



# Vue3 深度解析

## 前言

距离尤雨溪首次公开 Vue3 (vue-next)源码有一个多月了。青笔观察到，刚发布国庆期间，出现不少解读 Vue3 源码的文章。当然不少有追风蹭热之嫌，文章草草讲讲响应式原理，或者只是做了一些上层的导读，告诉读者应该先看哪再看哪。不能说这些文章就没有一点价值，它确实能够让你在短时间内，不用过多思考就能了解到一些 Vue3 重中之重的“干货”。但是过于干货的未必就是好的。因为干货通常是经过作者咀嚼过后的产物，大部分营养其实只被作者消化了。留给读者的只是一些看似很有料，实则没有营养的残渣。就像一块啃到只剩骨头的排骨。这样的文章通常适合于媒体传播，仅用于快速捕获眼球。但是对于想更细致了解 Vue3 的专业前端开发，这显然远远不够。

事实上，这不是青笔第一篇关于 Vue3 的文章。在 Vue3 公布后的第五天，也就是10月10号。青笔没有直接解读源码，而是从一个想要自己开发或参与 Vue3 项目的角度，讲到了构建 Vue3 所用到的构建工具和相关技术。文章不仅仅是给出最后的“干货”，而是把青笔在实践过程中的用到的方式方法，包括得到结果的每一行 `shell` 命令，`git` 技巧等。读者完全可以按照文章的脉络得到和青笔一样的结果。这样做是青笔自身多年软件开发经验，所坚持的一个观点，那就是“\*\*”技术不是用来看的，而是用来实操的“\*\*”。只有当你亲自实践才能真正理解其中的内涵，并且这也是最简单和行之有效的学习方式。那些读起来晦涩难懂，繁杂抽象的概念术语，其实最怕的就是被实操，因为一旦遇到一个身怀实操大法的读者，它的所有“江湖禁术”将被见招拆招，一一破解。

但是想要少走弯路，高效率地实践，前提是有一篇相关的文章。

### [从零开始构建 vue3](#)

本文依然坚持这样的准则，带你从实践角度来探秘 Vue3 的源码。你也可以理解为“授之以鱼不如授之以渔”。

## 1. 准备工作

为了顺利完成后面的实践。请先确保你的电脑已经安装了以下工具。

1. [git](#)
2. [node 10 及以上版本](#) (LTS版)
3. [yarn](#)
4. [lerna](#)
5. [typescript](#)

其中 `lerna` 和 `typescript` 使用 npm 进行全局安装。安装方式如下：

```
npm install -g lerna
npm install -g typescript
```

sh

## 2. 先人一步 体验 Vue3 搭建下一代网页应用

### 2.1 Composition API

事实上早在 Vue3 源码公布之前，Vue 官方已经透露了代表下一代 Vue 技术的 Vue3 将采取的新的接口使用方式。这种新的方式叫做 **Composition API**（组合式 API）。而与之相对应的经典 API 也是我们所熟知的 Vue 使用方式叫做 **Options API**（选项式 API）或 **Options-based API**（基于选项的 API）。

在经典的 **Options API** 中，我们使用一个具有 `data`，`methods` 等“选项”的 JS 对象来定义一个页面或者组件。这种简单直接的方式，在应用早期阶段，代码和业务逻辑较简单时，非常的友好亲民，这也是 Vue 以学习门槛较低而广受开发者亲昵的一个因素。但是，有过开发大型 Vue 应用的开发者应该心有体会。当页面的逻辑越来越多时，我们的组件将变得冗长复杂。很多本可以复用的逻辑和代码，你很难有一种使用起来非常舒适的方式来复用。亲笔自身实践，在 Vue2 中，组件逻辑和代码复用最常用的方式是混入 `mixin`，这虽然是一种可行的方式，但是这种方式显然从出生和 Vue 框架紧密耦合。当你想要将一个框架无区别的普通 JS 函数或者对象复用到 Vue2 开发的组件中时，你发现一切都是那么的不优雅。

基于满足在开发大型 Vue 应用中更优雅地复用已有代码的需求催生下，Vue3 **Composition API** 似乎是顺势而为，并且势在必得。

[vue-composition-api-rfc](#)

### 2.2 第一个 Composition API 应用

据官方介绍，Vue3 正式发布将在明年第一季度。但这并不影响我们提前使用 **Composition API** 开发一个简单的响应式 WEB 应用。

并且作为解读 Vue3 源码的前戏，我们将直接在最新源码上进行实操（你很快就会发现这样做的好处）。

#### 2.2.1 克隆源码与初始化

为了精简篇幅，这里直接整个给出所有命令。想了解更多细节，推荐青笔另一篇专栏文章 [《从零开始构建 vue3》](#)，里面有对相关细节的详细讲解。

```
# 克隆源码
git clone https://github.com/vuejs/vue-next.git
# 进入源码目录
cd vue-next
```

sh

```
# 安装依赖
yarn
# 初次构建
yarn build
# 建立项目内部 packages 软链
lerna bootstrap
```

这里需要特别讲到的是最后一步 `lerna bootstrap`，这里实际就是在项目根目录的 `node_modules` 创建了一个符号链接（或软链）`vue` 和一个 scope 目录 `@vue`。

在 macOS 或其他 linux 发行版上可以通过如下命令查看链接指向。

```
ls -l node_modules/ | grep vue
ls -l node_modules/@vue
```

sh

可以看到 `vue` 和 `@vue` 下的符号链接分别指向了源码目录 `packages/` 下对应的目录（文件夹）。

这样，我们就可以在 Vue3 正式发布到 npm 前，直接使用源码里的各个 package，等效于使用从 npm 安装的其他依赖。并且，由于 Vue3 使用 Typescript 编写，里面已经安装和提供编写 Typescript 所有需要开发依赖和配置。因此，我们可以在源码项目里使用和 Vue3 源码一样的方式书写 Typescript 程序。不用担心，即使还不熟悉 Typescript 也不影响继续阅读本文。

### 2.2.2 编写第一个 Vue3 Composition API 网页

为了不污染了 Vue3 源码目录结构。我们可以创建一个新的分支。

```
git checkout -b examples
```

sh

在根目录下创建 `examples` 目录，用于存放示例代码。

```
mkdir examples
```

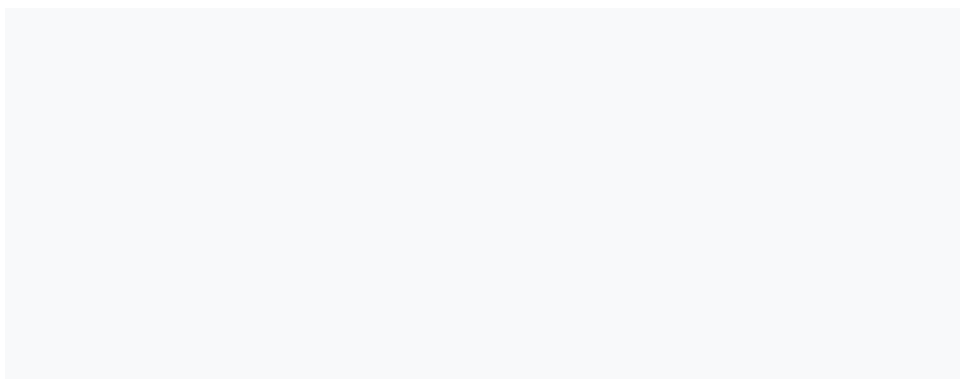
sh

新建文件 `./examples/composition.html`，添加如下内容：

```
<html>
<head><title>vue3 - hello composition!</title></head>
<body>
  <div id="app"><p>{{ state.text }}</p></div>
  <script src="../node_modules/vue/dist/vue.global.js"></script>
  <script>
    const { createApp, reactive, onMounted } = Vue
    const state = reactive({ text: 'hello world!' })
    const App = {
      setup () {
        onMounted(() => {
          console.log('onMounted: hello world!')
        })
        return { state }
      }
    }
    createApp().mount(App, '#app')
  </script>
</body>
</html>
```

html

使用 Chrome 浏览器打开这个 html 文件。在控制台可以访问我们定义的全局变量 `state`。可以任意修改 `state.text` 的值，你会看到网页显示的文本会随着新的赋值而变化。



恭喜你！你已经成功使用 `Vue3 Composition API` 编写了一个响应式 Web 应用。

可以看到不同于 Vue2 选项API丑陋的组件定义。`Vue3 Composition API` 提供一系列 Api 函数，通过简单组合（这也是 Composition 的含义所在），就构建了一个 Vue3 响应式 Web 应用，一切看起来那么自然舒服。可以

预见，随着函数式编程的日趋流行，[Vue3 Composition API](#) 势必成为构建下一代 Vue 应用的首选和主流方式。

### 3. 源码探秘

看过青笔专栏[《从零开始构建 vue3》](#)的读者应该知道，Vue3 源码分为几个不同的 [package](#)，存放在目录 [./packages/](#) 下，并使用 [lerna](#) 来管理多 [package](#) 项目。

```
packages/  
├─ compiler-core  
├─ compiler-dom  
├─ compiler-sfc  
├─ reactivity  
├─ runtime-core  
├─ runtime-dom  
├─ server-renderer  
├─ shared  
└─ vue
```

其中 [compiler-sfc](#) 是 Vue 单文件组件（也就是我们在 Webpack 下使用的 .vue 文件）的实现，[server-renderer](#) 是服务端渲染的源码，这两个部分截止本文写作时，还未完成；[shared](#) 是各个 [package](#) 共享的实用库（相当于我们平时使用的 utils），里面封装的都是一些例如判断是否是数组，是否对象和函数等通用函数，因此从理解 Vue3 源码角度，可以不去关注；而 [vue](#) 就是最终要发布的 [Vue3](#) 的包，但是从源码来看，这仅仅是内部模块对外的导出出口，它的源码也只有一个 [index.ts](#) 文件，通过这个文件我们可以知道，最终 [Vue3](#) 对外提供了哪些接口，也就是前面我们创建 [Composition API](#) 网页里面使用的全局对象 [Vue](#) 里支持的 API 函数。

缩小我们的关注范围，构成 [Vue3](#) 最核心的是以下 5 个 package:

- [reactivity](#)
- [compiler-core](#)
- [runtime-core](#)
- [compiler-dom](#)
- [runtime-dom](#)

而这其中前 3 个 package 即 [reactivity](#)，[compiler-core](#)，[runtime-core](#) 又是 [Vue3](#) 核心中的核心（正如 core 一词所表示的含义）。可以说这 3 个 package 是构建整个 [Vue3](#) 项目乃至整个 [Vue3](#) 生态的最底层依赖和基石。为了更加生动的理解这句话的含义。我设想一个这样的画面。

在一个秋高气爽的午后，尤雨溪同学抱着自己 13 英寸的 macBookPro 来到自己最常光顾的咖啡店，点了一杯拿铁。打开 VSCode 准备撸代码，冥冥之中看到了一个叫做 AngularJs 的东东。突然一个念头闪现在尤同学的脑海中。“wokao，这家伙，得劲啊! 我也弄一个...”。经过一段苦思冥想。“本尤要做一个更屌的，不仅用于构建 WEB 界面，还能使用前端熟悉的 html 模版构建手机 App 等任何客户端界面”。而要达

到这个效果，必须在设计时就要把页面模版解析（编译）和渲染输出进行解耦，于是，尤同学新建一个文件夹，命名为 `compiler-core`，用于存放实现将使用 `html` 编写的模版编译成 `抽象语法树` 和 `渲染桥接函数`（用于解耦渲染函数实现的 `桥`）的代码，有了模版编译解析，仅仅只有渲染层的抽象，但还需针对应用级别进行抽象，来运行应用，于是尤同学新建了第二个文件夹，命名为 `runtime-core`，用于存放创建应用和应用渲染器的抽象，这其中也包含了构成应用的组件和节点的抽象。到这一步，一个从 `html` 模版（字符串）构建应用视图界面的抽象已经完成，但是为了将视图显示的内容与数据进行绑定，实现修改数据时，就能响应式地改变视图内容，还需要一个响应数据变化的模块，于是尤同学又新建了第三个文件夹，命名为 `reactivity`，经过技术分析，尤同学认为当前使用 ES6 的新特性 `Proxy` 来实现数据响应是最优雅的方式，于是尤同学决定在这个文件夹里存放管理所有基于 `Proxy` 封装的响应式模块。不同于前两个 `package` 是对平台和环境的抽象，`reactivity` 是一个具像的实现，正如我们前面使用 `Composition API` 构建的 `hello world` 网页中使用的 `reactive` 函数就是导出自 `reactivity`。至此，用于实现构建任何用户界面的底层抽象和响应式数据模型已经完成。距离将这个视图设计方式应用到最终的产品中，还差一个将抽象的平台无差别的 `compiler-core` 和 `runtime-core` 的平台级实现。但是要实现所有平台的视图渲染，可不是一个小的工作量，前提你要会相关平台界面开发，例如 `IOS APP` 或 `Android APP` 的界面开发。可溪，尤同学只学过 `Web` 前端。于是，尤同学先从自己熟悉的入手，添加了两个用于在浏览器下环境的模块渲染和应用运行时实现，即 `compiler-dom` 和 `runtime-dom`。"...不知不觉，又过了一个秋"。尤同学终于将一年前那个设想在浏览器环境下实现，但是，距离最终目标显然还有一段路要走。尤同学接下来首要任务是先实现单文件组件 `package` 和服务端渲染 `pacakge`，来满足在 `Webpack` 环境更好开发 `Vue3` 应用，以及需要 `SEO` 场景的服务端渲染应用。

温馨提示：本剧情纯属虚构，甚至有点好笑 ^^!

看完这段虚构剧情，想必你已经对当前 `Vue3` 中 5 个最重要的模块有了一个比较清晰的理解。最后，用一张图来总结它们之间的关系。图片中箭头代表依赖。事实上，我们最终使用的 `vue` `package` 就是在浏览器下运行的，因此，`vue` 直接依赖于 `compiler-dom` 和 `runtime-dom`。而 `vue` 到 `reactive` 依赖使用了虚线，是因为，`vue` 不是直接依赖于 `reactivity`，而是通过导出所有 `runtime-dom` 的导出，而 `runtime-dom` 又导出了所有 `runtime-core`，其中包含了 `reactivity` 中创建响应式对象的 `reactive` 函数，通过这种方式间接导出了 `reactive`，也就是前文 `hello-world` `WEB` 应用中使用的函数。

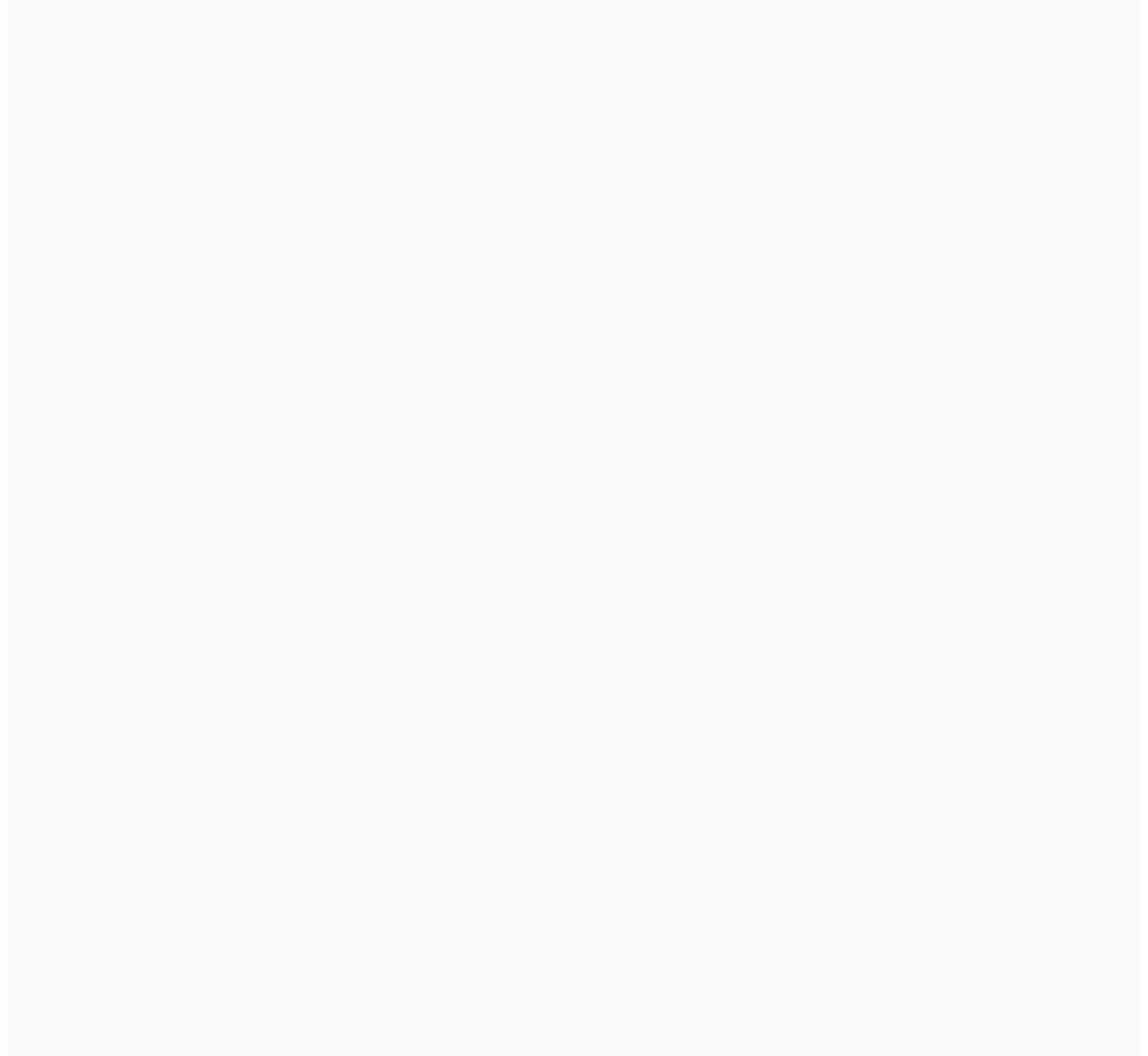
## 4. createApp

我们已经知道构成 Vue3 最核心的 5 个 package 的分工和依赖关系。但是它们之间具体如何相互“协作”，来完成一个完整的 WEB APP 的创建呢。我们以前文使用 **Composition API** 创建的 **hello world** 网页应用为例。以下摘取的是 Javascript 代码部分（这里使用了 ES6 的语法编写）。

```
const { createApp, reactive, onMounted } = Vue
const state = reactive({ text: 'hello world!' })
const App = {
  setup () {
    onMounted(() => {
      console.log('onMounted: hello world!')
    })
    return { state }
  }
}
createApp().mount(App, '#app')
```

js

我们看到最后一行代码，使用了一个 **createApp** 工厂函数创建了一个 Vue3 应用实例，然后将使用 **Composition API** 编写的应用根组件 **App** 挂载到 ID 为 **app** 的 Dom 元素上。这个过程在 **Vue3** 内部是如何传递的，或者说我们前面说的 5 个 package 之间如何协作来完成这个 App 创建的。下面是青笔逐行代码追踪后画出了这样一个调用关系图。



图中添加了背景色的部分是一些比较代表各 package 发挥关键作用的部分。其中黄色部分是 Vue3 在应用中导出的 Api；橙色部分是 `runtime-core` 中创建运行时渲染器；青色部分是 `compiler-core` 及 `compiler-dom` 中用于将模版字符串编译成渲染函数的抽象语法树及 dom 渲染实现；绿色部分是 `reactivity` 导出的两个基本的响应式 API，`reactive` 函数用于传入一个非响应式普通 JS 对象，返回一个响应式数据对象，而 `isReactive` 函数用于判断一个对象是否是一个响应式对象。

## 5. Typescript

---

我们知道 `Vue3` 使用 `Typescript` 编写。但是，这不并不意味着我们必须从头到尾先把 `Typescript` 学习一遍，才能看懂 `Vue3` 的源码。众所周知，Javascript 是一门弱类型的语言，这样带来的好处是减少代码“噪声”（与要实现功能无关的语法成分），让开发者专注于业务逻辑的实现，写出更加简洁易懂的代码；但凡事皆有利弊，当编写对稳定性和安全性有更高要求的大型软件时，类型灵活多变反而成了滋生疑难 BUG 的温床。因此就有了 `Typescript` 这样的强类型的语言，不过它仅仅是 Javascript 的超集，就是说任何合法的 `Javascript` 代码同时也是合法的 `Typescript`。`Typescript` 的核心就是在 `Javascript` 语法的基础上增加了对数据类型的约



束，以及新增一些数据类型（如：元组，枚举，Any等），接口类型（Interface）。而掌握 **Typescript** 的真正难点在于掌握在不同场景下限定类型的方式。具体而言就是变量申明，函数传参，函数返回值，复合（Array，Set，Map，WeakSet，WeakMap）元素类型，接口类型和类型别名。

以下给出了 **Typescript** 最常用也最基本的类型使用方式。

```
ts

// 变量申明
let num: number = 1
let str: string = 'hello'
// 函数参数类型与返回值类型
function sum(a: number, b: number): number {
    return a + b
}
// 复合元素类型
let arr: Array<number> = [1, 2, 3]
let set: Set<number> = new Set([1, 2, 2])
let map: Map<string, number> = new Map([['key1', 1], ['key2', 2]])
// 接口类型
interface Point {
    x: number
    y: number
}
const point: Point = { x: 10, y: 20 }
// 类型别名
type mathfunc = (a: number, b: number) => number
const product: mathfunc = (a, b) => a * b

console.log(num, str, arr, set, map, sum(1, 2), product(2, 3), point)
```

以上的例子，还是比较简单易懂的。个人觉得最 **Typescript** 最难理解的类型，也是 **Vue3** 源码阅读起来最大的障碍是 **泛型 (Generics)**。泛型是一种基于类型的组件（这里的组件是指代码中可复用单元，如函数等）复用机制，这么说有些抽象，简单来说，可以理解为类型变量。通常用于函数，作用类似于面向对象编程里的**函数重载**。

既然说在 Typescript 里范型就像类型变量，那么这个变量如何定义和使用，下面举个例子。

函数 **identity()** 接受 string 类型参数，并返回自身，也是 string 类型。

```
ts

function identity(arg: string): string {
    return arg
}
```

现在不希望参数和返回类型固定为 string，同时又希望能限定类型，最好的办法就是使得类型可变，或者说把类型定义为一个变量。这就是所谓的**泛型**。那么这个“类型变量”在哪定义，答案是在函数名称后面，插入一对尖括号"<>", 并在尖括号里定义这个变量，然后就可以将后面参数和返回类型用这个“类型变量替换”。如下：

```
function identity<T>(arg: T): T {
  return arg
}

console.log(identity<string>('hello'))
console.log(identity<number>(100))
// 也可省略类型部分
console.log(identity('hello'))
console.log(identity(100))
```

想了解更多范型的使用场景，可参考[官方文档](#)

如果认真掌握以上 Typescript 的类型使用，那么基本就可以读懂 **Vue3** 的源码了。虽然，这里列举的特性并非 Typescript 的全部，但是，剩下的已经不影响正确的理解源码，并且相比直接看完并掌握所有 Typescript 的特性，通过阅读 **Vue3** 源码能让你更快速地掌握最重要的特性和最佳实践方法，可谓一举两得。

## 6. 实践理解源码核心部分

说了这么多，最后通过 3 个示例代码，实践总结和加深理解 **Vue3** 最核心 3 个模块的作用，作文本文的收尾。

### 6.1 reactivity

在 `./examples` 目录新建文件 `reactivity.ts`，粘贴如下代码：

```
import { reactive, isReactive } from '@vue/reactivity'

const content = { text: 'hello' }
const state = reactive(content)

console.log('content is reactive: ', isReactive(content))
console.log('state is reactive: ', isReactive(state))

console.log('state ', state)
content.text = 'world'
console.log('state ', state)
```

编译运行：

```
tsc reactivity.ts && node reactivity.js
```

## 6.2 compiler-core

在 `./examples` 目录新建文件 `compiler-core.ts`，粘贴如下代码：

```
import { baseCompile as compile } from '@vue/compiler-core'

const template = '<p>{{ state.text }}</p>'
const { ast, code } = compile(template)

console.log('ast\n----')
console.log(ast)
console.log('code\n----')
console.log(code)
```

ts

编译运行：

```
tsc compiler-core.ts && node compiler-core.js
```

sh

## 6.3 runtime-core

在 `./examples` 目录新建文件 `runtime-core.ts`，粘贴如下代码：

```
import { createRenderer } from '@vue/runtime-core'

const patchProp = function (el: Element, key: string, nextValue: any, prevValue: any, isSVG: boolean)
const nodeOps = {
  insert: (child: Node, parent: Node, anchor?: Node) => {},
  remove: (child: Node) => {},
  createElement: (tag: string, isSVG?: boolean) => {},
  createText: (text: string) => {},
  createComment: (text: string) => {},
  setText: (node: Text, text: string) => {},
  setElementText: (el: HTMLElement, text: string) => {},
  parentNode: (node: Node) => {},
  nextSibling: (node: Node) => {},
  querySelector: (selector: string) => {}
}

const { createApp } = createRenderer({
  patchProp,
  ...nodeOps
})

console.log(createApp())
```

编译运行：

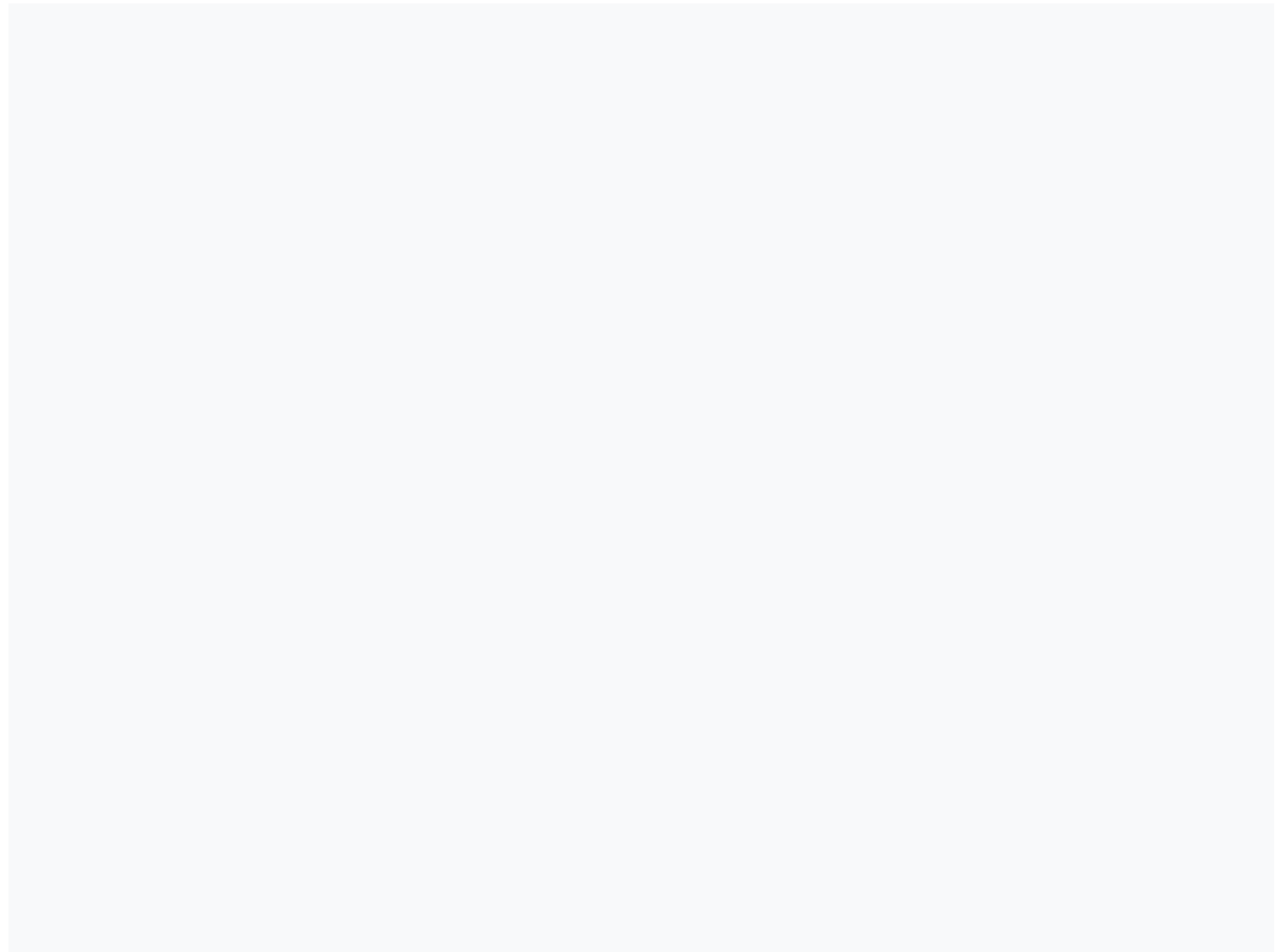
```
tsc runtime-core.ts && node runtime-core.js
```

sh

## 总结

本文从使用 **Vue3** 组合式API搭建第一个响应式 Web 应用开篇，由浅入深，先后讲解了构成 **Vue3** 最重要的 5 个 package 的分工和依赖，并进一步道出构成 **Vue3** 及构建 **Vue3** 生态 3 个最底层的 package，并通过编造一段有趣的故事来帮助读者理解 **Vue3** 的本质。为了扫除读者深入阅读 **Vue3** 源码的心理障碍，增加了针对 **Vue3** 源码所需要掌握的 Typescript 基础知识。最后，通过动手编写 3 个示例代码，分别给出 **Vue3** 响应式数据，模版编译和创建运行时应用最重要的接口，引导读者动手调试 **Vue3** 核心代码，来真正吃透 **Vue3** 的核心原理。

如果你对 **Vue3** 源码和最新发展感兴趣，可以**关注作者微信**，回复：**vue**，加入“Vue3 前端技术交流群”，和作者一起深入探讨学习。



关注作者微信

相关推荐：[《从零开始构建 vue3》](#)

评论

输入评论...