-next.git
```

注：git@github.com 为 ssh 下载，若未设置免密，则可使用 HTTPS 下载。

（3）安装项目依赖；

（4）安装 yarn 插件；

```
npm install yarn -g
```

（5）安装依赖；

```
yarn install
```

安装依赖时，如果提示需安装 pnpm，则会在执行时提示“This repository requires using pnpm as the package manager for scripts to work properly.”，如图 3.1 所示。pnpm 命令可以通过 npm 安装。

```
zhangtinghangdeMacBook-Pro:vue3-detail zhang$ yarn install
yarn install v1.22.10
info No lockfile found.
$ node ./scripts/preinstall.js
This repository requires using pnpm as the package manager  for scripts to work properly.

error Command failed with exit code 1.
info Visit https://yarnpkg.com/en/docs/cli/install for documentation about this command.
```

图 3.1 报错信息

（6）安装 pnpm；

```
npm install -g pnpm
```

使用 pnpm 时需注意，pnpm 命令对 node 版本有依赖，例如，在本地执行 pnpm 时，提示 node 版本过低，此时需要升级 node 版本，以支持依赖，提示信息如图 3.2 所示。

```
zhangtinghangdeMacBook-Pro:vue3-detail zhang$ pnpm install
ERROR: This version of pnpm requires at least Node.js v12.17
The current version of Node.js is v12.16.3
Visit https://r.pnpm.io/comp to see the list of past pnpm versions with respective Node.js version support.
```

图 3.2 提示信息

（7）升级 node 版本。

本书的操作环境安装有 nvm，可以直接切换多个版本 node，关于 node 升级此处不过多介绍，完成 node 版本升级后，通过 pnpm 安装依赖。

3.1.1 代码调试

依赖包安装完成后即可运行 vue 代码，执行 npm run build 打包最新的代码。完成打包后可以在 packages/vue/dist 内找到对应文件，打包结果如图 3.3 所示。

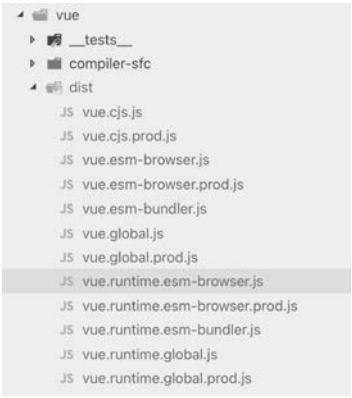

图 3.3 打包结果

完成打包后，新建 html 文件并引入编译好的 vue.global.js 文件，在 html 文件内实现简单的内容：渲染一个按钮和提示语，单击按钮后可以反转提示语的字母位置。该 demo 实现十分简单，但是通过该操作可以帮助读者探究 Vue3 内部运行原理，覆盖 Vue3 的核心逻辑。整体代码如下：

```
<!DOCTYPE html>
<html>
  <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <meta http-equiv="X-UA-Compatible" content="ie=edge">
      <title>Hello Vue3.0</title>
      <style>
          body,
          #app {
              text-align: center;
              padding: 30px;
          }
      </style>
      <script src="./packages/vue/dist/vue.global.js"></script>
  </head>
  <body>
    <h3>reactive</h3>
    <div id="app"></div>
  </body>
  <script>
    const { createApp, reactive } = Vue;
    const App = {
      template: `<button @click="click">reverse</button><div style="margin-top: 20px">
{{ state.message }}</div>`,
      setup() {
        console.log("setup");
        const state = reactive({
          message: "Hello Vue3!!"
```

```
        });
        click = () => {
          state.message = state.message.split("").reverse().join("");
        };
        return {
          state,
          click
        };
      }
    };
    createApp(App).mount("#app");
  </script>
</html>
```

上述代码使用了 Vue3 的 createApp()函数和 reactive()函数，createApp()函数完成组件的初始化、挂载、更新和内部 setup()函数执行等过程。reactive()函数完成内部响应式数据的实现。该 demo 实现的功能很简单，但覆盖了整个 Vue3 的核心逻辑。

上述页面完成后，即可在浏览器打开页面进行调试。查看 Source 标签页可以发现 Vue3 源码全在一个文件内不利于阅读和调试，需手动开启 sourceMap。

3.1.2 开启 sourceMap

开启 sourceMap 需要修改 rollup.config.js 文件内容，在 createConfig 方法内配置 output.sourcemap=true，设置 sourceMap 如图 3.4 所示。

```
92  output.exports = isCompatPackage ? 'auto' : 'named'
93  //output.sourcemap = !!process.env.SOURCE_MAP
94  output.externalLiveBindings = false
95  output.sourcemap = true
96
97  if (isGlobalBuild) {
98    output.name = packageOptions.name
99  }
```

图 3.4 设置 sourceMap

在 tsconfig.json 中配置 sourceMap 输出，将 sourceMap 从 false 改为 true，打开 sourceMap，如图 3.5 所示。

```
1  {
2    "compilerOptions": {
3      "baseUrl": ".",
4      "outDir": "dist",
5      "sourceMap": true,
6      "target": "es2016",
7      "useDefineForClassFields": false,
8      "module": "esnext",
9      "moduleResolution": "node",
10     "allowJs": false,
11     "strict": true,
12     "noUnusedLocals": true,
13     "experimentalDecorators": true,
```

图 3.5 打开 sourceMap

完成后重新打包即可开启 sourceMap，页面涉及的 Vue 源码如图 3.6 所示。

3.1.3 总结

本节主要介绍 Vue3 源码调试的步骤。本节的设置和调试作用将贯穿全文，后续源码解读和分析均会使用该 demo，因此学习并开始 Vue 源码调试十分重要，建议读者通过对源码的调试(debug)，学习 Vue3 的内部实现，提升自己的动手和代码阅读能力。

图 3.6　Vue 源码

后面将以表格的形式列出每个函数对应的文件，由于文件路径较长，为了便于阅读，以简写路径代替真实路径。

3.2　createApp()函数

3.2.1　涉及文件

createApp()函数涉及文件路径如表 3.1 所示。

表 3.1　createApp()函数涉及文件路径

名　称	简写路径	文件路径
createApp	$ runtime-dom_index	$ root/packages/runtime-dom/src/index.ts
ensureRenderer	$ runtime-dom_index	$ root/packages/runtime-dom/src/index.ts
baseCreateRenderer	$ runtime-core_render	$ root/packages/runtime-core/src/render.ts
render	$ runtime-core_render	$ root/packages/runtime-core/src/render.ts
createAppAPI	$ runtime-core_apiCreateApp	$ root/packages/runtime-core/src/apiCreateApp.ts

3.2.2　调用 createApp()函数

调用 createApp()函数返回一个应用实例，它的实现在 $ runtime-dom_index.ts 文件内，Vue3 源码内的 createApp()函数内部主要创建实例并对实例的 mount()方法进行重写，代码如下所示：

```
export const createApp = ((...args) => {
  // 创建渲染器并调用渲染器的 createApp()方法创建 app 实例
  const app = ensureRenderer().createApp(...args);
  // 重写 app 的 mount()方法
  const { mount } = app;
  app.mount = (containerOrSelector: Element | string): any => {
    // ...
```

```
    };
    // 返回 app
    return app;
}) as CreateAppFunction < Element >;
```

上述代码,主要完成两项工作:

(1) 初始化 app 实例;

(2) 重写 app 实例的 mount()方法。

在初始化 app 时,首先创建渲染器,并且调用渲染器的 createApp()方法,查看 ensureRenderer()函数实现逻辑,具体代码如下:

```
// $ runtime - dom_index
function ensureRenderer() {
    // 若 renderer 存在则直接返回,若不存在则创建
    return renderer ||
    (renderer = createRenderer < Node, Element >(rendererOptions))
}
```

该代码采用单例模式实现,若 renderer 存在则直接返回,若不存在则调用 createRenderer()函数创建。使用单例模式优化可以避免重复创建 renderer,减少性能消耗。调用 createRenderer()函数时传入 rendererOptions 对象,该对象内含有与 DOM 相关的方法,在 DOM 渲染等时将会大量使用。在查看 createRenderer()函数前,先简单介绍该对象内的实现。该对象由 3 个对象合并而来,分别是 patchProp、forcePatchProp 和 nodeOps,patch(class、style 等)的处理方法在前两个对象内,DOM 的处理方法在 nodeOps 对象内。

rendererOptions 生成代码实现如下:

```
const rendererOptions = extend({ patchProp, forcePatchProp }, nodeOps)
```

rendererOptions 流程图如图 3.7 所示。

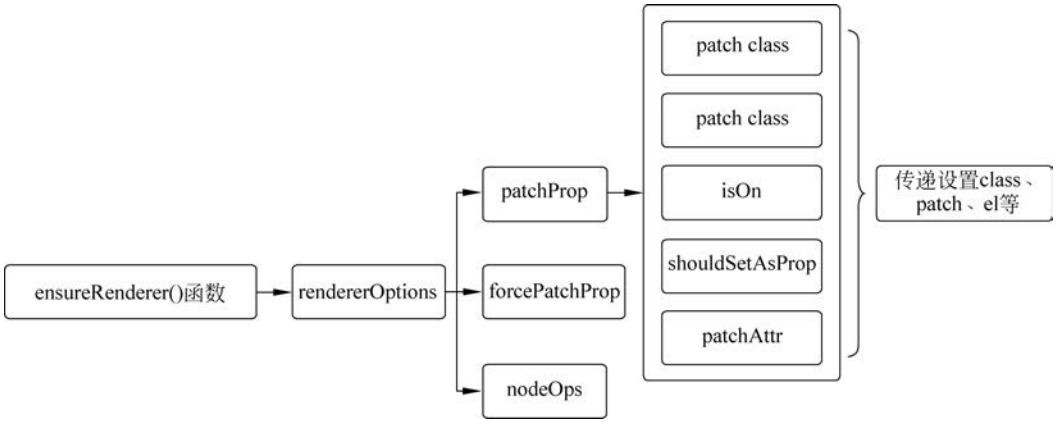

图 3.7 rendererOptions 流程图

3.2.3 调用 createRenderer()函数

继续介绍创建 createApp()函数的主线逻辑,调用 $ runtime-core _ render 文件内 createRenderer()函数。该函数采用高阶函数的方式,在函数内部返回另外一个函数,具体代码如下:

```
// $ runtime - core_render
export function createRenderer <
```

```
  HostNode = RendererNode,
  HostElement = RendererElement
>(options: RendererOptions<HostNode, HostElement>) {
  return baseCreateRenderer<HostNode, HostElement>(options)
}
```

在返回的函数 baseCreateRenderer()中,既可以传入浏览器平台的 options,也可以传入小程序等其他环境的 options,实现对上层平台 API 的解耦。

在 createRenderer()函数所在的文件内可以找到 3 个 baseCreateRenderer()函数定义,这是 TypeScript 内函数重载的写法。可以根据传入参数类型选择不同的函数体执行。涉及逻辑代码如下:

```
// $ runtime-core_render
// overload 1: no hydration
function baseCreateRenderer<
  HostNode = RendererNode,
  HostElement = RendererElement
>(options: RendererOptions<HostNode, HostElement>): Renderer<HostElement>
// overload 2: with hydration
function baseCreateRenderer(
  options: RendererOptions<Node, Element>,
  createHydrationFns: typeof createHydrationFunctions
): HydrationRenderer
// implementation
function baseCreateRenderer(
  options: RendererOptions,
  createHydrationFns?: typeof createHydrationFunctions
): any {
…
}
```

此处着重介绍第三个 baseCreateRenderer()函数的实现逻辑:

(1) 从 options 中取出平台相关的 API;

(2) 创建 patch 函数;

(3) 创建 render 函数;

(4) 返回一个渲染器对象。

上述 4 点简单概括了 baseCreateRenderer()函数所做工作,对函数精简后的主要逻辑代码如下:

```
// $ runtime-core_render
function baseCreateRenderer(
  options: RendererOptions,
  createHydrationFns?: typeof createHydrationFunctions
): any {
    // 从 options 中取出传递进来的 API
    const { … } = options
    // 创建 patch 函数
    const patch: PatchFn = ( … ) => {}
    const mountElement = () => {}
    const patchProps = () => {}
    const mountComponent: MountComponentFn = () => {}
    const updateComponent = () => {}
    const unmount: UnmountFn = () => {}
    // 创建 render 函数
```

```
    const render: RootRenderFunction = (vnode, container) => {}
   // 返回渲染器对象
   return {
    render,
    hydrate,
    createApp: createAppAPI(render, hydrate)
  }
}
```

该函数内部需要实现很多内置方法,包括第 2 章介绍的 mount 挂载、props 处理、unmount 卸载、update 更新、patch 和 render 等。由于代码较为繁杂,不利于理解主体逻辑,在后续内容将逐步细化分析,此处暂不给出具体实现。

注:hydrate 参数主要在 SSR 时使用。将后端渲染中的纯字符串转换为可展示的 HTML 页面,这个过程就是 hydrate。

在查看 createAppAPI()函数实现前,先查看 render 函数,它用于内部的 render 渲染操作,根据组件状态判断执行挂载还是更新,该函数内部的实现代码如下:

```
// $ runtime - core_render
const render: RootRenderFunction = (vnode, container) => {
    if (vnode == null) {
      if (container._vnode) {
        unmount(container._vnode, null, null, true)
      }
    } else {
      patch(container._vnode || null, vnode, container)
    }
    flushPostFlushCbs()
    container._vnode = vnode
  }
```

上述代码与第 2 章介绍 watchEffect()函数内传入的匿名函数十分相似。此处调用 render 函数将会执行 unmount()、patch()和 flushPostFlushCbs()函数。

继续查看 createAppAPI()函数,它位于 $ runtime-core_apiCreateApp 文件内。通过查看该函数的实现,可以发现内部又返回了一个 createApp()函数。此处多次返回函数的设计主要是采用函数柯里化的方式持有 render 函数以及 hydrate 参数,避免了用户在应用内使用时需要手动传入 render 函数给 createApp。

根据 createAppAPI()函数传递的参数可知,render 函数被传递到该函数内部,涉及的关键代码如下:

```
// $ runtime - core_apiCreateApp
export function createAppAPI < HostElement >(
  render: RootRenderFunction,
  hydrate?: RootHydrateFunction
): CreateAppFunction < HostElement > {
  return function createApp(rootComponent, rootProps = null) {
    const context = createAppContext()
    const installedPlugins = new Set()
    let isMounted = false
    const app: App = (context.app = {
      get config() {},
      set config(v) {},
      use(plugin: Plugin, ...options: any[]) {},
      mixin(mixin: ComponentOptions) {},
```

```
      component(name: string, component?: Component): any {},
      directive(name: string, directive?: Directive) {},
      mount(rootContainer: HostElement, isHydrate?: boolean): any {},
      unmount() {},
      provide(key, value) {}
    })
    return app
  }
```

该函数内部实现代码较多，内部主要做如下处理：

(1) 判断 rootProps 类型，调整其值；

(2) 通过 createAppContext 创建 context 上下文；

(3) 初始化已安装插件 installedPlugins 和 isMounted 状态对象；

(4) 通过对象字面量的方式创建一个完全实现 app 实例接口的对象并返回。

其中创建 context 上下文内容如下：

```
export function createAppContext(): AppContext {
  return {
    app: null as any,
    config: {
      isNativeTag: NO,
      performance: false,
      globalProperties: {},
      optionMergeStrategies: {},
      errorHandler: undefined,
      warnHandler: undefined,
      compilerOptions: {}
    },
    mixins: [],                          // 混入
    components: {},                      // 组件
    directives: {},                      // 指令
    provides: Object.create(null),       // 注入对象
    optionsCache: new WeakMap(),         // options 缓存
    propsCache: new WeakMap(),           // props 缓存
    emitsCache: new WeakMap()            // emits 缓存
  }
}
```

该函数初始化 app 全局方法，包括全局混入、全局组件、全局指令和全局注入等。完成 context 上下文创建后，再查看创建 app 实例的部分，代码如下：

```
function createApp(rootComponent, rootProps = null) {
  …
  // 创建 context 上下文
  const context = createAppContext()
  // 创建插件安装 set
  const installedPlugins = new Set()
  // 是否挂载
  let isMounted = false
const app: App = (context.app = {
  …
  use(plugin: Plugin, ...options: any[]) { … }
  mixin(mixin: ComponentOptions) { … }
  component(name: string, component?: Component): any { … }
  directive(name: string, directive?: Directive) { … }
  mount(rootContainer: HostElement, isHydrate?: boolean): any { … }
```

```
    unmount() { ... }
    provide(key, value) { ... }
    return app
    }
}
```

组件内的方法和属性大部分在此处声明和初始化,该函数主要涉及:

(1) 基本属性,包括_uid(唯一标识)、props、配置设置等;

(2) 插件安装,通过调用 use()方法处理插件相关内容;

(3) 混入函数,通过 mixin()函数处理混入相关内容;

(4) 组件内容,通过 component()函数处理;

(5) 指令内容,通过 directive()函数注册全局指令;

(6) 挂载组件,通过 mount()函数进行挂载;

(7) 卸载组件,通过 unmount()函数进行卸载;

(8) 注入组件,通过 provide()函数进行注入。

通过该函数,可以大致了解 Vue 的内部情况,完成初始化操作后,返回 app 对象。

此处执行初始化操作,且为非 SSR 模式,所以设置 isMounted=false,isHydrate=false。通过 createVNode 方法创建根组件 VNode,并由 render 函数从根组件 VNode 进行渲染,传入 VNode、rootContainer 参数,完成渲染后,将 isMounted 标识为 true(已挂载),然后绑定根实例和根容器,最后返回根组件的代理,完成挂载。主要逻辑代码如下:

```
mount(
  rootContainer: HostElement,
  isHydrate ? : boolean,
  isSVG ? : boolean
): any {
  if (!isMounted) {
    ...
    const vnode = createVNode(
      rootComponent as ConcreteComponent,
      rootProps
    )
    vnode.appContext = context
    ...
    if (isHydrate && hydrate) {
      hydrate(vnode as VNode<Node, Element>, rootContainer as any)
    } else {
      render(vnode, rootContainer, isSVG)
    }
    isMounted = true
    app._container = rootContainer;
    (rootContainer as any).__vue_app__ = app
    return getExposeProxy(vnode.component!) || vnode.component!.proxy
  }
  ...
}
```

注:此处的 render 方法是上文 createAppAPI 方法特有的,该方法是由上文传入,所以此处可以直接调用。

app 初始化完成并且调用 mount()方法后,回到本节开始的地方,会发现通过 app 实例解构出 mount()方法后对其进行了重写,整个过程通过处理容器元素、判断根组件是否存在模板

或渲染函数、清空容器内容 3 个步骤，确保原 mount()方法的正常执行。在完成这部分工作后 createApp(app).mount()全流程就完成了。重写 mount()方法的具体代码如下：

```
app.mount = (containerOrSelector: Element | string): any => {
    // 标准化容器元素 element | string --> element
    const container = normalizeContainer(containerOrSelector)
    // 若找不到元素,则直接 return
    if (!container) return
    // 拿到 app 组件
    const component = app._component
    // 如果既不是函数组件也没有 render 和模板,则取容器元素的 innerHTML 当作模板
    if (!isFunction(component) && !component.render && !component.template) {
      component.template = container.innerHTML
      ...
    }
    // 在挂载前清空容器的 innerHTML
    container.innerHTML = ''
    // 执行挂载,得到返回的代理对象
    const proxy = mount(container, false, container instanceof SVGElement)
    if (container instanceof Element) {
      container.removeAttribute('v-cloak')
      container.setAttribute('data-v-app', '')
    }
    return proxy
}
```

在对 mount 重写时，主要对 DOM 相关逻辑进行处理。例如，通过 innerHTML 将初始化完成的内容挂载到页面上等操作，是为了将浏览器平台的内容与核心内容进行分离，也方便用户自定义 mount 挂载逻辑，使之应用到不同的平台，为实现跨平台提供帮助。

Vue 初始化 app 实例如图 3.8 所示。

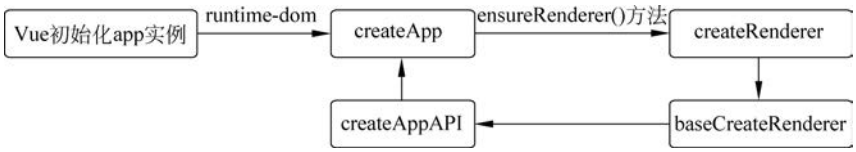

图 3.8 Vue 初始化 app 实例

3.2.4 总结

本节主要讲解整个 Vue 初始化 app 实例的流程，结合第 2 章对 createApp 核心内容和调用流程的讲解，相信读者对整个 Vue 的初始化有了深刻的认识。熟悉该调用流程后，我们将在后面展开介绍内部的实现原理。本节内也提到了很多设计模式等的使用，在阅读源码的过程中，可以多进行学习，并应用到自己的项目中，本节主要使用的技巧包括但不限于，函数柯里化、单例模式和函数重载的应用。

3.3 mounted 挂载

3.3.1 涉及文件

mounted 挂载涉及文件路径如表 3.2 所示。

表 3.2 mounted 挂载涉及文件路径

名 称	简写路径	文件路径
mount	$ runtime-core_apiCreateApp	$ root/packages/runtime-core/src/apiCreateApp.ts
processComponent	$ runtime-core_renderer	$ root/packages/runtime-core/src/renderer.ts
setupComponent	$ runtime-core_components	$ root/packages/runtime-core/src/component.ts
setupStatefulComponent	$ runtime-core_components	$ root/packages/runtime-core/src/component.ts
finishComponentSetup	$ runtime-core_components	$ root/packages/runtime-core/src/component.ts

本节将详细介绍 Vue3 中 mounted 挂载时的整体流程,通过该流程了解组件挂载的详细情况。下面对涉及的虚拟 DOM 和 VNode 的相关知识进行简单介绍。

mount(挂载)内部将 VNode 转换为真实的 DOM 结构再挂载到 DOM 根节点内,mount 的调用在 createApp 函数内已有简单介绍,对 render 函数的作用也有一定了解,本节将继续围绕该内容展开介绍。

虚拟 DOM 本质是通过 JavaScript 对象描述真实 DOM 的结构和属性。因为在 DOM 操作的过程中会涉及页面的重绘和回流,所以引入虚拟 DOM 减少对 DOM 对象的操作,降低性能损耗。如果只是对页面内某个极小部分修改就造成整个 DOM 树的重绘,那么会有极大的性能损耗。虚拟 DOM 概念从 Vue2 开始引入,在真实操作 DOM 前对需要修改的内容进行判断,保证只有 DOM 结构的最小化修改,减少重绘和回流的次数,以达到节约性能损耗的目的。在 Vue3 中依然保留虚拟 DOM,并且对其进行大幅度优化。

VNode 是虚拟 DOM 的外在表现,对 VNode 的增、删、改和查可映射为对 DOM 的增、删、改和查。整个渲染核心均是围绕虚拟 DOM 展开的,通过 VNode 的形式呈现。该对象声明位置在 $ runtime-core_vnode 文件内,具体声明及含义可在 4.1.2 节查看。

对虚拟 DOM 和 VNode 有简单了解后,再回到 $ runtime-core_apiCreateApp 文件内查看与 mount 相关的内容。mount 内部首先会判断是否已经挂载,如果已挂载,则结束 mount 函数执行。如果未挂载,则创建一个 VNode,再执行 render 渲染。mount 的挂载本质是调用 apiCreateApp()函数,该函数简化后代码如下:

```
// $ runtime-core_apiCreateApp
mount(rootContainer: HostElement, isHydrate?: boolean): any {
  if (!isMounted) {
    // 创建根组件 VNode
    const vnode = createVNode(
      rootComponent as ConcreteComponent,
      rootProps
    )
    // 在根节点 VNode 上存储应用上下文
    vnode.appContext = context
    // HMR 根节点重载
    if (__DEV__) {
      context.reload = () => {
        render(cloneVNode(vnode), rootContainer)
      }
    }
    if (isHydrate && hydrate) {
```

```
        hydrate(vnode as VNode < Node, Element >, rootContainer as any)
      } else {
      // 从根组件 VNode 开始渲染
        render(vnode, rootContainer)
      }
      // 标识已挂载
      isMounted = true
      // 绑定根实例和根容器
      app._container = rootContainer
      // 调试工具和探测
      ;(rootContainer as any).__vue_app__ = app
      if (__DEV__ || __FEATURE_PROD_DEVTOOLS__) {
        devtoolsInitApp(app, version)
      }
      // 返回根组件的代理
      return vnode.component!.proxy
    } else if (__DEV__) {
      // ...
    }
  }
```

上述代码通过 createVNode()函数创建 VNode 上下文,将创建好的上下文通过 render 函数进行渲染。此处 mounted 挂载将分为 3 个步骤:

(1) 根据根组件信息创建根组件 VNode,在根节点 VNode 上存储应用上下文;

(2) 从根组件 VNode 开始递归渲染生成 DOM 树并挂载到根容器上;

(3) 处理根组件挂载后的相关工作并返回根组件的代理。

注:此 render 函数是内部渲染函数,内部利用递归的方式将 VNode 转换为 DOM 结构使用,与后面将模板转换为 VNode 的 render 渲染函数无任何关联。

3.3.2 创建根组件 VNode

通过调用 createVNode()函数实现对根组件 VNode 的创建,该函数位于 $ runtime-core_vnode 文件内,实现代码如下:

```
// $ runtime - core_vnode
export const createVNode = (__DEV__
  ? createVNodeWithArgsTransform
  : _createVNode) as typeof _createVNode
```

该函数内部首先判断是否为开发环境,然后根据环境输出代码的执行情况,最后调用_createVNode()函数创建一个 VNode 并返回。_createVNode()函数的内部实现逻辑主要包括:

(1) 对 class 和 style 进行标准化;

(2) 对 VNode 的形态类型进行确定(patch 时依赖);

(3) 通过对象字面量的方式来创建 VNode;

(4) 标准化子节点;

(5) 完成创建后,返回 VNode。

注:此处的标准化操作十分有必要,并且是提升遍历速度的措施之一。此处标准化后,在后续对 VNode 进行递归遍历时,可以减少判断分支,提高递归效率和减少其他未知情况的影响,进而提升遍历稳定性、遍历速度和代码可读性。

因_createVNode()函数内容较多,此处不便展开,将放到 4.1.2 节详细介绍,并对较核心

的内容进行注解。

3.3.3 递归渲染

完成根组件创建后,开始调用 render 函数渲染。render 函数是以参数的形式传入 createAppAPI()函数内,因此在调用 createAppAPI()的地方,可以查到对应 render 函数的实现。代码位于 $ runtime-core_render 文件内,定义在 baseCreateRenderer()函数内,精简后的代码如下:

```
$ runtime - core_render
const render: RootRenderFunction = (vnode, container) => {
    if (vnode == null) {
      // 若无新的 VNode,则卸载组件
      if (container._vnode) {
        unmount(container._vnode, null, null, true)
      }
    } else {
      // 挂载分支
      patch(container._vnode || null, vnode, container)
    }
    // 执行 postFlush 任务队列
    flushPostFlushCbs()
    // 保存当前渲染完毕的根 VNode 在容器上
    container._vnode = vnode
  }
```

上述代码调用 render 函数执行卸载和挂载逻辑,因介绍 mounted 挂载逻辑,此处默认有 VNode 传入,执行 patch 函数。

patch 函数的主要逻辑代码如下:

```
const patch: PatchFn = (
    // 旧 VNode
    n1,
    // 新 VNode
    n2,
    // 挂载容器
    container,
    // 锚点
    anchor = null,
    parentComponent = null,
    parentSuspense = null,
    isSVG = false,
    slotScopeIds = null,
    optimized = __DEV__ && isHmrUpdating ? false : !!n2.dynamicChildren
) => {
    if (n1 === n2) {
      return
    }
    // 判断不是同一个 VNode,直接卸载旧 VNode
    if (n1 && !isSameVNodeType(n1, n2)) {
      // 获取插入的标识位
      anchor = getNextHostNode(n1)
      unmount(n1, parentComponent, parentSuspense, true)
      n1 = null
    }
    if (n2.patchFlag === PatchFlags.BAIL) {
      optimized = false
```

```
        n2.dynamicChildren = null
      }
      // 取出关键信息
      const { type, ref, shapeFlag } = n2
      switch (type) {
        case Text:
          // 处理文本节点...
          break
        case Comment:
          // 处理注释节点...
          break
        case Static:
          // 处理静态节点...
          break
        case Fragment:
          // 处理 fragment ...
          break
        default:
          // 判断 VNode 类型
          if (shapeFlag & ShapeFlags.ELEMENT) {
            // 处理元素节点
            processElement(...)
          } else if (shapeFlag & ShapeFlags.COMPONENT) {
            // 处理组件
            processComponent(...)
          } else if (shapeFlag & ShapeFlags.TELEPORT) {
            // 处理 teleport 组件
            ;(type as typeof TeleportImpl).process(...)
          } else if (__FEATURE_SUSPENSE__ && shapeFlag & ShapeFlags.SUSPENSE) {
            // 处理 suspense 组件
            ;(type as typeof SuspenseImpl).process(...)
          } else if (__DEV__) {
            warn('Invalid VNode type:', type, `($ {typeof type})`)
          }
      }
    }
```

整个 patch 函数主要是判断不同的 VNode 类型(此处的 VNode 类型即创建时初始化的类型),并根据不同类型执行不同的分支,实现递归整个 VNode 的逻辑。Vue3 在处理 VNode 的 shapeFlag 时采用了位运算的方式,关于位运算的知识,将在 7.6 节介绍。根据调试情况来看,程序执行逻辑最终会进入 processComponent()函数中,该函数具体实现如下:

```
const processComponent = (...) => {
  if (n1 == null) {
    ...
    // 挂载组件
    mountComponent(
      n2,
      container,
      anchor,
      parentComponent,
      parentSuspense,
      isSVG,
      optimized
    )
  } else {
```

```
      // 更新组件
      updateComponent(n1, n2, optimized)
    }
  }
```

本节主要关注挂载逻辑,处理 mounted 挂载的 mountComponent()函数,具体实现代码如下:

```
const mountComponent: MountComponentFn = (
      initialVNode,
      container,
      anchor,
      parentComponent,
      parentSuspense,
      isSVG,
      optimized
    ) => {
      const compatMountInstance =
      __COMPAT__ && initialVNode.isCompatRoot && initialVNode.component
      // 创建组件上下文实例
      const instance: ComponentInternalInstance =
       compatMountInstance ||
      (initialVNode.component = createComponentInstance(
        initialVNode,
        parentComponent,
        parentSuspense
      ));
      …
      // 启动组件
      setupComponent(instance);
      …
      // setup 函数为异步的相关处理,忽略相关逻辑
      if (__FEATURE_SUSPENSE__ && instance.asyncDep) {
        parentSuspense && parentSuspense.registerDep(instance, setupRenderEffect);
        if (!initialVNode.el) {
          const placeholder = (instance.subTree = createVNode(Comment));
          processCommentNode(null, placeholder, container!, anchor);
        }
        return;
      }
      // 启动带副作用的 render 函数
      setupRenderEffect(
        instance,
        initialVNode,
        container,
        anchor,
        parentSuspense,
        isSVG,
        optimized
      );
    };
```

在上述代码中,mountComponent()函数内部主要完成如下 3 项工作:

(1) 创建组件上下文实例;

(2) 执行组件 setup 函数;

(3) 创建带副作用的 render 函数。

接下来详细介绍上述 3 点内容。

3.3.4 创建组件上下文实例

```
export function createComponentInstance(
  vnode: VNode,
  parent: ComponentInternalInstance | null,
  suspense: SuspenseBoundary | null
) {
  // 根据条件获取上下文,可能的值包括父组件、组件自身或初始化完成的默认值
  const appContext =
    (parent ? parent.appContext : vnode.appContext) || emptyAppContext
  // 通过对象字面量创建 instance
  const instance: ComponentInternalInstance = {...}
  instance.ctx = { _: instance }
  instance.root = parent ? parent.root : instance
  instance.emit = emit.bind(null, instance)
  // 自定义元素特殊处理
  if (vnode.ce) {
    vnode.ce(instance)
  }
  return instance
}
```

该方法返回通过对象字面量创建的 instance。为帮助读者阅读源码及了解相关特性,我们可以通过 instance 的 interface 来了解组件实例包含的属性。ComponentInternalInstance 的 interface 定义如下:

```
export interface ComponentInternalInstance {
  // 组件唯一 id
  uid: number
  // 组件类型
  type: ConcreteComponent
  // 父组件实例
  parent: ComponentInternalInstance | null
  // 根组件实例
  root: ComponentInternalInstance
  // 根组件上下文
  appContext: AppContext
  // 在父组件的 Vdom 树中表示该组件的 VNode
  vnode: VNode
  // 父组件更新该 VNode 后的新 VNode
  next: VNode | null
  // 当前组件 Vdom 树的根 VNode
  subTree: VNode
  // 副作用渲染函数
  effect: ReactiveEffect
  // 副作用函数的运行方法传递给调度程序
  update: SchedulerJob
  // 返回 Vdom 树的渲染函数
  render: InternalRenderFunction | null
  // ssr 渲染方法
  ssrRender?: Function | null
  // 保存此组件为其后代提供的值
  provides: Data
  // 保存当前 VNode 的 effect 副作用函数,用于在组件卸载时清理相关 effect 副作用函数
```

```
  scope: EffectScope
  // 缓存代理访问类型,以避免 has Oun Property 调用
  accessCache: Data | null
  // 缓存依赖_ctx 但不需要更新的渲染函数
  renderCache: (Function | VNode)[]
  // 已注册的组件,仅适用于带有 mixin 或 extends 的组件
  components: Record < string, ConcreteComponent > | null
  // 注册指令表
  directives: Record < string, Directive > | null
  // 保存过滤方法
  filters?: Record < string, Function >
  // 保存属性设置
  propsOptions: NormalizedPropsOptions
  // 保存 emit 信息
  emitsOptions: ObjectEmitsOptions | null
  // 是否继承属性
  inheritAttrs?: Boolean
  // 是否自定义属性
  isCE?: Boolean
  // 特定自定义元素的 HMR 方法
  ceReload?: (newStyles?: string[]) = > void
  // 组件代理相关
  proxy: ComponentPublicInstance | null
  // 通过 exposed 公开属性
  exposed: Record < string, any > | null
  exposeProxy: Record < string, any > | null
  withProxy: ComponentPublicInstance | null
  ctx: Data
  // 内部状态
  data: Data
  props: Data
  attrs: Data
  slots: InternalSlots
  refs: Data
  emit: EmitFn
  // 收集带 .once 的 emit 事件
  emitted: Record < string, boolean > | null
  // props 默认值
  propsDefaults: Data
  // setup 相关函数状态
  setupState: Data
  devtoolsRawSetupState?: any
  setupContext: SetupContext | null
  // suspense 相关组件
  suspense: SuspenseBoundary | null
  suspenseId: number
  asyncDep: Promise < any > | null
  asyncResolved: boolean
  // 生命周期相关标识
  isMounted: boolean
  isUnmounted: boolean
  isDeactivated: boolean
  // 生命周期钩子函数
  ...
}
```

因组件属性较多,此处进行简单注解以便于读者理解。许多属性在日常开发过程中均可

能涉及，但此处不需要完全记住，对相关属性有了解后，后续若有需要，可以查阅相关文档。

介绍完 interface 属性定义后，继续回到 mountComponent()函数。得到初始化数据后，开始执行 setupComponent()函数，该函数主要初始化 props 和 slots，再执行 setup 和兼容(Vue2)options API 的方法，最终得到 instance 上下文实例。此处简单查看 setupComponent()函数的实现，具体解析将放在 3.4 节中介绍。

setupComponent()代码逻辑如下：

```
export function setupComponent(
  instance: ComponentInternalInstance,
  isSSR = false
) {
  isInSSRComponentSetup = isSSR
  const { props, children, shapeFlag } = instance.vnode;
  // 是否是包含状态的组件
  const isStateful = shapeFlag & ShapeFlags.STATEFUL_COMPONENT;
  // 初始化 Props
  initProps(instance, props, isStateful, isSSR);
  // 初始化 Slots
  initSlots(instance, children);
  // 判断是包含有状态的方法，执行对应逻辑
  const setupResult = isStateful
    ? setupStatefulComponent(instance, isSSR)
    : undefined;
  isInSSRComponentSetup = false
  return setupResult;
}
```

上述代码执行完成后，对于数据的处理包括 instance、props、slots 和 state。继续调用带副作用的 render 函数，在该函数内创建带副作用的渲染函数并保存在 update()方法中，然后调用 update()方法。在 update()方法内根据挂载情况，判断执行挂载或更新逻辑。主要代码逻辑如下：

```
// $ runtime-core_renderer
const setupRenderEffect: SetupRenderEffectFn = (
    instance,
    initialVNode,
    container,
    anchor,
    parentSuspense,
    isSVG,
    optimized
  ) => {
    const componentUpdateFn = () => {
      if (!instance.isMounted) {
        // 创建
        ...
      } else {
        // 更新
        ...
      }
    }
    // 渲染时创建 effect 副作用函数
    const effect = (instance.effect = new ReactiveEffect(
      componentUpdateFn,
      () => queueJob(instance.update),
```

```
      instance.scope // 在组件的影响范围内跟踪它
    ))
    const update = (instance.update = effect.run.bind(effect) as SchedulerJob)
    update.id = instance.uid
    // allowRecurse
    // 组件渲染 effect 应该允许递归更新
    toggleRecurse(instance, true)
    // 执行 update()
    update()
  }
```

setupRenderEffect()函数内保存 effect 副作用函数,整个副作用函数通过 componentUpdateFn()函数实现,完成后将 effect.run 方法绑定到当前上下文的 update()方法上,完成后执行 update()方法,触发 effect 副作用函数的执行(挂载和更新,均会触发)。componentUpdateFn()函数挂载逻辑涉及的代码如下:

```
$ runtime-core_renderer
const componentUpdateFn = () => {
  if (!instance.isMounted) {
    let vnodeHook: VNodeHook | null | undefined
    const { el, props } = initialVNode
    const { bm, m, parent } = instance
    const isAsyncWrapperVNode = isAsyncWrapper(initialVNode)
    toggleRecurse(instance, false)
    // beforeMount 钩子函数
    if (bm) {
      invokeArrayFns(bm)
    }
    // onVnodeBeforeMount
    if (
      !isAsyncWrapperVNode &&
      (vnodeHook = props && props.onVnodeBeforeMount)
    ) {
      invokeVNodeHook(vnodeHook, parent, initialVNode)
    }
    if (
      __COMPAT__ &&
      isCompatEnabled(DeprecationTypes.INSTANCE_EVENT_HOOKS, instance)
    ) {
      instance.emit('hook:beforeMount')
    }
    toggleRecurse(instance, true)
    // 服务器端渲染处理
    if (el && hydrateNode) {
      const hydrateSubTree = () => {
        instance.subTree = renderComponentRoot(instance)
        hydrateNode!(
          el as Node,
          instance.subTree,
          instance,
          parentSuspense,
          null
        )
      }
      // 服务器端渲染不用将渲染调用移到异步回调中,因为在服务器端只调用一次,后续不会触发更改
      if (isAsyncWrapperVNode) {
```

```
    ;(initialVNode.type as ComponentOptions).__asyncLoader!().then(
      () => !instance.isUnmounted && hydrateSubTree()
    )
  } else {
    hydrateSubTree()
  }
} else {
  // 渲染子树
  const subTree = (instance.subTree = renderComponentRoot(instance))
  // patch 子树
  patch(
    null,
    subTree,
    container,
    anchor,
    instance,
    parentSuspense,
    isSVG
  )
  initialVNode.el = subTree.el
}
// mounted 钩子函数
if (m) {
  queuePostRenderEffect(m, parentSuspense)
}
// onVnodeMounted
if (
  !isAsyncWrapperVNode &&
  (vnodeHook = props && props.onVnodeMounted)
) {
  const scopedInitialVNode = initialVNode
  queuePostRenderEffect(
    () => invokeVNodeHook(vnodeHook!, parent, scopedInitialVNode),
    parentSuspense
  )
}
if (
  __COMPAT__ &&
  isCompatEnabled(DeprecationTypes.INSTANCE_EVENT_HOOKS, instance)
) {
  queuePostRenderEffect(
    () => instance.emit('hook:mounted'),
    parentSuspense
  )
}
// 处理钩子函数
if (initialVNode.shapeFlag & ShapeFlags.COMPONENT_SHOULD_KEEP_ALIVE) {
  instance.a && queuePostRenderEffect(instance.a, parentSuspense)
  if (
    __COMPAT__ &&
    isCompatEnabled(DeprecationTypes.INSTANCE_EVENT_HOOKS, instance)
  ) {
    queuePostRenderEffect(
      () => instance.emit('hook:activated'),
      parentSuspense
    )
  }
```

```
        }
        // 标识为已挂载
        instance.isMounted = true
        // 仅挂载对象参数以防止内存泄漏
        initialVNode = container = anchor = null as any
      }
    }
```

ComponentUpdateFn()函数内部主要涉及挂载和更新流程,对挂载逻辑进一步分析可以看到,虽然上述代码较多,但是大部分在处理回调钩子函数。挂载流程实际上主要分为两个步骤:

(1) 渲染组件子树;

(2) patch 子树。

渲染组件子树时,主要涉及 renderComponentRoot()函数,根据代码执行逻辑找到该函数的实现。该函数位于 $ runtime-core_componentRenderUtils 文件内。renderComponentRoot()函数接收一个 instance 作为参数,返回渲染结果(根组件的 VNode)。了解该函数的作用后,再深入查看对应的实现逻辑,主要代码如下:

```
export function renderComponentRoot(
  instance: ComponentInternalInstance
): VNode {
  const {
    ...
  } = instance
  let result
  let fallthroughAttrs
  const prev = setCurrentRenderingInstance(instance)
  try {
    // 如果 vnode 组件是有状态组件
    if (vnode.shapeFlag & ShapeFlags.STATEFUL_COMPONENT) {
      // withProxy 只适用于在'with'作用域下运行时编译的渲染函数
      const proxyToUse = withProxy || proxy
      result = normalizeVNode(
        render!.call(
          proxyToUse,
          proxyToUse!,
          renderCache,
          props,
          setupState,
          data,
          ctx
        )
      )
      fallthroughAttrs = attrs
    } else {
      const render = Component as FunctionalComponent
      // 直接合并 props、slot、attrs 和 emit 即可
      result = normalizeVNode(
        render.length > 1
          ? render(
              props,
              { attrs, slots, emit }
            )
          : render(props, null as any /* we know it doesn't need it */)
```

```
        )
        fallthroughAttrs = Component.props
          ? attrs
          : getFunctionalFallthrough(attrs)
      }
    } catch (err) {
      blockStack.length = 0
      handleError(err, instance, ErrorCodes.RENDER_FUNCTION)
      result = createVNode(Comment)
    }
    // 在开发模式下属性合并注释被保留,可以在根元素旁边添加注释,使其成为一个片段
    let root = result
    let setRoot: ((root: VNode) => void) | undefined = undefined
    if (fallthroughAttrs && inheritAttrs !== false) {
      const keys = Object.keys(fallthroughAttrs)
      const { shapeFlag } = root
      if (keys.length) {
        //如果是元素或组件类型
        if (shapeFlag & (ShapeFlags.ELEMENT | ShapeFlags.COMPONENT)) {
          if (propsOptions && keys.some(isModelListener)) {
            // 如果 v-model 监听属性和 props 属性相同,则优先使用 v-model 处理
            fallthroughAttrs = filterModelListeners(
              fallthroughAttrs,
              propsOptions
            )
          }
          root = cloneVNode(root, fallthroughAttrs)
        }
      }
    }
    // 合并处理当前节点和 props 传递的 class 和 style 内容
    if (
      __COMPAT__ &&
      isCompatEnabled(DeprecationTypes.INSTANCE_ATTRS_CLASS_STYLE, instance) &&
      vnode.shapeFlag & ShapeFlags.STATEFUL_COMPONENT &&
      root.shapeFlag & (ShapeFlags.ELEMENT | ShapeFlags.COMPONENT)
    ) {
      const { class: cls, style } = vnode.props || {}
      if (cls || style) {
        root = cloneVNode(root, {
          class: cls,
          style: style
        })
      }
    }
    // 继承指令
    if (vnode.dirs) {
      root.dirs = root.dirs ? root.dirs.concat(vnode.dirs) : vnode.dirs
    }
    // 继承过渡数据
    if (vnode.transition) {
      root.transition = vnode.transition
    }
    result = root
    setCurrentRenderingInstance(prev)
    return result
}
```

注：此处的 render 渲染函数是组件的渲染函数，用于将 template 转换为 VNode 使用，并非 3.3.1 节中的 render 分发函数。

上述代码的核心是调用 instance 的 render 方法进行处理，通过调用 render 渲染函数完成转换过程得到 VNode 后，将执行第(2)步 patch 子树。

3.3.5 patch 子树

patch 子树的作用是将 VNode 通过递归的方式挂载为 VNode Tree 并转换为渲染函数的过程。在转换过程中，会根据子树的类型进行 patch 分发，调用不同的逻辑渲染。关于 patch 函数的具体实现，将会在 4.2 节着重介绍。

在完成 patch 子树步骤后，会再次调用 patch 函数，此时将会得到根组件的真实 DOM 节点，即 id＝"root"的 div 元素。根据传入的类型可知，本次 patch 函数将会分发到 processElement()函数内。该函数主要根据判断结果，执行对元素的挂载或更新。具体代码逻辑如下：

```
// $ runtime - core_renderer
const processElement = (...) => {
  isSVG = isSVG || (n2.type as string) === 'svg'
  if (n1 == null) {
    // 挂载元素
    mountElement(...)
  } else {
    // 更新元素
    patchElement(...)
  }
}
```

此处为首次渲染，会通过 mountElement()函数对节点元素进行挂载，涉及代码逻辑如下：

```
const mountElement = (
  vnode: VNode,
  container: RendererElement,
  anchor: RendererNode | null,
  parentComponent: ComponentInternalInstance | null,
  parentSuspense: SuspenseBoundary | null,
  isSVG: boolean,
  slotScopeIds: string[] | null,
  optimized: boolean
) => {
  let el: RendererElement;
  let vnodeHook: VNodeHook | undefined | null;
  const {
    type,
    props,
    shapeFlag,
    transition,
    patchFlag,
    dirs,
  } = vnode;
  // 判断 VNode 是否属于静态标识
  if (
    !__DEV__ &&
    vnode.el &&
```

```
    hostCloneNode !== undefined &&
    patchFlag === PatchFlags.HOISTED
  ) {
    // 如果一个 vnode 有非空的 el,则意味着正在被重用.只有静态的 vnode 可以被重用,所以挂载的
    // DOM 节点应该是完全相同的,并且只在生产环境使用
    el = vnode.el = hostCloneNode(vnode.el)
  } else {
      el = vnode.el = hostCreateElement(
        vnode.type as string,
        isSVG,
        props && props.is,
        props
      );
      // 文本节点的 children
      if (shapeFlag & ShapeFlags.TEXT_CHILDREN) {
        // 如果是文本类型的子节点,则直接设置子节点的内容
        hostSetElementText(el, vnode.children as string);
      } else if (shapeFlag & ShapeFlags.ARRAY_CHILDREN) {
        // 数组型的 children
        mountChildren(
          vnode.children as VNodeArrayChildren,
          el,
          null,
          parentComponent,
          parentSuspense,
          isSVG && type !== "foreignObject",
          slotScopeIds,
          optimized
        );
      }
      if (dirs) {
        invokeDirectiveHook(vnode, null, parentComponent, 'created')
      }
      // 设置 props
      if (props) {
        for (const key in props) {
          if (key !== 'value' && !isReservedProp(key)) {
            hostPatchProp(
              el,
              key,
              null,
              props[key],
              isSVG,
              vnode.children as VNode[],
              parentComponent,
              parentSuspense,
              unmountChildren
            );
          }
        }
        if ('value' in props) {
          hostPatchProp(el, 'value', null, props.value)
        }
        if ((vnodeHook = props.onVnodeBeforeMount)) {
          invokeVNodeHook(vnodeHook, parentComponent, vnode)
        }
      }
```

```
        // 触发钩子函数
        if (dirs) {
          invokeDirectiveHook(vnode, null, parentComponent, 'beforeMount')
        }
        // 设置 scopeId
        setScopeId(el, vnode, vnode.scopeId, slotScopeIds, parentComponent)
      }
      // 判断组件需要触发 transition 的钩子函数
      const needCallTransitionHooks =
        (!parentSuspense || (parentSuspense && !parentSuspense.pendingBranch)) &&
        transition &&
        !transition.persisted
      // 如果涉及动画触发,则触发 beforeEnter
      if (needCallTransitionHooks) {
        transition!.beforeEnter(el)
      }
      // 插入容器元素中
      hostInsert(el, container, anchor)
      if (
        (vnodeHook = props && props.onVnodeMounted) ||
        needCallTransitionHooks ||
        dirs
      ) {
        // 元素插入完成,需触发 transition 的 enter 效果
        queuePostRenderEffect(() => {
          vnodeHook && invokeVNodeHook(vnodeHook, parentComponent, vnode)
          needCallTransitionHooks && transition!.enter(el)
          dirs && invokeDirectiveHook(vnode, null, parentComponent, 'mounted')
        }, parentSuspense)
      }
    }
```

上述代码的主要逻辑如下：

(1) 创建 el 元素；

(2) 根据 children 类型,处理子节点；

(3) 处理 props；

(4) 将元素插入 DOM 节点内。

mountElement()函数内的其他流程逻辑都较为清晰和简单,下面简单介绍 mountChildren()函数的实现。该函数内部通过循环判断 children 数组,直接调用 patch 函数递归挂载组件,具体代码实现逻辑如下：

```
const mountChildren: MountChildrenFn = (
  children,
  container,
  anchor,
  parentComponent,
  parentSuspense,
  isSVG,
  slotScopeIds,
  optimized,
  start = 0
) => {
  for (let i = start; i < children.length; i++) {
    // 标准化 VNode
    const child = (children[i] = optimized
```

```
            ? cloneIfMounted(children[i] as VNode)
            : normalizeVNode(children[i]))
        // 直接调用 patch()
        patch(
          null,
          child,
          container,
          anchor,
          parentComponent,
          parentSuspense,
          isSVG,
          slotScopeIds,
          optimized
        );
      }
    };
```

完成渲染后，会返回 mount 并执行重写 mount 的方法，将元素挂载到页面上。整个 mount 方法挂载流程如图 3.9 所示。

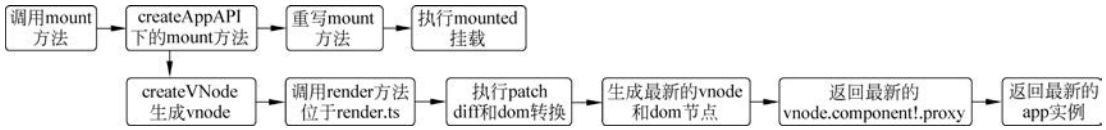

图 3.9 mount 方法挂载流程

3.3.6 总结

整个 mounted 挂载过程介绍完成。通过本节的学习，可对 Vue3 中整个 mounted 实现过程有深入的认识。本节内涉及的函数较多，函数内实现代码逻辑较多，可以结合流程图帮助理解流程。本节通过重写 mount 方法，将核心的处理流程和平台的 API 进行分离，通过重写的方法巧妙地将平台方法和核心方法分离，该写法使得核心方法不局限于 Web 端，可进行多端扩展。该方法和技巧值得学习，可根据实际情况应用到自身项目内。

3.4 setup 函数

setup 函数涉及文件路径如表 3.3 所示。

表 3.3 setup 函数涉及文件路径

名 称	简写路径	文件路径
mountComponent	$ runtime-core_render	$ root/package/runtime-core/src/render. ts
setupComponent	$ runtime-core_components	$ root/package/runtime-core/src/component. ts
PublicInstanceProxyHandlers	$ runtime-core_componentPublicInstance	$ root/package/runtime-core/src/componentPublicInstance. ts
createSetupContext	$ runtime-core_components	$ root/package/runtime-core/src/component. ts
reactivity/src/ref. ts	$ reactivity_ref	$ root/package/reactivity/ref. ts
finishComponentSetup	$ runtime-core_Component	$ root/packages/runtime-core/src/component. ts

3.4.1 涉及文件

setup 函数是 Vue3 新推出的函数，该函数将用来承载 composition API，同时替代 beforeCreated 和 created 生命周期函数。Vue3 内大多数方法和函数均可以在该函数内使用。该函数仅会在组件启动的时候执行一次，内部会执行初始化，并确认组件与视图的数据、行为和作用。setup 函数作为桥梁存在，连接模板渲染、数据处理和响应式数据监听。

3.4.2 mountComponent()函数

在前面介绍 mounted 挂载执行逻辑时，提到内部将会触发 setupComponent()函数，在该函数内创建好 instance 后，将启动 setup 处理内部暴露和声明的内容。setup 处理代码位于 $ runtime-core_render 文件的 mountComponent()函数内。对应代码片段如下：

```
// $ runtime - core_render
// 创建组件实例
const instance: ComponentInternalInstance =
  compatMountInstance ||
(initialVNode.component = createComponentInstance(
  initialVNode,
  parentComponent,
  parentSuspense
));
// 启动组件
setupComponent(instance);
// 启动带副作用的 render 函数
setupRenderEffect(
  instance,
  initialVNode,
  container,
  anchor,
  parentSuspense,
  isSVG,
  optimized
);
```

3.4.3 setupComponent()函数

根据上述代码逻辑，创建组件实例完成后将调用 setupComponent()函数，该函数位于 $ runtime-core_components 文件内，函数实现逻辑如下：

```
// $ runtime - core_components
export function setupComponent(
  instance: ComponentInternalInstance,
  isSSR = false
) {
  isInSSRComponentSetup = isSSR
  const { props, children } = instance.vnode
  const isStateful = isStatefulComponent(instance)
  // 初始化 props
  initProps(instance, props, isStateful, isSSR)
  // 初始化插槽
  initSlots(instance, children)
  // 执行 setup
  const setupResult = isStateful
```

```
    ? setupStatefulComponent(instance, isSSR)
    : undefined
  isInSSRComponentSetup = false
  // 返回 setup 执行结果
  return setupResult
}
```

该函数的主要逻辑如下：

(1) 初始化 props；

(2) 初始化插槽 slot；

(3) 根据条件判断是否为带状态的组件选择是否执行 setup；

(4) 返回 setup 执行结果。

根据判断条件可知，核心方法是 setupStatefulComponent()，此处暂不介绍 props 和 slot 的初始化方法，直接在当前文件内查看 setupStatefulComponent()函数的实现。主要代码逻辑如下：

```
// $ runtime - core_components
function setupStatefulComponent(
  instance: ComponentInternalInstance,
  isSSR: boolean
) {
  const Component = instance.type as ComponentOptions
  // 初始化缓存
  instance.accessCache = Object.create(null)
  // 创建组件的渲染上下文代理
  instance.proxy = markRaw(new Proxy(instance.ctx, PublicInstanceProxyHandlers))
  // 调用 setup
  const { setup } = Component
  if (setup) {
    // 按需创建 setup 第二个参数
    const setupContext = (instance.setupContext =
      setup.length > 1 ? createSetupContext(instance) : null)
    // 设置当前组件实例
    setCurrentInstance(instance)
    pauseTracking()
    // 调用 setup
    const setupResult = callWithErrorHandling(
      setup,
      instance,
      ErrorCodes.SETUP_FUNCTION,
      [__DEV__ ? shallowReadonly(instance.props) : instance.props, setupContext]
    )
    resetTracking()
    unsetCurrentInstance()
    // 处理 setup 的执行结果
    if (isPromise(setupResult)) {
      setupResult.then(unsetCurrentInstance, unsetCurrentInstance)
      if (isSSR) {
        // 在 ssr(服务器端渲染)环境下，等待 promise 返回再处理 setup 的执行结果
        return setupResult.then((resolvedResult: unknown) => {
          handleSetupResult(instance, resolvedResult, isSSR)
        })
        .catch(e => {
          handleError(e, instance, ErrorCodes.SETUP_FUNCTION)
        })
      } else if (__FEATURE_SUSPENSE__) {
```

```
            // 保存异步依赖
            instance.asyncDep = setupResult
          }
        } else {
          // 更详细地处理 setup 执行结果
          handleSetupResult(instance, setupResult, isSSR)
        }
      } else {
        // 完成组件启动的后续工作
        finishComponentSetup(instance, isSSR)
      }
    }
```

该函数的主要逻辑如下：

(1) 初始化代理缓存；

(2) 创建渲染上下文代理；

(3) 调用 setup 函数；

(4) 处理 setup 函数执行结果；

(5) 完成组件启动。

3.4.4 初始化代理上下文

通过 Proxy 创建代理上下文，Proxy 对象是 ES6 新增加的属性，主要用于创建一个对象的代理。该对象不支持 IE11 以下的浏览器，因此 Vue3 将不兼容老浏览器。使用该对象可以无损害地劫持对象的部分属性，并重写该属性，达到数据动态监听的目的，同时也是响应式数据的核心。

根据 Proxy 语法要求，传入两个参数：target 和 handler。其中 target 是需要被代理的对象，handler 主要是重写或自定义拦截函数，达到拦截 target 对象并且实现自定义逻辑的目的。具体写法如下：

```
const p = new Proxy(target, handler)
```

简单介绍完 Proxy 语法后，查看其具体实现，涉及代码如下：

```
instance.proxy = markRaw(new Proxy(instance.ctx, PublicInstanceProxyHandlers))
```

被代理对象是 instance.ctx，传入的拦截函数是 PublicInstanceProxyHandlers()。该函数位于 $ runtime-core_componentPublicInstance 文件内，内部实现代码较多，核心逻辑主要实现了 get、set、has 方法的监听，核心代码如下：

```
// $ runtime - core_componentPublicInstance
export const PublicInstanceProxyHandlers: ProxyHandler < any > = {
  get({ _: instance }: ComponentRenderContext, key: string) { … }
  set(
    { _: instance }: ComponentRenderContext,
    key: string,
    value: any
  ): boolean { … }
  has(
      {
        _: { data, setupState, accessCache, ctx, appContext, propsOptions }
      }: ComponentRenderContext,
      key: string
  ) { … }
}
```

3.4.5 get 方法

该方法内需要传入 instance 上下文和需要获取的 key 值。若 key 值不是以 $ 开头，则根据 key 值判断是否缓存在 accessCache 中，若存在，则根据 setup、data、ctx 和 props 的顺序查询对应值并返回，若不存在则进一步判断。

```
let normalizedProps
if (key[0] !== '$ ') {
  const n = accessCache![key]
  if (n !== undefined) {
    switch (n) {
      case AccessTypes.SETUP:
        return setupState[key]
      case AccessTypes.DATA:
        return data[key]
      case AccessTypes.CONTEXT:
        return ctx[key]
      case AccessTypes.PROPS:
        return props![key]
      // default: just fallthrough
    }
  }
  ...
}
```

根据同样的顺序判断 setup、data、props 和 ctx 对象上的属性是否有值，若存在则直接返回，若不存在则开始全局扫描。

```
else if (setupState !== EMPTY_OBJ && hasOwn(setupState, key)) {
  accessCache![key] = AccessTypes.SETUP
  return setupState[key]
} else if (data !== EMPTY_OBJ && hasOwn(data, key)) {
  accessCache![key] = AccessTypes.DATA
  return data[key]
} else if (
  // props
  (normalizedProps = instance.propsOptions[0]) &&
  hasOwn(normalizedProps, key)
) {
  accessCache![key] = AccessTypes.PROPS
  return props![key]
} else if (ctx !== EMPTY_OBJ && hasOwn(ctx, key)) {
  accessCache![key] = AccessTypes.CONTEXT
  return ctx[key]
} else if (!__FEATURE_OPTIONS_API__ || shouldCacheAccess) {
  accessCache![key] = AccessTypes.OTHER
}
```

若在缓存内未找到对应 key，则开始遍历使用 $ 表示的公开属性并执行对应的逻辑。在介绍内部实现前，先查看以 $ 开头的公开属性：

```
export const publicPropertiesMap: PublicPropertiesMap = extend(
  Object.create(null),
  {
    $ : i => i,
    $ el: i => i.vnode.el,
```

```
    $data: i => i.data,
    $props: i => (__DEV__ ? shallowReadonly(i.props) : i.props),
    $attrs: i => (__DEV__ ? shallowReadonly(i.attrs) : i.attrs),
    $slots: i => (__DEV__ ? shallowReadonly(i.slots) : i.slots),
    $refs: i => (__DEV__ ? shallowReadonly(i.refs) : i.refs),
    $parent: i => getPublicInstance(i.parent),
    $root: i => getPublicInstance(i.root),
    $emit: i => i.emit,
    $options: i => (__FEATURE_OPTIONS_API__ ? resolveMergedOptions(i) : i.type),
    $forceUpdate: i => () => queueJob(i.update),
    $nextTick: i => nextTick.bind(i.proxy!),
    $watch: i => (__FEATURE_OPTIONS_API__ ? instanceWatch.bind(i) : NOOP)
  } as PublicPropertiesMap
)
```

publicPropertiesMap 对象定义公开的内部属性,涉及常见的 $el、$data 等。

若为公开属性,并且是 $attrs,则调用 track 收集依赖;若是 css module,则暂时不处理,直接返回;若是用户自定义的以 $ 开头的属性,则查询上下文 ctx 并返回;若是全局属性,则查询全局对象并返回。

此处最核心的地方是使用 track 进行依赖的收集,具体代码实现如下:

```
const publicGetter = publicPropertiesMap[key]
let cssModule, globalProperties
// public $xxx properties
if (publicGetter) {
if (key === '$attrs') {
    track(instance, TrackOpTypes.GET, key)
    __DEV__ && markAttrsAccessed()
  }
  return publicGetter(instance)
} else if (
  // css module (injected by vue-loader)
  (cssModule = type.__cssModules) &&
  (cssModule = cssModule[key])
) {
  return cssModule
} else if (ctx !== EMPTY_OBJ && hasOwn(ctx, key)) {
  // user may set custom properties to `this` that start with `$`
  accessCache![key] = AccessTypes.CONTEXT
  return ctx[key]
} else if (
  // global properties
  ((globalProperties = appContext.config.globalProperties),
  hasOwn(globalProperties, key))
) {
  if (__COMPAT__) {
    const desc = Object.getOwnPropertyDescriptor(globalProperties, key)!
    if (desc.get) {
      return desc.get.call(instance.proxy)
    } else {
      const val = globalProperties[key]
      return isFunction(val) ? val.bind(instance.proxy) : val
    }
  } else {
    return globalProperties[key]
  }
}
```

3.4.6 set方法

完成get方法处理后，继续查看set方法的实现。相较于get方法，set方法的内部逻辑判断更加简单，只要判断传入值类型并保存到对应类型属性中即可。具体实现代码如下：

```
set(
  { _: instance }: ComponentRenderContext,
  key: string,
  value: any
): boolean {
  const { data, setupState, ctx } = instance
  if (setupState !== EMPTY_OBJ && hasOwn(setupState, key)) {
    setupState[key] = value
  } else if (data !== EMPTY_OBJ && hasOwn(data, key)) {
    data[key] = value
  } else if (hasOwn(instance.props, key)) {
    __DEV__ &&
      warn(
        `Attempting to mutate prop "$ {key}". Props are readonly.`,
        instance
      )
    return false
  }
  if (key[0] === '$ ' && key.slice(1) in instance) {
    __DEV__ &&
      warn(
        `Attempting to mutate public property "$ {key}". ` +
          `Properties starting with $  are reserved and readonly.`,
        instance
      )
    return false
  } else {
    if (__DEV__ && key in instance.appContext.config.globalProperties) {
      Object.defineProperty(ctx, key, {
        enumerable: true,
        configurable: true,
        value
      })
    } else {
      ctx[key] = value
    }
  }
  return true
}
```

set方法解析出data、setupState和ctx属性，并将key的名称与解析出的属性名称进行对比。若相同，则直接保存；若不相同，则判断key的名称与instance的props属性是否相等，若相等则结束set方法的执行，并在开发模式下抛出不能修改的警告。

完成上述判断后，若key以$开头且名称在instance上下文内，则直接返回false，结束set方法执行。若不是以$开头，则进一步判断是否为开发模式且为全局属性，若是则通过instance.appContext.config.globalProperties对象保存，并通过Object.defineProperty方法实现对应值的响应式监听。

3.4.7 has 方法

完成 set 代码解析后,继续查看 has 方法的实现。只需要判断对应 key 是否有缓存,具体代码如下:

```
has(
  {
    _: { data, setupState, accessCache, ctx, appContext, propsOptions }
  }: ComponentRenderContext,
  key: string
) {
  let normalizedProps
  return (
    accessCache![key] !== undefined ||
    (data !== EMPTY_OBJ && hasOwn(data, key)) ||
    (setupState !== EMPTY_OBJ && hasOwn(setupState, key)) ||
    ((normalizedProps = propsOptions[0]) && hasOwn(normalizedProps, key)) ||
    hasOwn(ctx, key) ||
    hasOwn(publicPropertiesMap, key) ||
    hasOwn(appContext.config.globalProperties, key)
  )
}
```

3.4.8 调用 setup 函数

完成代理创建后,开始调用 setup 函数。获取到 setup 函数后,根据 setup 的值判断是否有第二个值传入,若有则调用 createSetupContext()函数初始化,若没有则直接返回 null。createSetupContext()函数简化后的代码如下:

```
// $ runtime-core_components
export function createSetupContext(
  instance: ComponentInternalInstance
): SetupContext {
  const expose: SetupContext['expose'] = exposed => {
    instance.exposed = exposed || {}
  }
  return {
    get attrs() {
       return attrs || (attrs = createAttrsProxy(instance))
     },
    slots: instance.slots,
    emit: instance.emit,
    expose
  }
}
```

该方法返回 4 个属性,分别为 attrs、slots、emit 和 expose。所以 setup 的 ctx 上只能获取到这 4 个属性。

完成 ctx 的处理后开始执行 setup 函数,通过 callWithErrorHandling 函数包裹执行,方便捕获异步执行时抛出的错误。完成执行后开始处理执行结果 setupResult,判断是否为 promise 对象,若是则执行 then 方法,然后执行 handleSetupResult()函数,将 setupResult 返回值作为参数传入 handleSetupResult()函数;如果不是则直接调用 handleSetupResult()函数将 setupResult 的返回值作为参数传入。

handleSetupResult()函数内部根据传入值判断是函数还是对象。若是函数,则将该函数挂载到 instance. render 上;若是对象,则通过 proxyRefs()函数将对象保存在 instance. setupState 属性中。完成该步骤后,调用 finishComponentSetup()函数处理 setup 完成后的其他步骤。

在查看 finishComponentSetup()函数前,先简单介绍 proxyRefs()函数的内部实现。由函数名称可知,该函数的主要作用是将一个对象变成响应式对象,具体代码如下:

```
// reactivity_ref
export function proxyRefs < T extends object >(
  objectWithRefs: T
): ShallowUnwrapRef < T > {
  return isReactive(objectWithRefs)
    ? objectWithRefs
    : new Proxy(objectWithRefs, shallowUnwrapHandlers)
}
```

如果需要将一个对象变为响应式,那么最核心的方法是使用 new Proxy()实例,与前面介绍的响应式方法实现类似,同样传入 shallowUnwrapHandlers 对象,该对象内部也是通过劫持 set 和 get 方法,并利用 Reflect 内置对象来保存数据。

完成 setup 函数执行后通过 finishComponentSetup()函数处理后续逻辑。在该函数内部处理 render 渲染函数,依次执行如下步骤。

(1) 获取 render 属性值;

(2) 判断运行环境是否为 SSR,若不是则进一步判断;

(3) 是否有 complie()函数,是否有 template 字段,是否无 render 渲染函数,该逻辑主要判断当前 VNode 是否需要将 template 模板字符串转换为 render 渲染函数,对于完整版的 Vue3,会在此处注入 complie()转换函数,执行 template 模板字符串的渲染。在 7.1 节将着重介绍 complie 的注入逻辑,介绍 template 模板字符串到 render 渲染函数的转换。

上述判断完成后,将 template 模板转换为 VNode 树,再转换为 html DOM 树,主要逻辑为:

(1) 获取 template;

(2) 根据当前上下文的设置,合并 options;

(3) 调用 compile 函数,将模板编译为 render 渲染函数。

render 函数内主要处理页面渲染模板的转换,将 template 模板转换为对应的 VNode,再将 VNode 转换为对应的 DOM 树。完成后判断是否有_rc 属性,重新调用 proxy 实现 ctx 的代理函数,并且判断是否为 2. x 版本,通过 applyOptins 处理 instance,兼容 Vue2 的语法。

3.4.9 finishComponentSetup()函数

finishComponentSetup()函数内执行 render 渲染逻辑,将 VNode 转换为真实 DOM 结构,并将转换完成的 DOM 节点挂载到根节点上。

finishComponentSetup()函数逻辑如下:

```
// $ runtime - core_component
export function finishComponentSetup(
  instance: ComponentInternalInstance,
  isSSR: boolean,
  skipOptions?: boolean
) {
```

```
  const Component = instance.type as ComponentOptions
  if (__COMPAT__) {
    convertLegacyRenderFn(instance)
  }
  if (!instance.render) {
    // SSR 的实时编译由服务器端渲染器完成
    // 若 compile 存在,且没有 render 渲染函数,在不是 SSR 的情况下开始模板编译
    if (!isSSR && compile && !Component.render) {
      const template =
        (__COMPAT__ &&
          instance.vnode.props &&
          instance.vnode.props['inline-template']) ||
        Component.template
      if (template) {
        // 判断是否为自定义元素,和渲染属性设置进行合并
        const { isCustomElement, compilerOptions } = instance.appContext.config
        const { delimiters, compilerOptions: componentCompilerOptions } =
          Component
        const finalCompilerOptions: CompilerOptions = extend(
          extend(
            {
              isCustomElement,
              delimiters
            },
            compilerOptions
          ),
          componentCompilerOptions
        )
        if (__COMPAT__) {
          // 将兼容属性传入 compiler
          finalCompilerOptions.compatConfig = Object.create(globalCompatConfig)
          if (Component.compatConfig) {
            extend(finalCompilerOptions.compatConfig, Component.compatConfig)
          }
        }
        // 生成渲染函数
        Component.render = compile(template, finalCompilerOptions)
      }
    }
    instance.render = (Component.render || NOOP) as InternalRenderFunction
    // 兼容使用'with'作用域的在运行时进行编译的渲染函数
    if (installWithProxy) {
      installWithProxy(instance)
    }
  }
  // 兼容 2.x 版本属性
  if (__FEATURE_OPTIONS_API__ && !(__COMPAT__ && skipOptions)) {
    setCurrentInstance(instance)
    pauseTracking()
    applyOptions(instance)
    resetTracking()
    unsetCurrentInstance()
  }
}
```

得到 render 渲染函数后,使用该 render 函数进行渲染。继续回到 mountComponent 函数逻辑内,完成 setupComponent 函数后,通过 setupRenderEffect 函数启动带副作用的 render

函数。这里之所以叫带副作用的 render 函数，是因为 effect 副作用函数在此处传入，具体代码如下：

```
const setupRenderEffect: SetupRenderEffectFn = (
    instance,
    initialVNode,
    container,
    anchor,
    parentSuspense,
    isSVG,
    optimized
  ) => {
    const componentUpdateFn = () => {
      if (!instance.isMounted) {
        // 创建
        ...
      } else {
        // 更新
        ...
      }
    }
    // 渲染时创建 effect 副作用函数
    const effect = (instance.effect = new ReactiveEffect(
      componentUpdateFn,
      () => queueJob(instance.update),
      instance.scope       // 在组件的影响范围内跟踪它
    ))
    const update = (instance.update = effect.run.bind(effect) as SchedulerJob)
    update.id = instance.uid
    // allowRecurse
    // 组件渲染 effects 应该允许递归更新
    toggleRecurse(instance, true)
    // 执行 update
    update()
  }
```

此步骤与 effect 副作用函数关联，整个代码的执行逻辑也基本清晰。在 mounted 挂载的时候，effect 副作用函数会调用 update 方法，触发内部的 effect 副作用函数执行，该函数在组件挂载和更新的时候均会执行。

如果引入的 Vue3 不是 runtime 版本，则会在此处对 compile 函数进行注入。以函数注入的方式进行挂载，可以更加优雅地完成模板渲染和 runtime-core 核心逻辑的解构。

3.4.10 总结

完成该函数的解析后，整个 setup 执行步骤即处理完成。由分析结果可知，该函数主要处理不同类型的情况，执行 setup 函数后，再调用渲染函数，将模板转换为 render 对象内容，以备后续使用。setup 函数内使用了许多优化措施，包括通过 accessCache 缓存已挂载数据，加快已处理数据取值；根据 setup 传入参数个数动态创建 setupContext。通过 proxy 对象，解决 Vue2 遗留的响应式数据问题的操作。

本节对响应式数据的依赖收集 track 和派发更新 trigger 做了简单介绍。通过对本节的学习，读者应该对 Vue3 整个执行和渲染逻辑有了全面的了解。此处介绍的也是整个 Vue3 最核心的地方，后续大部分章节将围绕该流程进行更加深入的展开。在后面的学习中可以参考本

节的实现逻辑。比如最核心的 diff 其实就是对比新旧 VNode 的过程,响应式数据就是通过 new proxy 实例,劫持 get、set 和 has 等方法,实现依赖收集(track)和派发更新(trigger)的过程,调用过程如图 3.10 所示。

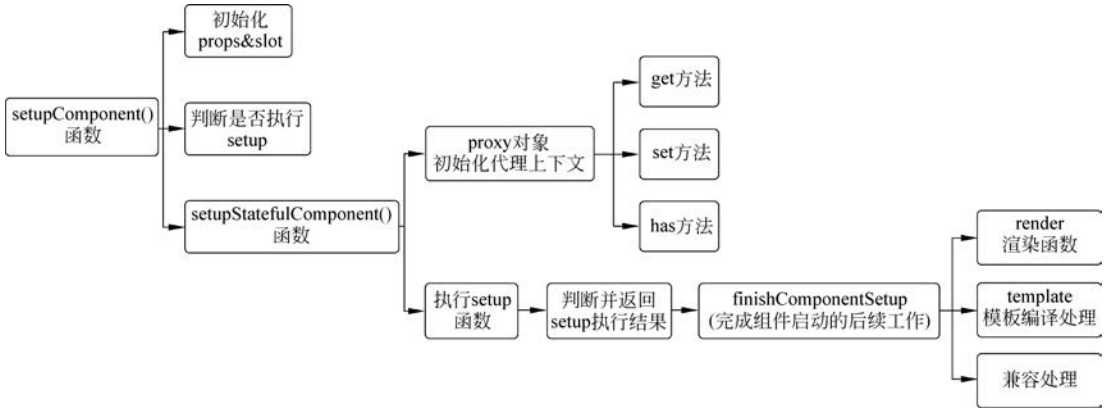

图 3.10　调用过程

3.5　update 方法

介绍完 app 初始化和 mounted 挂载流程后,本节介绍 Vue3 内执行 update 的整体流程。在 Vue3 中引入 VNode 来处理虚拟 DOM 结构到真实 DOM 树之间的关系。update 方法是通过对比新、旧 VNode 树的变化,找出不一致的地方调用渲染函数,对真实的 DOM 进行局部修改,避免大面积修改页面 DOM,以及页面重绘或回流,提高渲染性能,从而以最小的代价完成页面的操作,更快地完成页面的更新。

3.5.1　涉及文件

update 方法涉及文件路径如表 3.4 所示。

表 3.4　update 方法涉及文件路径

名　　称	简 写 路 径	文 件 路 径
setupRenderEffect	$ runtime-core_render	$ root/packages/runtime-core/src/renderer.ts
updateComponentPreRender	$ runtime-core_render	$ root/packages/runtime-core/src/renderer.ts
updateComponent	$ runtime-core_render	$ root/packages/runtime-core/src/renderer.ts
shouldUpdateComponent	$ runtime-core_componentRenderUtils	$ root/packages/runtime-core/src/componentRenderUtils.ts
processElement	$ runtime-core_renderer	$ root/packages/runtime-core/src/renderer.ts
patchChildren	$ runtime-core_renderer	$ root/packages/runtime-core/src/renderer.ts
patchProps	$ runtime-dom_patchProp	/package/runtime-dom/src/patchProp.ts
patchChildren	$ runtime-core_renderer	$ root/packages/runtime-core/src/renderer.ts

3.5.2 setupRenderEffect()函数

当组件内部状态变化、父组件更新子组件或触发 forceUpdate 时，将触发组件的 update 方法。该方法在组件初始化的时候已经挂载，关于组件对应方法的实现，已经在介绍 render 函数时有过说明。

下面查看组件更新的实现步骤，具体代码逻辑位于 $ runtime-core_render. ts 文件内。update 实现代码较多，此处先了解该方法的实现，简化后的代码如下：

```
// $ runtime - core_render
const setupRenderEffect: SetupRenderEffectFn = (
  instance,
  initialVNode,
  container,
  anchor,
  parentSuspense,
  isSVG,
  optimized
) => {
  // 创建执行带副作用的渲染函数并保存在 update 属性中
  instance.update = effect(function componentEffect() {
    if (!instance.isMounted) {
      // 挂载组件
    } else {
      // 组件更新
      // 组件自身发起的更新，next 属性值为 null
      // 父组件发起的更新，next 属性值为当前组件更新后 VNode
      let { next, vnode } = instance;
      let originNext = next;
      if (next) {
        // 如果存在 next，则需要更新组件实例相关信息
        // 修正 instance 和 nextVNode 的指向关系
        // 更新 Props 和 Slots
        updateComponentPreRender(instance, next, optimized);
      } else {
        next = vnode;
      }
      // 渲染新的子树
      const nextTree = renderComponentRoot(instance);
      const prevTree = instance.subTree;
      instance.subTree = nextTree;
      next.el = vnode.el;
      // diff 子树
      patch(
        prevTree,
        nextTree,
        // 排除 teleport 的情况，及时获取父节点
        hostParentNode(prevTree.el!)!,
        // 排除 fragement 情况，及时获取下一个节点
        getNextHostNode(prevTree),
        instance,
        parentSuspense,
        isSVG
      );
      next.el = nextTree.el;
```

```
    }
  }, EffectOptions);
```

上述代码的主要实现逻辑如下：

(1) 判断是否为自身执行的更新，若有 next 则代表是父组件发起更新，若无则代表自身触发更新；

(2) 修正 instance 和 nextVNode 的指向关系；

(3) 渲染子树；

(4) diff 子树。

具体查看该函数内部实现，在修正指向关系时，会调用 updateComponentPreRender()函数，并传入 instance、next 和 optimized 参数。该函数将传入的 next(nextVnode)赋值到 instance 对象的 VNode 上，将 instance 上的 next 设置为 null，以达到修正 instance 属性 VNode 指向的目的。完成后调用 updateProps 和 updateSlots 更新组件的 props 和 slots 对象。

3.5.3 updateComponentPreRender()函数

执行当前 VNode 的 update 方法，具体代码实现如下：

```
// $ runtime-core_render
const updateComponentPreRender = (
    instance: ComponentInternalInstance,
    nextVNode: VNode,
    optimized: boolean
  ) => {
    nextVNode.component = instance
    const prevProps = instance.vnode.props
    instance.vnode = nextVNode
    instance.next = null
    updateProps(instance, nextVNode.props, prevProps, optimized)
    updateSlots(instance, nextVNode.children, optimized)
    pauseTracking()
    // props update may have triggered pre-flush watchers.
    // flush them before the render update.
    flushPreFlushCbs(undefined, instance.update)
    resetTracking()
  }
```

关于 props 和 slot 的 update 方法，将在 8.4 节和 8.5 节详细介绍，此处暂不展开。完成指向关系修正后，继续执行后续逻辑。通过 renderComponentRoot()函数渲染新的子树，使用当前子树(prevTree)和新子树(nextTree)作为参数调用 patch 函数，该函数主要对新旧子树进行对比和判断。patch 完成后，将 next.el 重新赋值给当前实例 next。

```
next.el = nextTree.el
```

在 3.3 节中曾经介绍过 patch 内部主要对不同的类型进行分发，组件 VNode 将执行 processComponent()函数，该函数的内部判断传入的 VNode 是否为 null，判断执行挂载还是更新逻辑，具体实现代码如下：

```
const processComponent = (...) => {  if (n1 == null) {
    // 组件挂载
    mountComponent(...)
  } else {
```

```
    // 组件更新
    updateComponent(n1, n2, optimized)
  }
}
```

3.5.4 updateComponent()函数

updateComponent()函数内部判断是否需要更新,若需要更新则通过异步方式进行更新;若不需要更新则重置实例指向,更新 instance 和 VNode 的关系后结束。具体实现代码如下:

```
// $ runtime - core_render
const updateComponent = (n1: VNode, n2: VNode, optimized: boolean) => {
  const instance = (n2.component = n1.component)!;
  // 组件是否需要更新
  if (shouldUpdateComponent(n1, n2, optimized)) {
    instance.next = n2;
    // 去除异步队列中的子组件,避免重复更新
    invalidateJob(instance.update);
    // 同步执行组件更新
    instance.update();
  } else {
    // 不需要更新,只需要更新 instance 和 VNode 的关系
    n2.component = n1.component;
    n2.el = n1.el;
    instance.vnode = n2;
  }
};
```

注:因当前组件也有可能状态更改,触发更新但是还未执行,因此清理该组件的更新任务。

3.5.5 shouldUpdateComponent()函数

shouldUpdateComponent()函数通过组件 slots 及 props 判断是否执行更新操作,若需要执行更新,则设置组件的 next VNode 并且将异步更新队列中的该组件更新任务清除,以防止重复更新。

下面简单介绍 shouldUpdateComponent()函数的内部实现逻辑,代码如下:

```
// $ runtime - core_componentRenderUtils
export function shouldUpdateComponent(
  prevVNode: VNode,
  nextVNode: VNode,
  optimized?: boolean
): boolean {
  const { props: prevProps, children: prevChildren, component } = prevVNode
  const { props: nextProps, children: nextChildren, patchFlag } = nextVNode
  const emits = component!.emitsOptions
  if (__DEV__ && (prevChildren || nextChildren) && isHmrUpdating) {
    return true
  }
  // 若涉及指令或 transition,则需更新
  if (nextVNode.dirs || nextVNode.transition) {
    return true
  }
// 涉及优化的情况下,根据状态判断是否需要更新
```

```
  if (optimized && patchFlag >= 0) {
    …
}else{
    …
}
```

上述代码在有 patchflag 优化的情况下,进一步判断优化类型,涉及动态插槽、props 判断和模板编译阶段优化。具体代码实现如下:

```
if (patchFlag & PatchFlags.DYNAMIC_SLOTS) {
      // 动态插槽情况
      return true;
    }
    if (patchFlag & PatchFlags.FULL_PROPS) {
      // 全量 props 的情况
      if (!prevProps) {
        // 若没有旧 Props,则由新 Props 决定
        return !!nextProps;
      }
      // 若都存在,则查询有无变化
      return hasPropsChanged(prevProps, nextProps!);
    } else if (patchFlag & PatchFlags.PROPS) {
      // 在模板编译阶段优化动态 props
      const dynamicProps = nextVNode.dynamicProps!;
      for (let i = 0; i < dynamicProps.length; i++) {
        const key = dynamicProps[i];
        if (nextProps![key] !== prevProps![key]) {
          return true;
        }
      }
}
```

在不涉及 patchflag 优化的情况下(例如,手动创建 render 函数等),进一步对比是否有更新的情况,若有则直接强制更新,具体代码实现如下:

```
if (prevChildren || nextChildren) {
    if (!nextChildren || !(nextChildren as any).$ stable) {
      return true;
    }
  }
  // props 未改变
  if (prevProps === nextProps) {
    return false;
  }
  if (!prevProps) {
    // 没有旧 props ---> 由新 props 决定
    return !!nextProps;
  }
  if (!nextProps) {
    // 存在旧 props ---> 不存在新 props
    return true;
  }
  // 若新旧 props 都存在,则判断是否有变化
  return hasPropsChanged(prevProps, nextProps);
…
return false;
```

以上代码主要介绍 shouldUpdateComponent()函数内部实现,判断 props、slot 等是否需

要更新的情况。根据判断结果继续查看 updateComponent()函数的内部逻辑。若不需要更新则直接调整 VNode 指向,结束代码执行。具体代码实现如下:

```
n2.component = n1.component
n2.el = n1.el
instance.vnode = n2
```

若需要更新则判断是否为 Suspense 类型组件,若是 Suspense 类型组件则调用 updateComponentPreRender()函数,更新 slots 和 props 后异步执行 update 方法;若不是 Suspense 类型组件则直接执行当前实例的 update 方法。

3.5.6 processElement()函数

组件只是某一段具体 DOM 的抽象,经过不同类型的 patch 递归处理后,最后一次进行 diff 的必然是普通元素,因此直接关注普通元素的更新,即可查看整个更新逻辑的完整步骤。在 patch 函数中找到 processElement()逻辑分支,该分支方法实现了普通元素的更新流程,具体代码如下:

```
// $ runtime - core_renderer
const patchElement = (
    n1: VNode,
    n2: VNode,
    parentComponent: ComponentInternalInstance | null,
    parentSuspense: SuspenseBoundary | null,
    isSVG: boolean,
    optimized: boolean
  ) => {
    // 初始化变量
    const el = (n2.el = n1.el!)
    const oldProps = n1.props || EMPTY_OBJ
    const newProps = n2.props || EMPTY_OBJ
    // 更新 props
    patchProps(...)
    // 更新 children
    const areChildrenSVG = isSVG && n2.type !== 'foreignObject'
    patchChildren(...)
  }
```

为更好地查看核心逻辑,上述代码暂时将钩子函数相关的代码以及针对 patchFlags 的优化操作省略,该函数内主要实现如下逻辑:

(1) 更新 props;

(2) 更新 children。

由代码逻辑可知,patchProps 调用的是初始化时传递的 patchProp 函数,位于 $ runtime-dom_patchProp 文件内,主要是针对 class 和 style 以及指令事件等内容进行对比更新,当 props 更新完成后还需要对 children 进行更新。

3.5.7 patchChildren()函数

对 props 和 children 更新完成后就开始对整个 DOM 结构进行更新,此处直接查看 patchChildren()函数的实现代码如下:

```
// $ runtime - core_renderer
const patchChildren: PatchChildrenFn = (
```

```
  n1,
  n2,
  container,
  anchor,
  parentComponent,
  parentSuspense,
  isSVG,
  optimized = false
) => {
  // 初始化变量
  const c1 = n1 && n1.children;
  const prevShapeFlag = n1 ? n1.shapeFlag : 0;
  const c2 = n2.children;
  const { patchFlag, shapeFlag } = n2;
  // children 存在 3 种情况：文本节点、数组、无 children
  if (shapeFlag & ShapeFlags.TEXT_CHILDREN) {
    // 新 children 文本类型的子节点
    if (prevShapeFlag & ShapeFlags.ARRAY_CHILDREN) {
      // 旧 children 是数组,直接卸载
      unmountChildren(c1 as VNode[], parentComponent, parentSuspense);
    }
    if (c2 !== c1) {
      // 新旧都是文本类型,但是文本内容不相同直接替换
      hostSetElementText(container, c2 as string);
    }
  } else {
    if (prevShapeFlag & ShapeFlags.ARRAY_CHILDREN) {
      // 旧 children 是数组
      if (shapeFlag & ShapeFlags.ARRAY_CHILDREN) {
        // 新 children 是数组
        patchKeyedChildren(
          c1 as VNode[],
          c2 as VNodeArrayChildren,
          container,
          anchor,
          parentComponent,
          parentSuspense,
          isSVG,
          optimized
        );
      } else {
        // 不存在新 children,直接卸载旧 children
        unmountChildren(c1 as VNode[], parentComponent, parentSuspense, true);
      }
    } else {
      // 旧 children 可能是文本或者为空
      // 新 children 可能是数组或者为空
      if (prevShapeFlag & ShapeFlags.TEXT_CHILDREN) {
        // 如果旧 children 是文本类型,那么无论新 children 是哪种类型都需要先清除文本内容
        hostSetElementText(container, "");
      }
      // 此时原 DOM 内容应该为空
      if (shapeFlag & ShapeFlags.ARRAY_CHILDREN) {
        // 如果新 children 为数组,则直接挂载
        mountChildren(
          c2 as VNodeArrayChildren,
          container,
```

```
            anchor,
            parentComponent,
            parentSuspense,
            isSVG,
            optimized
          );
        }
      }
    }
  };
```

创建 VNode 时会对 children 进行标准化处理，在 diff children 的时候可以只考虑 children 的类型为数组、文本和空这 3 种情况。结合代码来看，其中 if 条件的设置也很巧妙，既包含所有情况，又能清晰地拆分出挂载、删除、对比 3 个操作。

满足 diff 条件将会执行 patchKeyedChildren 函数，关于该函数将在 4.3 节中对 diff 算法进行详细介绍。新旧 VNode 的 diff 操作完成后，对映射的 DOM 元素进行调整更新，完成组件的更新步骤。

3.5.8 总结

update 方法执行时，整个执行逻辑与 mount 类似，但在这个过程中将会经历新、旧 VNode 的 diff 过程，找出更新的内容点以最小的代价对 DOM 树进行更新操作。通过对本节的学习，可以掌握 Vue3 内部更新逻辑的处理，帮助读者理解 path 和 diff 的概念。

3.6 unmount 方法

前面介绍了 mount(挂载)方法和 update(更新)方法，本节将介绍 unmount 方法的实现。

3.6.1 涉及文件

unmount 方法涉及文件路径如表 3.5 所示。

表 3.5 unmount 方法涉及文件路径

名 称	简写路径	文件路径
unmount	$ runtime-core_render	$ root/package/runtime-core/src/renderer.ts
unmountChildren	$ runtime-core_render	$ root/package/runtime-core/src/renderer.ts

3.6.2 baseCreateRenderer()函数

unmount 方法同样位于 $ runtime-core_render 文件内，在 baseCreateRenderer()函数内部定义，该函数传入 5 个参数，分别为 vnode、parentComponent、parentSuspense、doRemove 和 optimized，传入参数类型如下：

```
type UnmountFn = (
  vnode: VNode,
  parentComponent: ComponentInternalInstance | null,
  parentSuspense: SuspenseBoundary | null,
  doRemove?: boolean,
  optimized?: boolean
) => void
```

根据传入参数可知,前面 3 个为必传参数,主要涉及 vnode 结构、parentComponent 组件和 parentSuspense 组件。该函数内涉及内容较多,函数内主要涉及逻辑如下:

若涉及 ref 数据,则将其设置为空;

若为 keepalive 缓存组件,则卸载;

若为 component 类型组件,则调用 unmountComponent()函数卸载;

若为 suspense 组件,则卸载;

若为 fragments、array 类型组件,则调用 unmountChildren()函数卸载;

若为 teleport,则在卸载组件的同时卸载子节点;

若为动态子节点,则调用 unmountChildren()函数卸载;

若为 fragment 类型,则调用 unmountChildren()函数卸载;

根据 doRemove 参数判断是否卸载整个 VNode。

上述逻辑根据类型判断待卸载 VNode 类型,并执行不同的卸载逻辑。

3.6.3 ref 数据

判断 VNode 是否有 ref,若有则调用 setRef,传入 null 值,清理该 ref 数据。具体代码实现如下:

```
if (ref != null) {
  setRef(ref, null, parentSuspense, null)
}
```

关于 setRef()的处理,可以在 $runtime-core_rendererTemplateRef 文件中看到具体实现,简单查看该函数的处理流程。

若传入的 ref 为数组类型,则对 ref 进行递归遍历,具体实现代码如下:

```
if (isArray(rawRef)) {
  rawRef.forEach((r, i) => 
    setRef(
      r,
      oldRawRef && (isArray(oldRawRef) ? oldRawRef[i] : oldRawRef),
      parentSuspense,
      vnode,
      isUnmount
    )
  )
  return
}
```

若 VNode 属于异步加载组件,且状态不是卸载,则直接跳过。因为异步组件的引用在内部组件,所以此处直接返回即可。

```
if (isAsyncWrapper(vnode) && !isUnmount) {
    return
}
```

若未传入 VNode,则直接设置 value 为 null,否则判断是否为组件类型,若是组件类型则提取组件的参数,若两者均不是则直接提取 el 参数并设置为 value。具体代码实现如下:

```
const refValue =
  vnode.shapeFlag & ShapeFlags.STATEFUL_COMPONENT
    ? getExposeProxy(vnode.component!) || vnode.component!.proxy
    : vnode.el
const value = isUnmount ? null : refValue
```

完成 value 值设置后，提取 oldRef 和 refs 以及 setup 状态，若存在 oldRef 则需要进行卸载。具体代码实现如下：

```
// unset old ref
if (oldRef != null && oldRef !== ref) {
  if (isString(oldRef)) {
    refs[oldRef] = null
    if (hasOwn(setupState, oldRef)) {
      setupState[oldRef] = null
    }
  } else if (isRef(oldRef)) {
    oldRef.value = null
  }
}
```

获取当前 rawRef 的 ref 属性，判断该属性的类型，如果是非空值，则直接设置为－1，标识卸载；如果是空值，则根据不同类型处理 ref 属性。具体代码实现如下：

```
// 如果是函数,则直接执行
if (isFunction(ref)) {
  callWithErrorHandling(ref, owner, ErrorCodes.FUNCTION_REF, [value, refs])
} else {
  const _isString = isString(ref)
  const _isRef = isRef(ref)
  // 如果是 string 或 Ref 类型
  if (_isString || _isRef) {
    const doSet = () => {
      ...
    }
    if (value) {
      // 如果是非空值,则设置 id 为 - 1(标识卸载)
      ;(doSet as SchedulerJob).id = -1
      queuePostRenderEffect(doSet, parentSuspense)
    } else {
      doSet()
    }
  } else if (__DEV__) {
    warn('Invalid template ref type:', ref, `($ {typeof ref})`)
  }
}
```

如果是字符串类型，则判断是否有 value 值，若有则通过 queuePostRenderEffect()函数排队执行 doSet()函数，若无 value 值则直接执行 doSet()函数。该函数主要设置 refs 内对应的 ref 属性值。具体代码实现如下：

```
const doSet = () => {
  // refInFor 标识
  if (rawRef.f) {
    // 判断 ref 的类型获取值
    const existing = _isString ? refs[ref] : ref.value
    // 如果标识卸载
    if (isUnmount) {
      // 如果是数组,则通过 remove 删除
      isArray(existing) && remove(existing, refValue)
    } else {
      // 如果不是数组
      if (!isArray(existing)) {
```

```
            // 如果是字符串
            if (_isString) {
              refs[ref] = [refValue]
            } else {
              // 如果是其他情况
              ref.value = [refValue]
              // 判断 ref 是否有 key
              if (rawRef.k) refs[rawRef.k] = ref.value
            }
          // 如果是数组但是不包含 refValue 值
          } else if (!existing.includes(refValue)) {
            existing.push(refValue)
          }
        }
      } else if (_isString) { // 如果是字符串类型
        refs[ref] = value
        if (hasOwn(setupState, ref)) {
          setupState[ref] = value
        }
      } else if (isRef(ref)) { // 如果是 ref 类型
        ref.value = value
        if (rawRef.k) refs[rawRef.k] = value
      } else if (__DEV__) { // 无效的 ref 类型
        warn('Invalid template ref type:', ref, `($ {typeof ref})`)
      }
    }
```

该函数根据传入参数来判断是设置 ref 还是清理 ref。若传入的是 null,则对应 value=null,将 ref 值设置为 null,以达到清理 ref 数据的目的。

3.6.4 keepalive 组件

完成 setRef 逻辑后,根据代码执行顺序继续查看后续处理。首先判断是否为 keepalive 组件,若是则在 parentComponent 实例上删除该组件缓存。主要代码如下:

```
if (shapeFlag & ShapeFlags.COMPONENT_SHOULD_KEEP_ALIVE) {
  ;(parentComponent!.ctx as KeepAliveContext).deactivate(vnode)
  return
}
```

deactivate 方法内主要调用 move 函数对元素节点进行移除,完成后调用对应钩子函数即可。涉及代码如下:

```
sharedContext.deactivate = (vnode: VNode) => {
  const instance = vnode.component!
  // 传入 null,进行清理
  move(vnode, storageContainer, null, MoveType.LEAVE, parentSuspense)
  // 通过异步队列的方式进行钩子函数的回调
  queuePostRenderEffect(() => {
    if (instance.da) {
      invokeArrayFns(instance.da)
    }
    const vnodeHook = vnode.props && vnode.props.onVnodeUnmounted
    if (vnodeHook) {
      invokeVNodeHook(vnodeHook, instance.parent, vnode)
    }
    // 标识为卸载状态
```

```
      instance.isDeactivated = true
    }, parentSuspense)
    if (__DEV__ || __FEATURE_PROD_DEVTOOLS__) {
      // Update components tree
      devtoolsComponentAdded(instance)
    }
  }
```

keepalive 组件调用 deactivate 卸载 VNode 后，结束 unmount 操作。若不是 keepalive 组件，则进一步判断组件类型。

3.6.5 component 组件

若为 component 类型，则调用 unmountComponent()函数处理，涉及代码如下：

```
if (shapeFlag & ShapeFlags.COMPONENT) {
  unmountComponent(vnode.component!, parentSuspense, doRemove)
} else { ... }
```

unmountComponent()函数内包括如下逻辑：

(1) 停止该组件上的 effects 副作用函数执行；

(2) 停止 update 队列并且调用组件卸载函数 unmount；

(3) 将当前组件标记为已卸载；

(4) 处理 suspense 类型组件。

该函数执行完成后，对应组件卸载即完成，主要代码实现如下：

```
const unmountComponent = (
    instance: ComponentInternalInstance,
    parentSuspense: SuspenseBoundary | null,
    doRemove?: boolean
) => {
  const { bum, effects, update, subTree, um } = instance
  // beforeUnmount 钩子函数回调触发
  if (bum) {
    invokeArrayFns(bum)
  }
  // 停止所有副作用函数
  if (effects) {
    for (let i = 0; i < effects.length; i++) {
      stop(effects[i])
    }
  }
  // 如果组件在异步设置之前被卸载，则 update 可能为空
  if (update) {
    stop(update)
    unmount(subTree, instance, parentSuspense, doRemove)
  }
  // unmounted 钩子函数回调
  if (um) {
    queuePostRenderEffect(um, parentSuspense)
  }
  queuePostRenderEffect(() => {
    instance.isUnmounted = true
  }, parentSuspense)
  // 判断 suspense 组件情况，从父 suspense 组件内删除，并且收集的依赖减少 1，如果是最后一个，则
  // 直接执行 resolve 表示结束；如果还有其他 suspense，则忽略
```

```
    if (
      __FEATURE_SUSPENSE__ &&
      parentSuspense &&
      parentSuspense.pendingBranch &&
      !parentSuspense.isUnmounted &&
      instance.asyncDep &&
      !instance.asyncResolved &&
      instance.suspenseId === parentSuspense.pendingId
    ) {
      parentSuspense.deps--
      if (parentSuspense.deps === 0) {
        parentSuspense.resolve()
      }
    }
  }
```

3.6.6 suspense 组件

如果是 suspense 类型组件,则调用 suspense 对象上的 unmount 卸载。

```
  if (__FEATURE_SUSPENSE__ && shapeFlag & ShapeFlags.SUSPENSE) {
    vnode.suspense!.unmount(parentSuspense, doRemove)
    return
  }
```

3.6.7 telport 组件

完成该处理后继续判断组件是否为 telport 类型,若为 telport 类型组件,则同上述处理,调用 telport 类型的 remove 方法进行卸载。然后根据传入的 doRemove 参数判断是否调用 remove 方法,删除对应 DOM 节点。完成后执行 unmount 钩子函数,涉及代码逻辑如下:

```
  if (shapeFlag & ShapeFlags.TELEPORT) {
    (vnode.type as typeof TeleportImpl).remove(
      vnode,
      parentComponent,
      parentSuspense,
      optimized,
      internals,
      doRemove
    )
  }
```

3.6.8 动态子组件等

如果是动态子组件,且是特定 fragment 或数组类型,则直接调用 unmountChildren()函数,传入不同参数进行处理,核心代码如下:

```
  if (
    dynamicChildren &&
    // #1153: fast path should not be taken for non-stable (v-for) fragments
    (type !== Fragment ||
      (patchFlag > 0 && patchFlag & PatchFlags.STABLE_FRAGMENT))
  ) {
    // fast path for block nodes: only need to unmount dynamic children.
    unmountChildren(
      dynamicChildren,
```

```
        parentComponent,
        parentSuspense,
        false,
        true
      )
    } else if (
      (type === Fragment &&
        (patchFlag & PatchFlags.KEYED_FRAGMENT ||
          patchFlag & PatchFlags.UNKEYED_FRAGMENT)) ||
      (!optimized && shapeFlag & ShapeFlags.ARRAY_CHILDREN)
    ) {
      unmountChildren(children as VNode[], parentComponent, parentSuspense)
    }
```

在 unmountChildren()函数内，无论是 fragment 或 array 类型，均通过 for 循环逐个调用 unmount 函数卸载。

3.6.9 总结

unmount 函数内部主要针对 VNode 的类型执行不同的卸载逻辑。熟悉本节内容有助于理解组件的卸载对不同类型的处理方法，进一步理解组件的挂载、更新和卸载的整体流程。